Multitasking on Cortex-M(0) class MCU

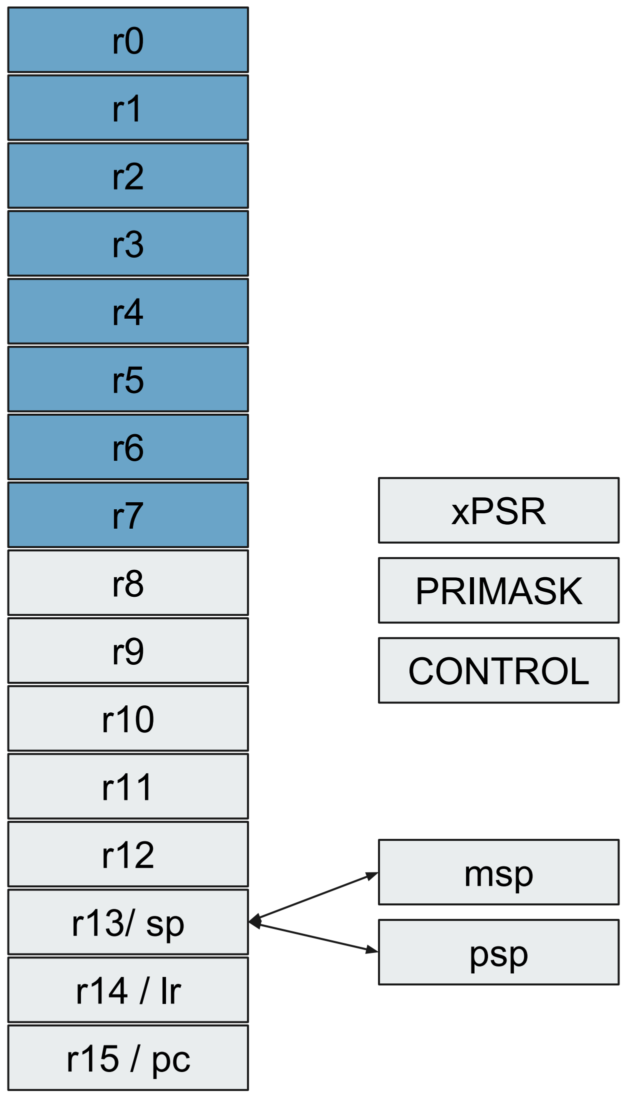
A deepdive into the Chromium-EC scheduler

$whoami

- Embedded Software Engineer at National Instruments
- We just finished our first product using Chromium-EC and future ones to come
- Other stuff I do:
    - fpga-mgr framework (co)maintainer for the linux kernel
    - random drive-by contributions to other projects

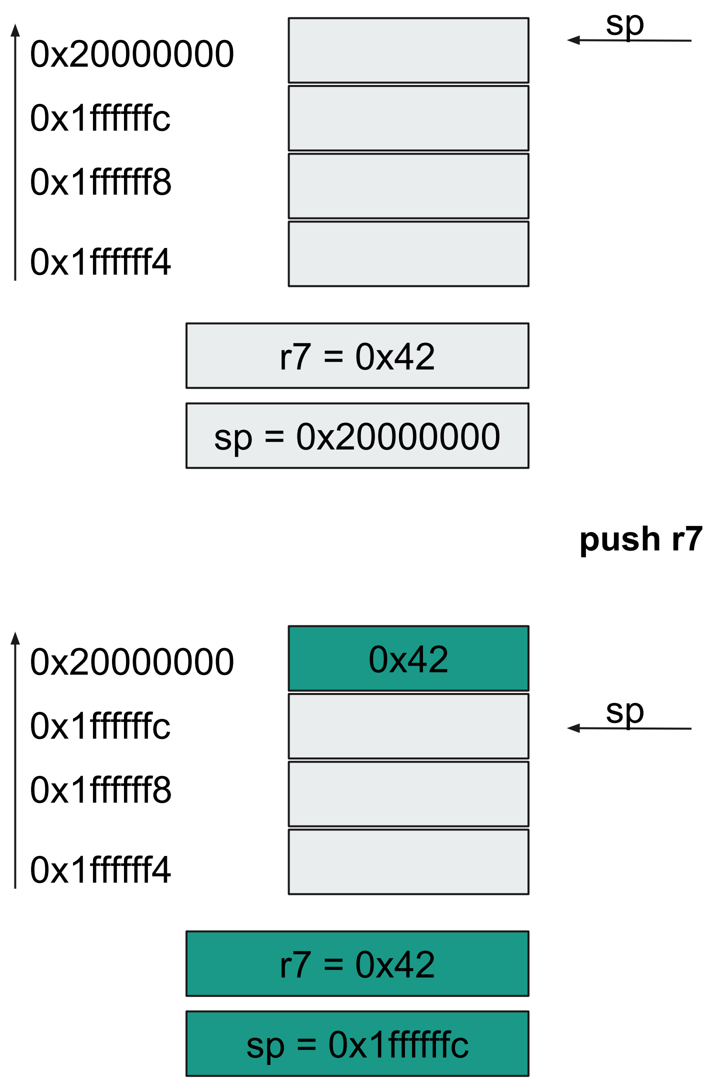
Cortex-M0 registers

- Cortex-M0 has 16 general purpose registers
  - Low registers (**r0-r7**)
  - High registers (**r8-r15**)
  - Undefined status at reset
  - Limited size in thumb, often only low regs
- **r13** is stack pointer for current context stack
  - It is banked, eigher **msp** or **psp**
- xPSR is depending on mode (later)
  - APSR (application program status register)
  - EPSR (exception program status register)
  - IPSR (interrupt program status register)
- PRIMASK (later)
- CONTROL (later)



Stack pointer (sp/r13)

- Used for accessing stack memory via **push** and **pop** instructions
- Can be modified / accessed like any other reg via **ldr, str, subs, adds**, ...
- Either **r13** or **sp** will work
- It is banked (later more)
- Always word aligned, i.e. lowest two bits will always read 0
- Stack is full descending



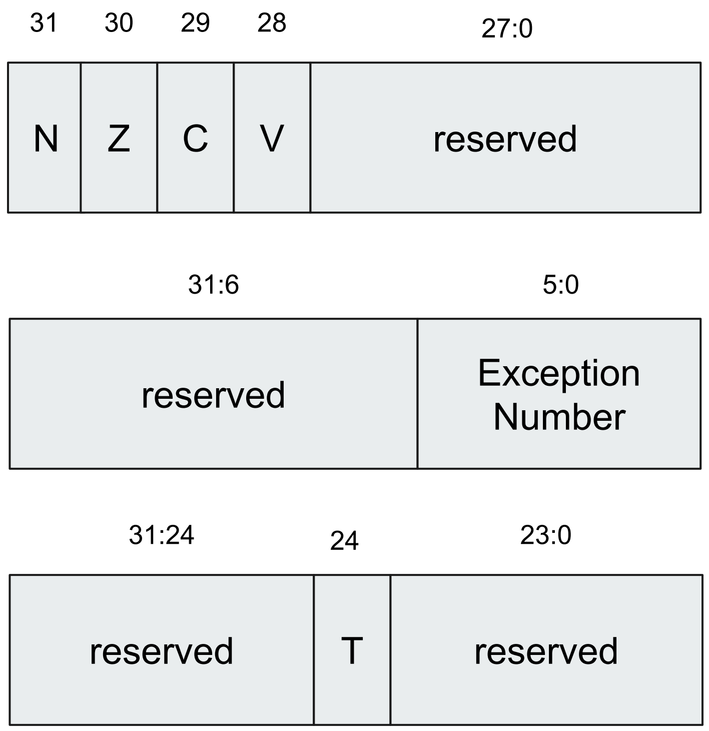
Link Register (r14) & Program Counter (r15)

