Interrupt-Driven Input/Output

Textbook: Chapter 11 (Interrupts)

ARM Cortex-M4 User Guide (Interrupts, exceptions, NVIC)

- Sections 2.1.4, 2.3 – Exceptions and interrupts
- Section 4.2 – Nested Vectored Interrupt Controller

STM32F4xx Tech. Ref. Manual:

- Chapter 8: External interrupt/wakeup lines
- Chapter 9: SYSCFG external interrupt config. registers

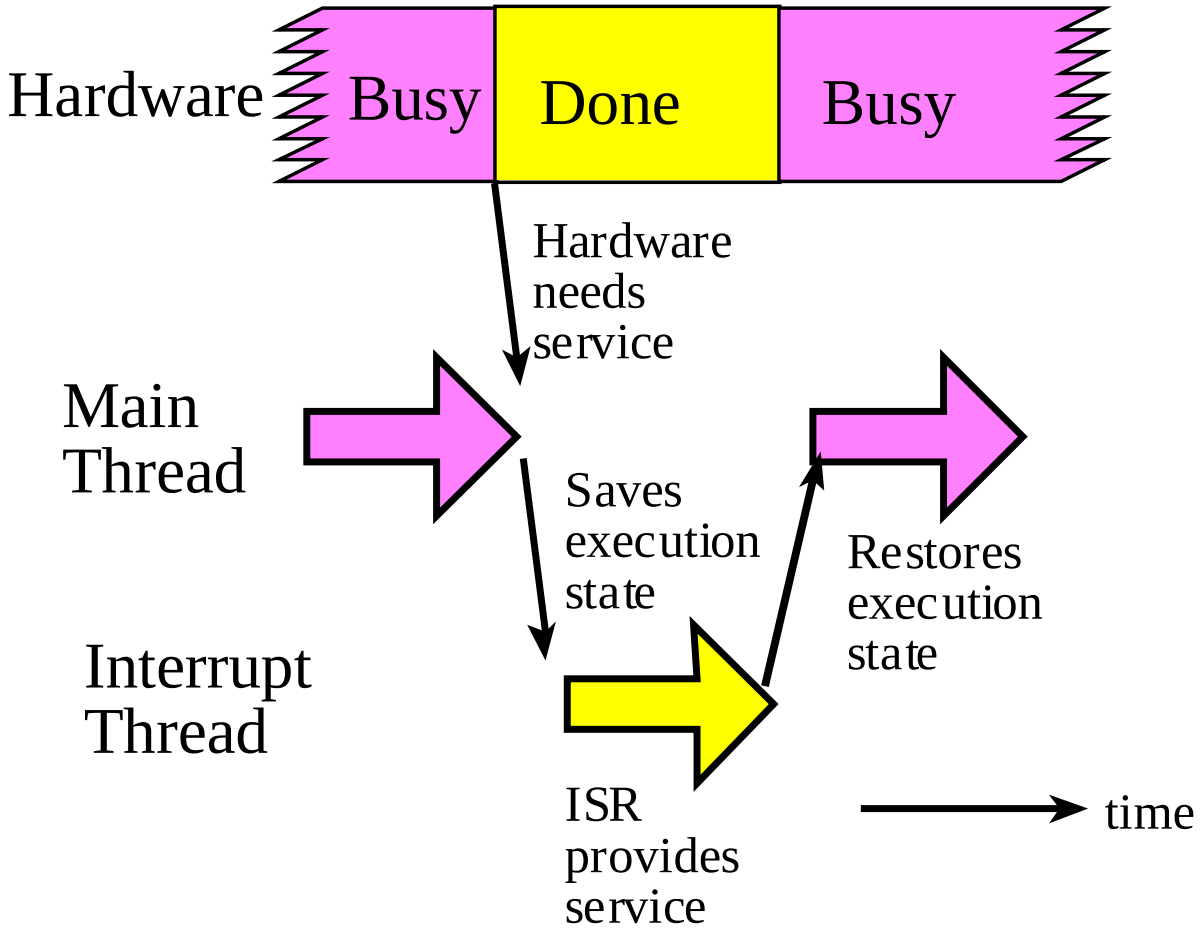
Interrupt-driven operations

- An **interrupt** is an event that initiates the automatic transfer of software execution from one program thread to an **interrupt handler**
- Event types:
  - Signal from a "device" (keyboard, data converter, etc.)
    - Device external to the CPU (possibly within a microcontroller)
    - Signals that a device needs, or is able to provide service
      (i.e. device goes from "busy" to "ready")
    - Asynchronous to the current program thread
    - Allow CPU to do other work until device needs service!
  - An internal event or "exception" caused by an instruction
    Ex. invalid memory address, divide by 0, invalid op code
  - A software interrupt instruction
    Ex. ARM Cortex SVC (supervisor call) instruction

