Mitchell Jones
Jesse Bahr
EcEn 425 – Lab4a
10/01/15

1)

Contexts will be saved on the current stack as well as registers and then bp and sp will be saved in the TCB. The sp saved will point to the last register saved or one above.

a) ISRs context must be saved on the stack that they are on. The task scheduler won't be run until the ISRs have stopped running.

b) Blocking functions will save context and registers on the stack as well and then save the value of the stack pointer in the TCB that points to the last register pushed.

2)

- The ISRs will save context and register contents on the stack but then bp and sp will be saved in the TCB. The sp saved will point to the last register saved or one above.
- Use the push instruction to save the registers and C code to move sp and bp values into the TCB.

```
ISRFunction:            ;Will be written in assembly
        push registers
        save bp and sp in TCB
        Call EnterISR
        Enable Interupts
        Call Handler
        Disable interrupts
        Call ExitISR    ;This will call the scheduler etc
```

3)

- The blocking functions will save context and register contents on the stack but then bp and sp will be saved in the TCB. The sp saved will point to the last register saved or one above.
- Use the push instruction to save the registers and C code to move sp and bp values into the TCB.

```
YkDelayTask{            //Will be written in assembly
        inline assembly:
                disable interrupts
                save context on stack
                push registers
                save bp and sp in TCB
        C:
                //Set the time to block the task etc
                enable interrupts
}
```

4)

- We don't want the dispatcher to save these values. We want it to simply point to the already saved values that the isr or blocking task created on the stack. The dispatcher can change the values of the stack and base pointers before it returns – to point to the saved registers of the task that it knows it wants to run using the TCB. A global variable indicating which task is

Mitchell Jones
Jesse Bahr
EcEn 425 – Lab4a
10/01/15

running will help us know what TCB to use.

```
YKDispatcher{
        Figure out what task we want to run
        Access the TCB of that task to get the stack and base pointers

        inline assembly:
            disable interrupts
            change the value of sp and bp to these values
            pop the registers
            iret      ; return using interrupt return
    }
```

5)
- The dispatcher can change the values of the stack and base pointers before it returns – to point to the saved registers of the task that it knows it wants to run using the TCB. A global variable indicating which task is running will help us know what TCB to use.

6)
- If no task switch needs to take place then the scheduler can just return to the ISR and the ISR can finish running.

7)
- In order to stop the function from running again we set a flag in its TCB to tell the scheduler not to run it until the number of ticks transpires and it is taken off the queue.

8)
- The dispatcher will need the tasks TCB that holds all of the information it needs. The scheduler could set a global pointer for the current TCB or pass the TCB to the dispatcher.

```
YKDispatcher(&TCB[index of highest priority task]);
```

9)
- No

10)
- The first time the task is dispatched it will not be different. The settings for the TCB will simply have been set in the YKNewTask

11)
- It will check the task TCB to see if the task is delayed, or blocked. The TCB will hold values that tell us whether or not the stack is ready.
- Ideally, blocked tasks will be pushed off of the queue for ready tasks that we are thinking about running.

12)
- Each tasks TCB will have a priority level and none will have the same priority
- The idle task runs if no other task of higher priority is ready

13)
- The scheduler will pass a pointer (address) to the dispatcher of the TCB for the task that it

has chosen to run.

14)

- If the ISR depth is 0 the ExitISR will call the scheduler in order to execute the highest priority ready task. To ensure proper count, increment and decrement the isr call depth is accurate (on enter and exit respectively).

15)

- The EnterISR and ExitISR increment and decrement and global variable that tracks the depth of our nested ISRs. We will simply check to see if the ISR is not at depth 0 in order to call the scheduler otherwise we will finish running the ISR as usual.

16)

- We will check the depth of the nested ISR variable and if it is not 0 then we will not rewrite new information to the TCB.

17)

- Struct {

```
unsigned short stackpointer;  //To help us know how to get to the correct stack frame
unsigned short basepointer;  //To help us know the base of the correct stack frame
unsigned short programcounter; // Returns us to the proper instruction in the code
unsigned short blockedFlag;          //To help us know if the task is is blocked
unsigned short readyFlag;            //To help us know if task is ready to run
                                     // We may op to use a single short
                                     // with different bits representing respective
                                     // Flags
unsigned short delayCount;           //Help us know if task is delayed
unsigned short stackBeginAddress;  //define a separate task for this task
unsigned short stackEndAddress;    //define a separate task for this task
} TCB;
```

18)

- We want to use a linked list that connects all of our TCBs for general purposes and then we want to make a queue of pointers to our TCBs for our ready tasks

19)

- If it is called before YKRun it could call the scheduler as long as the scheduler did not call the dispatcher. We don't want to actually run the tasks until YKRun is called.
- After YKRun is called it should call the scheduler which should call the dispatcher.

20)

- YKNewTask will initialize all of the values of the struct specified in our answer to question 17.

21)

- The registers should be initialized to something that corresponds to the stackBeginAddress and stackEndAddress for that task (Do this is YKNewTask) Set the segment registers to match those addresses. Our segment register should never be anything other than 0 though. The flag register should not be changed except we should make sure the interrupt bit is set. Initialize the TCB saved program counter to the beginning of the function.

Mitchell Jones
Jesse Bahr
EcEn 425 – Lab4a
10/01/15

```
void YKInitialize(void){
        /* initialize global variables */
        /*Create YKIdleTask TCB*/
        /*allocate the stack space for the YKEdleTask*/
}

void YKNewTask(void (* task)(void), void *taskStack, unsigned char priority){
        Create TCB for the task
        Create the stack space for task
        // Put in scheduler queue, but don't run yet.

}

void YKRun(void){
        Call the scheduler
        // Possibly pre load scheduler queue with startup // initial tasks.
}

void YKDelayTask(unsiged count){
        if count > 0
                //Set the task TCB.delayCount
                // Change state of Task
                if(after count is expired){
                        Make a software interrupt
                        That interrupt will make this task ready again.
                }
}

void YKEnterMutex(void){
        Disable interrupts
}

void YKExitMutex(void){
        Enable interrupts
}

void YKEnterISR(void){
        increment depth variable
}

void YKExitISR(void){
        decrement depth variable
```

Mitchell Jones
Jesse Bahr
EcEn 425 – Lab4a
10/01/15

```
        if(nesting level is 0)
                Call YKScheduler
}

void YKScheduler(void){
        figure out which task is highest priority ready task (using queue)
        call dispatcher sending the task pointer.
}


YKDispatcher(task* task){
                        Figure out what task we want to run (as directed by the scheduler)
                        Access the TCB of that task to get the stack and base pointers

                        inline assembly:
                                disable interrupts
                                change the value of sp and bp to these values
                                 sp =  tcb.stackpointer;
                                bp = tcb.stackpointer;
                                pop the registers
                                YKCtxSxCount
                                iret      ; return using interrupt return
                }

void YKTickHandler(void){
        Decrement delayCounters if nonzero
        increment YKTickNum
        deal with stuff for idleHandler if needed  // Make task available if delay is over.
        // Check to see if user tick handler needs to be used, then do so. (see if action needs to be taken
        on each clock tick.
}
```