- Link register **lr** is used with subroutine calls and exceptions (later)
- During subroutine call (using **bl / blx**), sequentially next **pc** value is loaded into lr
- **lr[0]** set indicates a return to Thumb state
- Some instructions need **lr[0]** to be set

- Reading it will give current instruction + 4 (pipeline)
- **pc[0]** should be zero, however **bx/blx** require it to be set, to make sure we stay in Thumb

Combined Program Status Register

- the **xPSR** is a combined register, where **apsr** is the application program status registers
- Bits 31:0 are the ALU flags
  - Not
  - Zero
  - Carry
  - Overflow
- The **ipsr** contains the exception number in the lower bits 5:0
  - if 0, then thread mode (later)
- The **epsr** is the exception program status register
- All of them can be accessed with **msr** / **mrs** instructions

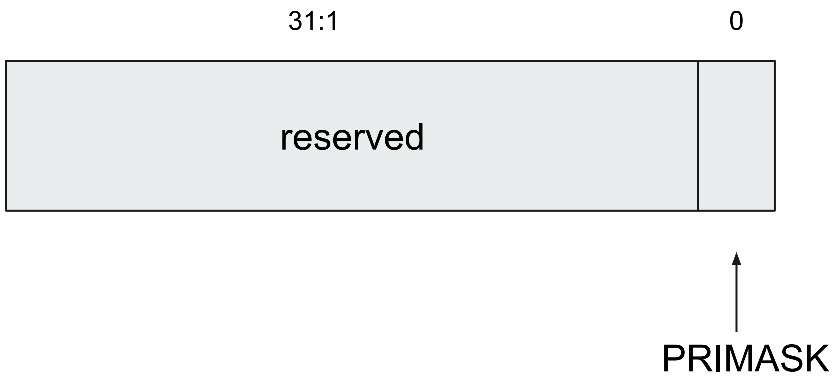


Calling convention

- **r0-r3** are the argument and scratch registers
- **r0-r1** are also the result registers (**r1** if result > word size)
- **r4-r8** are callee-save registers (i.e. callee gotta restore them on return)
- **r9** might be a callee-save register or not (on some variants of AAPCS it is a special register, ignore that)
- **r10-r11** are callee-save registers
- **r12-r15** are special registers (**r12** is intra procedure scratch register)

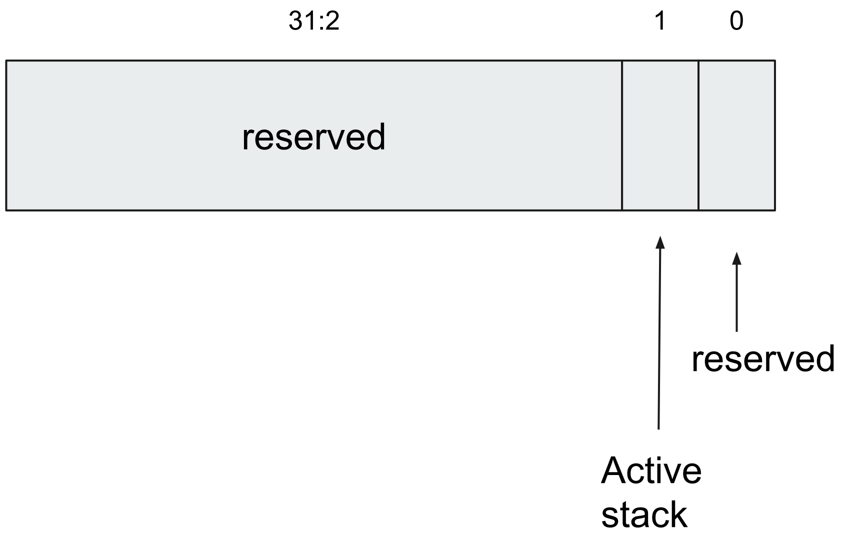
PRIMASK

- PRIMASK
  - If set **no exceptions** with programmable priority entered
  - If not set, no effect



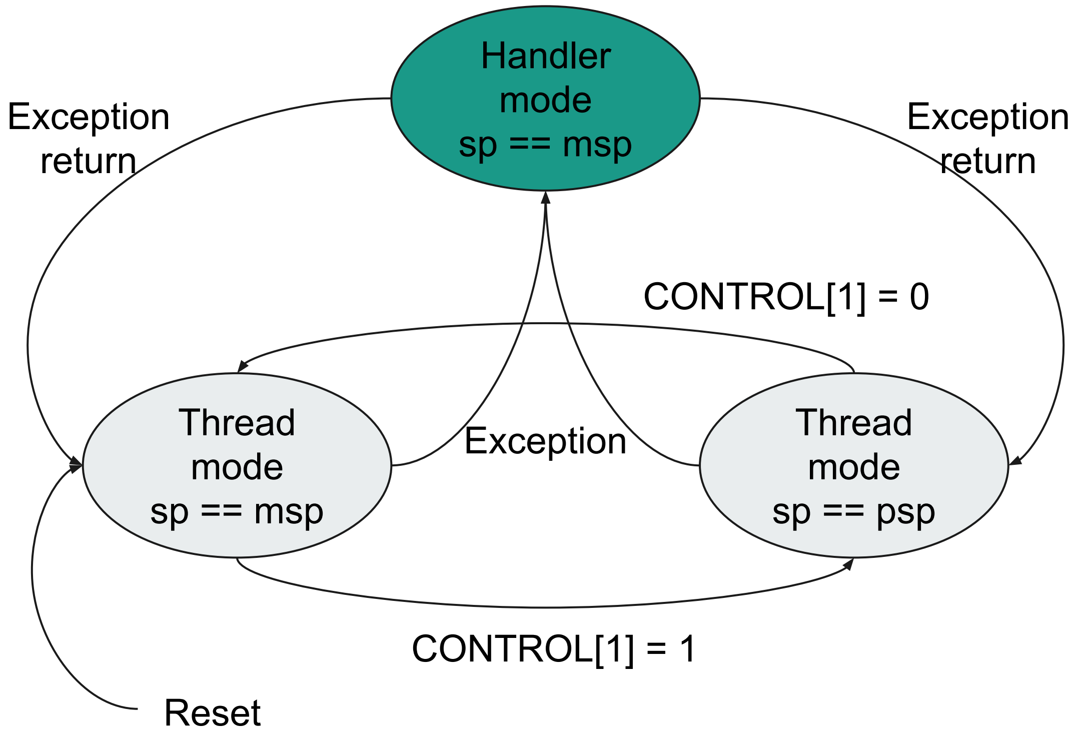
CONTROL

- Only one privilege level in Cortex-M0
- **CONTROL[0]** is reserved (priv. In M3)
- **CONTROL[1]**
    - if set **sp = psp**
    - if not set **sp = msp**
- Using **CONTROL[1]** one can switch between **psp** and **msp**