Interrupt I/O

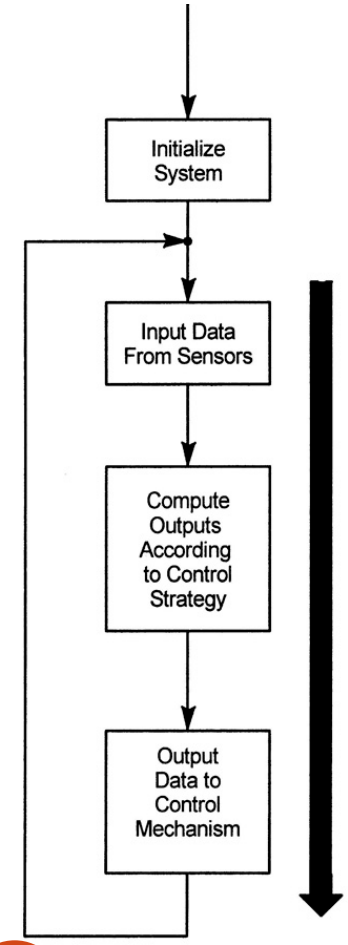
- Busy/wait is very inefficient.
  - CPU can't do other work while testing device.
  - Hard to do simultaneous I/O.
  - But – OK if the CPU has nothing else to do, or if the program cannot otherwise continue
- An interrupt handler is executed **only** when a device requires service

