Embedded Operating Systems

Condensed version of Embedded Operating Systems course.

Or how to write a TinyOS

Part 2 – Context Switching

John Hatch

Covered in Part One

- ARM registers and modes
- ARM calling standard
- Exception types
- Mode switching on exception
- Vector table
- Context saving and restoring
- Adjusting the return address for exceptions
- Data abort handler example

Switching Tasks – What's needed

Switching between tasks requires:

- Setting up stacks for each task
- Tracking stack pointers for each task
- Priming the stacks with an initial context
- Saving the full context
- Saving the task's stack pointer
- Switching tasks by switching stack pointers
- Restoring the full context for next task
- Using the timer to drive context switches

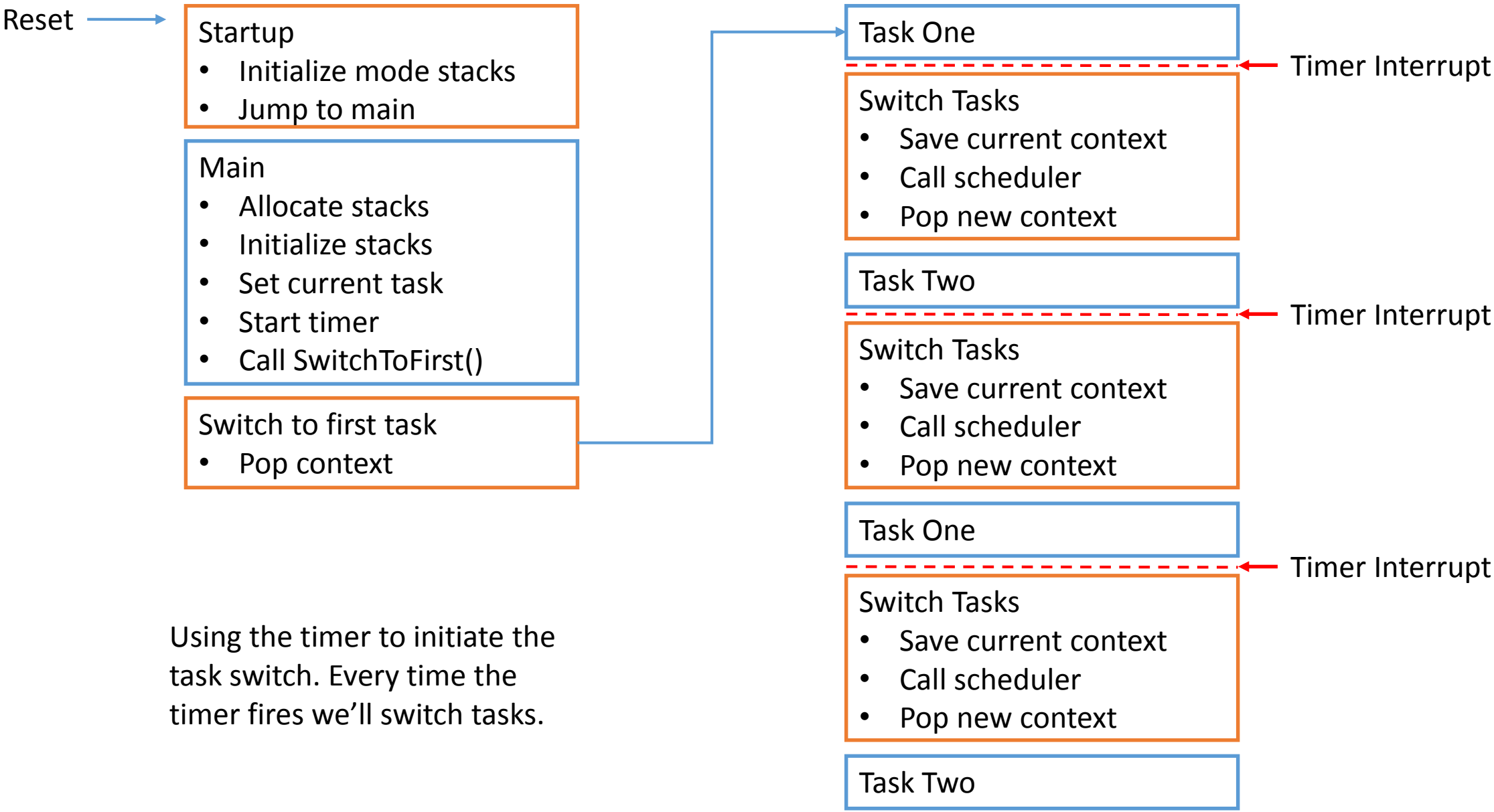
Some Important Points

- Each mode has its own stack pointer
  - So each mode can have own stack
  - Which we set up in the reset handler
- Which we can use as temporary scratch pad
- This will be important for interrupt handling and context switching
- It will give us a place to temporarily safe registers so that we can use them as variables.
- Which we will need to shuffle data between modes