Thumb State (overview)

- Two modes
  - **Handler** mode
  - **Thread** mode
- Handler mode always uses **msp** as stack
- **Thread** mode usage of stack depends on setting in control register
- After reset start out in **Thread** mode with **msp** active
- Thread to Handler mode transition via Exception



Exceptions

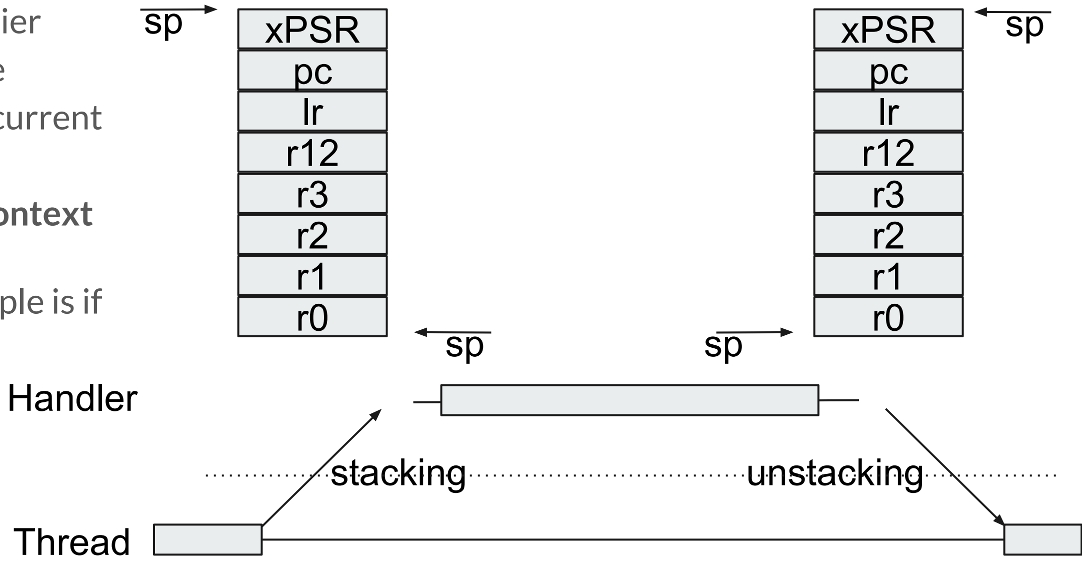
- Event that changes program flow
- Suspends current code, run handler, resume
- (some) exceptions on Cortex-M0 have fixed priorities
  - Reset
  - NMI
  - Hard Fault
- some exceptions have programmable priority
  - SysTick (later)
  - PendSV (later)
  - IRQs
- 0 is the highest priority
- PRIMASK can be used to mask interrupts
- Interrupts can be pending

Exceptions: Vectors

- Cortex-M0 processors support vectored exceptions
- Table contains addresses of handlers, processor fetches address on exception
- Processor jumps to correct handler, instead of having single handler
- Make sure to set a default one

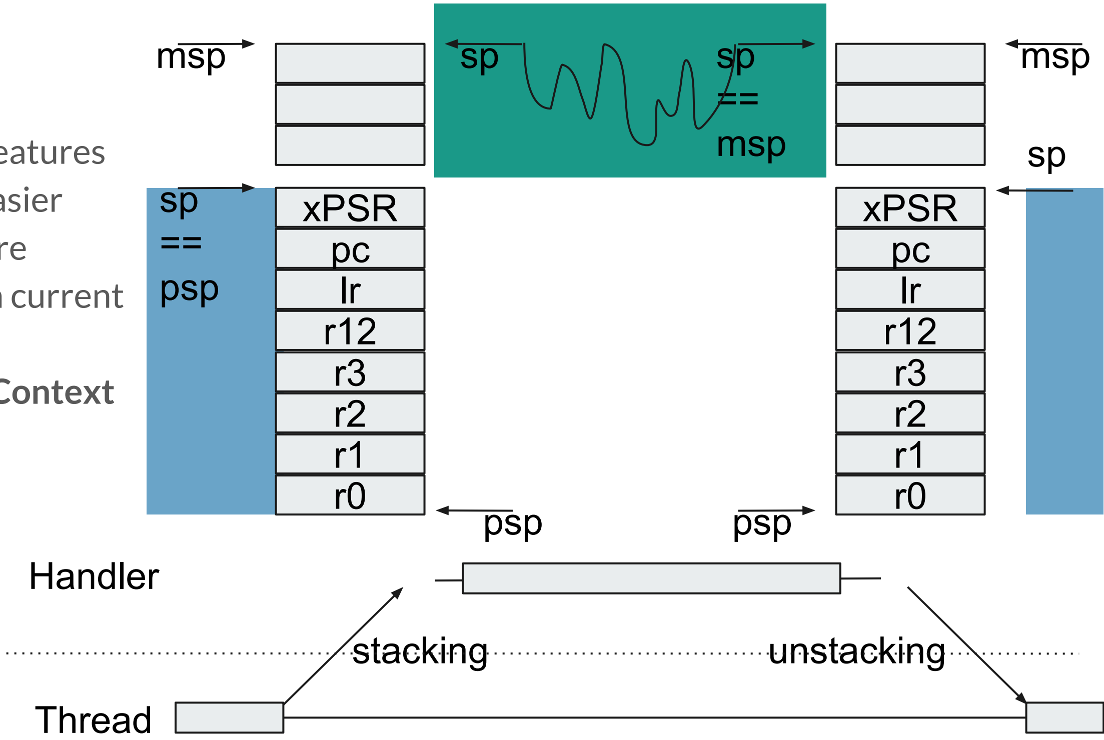
Exceptions (Stacking with MSP)

- Cortex-M(0) comes with hardware features that make dealing with exceptions easier
- On exception entry some registers are pushed onto the stack (depending on current mode)
- These registers form the **Exception Context**
    - **r0-r3, r12, lr, pc, xPSR**
- Stacking happens on current **sp** (example is if psp is not used)
- Makes nesting possible
- **Unstacking** happens based on **lr**



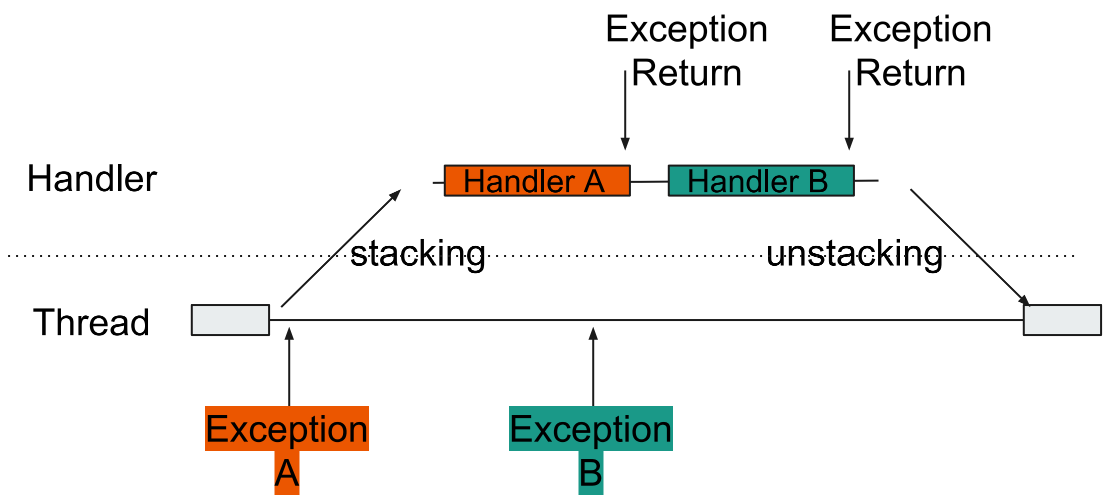
Exceptions (Stacking with PSP)