Interrupt Processing

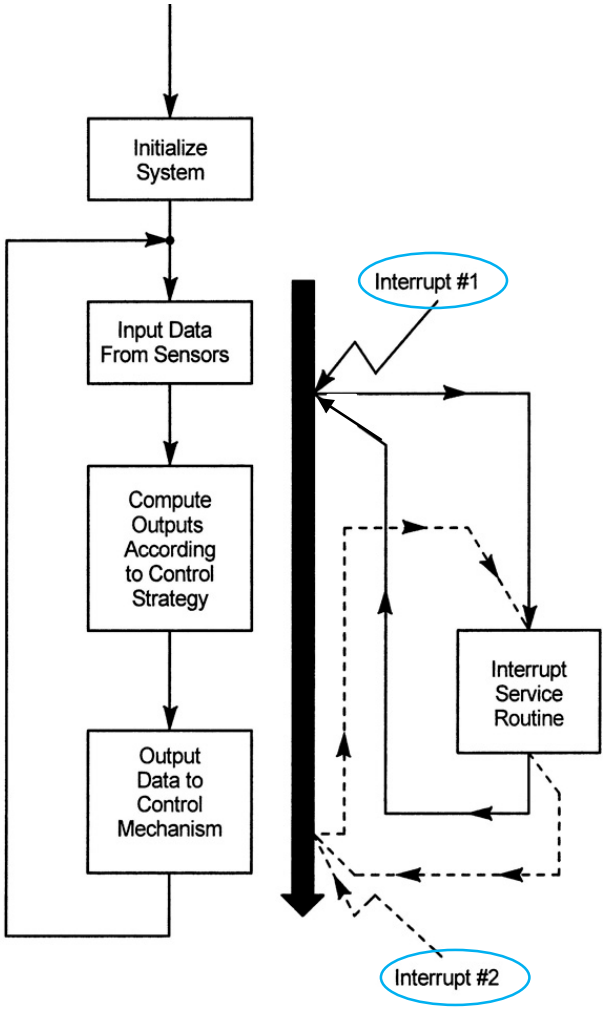


Interrupts in control systems

Continuous loop

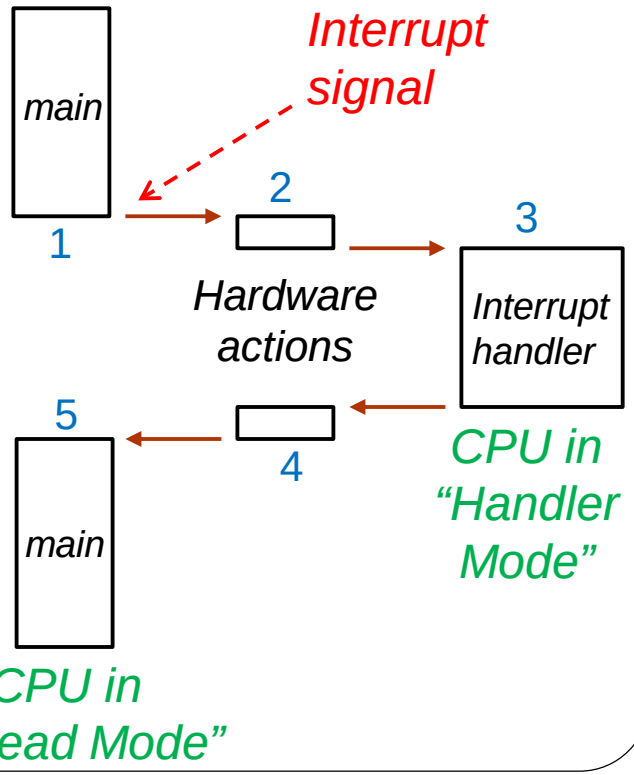


With interrupts



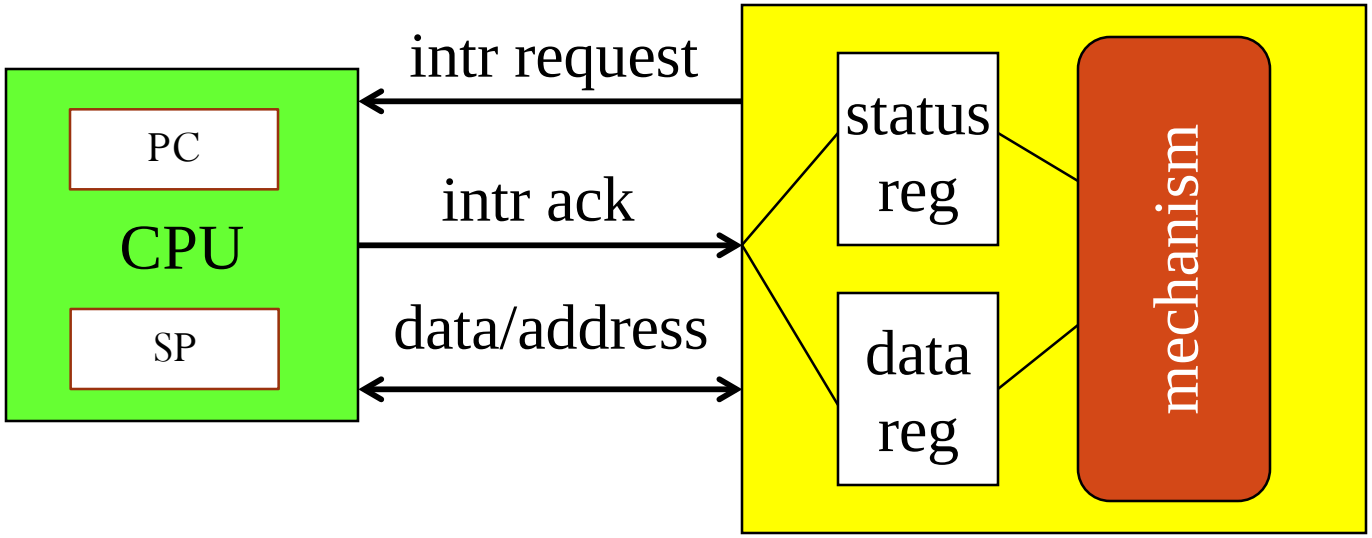
Handling an interrupt request

1. Suspend main thread
2. Save state and jump to handler
3. Execute interrupt handler
4. Restore state and return to main
5. Resume main thread