Simplifications

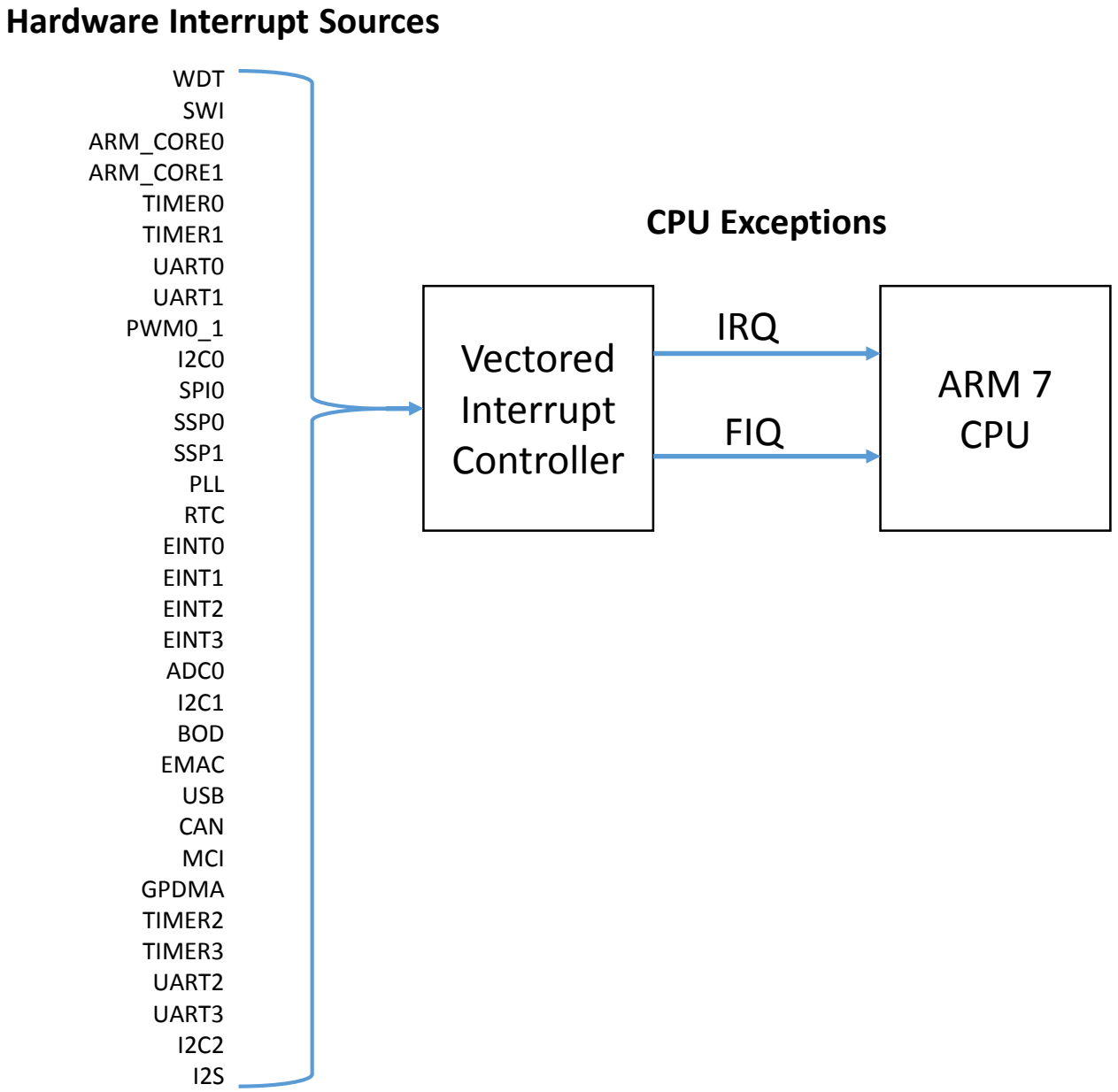
- No heap memory
- Mode stacks are allocated in the linker definition file and assigned in the reset handler
- Tasks never return. Implemented as endless loops.
- Stacks are allocated as stack globals
- Each timer interrupt will cause a task switch.

Simple Task Switching Overview



Using the timer to initiate the task switch. Every time the timer fires we'll switch tasks.

Hardware Interrupts to CPU Exceptions



- All hardware interrupts go through the VIC
- VIC handles 32 interrupt sources
- VIC is programmable
- Interrupts can fire either an IRQ or FIQ exception
- Each sources has a vector that can be programmed with the address of an interrupt handler.
- When the interrupt fires the IRQ handler gets the function address from the VIC and calls it.

Defining a Task

```
//
// TaskOne counts up from 0.
// Never exits.
//
void TaskOne(void)
{
    int count = 0;
    while(1)
    {
        printString("task one: ");
        print_uint32(count++);
        printString("\n");

        // delay
        for(int i=0; i<10000; i++);
    }
}
```

- A task is simply a function that does not return.
  - No where for it to return to since it starts the call chain for its context.
- Each task has its own stack
  - A stack is simply an array of bytes allocated before the task runs.
  - When the task starts the SP register must point to the highest free address in the stack.

Allocating a stack

```
// Allocate a stack
int stackOne[STACKSIZE];

// Calculate the beginning of the stack.
int taskOneSP = stackOne + STACKSIZE - 1;
```

- A simply way to allocate the stack is simply to define a global array.
- The stack pointer is initialized to the highest memory location in that array.
- Each task needs its own stack.
- The task will use its stack to store locals and make function calls.
- The stack will also be used to save the task's context when there is a task switch.

Tracking the running task

```
// The value of the current task's stack pointer
int currentSP;

// The ID of the current task
int currentTaskID;

// Array of stack pointers. One for each task.
int Tasks[NUMTASKS];
```

- A simple way to track which task is running is to have two global variables.
  - currentSP points to the stack of the current running task.
  - currentTaskID is the id of the current running task.
- The globals make it easy to share information between the C code and the assembly code.