- Cortex-M(0) comes with hardware features that make dealing with exceptions easier
- On exception entry some registers are pushed onto the stack (depending on current mode)
- These registers form the **Exception Context**
    - **r0-r3, r12, lr, pc, xPSR**
- Stacking happens on current **sp**
- Makes nesting possible
- **Unstacking** happens based on **lr**



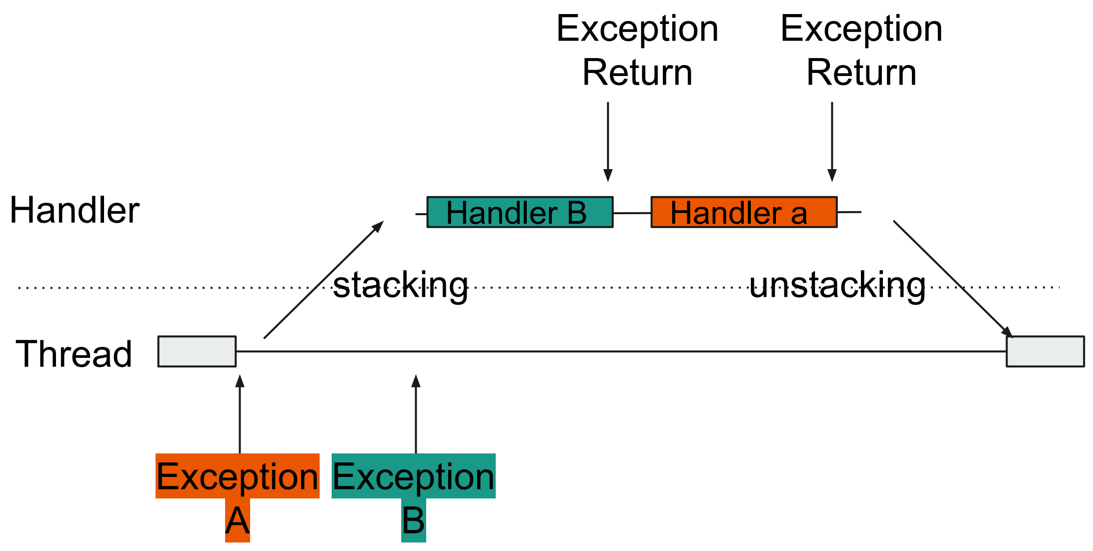
Exceptions (Tail Chaining)

- Speeds up exception servicing
- On completion, if there is a pending exception, unstacking is skipped
- New handler runs



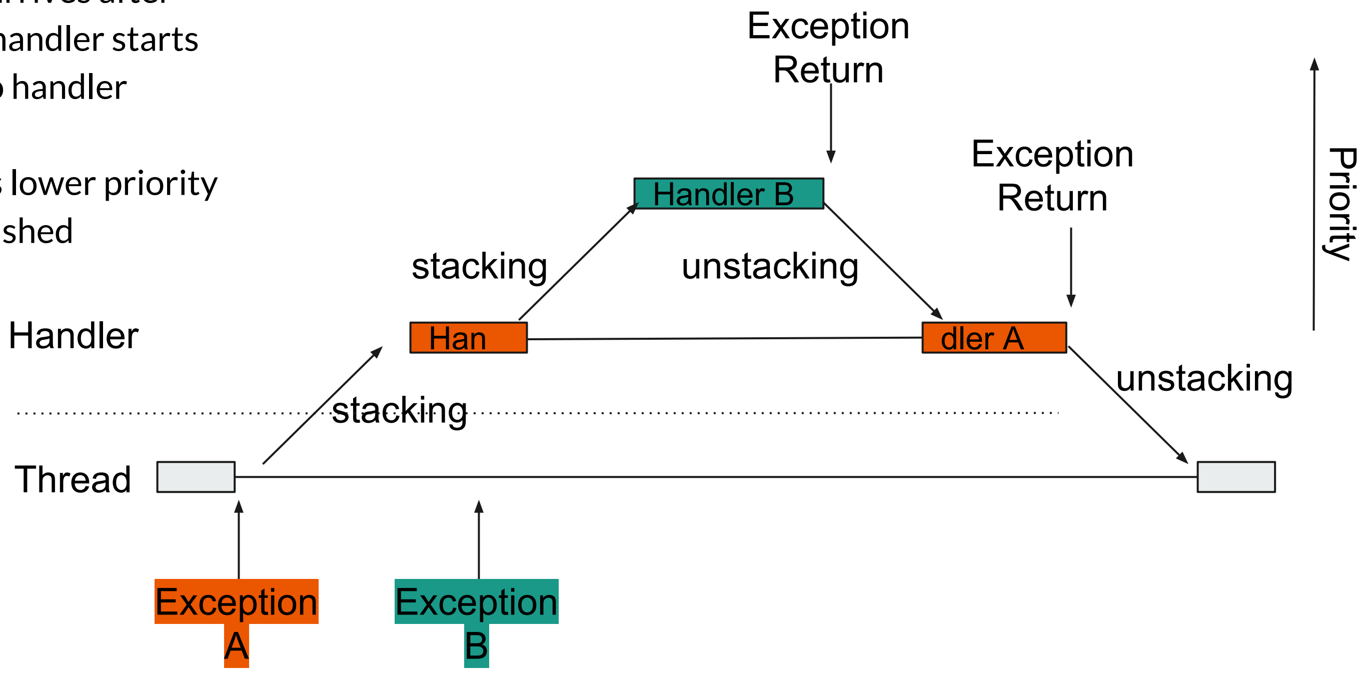
Exceptions (Late arrival)

- If higher priority exception arrives before execution of handler, but after stacking
- Stacking gets reused