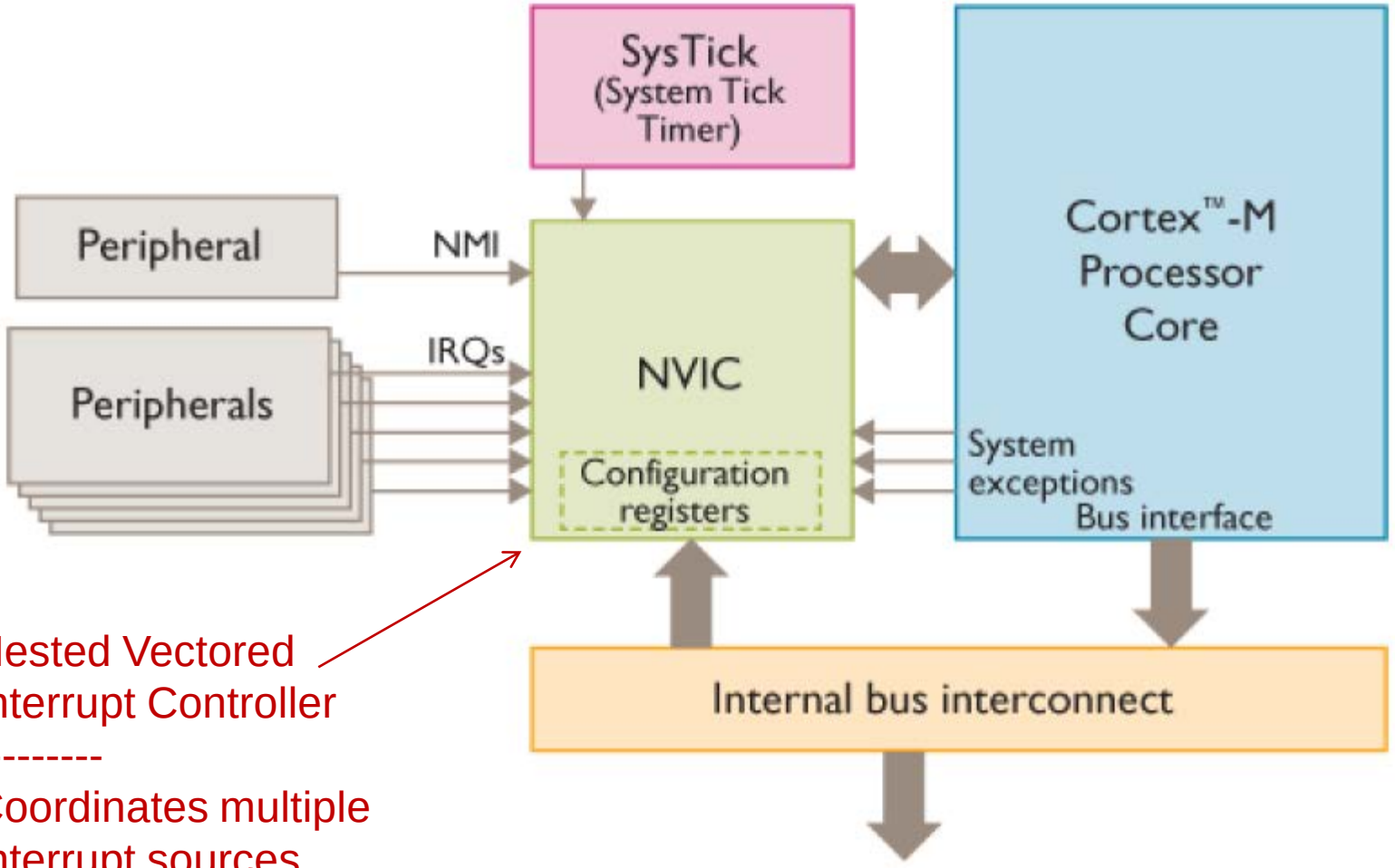


Interrupt interface

- CPU and device are connected by CPU bus.
- CPU and device handshake:
  - device asserts interrupt request;
  - CPU asserts interrupt acknowledge when it responds to the interrupt;
  - device de-asserts interrupt request.