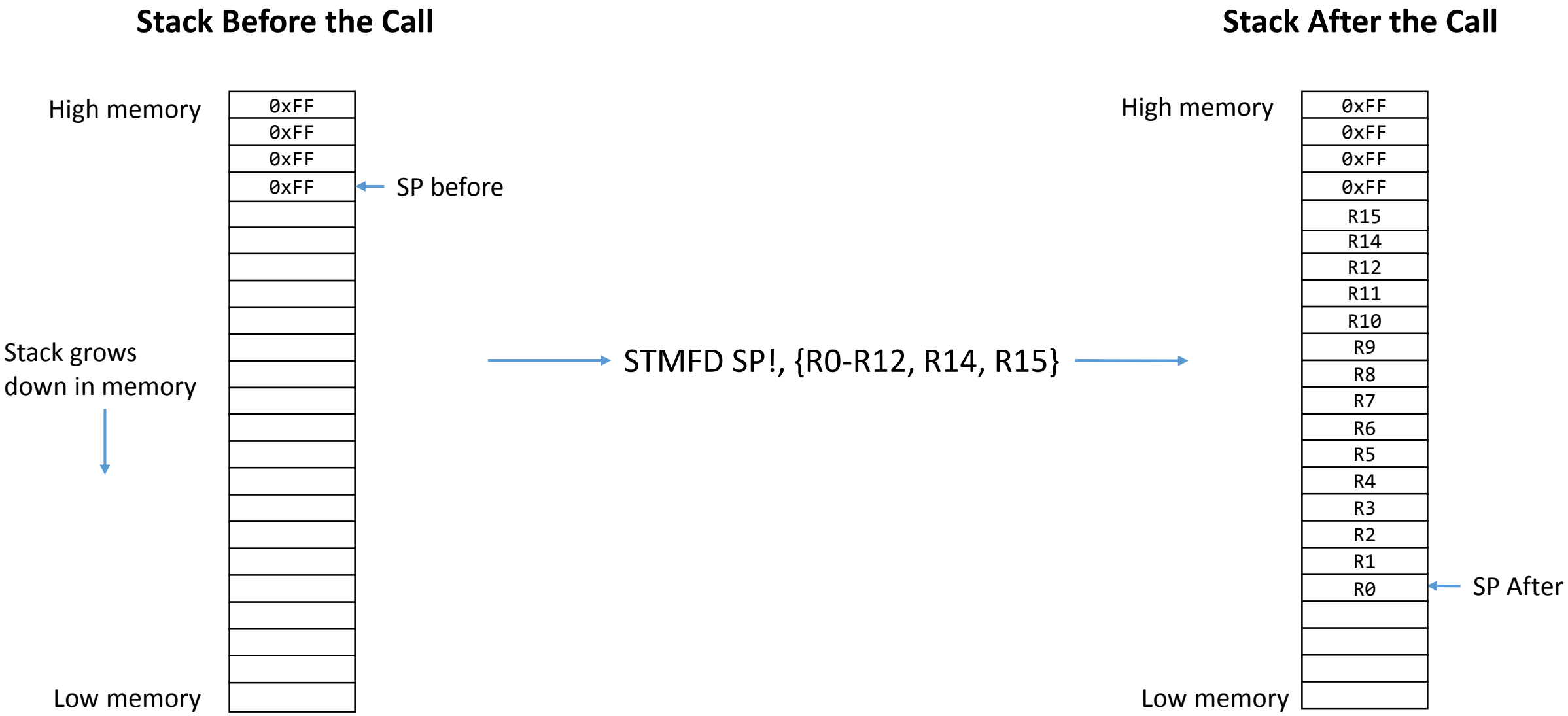
Saving a Task's Full Context

- When the timer interrupts we need to save the current context on to the current task's stack.
- We must save the full context, that is all the registers and the value in the SPSR.
- There are no instructions for saving SPSR directly to memory. The value has to be copied to a register first.
- The SPSR is the Interrupt Mode's SPSR.
  - We need to get the value over into the Supervisor Mode.
  - If we switch modes then we won't be able to see the Interrupt Mode's SPSR register.
  - So we need to save the SPSR value in a register before switching modes.
  - Be can't use any registers until their values have been saved.
- But we want SPSR to be the first thing saved so we'll need to juggle a little.
- We can do this be temporarily saving some registers to the Interrupt mode's stack. That will give use some registers to play with.

How Store Multiple and Load Multiple work

- To push all the register on to the stack we use the Store Multiple instruction.
  - STMFD SP!, {R0-R12, R14, R15}
- It always writes the values from the register from lowest to highest to the address pointed to by SP.
- That means the lowest register is at the lowest memory location, the highest register ends up at the highest memory location