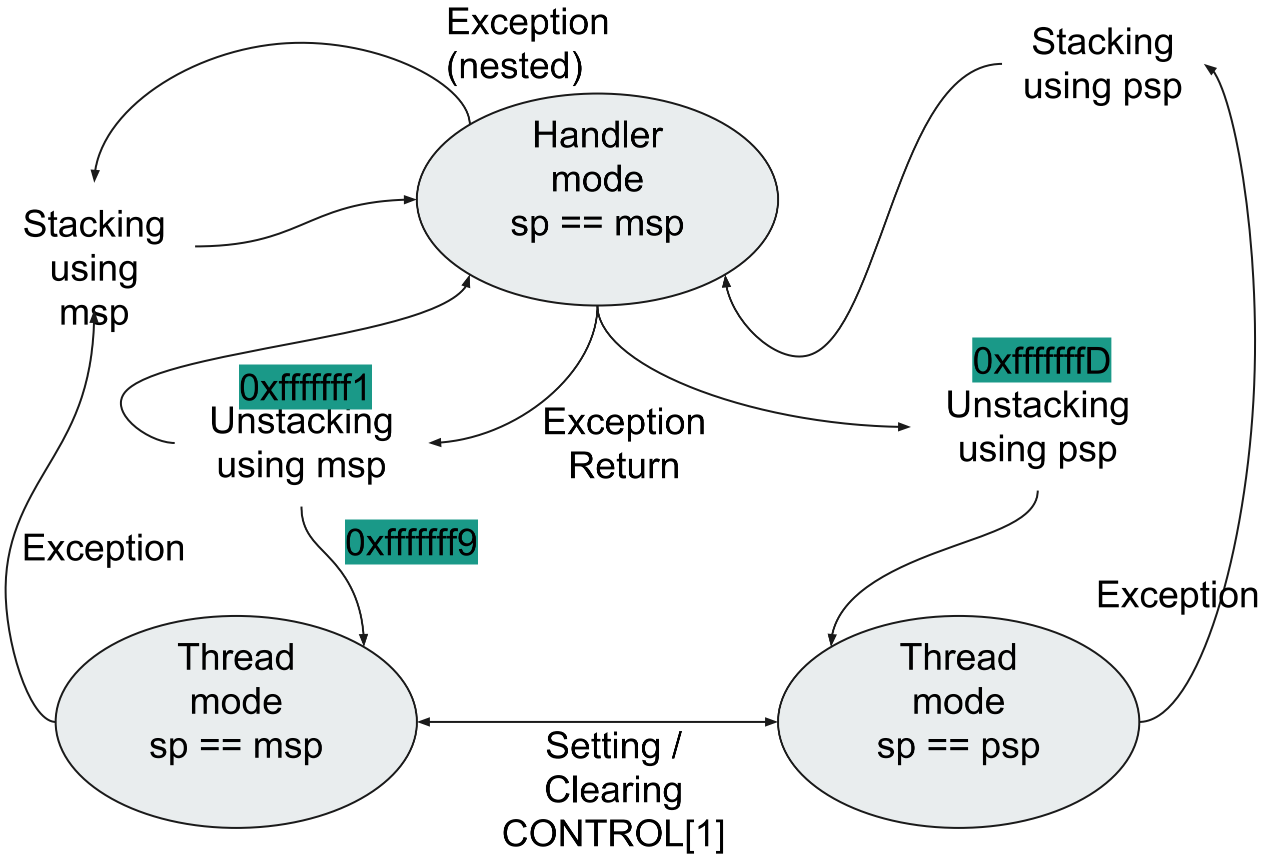
Exceptions (Nested)

- If higher priority exception arrives after Exception handler starts
- Higher Prio handler preempted
- Afterwards lower priority handler finished



Exceptions (EXC_RETURN)

- Depending on value of the **lr** different paths will be taken
- if **lr**
  - 0xfffffff1 unstacking **msp** to handler mode
  - 0xfffffff9 unstacking **msp** to thread mode
  - 0xfffffffd unstacking **psp** to thread mode



Startup code / Getting to main()

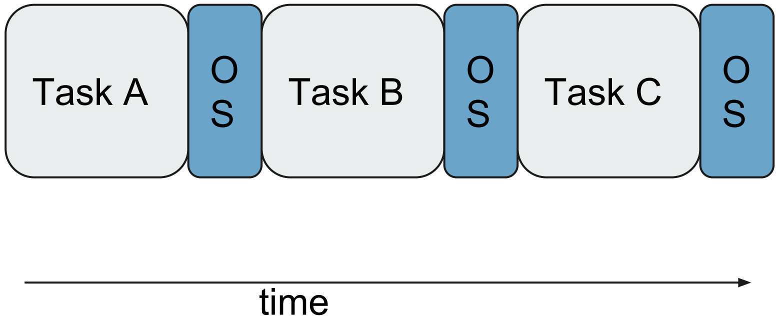
- On reset execution starts at reset vector
- Do a bunch of stuff before we can run ‘normal’ C
- Make sure we’re in the right state (MSP, Thumb, Privileged, …)
- Initialize bss section to zero
- Copy exception vectors to SRAM
- Copy initialized data section to SRAM
    - e.g. global variables
- Set initial stack pointer
- Jump to main()

Getting to main() in Chromium-EC

- After reset make sure CONTROL = 0
- Write bss to zero
- Copy over vectors
- Set vector table to SRAM
- Copy initialized data
- Go!

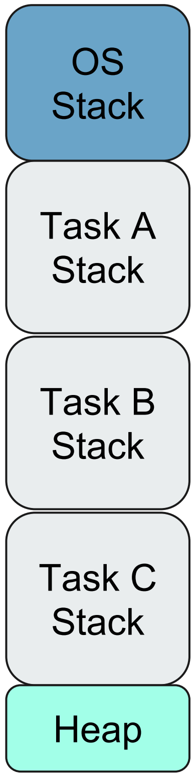
```
reset:
        movs r0, #0
        msr control, r0
        isb
        movs r0, #0
        ldr  r1,_bss_start
        ldr  r2,_bss_end
bss_loop:
        str  r0, [r1]
        adds r1, #4
        cmp  r1, r2
        blt bss_loop
        ldr r1, =vectors
        ldr r2, =sram_vtable
        movs r0, #0
vtable_loop:
        ldr r3, [r1]
        str r3, [r2]
        adds r1, #4
        adds r2, #4
        adds r0, #1
        cmp r0, #48
        blt vtable_loop
movs r0, #3
ldr r1, =0x40010000
str r0, [r1]

ldr r0,_ro_end
ldr r1,_data_start
ldr r2,_data_end

data_loop:
        ldr  r3, [r0]
        adds r0, #4
        str  r3, [r1]
        adds r1, #4
        cmp  r1, r2
        blt data_loop

        ldr r0, =stack_end
        mov sp, r0

        bl main
fini_loop:
        b fini_loop
```

Multitasking - Context Switching



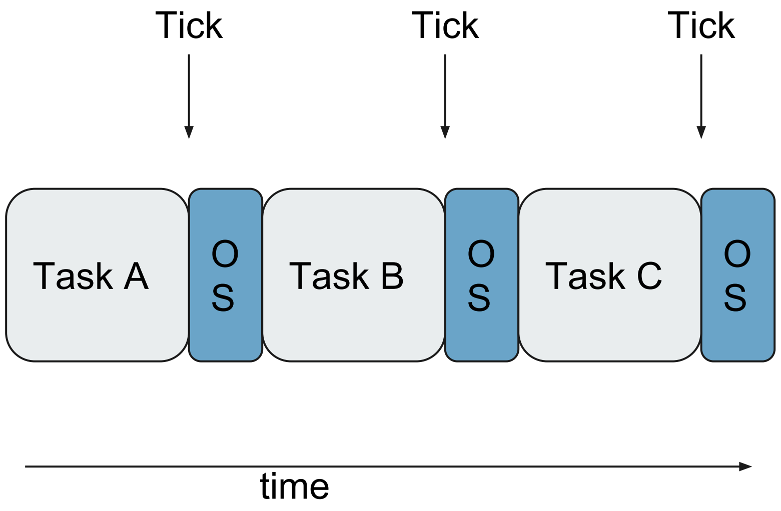
- General idea: Run tasks in a way that each of them can use processor exclusively
- As opposed to cooperative approaches, tasks don't need to be aware of each other
- To make that work, each of them will need state, i.e. context to be restored
- As seen before, Context: registers + stack
- OS decides who goes next