Cortex-M structure

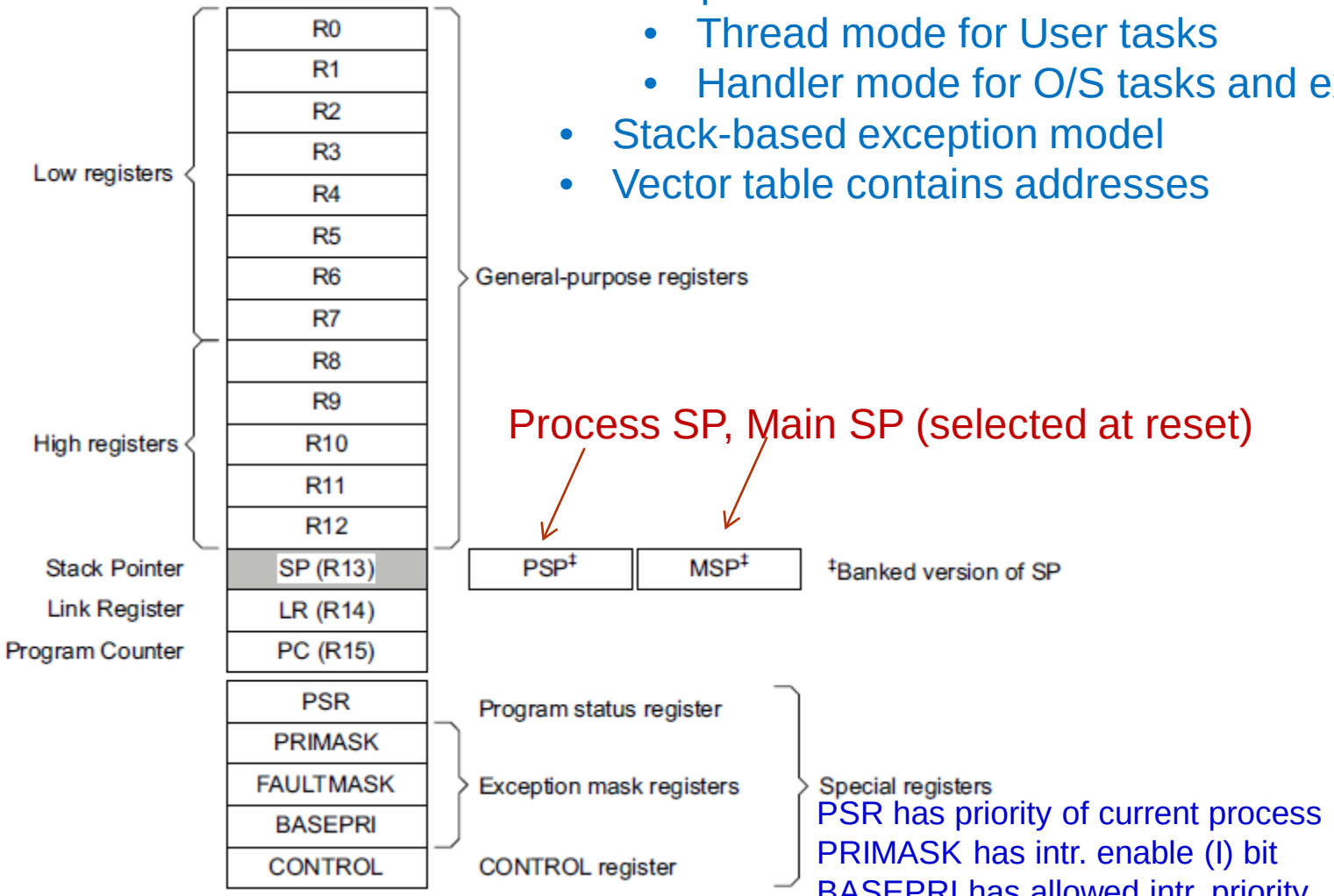


Nested Vectored
Interrupt Controller

Coordinates multiple
Interrupt sources

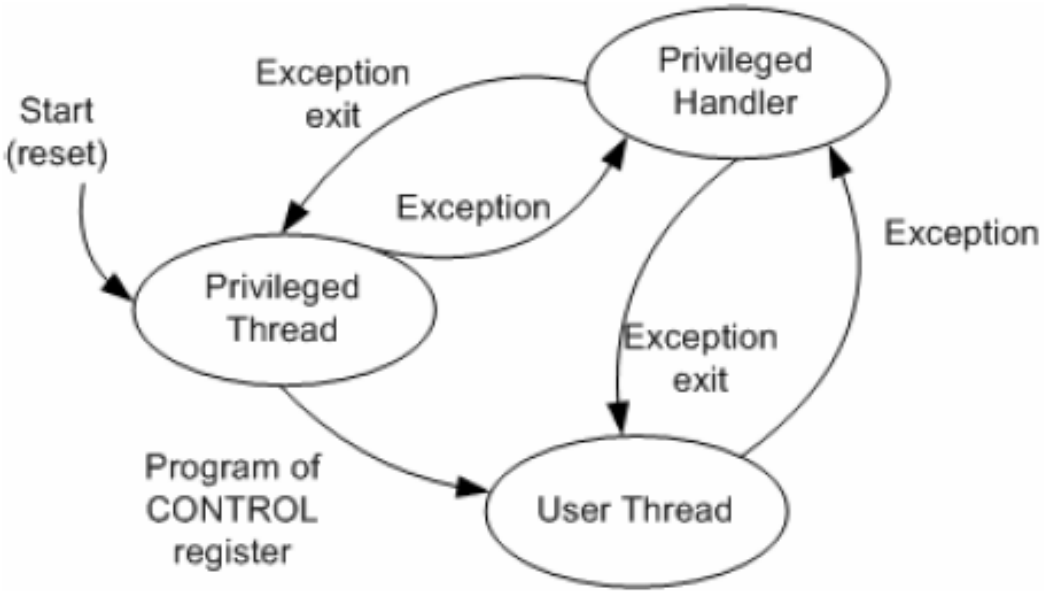
Cortex CPU registers

- Two processor modes:
  - Thread mode for User tasks
  - Handler mode for O/S tasks and exceptions
- Stack-based exception model
- Vector table contains addresses



Cortex-M4 processor operating modes

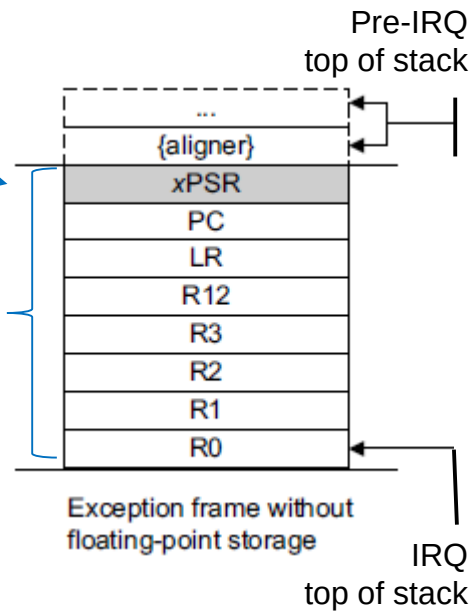
- Thread mode – normal processing
- Handler mode – interrupt/exception processing
- Privilege levels = User and Privileged
  - Supports basic “security” & memory access protection
  - Supervisor/operating system usually privileged



Cortex-M Interrupt Process

(much of this is transparent when using C)