How Store Multiple saves to the stack

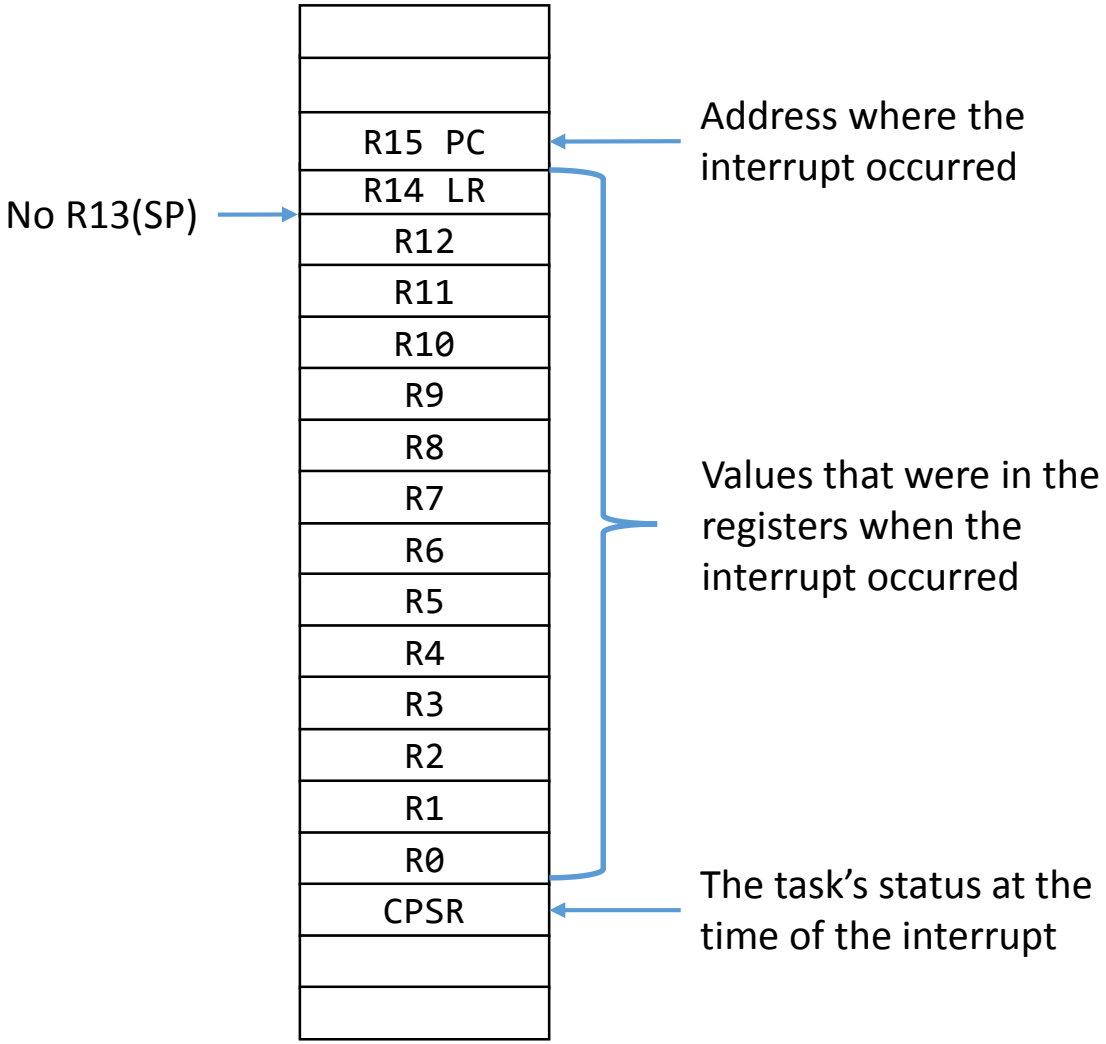


Full Context on the Stack

This is how we want the task's stack to look after we save the task's full context.

It has all the information we need to restore the task later.

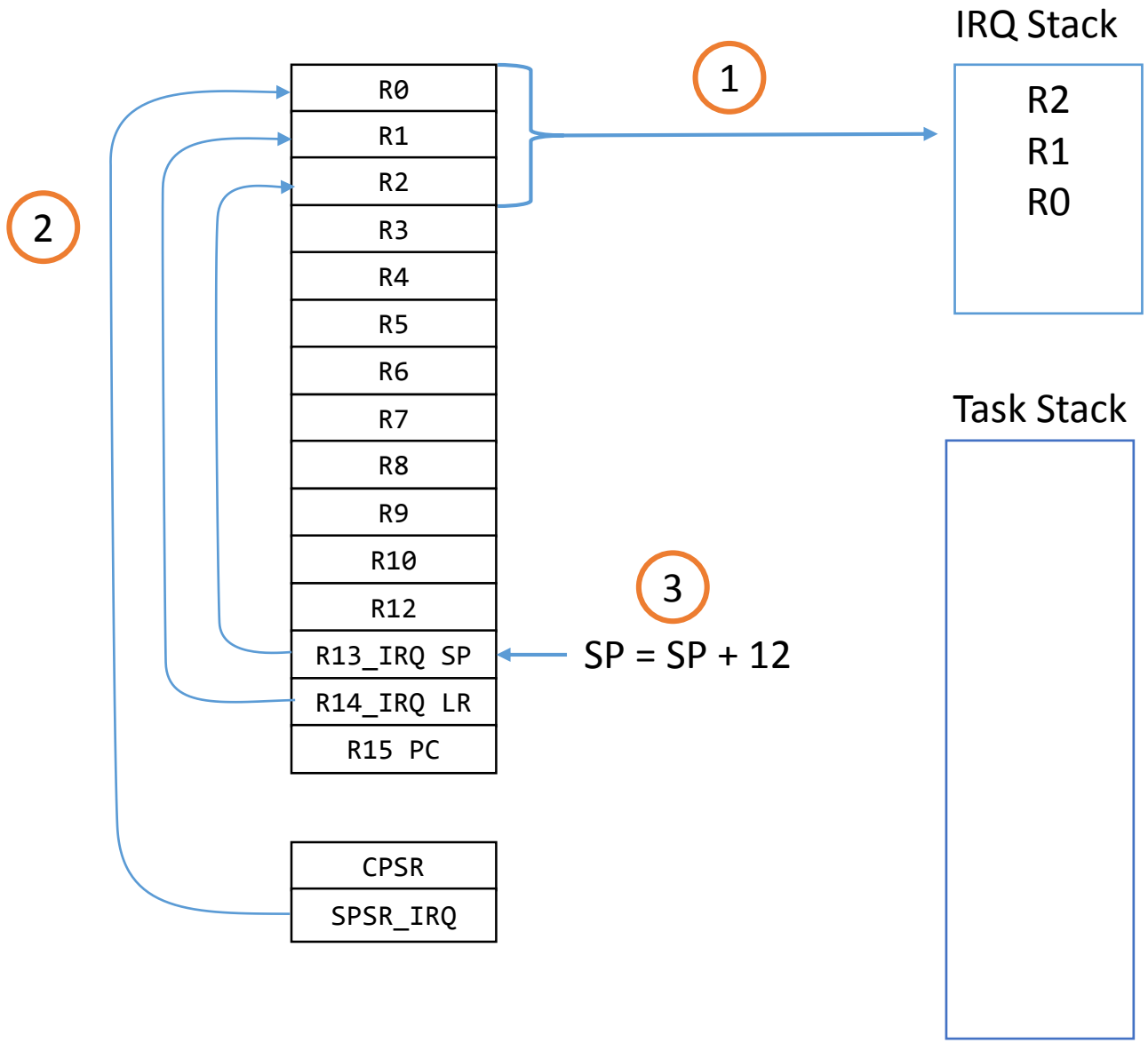
To get this we first push R0-12, LR, PC to the stack. Then we push the value saved in the Interrupt mode's SPSR. That value was the task's CPSR at the time of the interrupt.