Multitasking - Stack layout



- Need one stack per task
- Need one stack for OS
- OS stack needs to be large enough to deal with all Exceptions
- Task doesn’t need to know it’s own stack
- OS takes care of dealing with stack pointers
- Heap (malloc, free …) is optional

Multitasking - Systick



- Use a timer as periodic event source
- Use these events to run scheduler
- Correct prioritization is required
- So common that ARM provides (optional) one in ARMv6

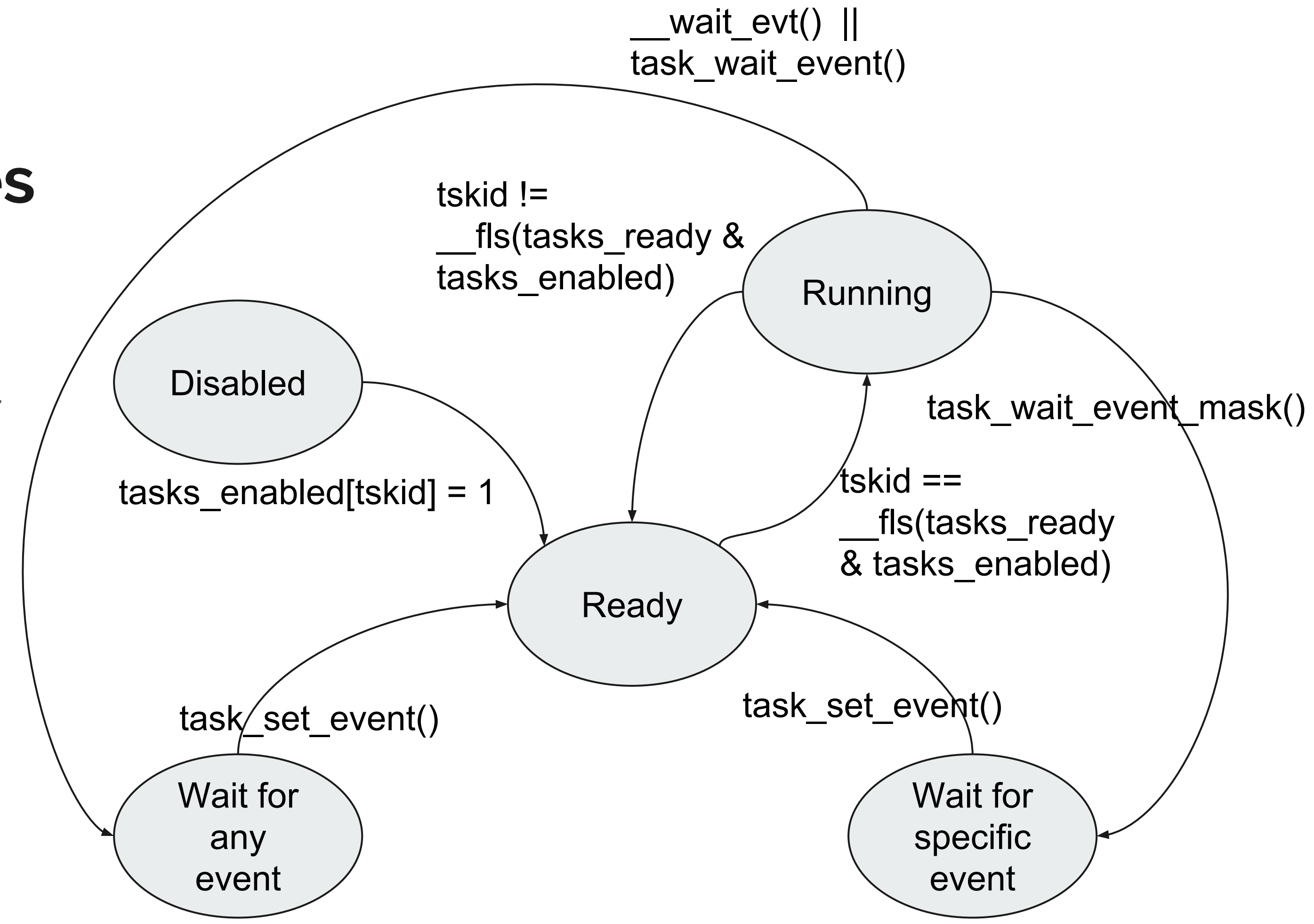
Multitasking in Chromium-EC

```
struct task {
        u32 sp;
        u32 events;
        ...
        u32 *stack;
};
```

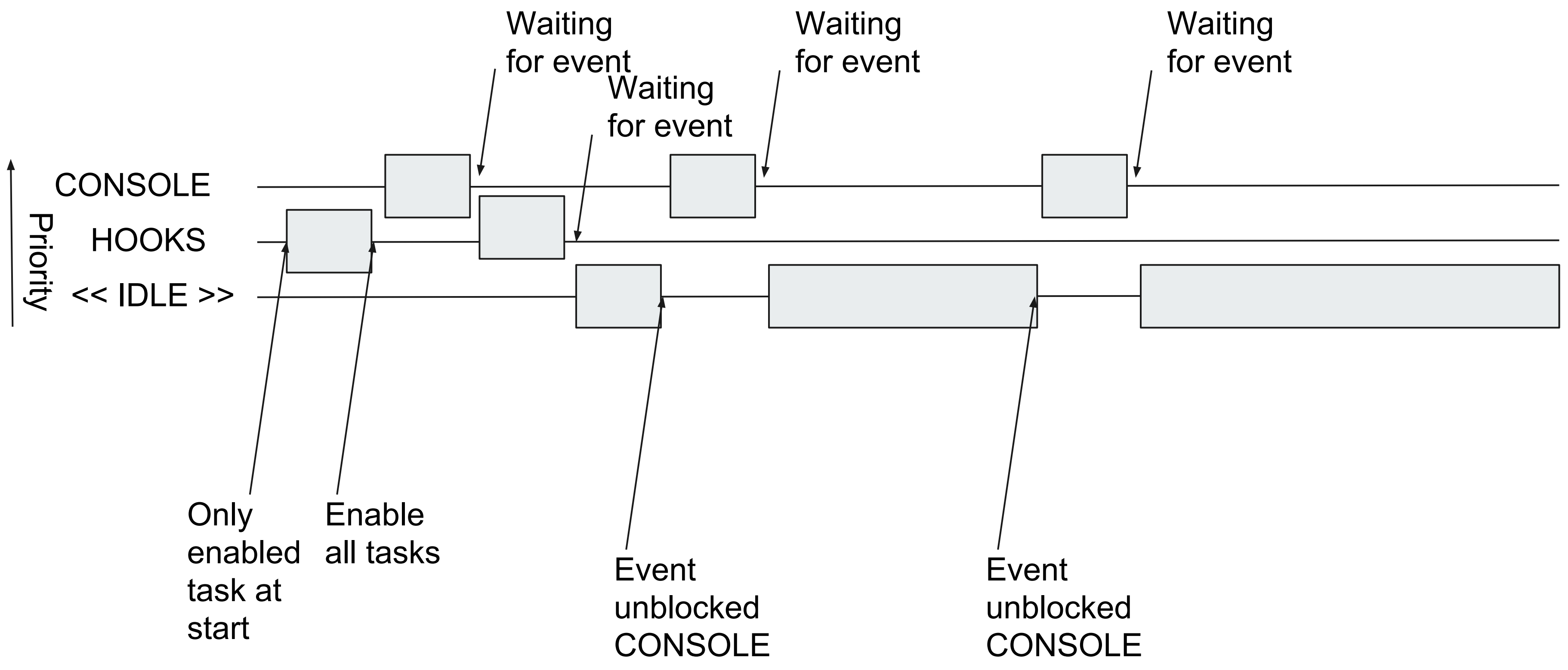
- Task struct as shown
- No heap
- Fixed priorities
- Events
  - Timer
  - Mutex
  - Wake
  - Peripherals
- using 32 bit or 16 bit hardware timer instead of systick