1. Interrupt signal detected by CPU
2. Suspend main program execution
   - finish current instruction
   - save CPU state (push registers onto stack)
   - set LR to 0xFFFFFFF9 (indicates interrupt return)
   - set IPSR to interrupt number
   - load PC with ISR address from vector table
3. Execute interrupt service routine (ISR)
   - save other registers to be used in the ISR[1]
   - clear the "condition" that requested the interrupt
   - perform the requested service
   - communicate with other routines via global variables
   - restore any registers saved by the ISR[1]
4. Return to and resume main program by executing ***BX LR***
   - saved state is restored from the stack, including PC (see next slide)

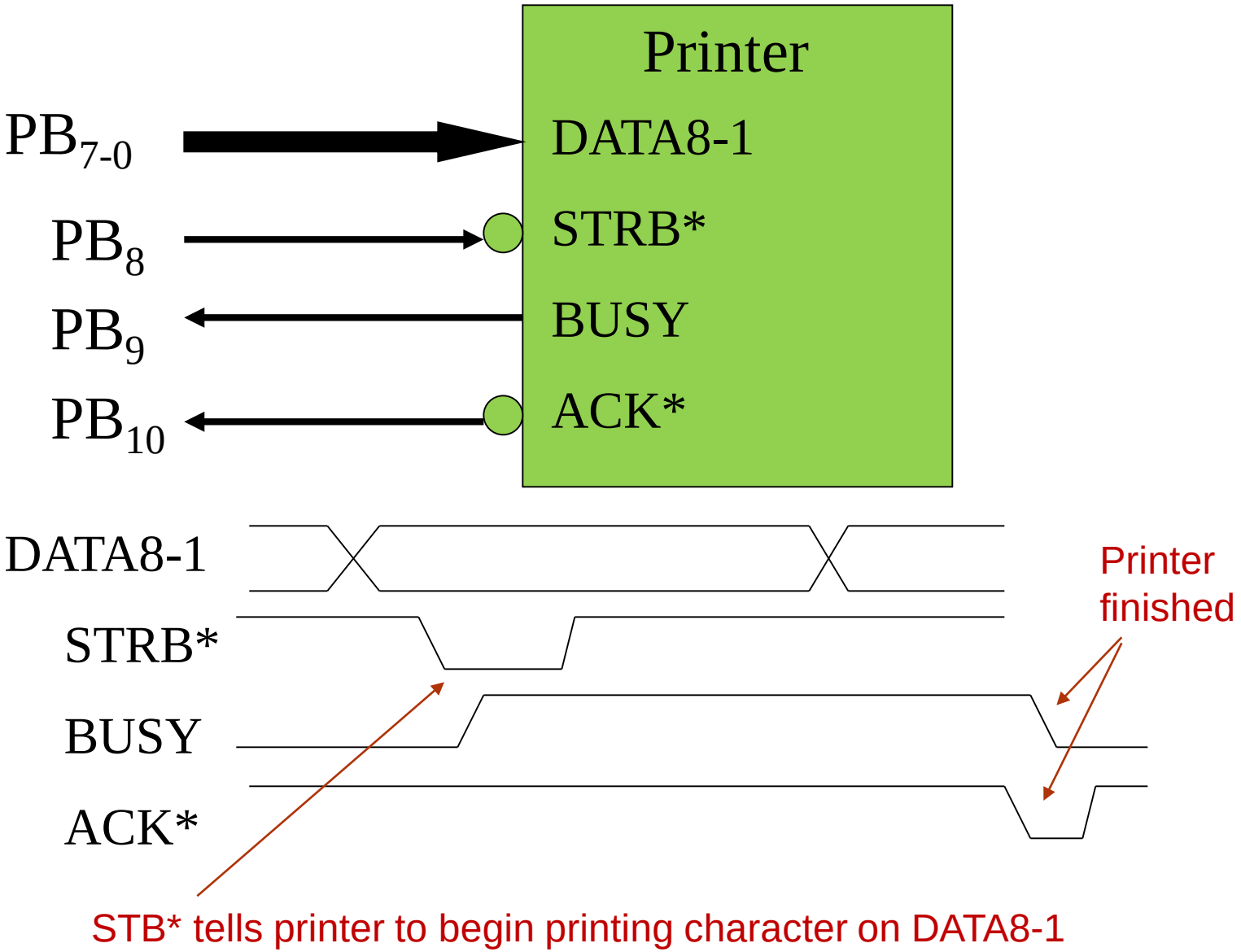


[1] C compiler takes care of saving/restoring registers

Exception return

- The exception mechanism detects when the processor has completed an exception handler.
- EXC_RETURN value (0xFFFFFFF9) loaded into LR on exception entry (after stacking PC and original LR)
  - Lowest 5 bits of EXC_RETURN provide information on the return stack and processor mode.
- Exception return occurs when:
  1. Processor is in Handler mode
  2. EXC_RETURN loaded to PC
  3. Processor executes one of these instructions:
     - LDM or POP that loads the PC
     - LDR with PC as the destination
     - BX using any register