Steps for a context switch

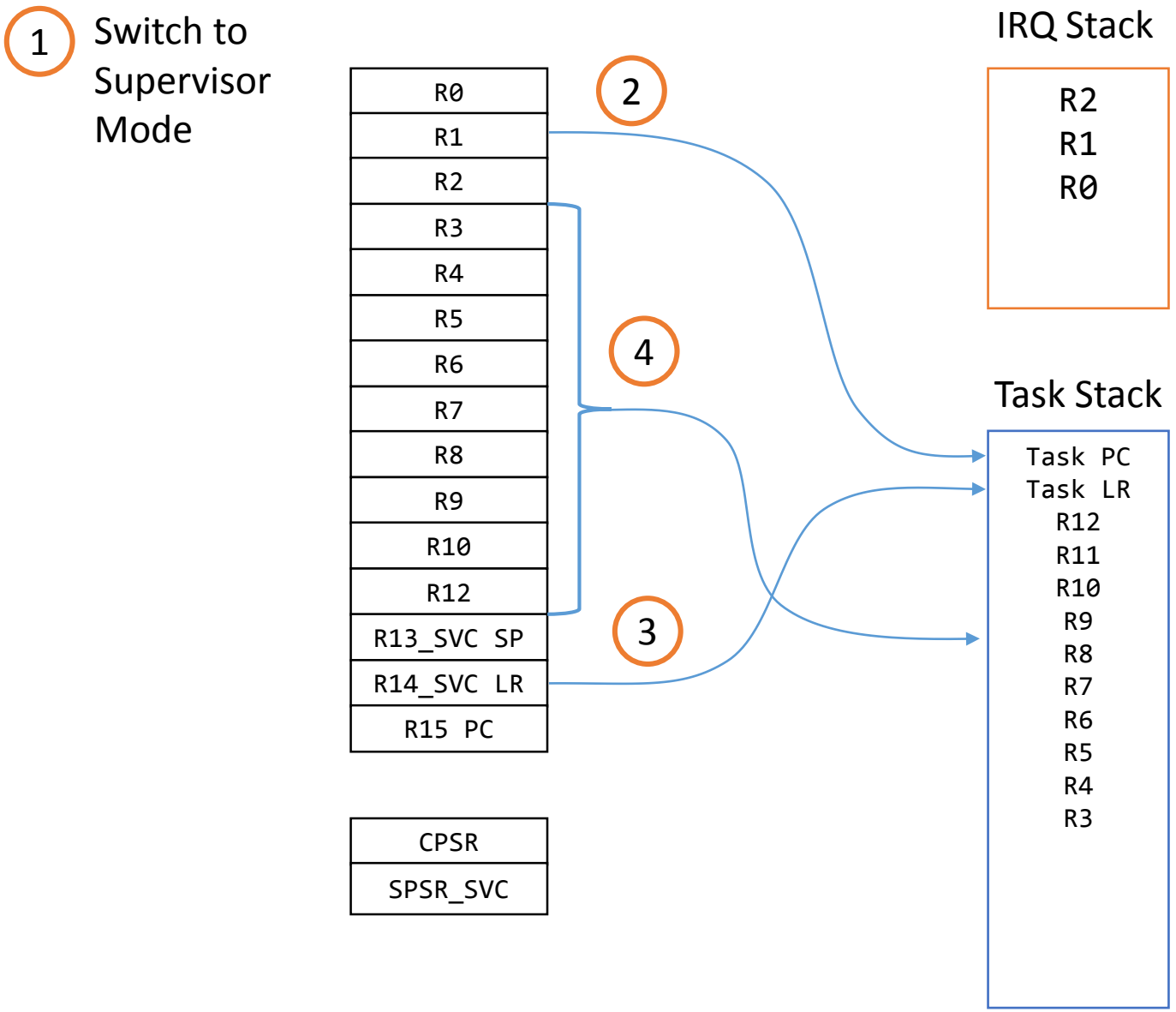
- Save some registers to the IRQ stack so that we can use the registers as variable
- Remember where we save the original values
- Save the SPRS, LR and SP for later in the freed registers
- Switch to the Supervisor mode
- Push registers to the task’s stack
- Retrieve the values saved on the IRQ stack and save them to the task’s stack
- Save the stack pointer to a global
- Call the scheduler who will save the value in the global, pick the next task, and set the global with the next task’s stack pointer.
- Pop the next task’s context of its stack