Task states

- **__fls()** gives first set bit in integer
- always run task with highest priority
- tskid is id of current task
- all tasks start out disabled, except HOOKS task
- **all scheduling** happens on event



Scheduling (example)



__schedule(int desched, int resched)

- Wrapper function for SVC call, passing desched in **r0**, and resched in **r1**
- Switches to handler mode (uses MSP)
- **void __schedule(int desched, int resched)**

```
{
        register int p0 asm("r0") = desched;
        register int p0 asm("r1") = resched;
        asm("svc 0");
}
```

svc_handler

- push **r3** and **lr** on stack
- call **__svc_handler** to figure out who goes next

```
svc_handler:
        push {r3, lr}
        bl __svc_handler
        ldr r3, =current_task
        ldr r1, [r3]
        cmp r0, r1
        beq svc_handler_return
        bl __switchto
svc_handler_return:
        pop {r3, pc}
```

__svc_handler (scheduling decision)

- if **desched** and !**current->events** current task is no longer ready
- task given by **resched** is now ready
- pick by priority amongst enabled tasks via **__fls(tasks_ready & tasks_enabled)**
- return **current** in **r0**

```
task_* __svc_handler(int desched, task_id resched)
{
        task_* current;

        current  = current_task;
        if (desched && !current->events)
                tasks_ready &= ~BIT(current - tasks);
        tasks_ready |= BIT(resched);
        next = __fls(tasks_ready & tasks_enabled);
        current_task = next;
        return current;
}
```

… back in svc_handler

- **__svc_handler** returned **current** in **r0**
- compare **current** in **r0 (return value)** with **next (changed by function call)** in **r3**
- if **context switch** required**, call __switchto**
- **pop r3** and **lr** (into **pc**)

```
svc_handler:
        push {r3, lr}
        bl __svc_handler
→       ldr r3, =current_task
        ldr r1, [r3]
        cmp r0, r1
        beq svc_handler_return
        bl __switchto
svc_handler_return:
        pop {r3, pc}
```

__switchto

- get **psp** for **current** task into **r2**
- store currently active **sp** (**msp**)in **r3**
- make **psp** our stack (from **r2**)
- push old context (remember only got thumb)

```
__switchto:
    mrs r2, psp
    mov r3, sp
    mov sp, r2
    push {r4-r7}
    mov r4, r8
    mov r5, r9
    mov r6, r10
    mov r7, r11
    push {r4, r7}
    mov r2, sp
    mov sp, r3
    str r2, [r0]
    ldr r2, [r1]
    ldmia r2!, {r4-r7}
    mov r8, r4
    mov r9, r5
    mov r10, r6
    mov r11, r7
    ldmia r2!, {r4-r7}
    msr psp, r2
    bx lr
```

__switchto

- store active **sp** into **r2**
- make **r3** (old **msp**) our stack again
- store **r2 (old psp)** into old task (first member of struct, pointer in **r0**)
- load new **sp** into **r2** (first member of struct, pointer in **r1**)
- restore **r8-r11,** then **r4-r7** from new stack
- make **r2** the new **psp**

remember:

```
struct task {
        u32 sp;
        [...]
};
```

```
__switchto:
        mrs r2, psp
        mov r3, sp
        mov sp, r2
        push {r4-r7}
        mov r4, r8
        mov r5, r9
        mov r6, r10
        mov r7, r11
        push {r4, r7}
        mov r2, sp
        mov sp, r3
        str r2, [r0]
        ldr r2, [r1]
        ldmia r2!, {r4-r7}
        mov r8, r4
        mov r9, r5
        mov r10, r6
        mov r11, r7
        ldmia r2!, {r4-r7}
        msr psp, r2
        bx lr
```

... back in svc_handler

- **pop r3** and **lr** (into **pc**) to return from exception

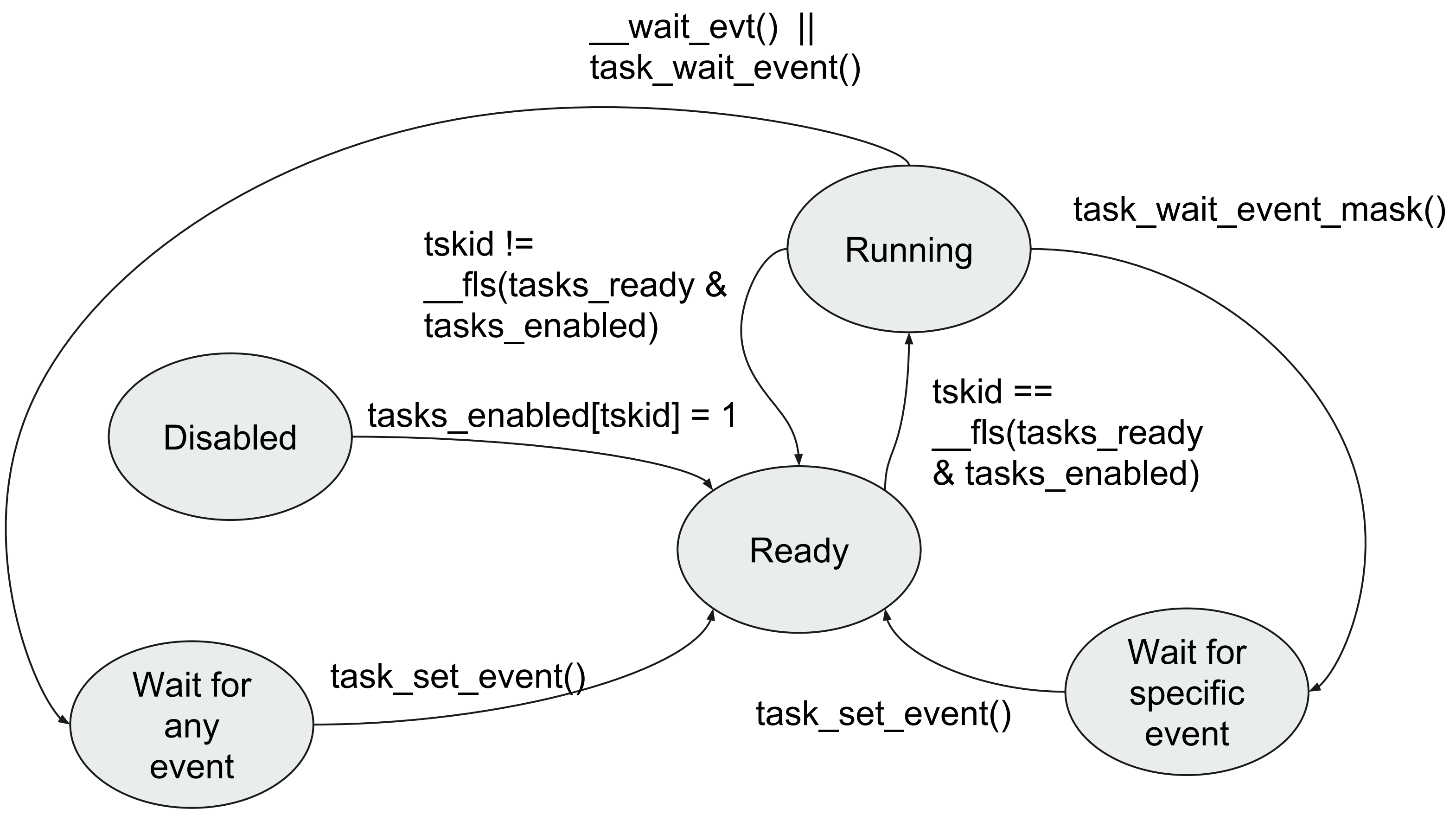
```
svc_handler:
        push {r3, lr}
        bl __svc_handler
        ldr r3, =current_task
        ldr r1, [r3]
        cmp r0, r1
        beq svc_handler_return
        bl __switchto
svc_handler_return:
        pop {r3, pc}
```