Example: Interrupt-driven printing



Initialize PB pins for printer

```
InitPrinter
;enable clock to GPIOB
        ldr     r0,=RCC                 ;clock control registers
        ldr     r1,[r0,#AHB1ENR]        ;get current values
        orr     r1,#0x02                ;enable GPIOB clock
        str     r1,[r0,#AHB1ENR]        ;update values
;PB7-0=outputs (data), PB8=output (STRB*), PB9-10 inputs
        ldr     r0,=GPIOB
        ldr     r1,[r0,#MODER]          ;get current MODER
        ldr     r2,=0x003fffff          ;clear bits for PB10-0
        bic     r1,r2                   ;clear bits
        ldr     r2,=0x00015555          ;PB10-9 input, PB8-0 output
        orr     r1,r2                   ;set bits
        str     r1,[r0,#MODER]          ;update MODER
;Set initial value of STRB* = 1
        mov     r1,#0x0100              ;select pin PB8 (STRB*)
        strh    r1,[r0,#BSRRL]          ;PB8 = STRB* = 1 initially
        bx      lr                      ;return
```

Program-controlled solution (no interrupt)

```
        ldr     r0,=GPIOB
        ldr     r1,=string      ;string = char array
Loop:   ldrb    r2,[r1],#1      ;get next character
        cmp     r2,#0           ;NULL?
        beq     Return          ;quit on NULL
        strb    r2,[r0,#ODR]    ;character to printer (PB7-PB0)
        mov     r2,#0x0100      ;strobe = PB8
        strh    r2,[r0,#BSRRH]  ;Reset PB8=0  (strobe pulse high-to-low)
        strh    r2,[r0,#BSRRL]  ;Set PB8=1 (strobe pulse low-to-high)
Wait:   ldrh    r2,[r0,#IDR]    ;check PB9 (BUSY)
        tst     r2,#0x0200      ;test BUSY bit
        bne     Wait            ;repeat while BUSY=1
        b       Loop            ;next character
Return: bx      lr
```

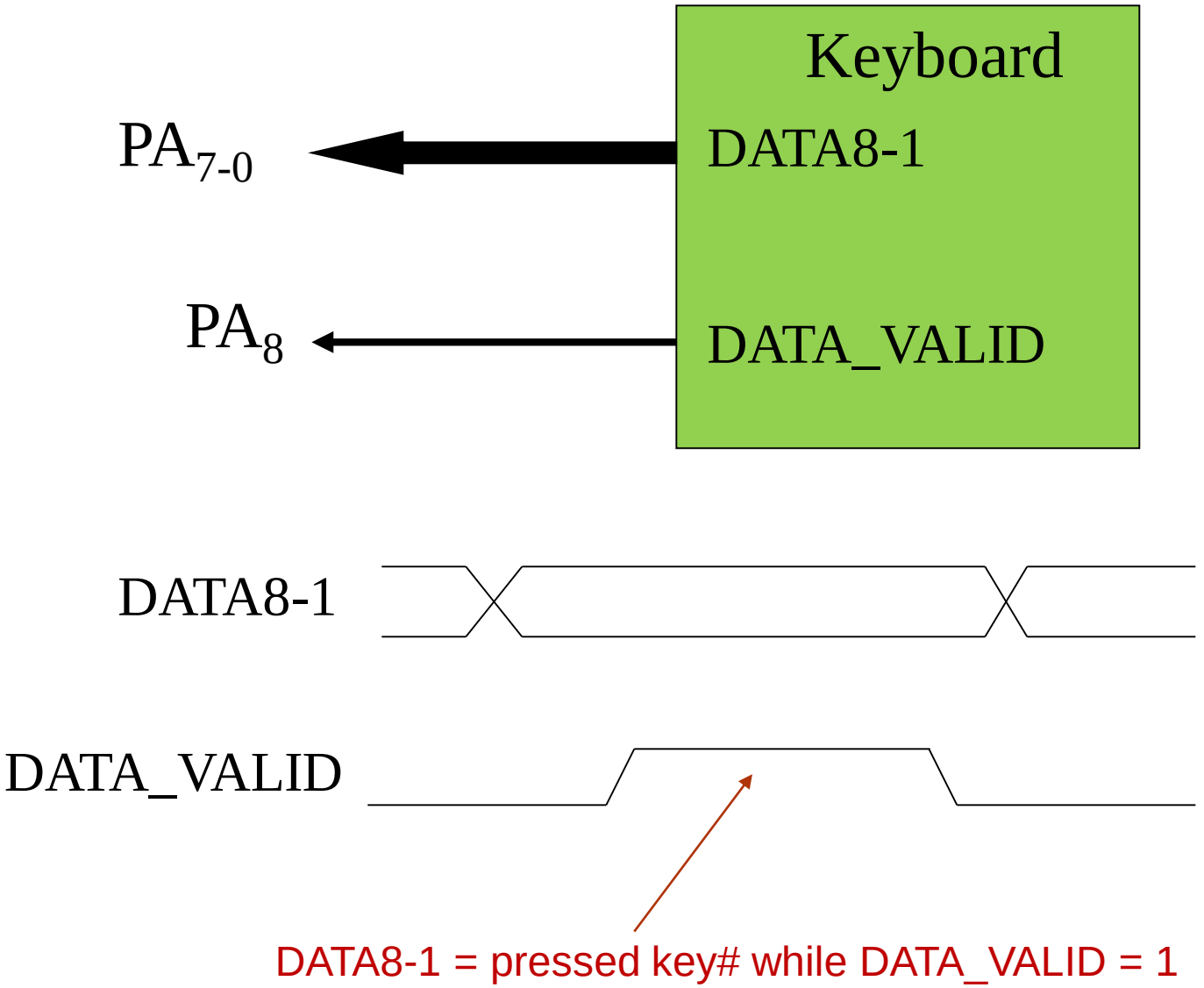
Time "lost" waiting for BUSY = 0.