First Step to Saving Context



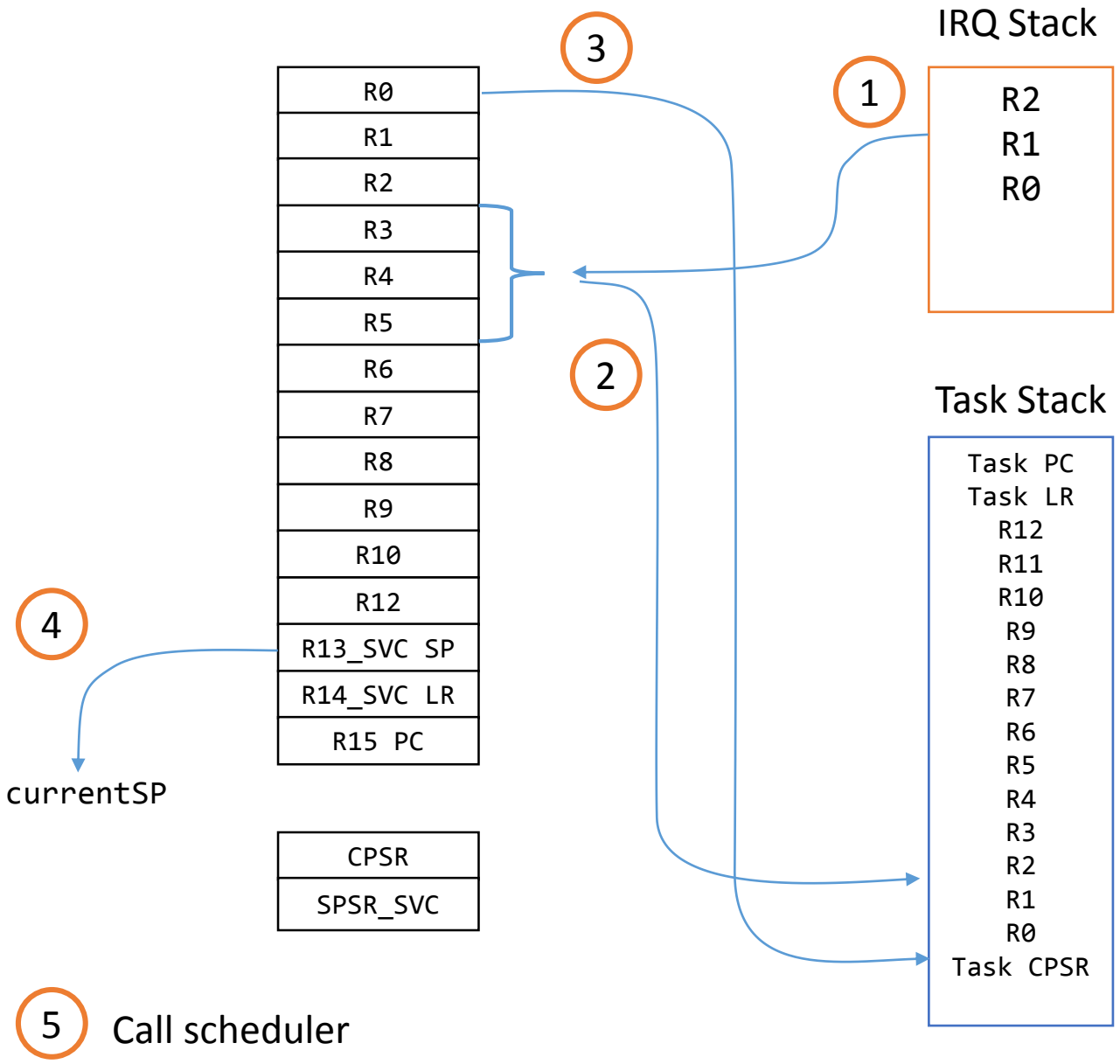
1. Save R0, R1, R2 to the IRQ stack so we can use the registers as variables.
   - The IRQ stack is temporary storage
2. Save SPSR, LR, and SP to R0, R1, R2
   - SP points to where we saved the original values of R0, R1, and R2 on the IRQ stack.
   - We'll need to retrieve the values later so we have to remember where they are saved.
3. Reset IRQ mode's SP
   - We saved SP to R2 so we don't need it anymore
   - We need to make sure the IRQ SP is ready for the next time by moving it back to where it was.
   - Since the stack grow down in memory we need to add 12 bytes to set it back to where it was.
   - 12 bytes is the size of 3 four byte registers.