how do we get this thing going?

- load scratchpad stack into **r2**
- make room for 17 regs (16 regs (r0-15), psr)
- make that **psp**
- switch thread mode stack pointer to **psp**
- **__task_start** was called with pointer to started_scheduling variable -> set it to 1
- call __schedule (remember r0 = 0 -> don't deschedule, r1 contains task_id to schedule)

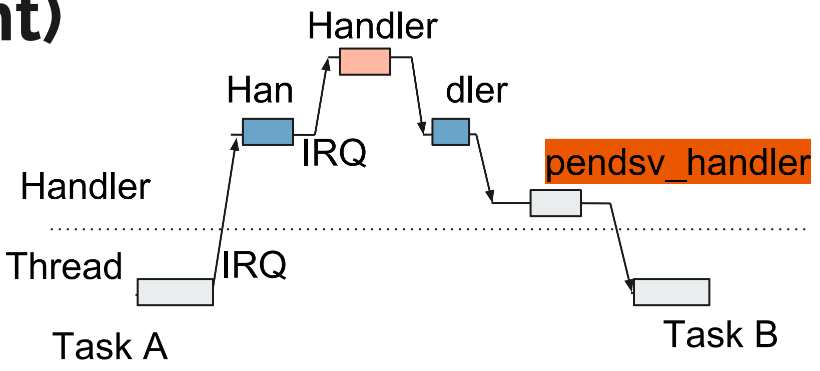
```
__task_start:
        ldr  r2,=scratchpad
        movs r3, #2
        adds r2, #17*4
        movs r1, #0
        msr  psp, r2
        movs r2, #1
        isb
        msr control, r3
        movs r3, r0
        movs r0, #0
        isb
        str  r2, [r3]
        bl __schedule
        movs r0, #1
        bx lr
```

Task states

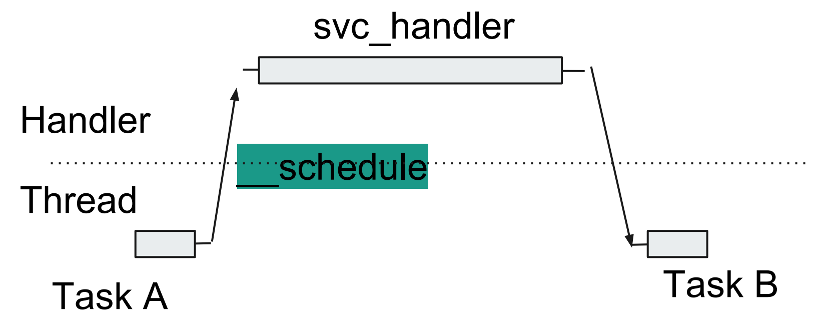

__wait_evt() ||
task_wait_event()
Running
task_wait_event_mask()
tskid !=
__fls(tasks_ready &
tasks_enabled)
Disabled
tasks_enabled[tskid] = 1
tskid ==
__fls(tasks_ready
& tasks_enabled)
Ready
Wait for
any
event
task_set_event()
task_set_event()
Wait for
specific
event

task_set_event(tskid, event)

- Event source is IRQ context
  - (atomically) set event flag in receiver task
  - mark task as ready
  - use pendSV to call __schedule() after IRQs are done
  - Example: Timer (process_timers)



- Event source is task context
  - (atomically) set event flag in receiver task
  - directly call __schedule()
  - Example: Mutex (mutex_unlock)



__wait_evt(timeout_us, resched)

- MUST not be called in IRQ context
- Arm a timer with timeout
- While (atomic) read of task->events == 0, deschedule ourselves, and reschedule ‘resched’
- If timer expires, return timeout
- Wrapped in helper

```
u32 task_wait_event(timeout_us)
{
        return __wait_evt(timeout_us, TASK_IDLE);
}
```

Example: usleep(u32 timeout)

```
u32 evt = 0;
u32 t0 = __hw_clock_source_read();

do {
	evt |= task_wait_evt(timeout);
} while ( !(evt & TASK_EVENT_TIMER) && __hw_clock_source_read() - t0 < timeout));

if (evt)
	atomic_or(task->events, evt & ~TASK_EVENT_TIMER);
```

Atomic operations

- Cortex-M0 does not have **strex** etc so all we can do is disable IRQ, modify, enable IRQ
- This then will look something like this (assuming address passed in r0, and value in r1)

```
#define ATOMIC_OP(asm_op, a, v) do {                    \
          uint32_t reg0;                                \
          __asm__ __volatile__("  cpsid i\n"            \
                              "   ldr  %0, [%1]\n"      \
                              #asm_op" %0, %0, %2\n"    \
                              "   str  %0, [%1]\n"      \
                              "   cpsie i\n"            \
                              : "=&b" (reg0)            \
                              : "b" (a), "r" (v) : "cc");    \
} while (0)
```

static inline void atomic_or(uint32_t volatile *addr, uint32_t bits)
{ ATOMIC_OP(orr, addr, bits); }

Timers / Timer Events

- Using 32 bit or 16 bit hardware timers
- Each task can have a timer, activating it via **timer_arm(timestamp_t tstamp, task_id_t tskid)**
- Set compare value for timer to closest deadline
- On interrupt call **process_timers()** which compares each tasks deadline with timer value, and expires timers as required
- Expired timers set timer event, i.e. task that called **task_wait_event()** or **task_wait_event_mask()** becomes ready
- To cancel a timer **timer_cancel()**

Future work

- Add OS awareness to OpenOCD (kinda works already, just gotta clean up)
- Port to RiscV (just because)
- Port to Microblaze

Questions?

email:
moritz.fischer@ettus.com / mdf@kernel.org

gpg-fingerprint:
135A 2159 8651 9D68 DA5B C3F1 958A 4C04 7304 62CC

github:
mfischer