Interrupt-driven solution

```
;Printer ISR – Send next character when ACK received from printer.
;     Saved_Pointer variable contains address of next character
PrintISR    ldr     r0,=Saved_Pointer    ;pointer variable address
            ldr     r1,[r0]              ;retrieve saved pointer
            ldrb    r2,[r1],#1           ;get next character
            str     r1,[r0]              ;save pointer for next interrupt
            cmp     r2,#0                ;NULL character?
            beq     Return               ;quit on NULL
            ldr     r0,=GPIOB            ;GPIOB register address block
            strb    r2,[r0,#ODR]         ;character to printer (PB7-PB0)
            mov     r2,#0x0100           ;strobe = PB8
            strh    r2,[r0,#BSRRH]       ;Reset PB8=0  strobe pulse high->low
            strh    r2,[r0,#BSRRL]       ;Set PB8=1 strobe pulse low->high
Return      bx      lr                   ;return from ISR
```

No new interrupt request if no new strobe pulse.

Example: Interrupt-driven keyboard



Initialize PA pins for keyboard

```
InitKeyboard
;enable clock to GPIOA
        ldr     r0,=RCC                 ;clock control registers
        ldr     r1,[r0,#AHB1ENR]        ;get current values
        orr     r1,#0x01                ;enable GPIOA clock
        str     r1,[r0,#AHB1ENR]        ;update values
;PA7-0=inputs (data), PA8=input (DATA_VALID)
        ldr     r0,=GPIOA
        ldr     r1,[r0,#MODER]          ;get current MODER
        ldr     r2,=0x0003ffff          ;clear bits for PA8-0
        bic     r1,r2                   ;clear bits for input mode
        str     r1,[r0,#MODER]          ;update MODER
        bx      lr                      ;return
```

Program-controlled solution (no interrupt)

```
;Read key numbers and store in String array until ENTER pressed
        ldr     r0,=GPIOA
        ldr     r1,=String      ;String = char array
Wait:   ldrh    r2,[r0,#IDR]    ;check PA8 = DATA_VALID
        tst     r2,#0x0100      ;test DAVA_VALID bit
        beq     Wait            ;repeat while DATA_VALID = 0
        and     r2,#0x00ff      ;mask DATA_VALID (key# = PA7-PA0)
;Homework problem: return code in r0 instead of the following
        mov     r3,#0           ;NULL character
        strb    r3,[r1]         ;save NULL in String (for now)
        cmp     r2,#0x0D        ;ENTER key?
        beq     Return          ;quit on ENTER
        strb    r2,[r1],#1      ;replace NULL with key#
        b       Wait            ;next character
Return: bx      lr
```

Time "lost" waiting for key press.

Interrupt-driven solution

```
;Key ISR – Get character when DATA_VALID pulsed.
;     Saved_Pointer variable contains address at which to store next character
KeyISR    ldr     r0,=Saved_Pointer  ;pointer variable address
          ldr     r1,[r0]            ;retrieve saved pointer
          ldr     r2,=GPIOA
          ldrb    r3,[r2,#IDR]       ;read key# = PA7-PA0
          mov     r4,#0              ;NULL character code
          strb    r4,[r1]            ;save NULL in String (for now)
          cmp     r3,#0x0D           ;ENTER key (ASCII code for ENTER)
          beq     Return             ;quit on ENTER
          strb    r3,[r1],#1         ;replace NULL with key#
          str     r1,[r0]            ;save incremented pointer
Return    bx      lr                 ;return from ISR
```

Main program setup

```
__main
        ; Configure the I/O ports

        ; Set up printing of a character string
        ldr     r0,=String          ; pointer to character string
        ldr     r1,=Saved_Pointer   ; variable address
        str     r0,[r1]             ; save string pointer for ISR
        cpsie   i                   ; enable interrupts
        bl      PrintISR            ; print the 1st character
                                    ; others printed when CPU interrupted
                                    ; when printer changes BUSY->READY
;******  rest of the program

        AREA    D1,DATA
Saved_Pointer   dcd     0
String          dcb     "This is a string",0
```