Second step to saving context



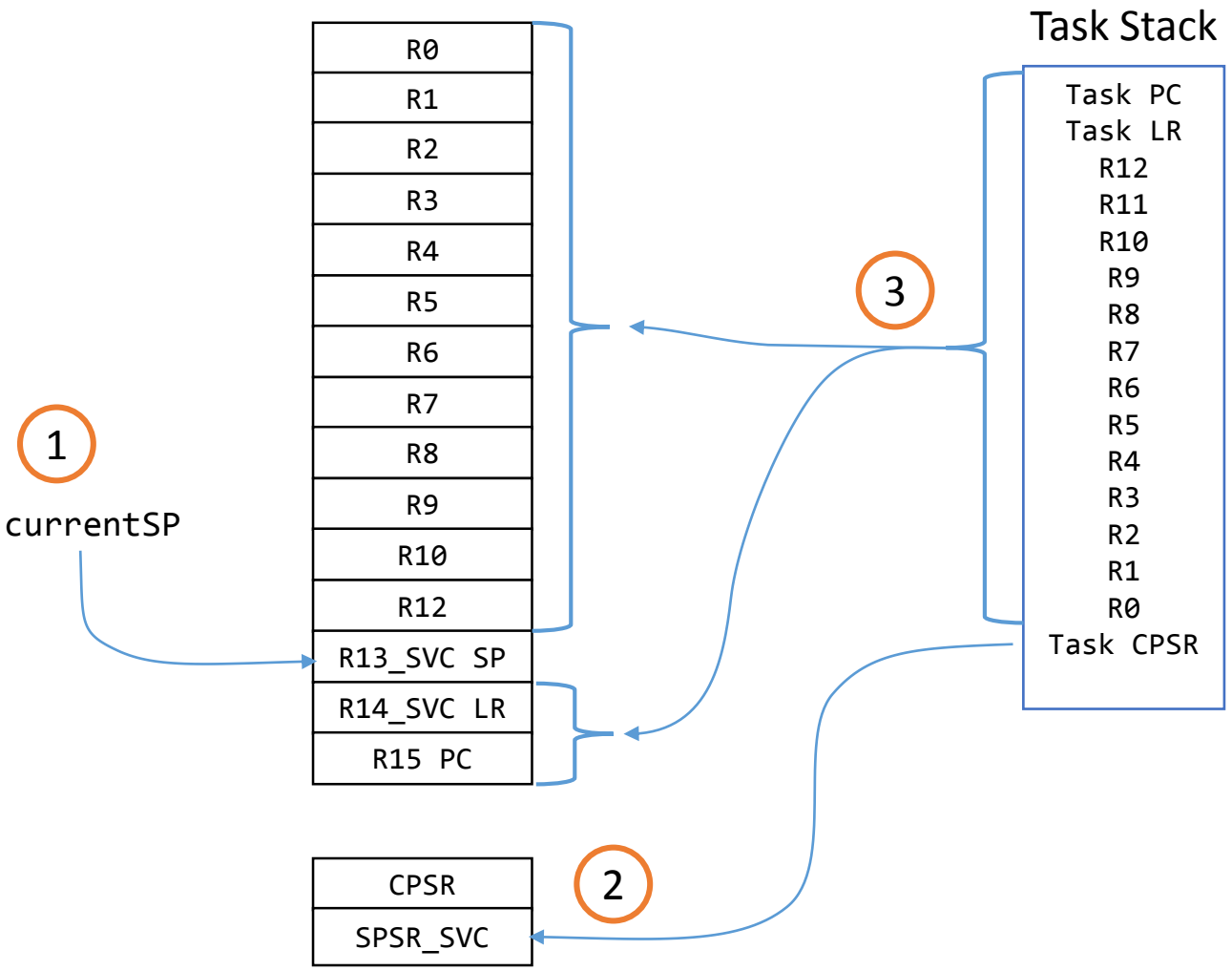
1. Switch to Supervisor Mode
2. Save R1 which has the task's PC value to the stack.
3. Save R14 which has the task's LR to the stack.
4. Save R3-R12 to the stack

Third Step to Saving Task Context



1. Pop the three values saved on the IRQ stack into R3-R5.
   - R2 points to the right position on the IRQ stack.
   - Use R2 as the stack pointer.
2. Save values in R3-R5 to the task's stack.
   - They were the original R0-R2 values.
3. Save value in R0 to the task's stack.
   - It is the task's CPSR at the moment it was interrupted.
4. Save SP to the global variable currentSP.
   - The task's full context is saved to its stack. SP points to the task's context.
   - The context's location is all we need to remember to restore the task later.
5. Call the scheduler.

Restoring a Task



1. Copy currentSP to R13
   - Sets SP to point to the next task's context.
2. Pop the first value off the stack and copy it to the SPRS
   - The first value is the task's CPSR.
3. Pop the context using the load multiple and the ^ flag.
   - Single instruction loads all the registers and copies the SPSR to the CPSR.

- The task switch is complete with everything restored correctly.

Observations

- The code popping the context and restoring the task assumes there is already a context.
  - So how do we start?
- The easiest thing to do is to first prime each stack with an initial context.
  - The initial context should have the same structure as saved context.
- The only values that matter are the PC and CPSR.
  - The PC should be the address of the task function
  - The CPSR should be set to the mode we want task to run in. In this case it is the Supervisor mode.
- Having a initial context means we don’t need to special case running a task for the first time.
- To start task switching we call a simple function that just pops the first context.

Main starts everything

```
int main(void)
{
    initHardware();

    // Setup initial stacks for tasks
    Tasks[0] = initialize_stack(stackOne, (void*)taskOne);
    Tasks[1] = initialize_stack(stackTwo, (void*)taskTwo);

    // Set current to first task
    currentTaskID = 0;
    currentSP = Tasks[currentTaskID];

    initialize_timer_tick();

    // Pop the first context to start task running.
    switch_to_current();

    // Never gets here.
    printString("Main completed.\n");
    while(1);
    return 0;
}
```

- Initializes two stacks.
  - One for each task.
- Sets the currentTaskID and currentSP to the first task.
- Turns on the hardware timer to interrupt every 100 milliseconds.
- Starts multitasking
  - Calls switch_to_current which pops the context pointed to by currentSP.
  - The call never returns since it switches execution over to the tasks.

Priming the Stack for Task Switching

```
int initialize_stack(int* stack, void* task_address)
{
    // Calculate the top of the stack.
    int *stkptr = (int*)(stack + STACKSIZE - 1);

    *(stkptr)   = (int)task_address;   // PC
    *(--stkptr) = (int)0x0e0e0e0e;     // R14 (LR)
    *(--stkptr) = (int)0x0c0c0c0c;     // R12
    *(--stkptr) = (int)0x0b0b0b0b;     // R11
    *(--stkptr) = (int)0x0a0a0a0a;     // R10
    *(--stkptr) = (int)0x09090909;     // R9
    *(--stkptr) = (int)0x08080808;     // R8
    *(--stkptr) = (int)0x07070707;     // R7
    *(--stkptr) = (int)0x06060606;     // R6
    *(--stkptr) = (int)0x05050505;     // R5
    *(--stkptr) = (int)0x04040404;     // R4
    *(--stkptr) = (int)0x03030303;     // R3
    *(--stkptr) = (int)0x02020202;     // R2
    *(--stkptr) = (int)0x01010101;     // R1
    *(--stkptr) = (int)0x00000000;     // R0
    *(--stkptr) = ARM_SVC_MODE_ARM;    // Initial CPSR

    return (int)(stkptr);
}
```

- The stack grows down in memory so we want to start by calculating the top of the stack.
- Then fill in the initial data starting at the top and working down.
- The important values are the task_address and the initial CPSR.
- LR or return address doesn't matter since the task is a infinite loop and should not exit.
- Number values are simply markers that can help in debugging.

Starting the first task

```
/*
 *  Switch to current task without saving previous context.
 *  Used to run the first task the first time.
 *
 *  Pops the initial context setup up by the initialize_stack function.
 *  That means this function will act like a return to the task.
 *
 */
switch_to_current:
                                    /* Switch to the current task's stack. */
    ldr   r0, =currentSP            /* Load r0 with the address for currentSP */
    ldr   sp, [r0]                  /* Load sp with the value stored at the address */
                                    /* sp now points to the task's stack. */

                                    /* Set SPSR to the CPSR for the task. */
    ldmfd sp!, {r0}                 /* Pop the first value from the stack into r0. */
                                    /* That value was the CPSR for the task. */
    msr   spsr_cxsf, r0             /* Load SPSR with that CPSR value in r0. */

                                    /* Run task. */
    ldmfd sp!, {r0-r12, lr, pc}^    /* Pop the rest of the stack setting regs and pc for the task */
                                    /* Acts like a call the task. */
```

Timer Interrupt

- Timer is set to interrupt every 100 milliseconds.
- The timer interrupt causes an IRQ exception.
- The CPU switches the IRQ mode and jumps to the IRQ vector.
- The IRQ vector is set to the address of the context switching function called `SwitchingIRQHandler`.
- The handler will save the current context, call the scheduler, then restore the context selected by the scheduler.

Context Switcher

```
SwitchingIRQHandler:
                                        /* Current mode: IRQ mode with IRQ stack */
    sub     lr, lr, #4                  /* LR offset to return from this exception: -4. */
    stmfd   sp!, {r0-r2}                /* Push working registers r0-r3.  */

    mrs     r0, spsr                    /* Save task's cpsr (stored as the IRQ's spsr) to r0 */
    mov     r1, lr                      /* Save lr which is the task's pc to r1 */

    mov     r2, sp                      /* Save exception's stack pointer to r2, used to retrieve the data off of the IRQ stack. */
    add     sp, sp, #(3 * 4)            /* Reset IRQ's stack back to where it was so it will be ready next time. */

    msr     cpsr_c, #(INT_DISABLED | MODE_SVC)  /* Change to SVC mode with interrupts disabled. */

                                        /* Current mode: SVC mode with SVC stack. This is the stack of interrupted task. */
    stmfd   sp!, {r1}                   /* Push task's PC */
    stmfd   sp!, {lr}                   /* Push task's LR */
    stmfd   sp!, {r3-r12}               /* Push task's R4-R12 */

    ldmfd   r2!, {r3-r5}                /* Move r0-r3 from exception stack to task's stack. First load r3-r5 with r0-r2 */
    stmfd   sp!, {r3-r5}                /* then by pushing them onto the task's stack */

    stmfd   sp!, {r0}                   /* Push task's CPSR, which was the IRQ mode's SPSR to the task stack. */

    mov     r0, sp                      /* Save task's stack pointer for call to switcher */
    bl      scheduler                   /* Call the task scheduler which sets currentSP and resets the timer. */

    ldr     r0, =currentSP              /* Set sp to the new task's sp by reading the value stored in currentSP. */
    ldr     sp, [r0]

    ldmfd   sp!, {r0}                   /* Pop task's CPSR and restore it to spsr. */
    msr     spsr_cxsf, r0               /* spsr will be moved to cpsr when we pop the context with ldmfd */

    ldmfd   sp!, {r0-r12, lr, pc}^      /* Pop task's context. Restores regs and cpsr which reenables interrupts. */
```

Simple Scheduler

```
// Simple round robin scheduler.
//
//   Saves currentSP, increments the task ID
//   Set current SP to the next task.
//
void scheduler()
{
    // Reset timer
    WRITEREG32(T0IR, 0xFF);

    Tasks[currentTaskID++] = currentSP;

    if (currentTaskID > NUMTASKS)
    {
        currentTaskID = 0;
    }

    currentSP = Tasks[currentTaskID];
}
```

- All the scheduler needs to do to track each task is to save each task's stack pointer.
  - The stack pointer points to where the task's context is saved.
  - The scheduler uses an array to store the stack pointers.
  - The task id is the index to where its data is saved.
- To switch task all the scheduler needs to do is set the global currentSP.
  - The interrupt routine will restore the context pointed to by currentSP.
- The scheduler returns back to the interrupt handler.

Summary

- Tasks are functions that don’t return.
- Each task has a stack.
- The stack allocated statically and is global.
- The stack is primed with an initial context containing the task’s address.
- The timer is turned on.
- The timer causes IRQ interrupts
- The IRQ interrupts are handled by a context switching function
- The function saves the current context, calls the scheduler, and then restores the context selected by the scheduler.
- The scheduler saves the pointer to the context to an array and retrieves the next task’s stack pointer.
- The switching function restores the new context.

Next Steps

- Quantums
  - Quantums are the period of time a task gets to run before being switched.
  - Current setup switches task on each interrupt.
  - Better to count interrupts then switch.
  - Allows clock to run at a different rate then switching
  - Timer can run at 1 milliseconds and switches only every 100 interrupts
  - Then the quantum is 100 milliseconds.
- Delays
  - Add a functions that puts the task on a wait list for a set time.
  - Task does not run while on wait.
  - Delay count is decreased every interrupt.
  - When delay count is zero the task is placed back on the run list.
- Events
  - Semaphores are used to synchronize tasks, they are essential in a multi-tasking OS.
  - Semaphores are acquired or released.
  - Tasks that acquire the semaphore can run.
  - Tasks are blocked if they can’t get the semaphore.
  - By acquiring and releasing semaphores tasks can access shared resources without corrupting data.