CHAPTER 8

Programming in Assembly
Part 4: I/O Programming

Relative Speed

- I/O devices are sometimes mechanical devices (e.g., solenoids, relays, etc.) that take a long time to perform an action.

- The computer performs operations orders of magnitude faster than the I/O devices.

- Synchronization: The CPU must wait for the I/O device to finish each command before issuing the next.

Asynchronous Events

- The events that determine when an input device has data available or when an output device needs data are independent of the CPU.

- I/O programming, therefore, requires "hand-shaking" between the CPU and the I/O device to coordinate the transfer so that data is transferred reliably.

Time Behavior

Data Rate	*Pattern*	*Increasing Time →*
Low	Random	
	Periodic	
High	Random	
	Periodic	

I/O Data Ports

Output Device: Wait for status port to indicate that device is ready to accept data, then copy data to output data port.

- Example: UART Transmit Holding Register

Input Device: Wait for status port to indicate that data is available, then copy data from the input data port.

- Example: UART Receive Buffer Register

I/O Status Ports

(Example: UART Line Status Register)

- Output Device: Status port contains a single bit that indicates whether or not the device is ready to accept new data from the data port.

7	6	5	4	3	2	1	0
Receiver Error	Trans. Empty	THRE	Break	Framing Error	Parity Error	Overrun Error	Data Ready

THRE=1 when device is ready to receive new data; writing new data into the output data port resets this bit to 0.

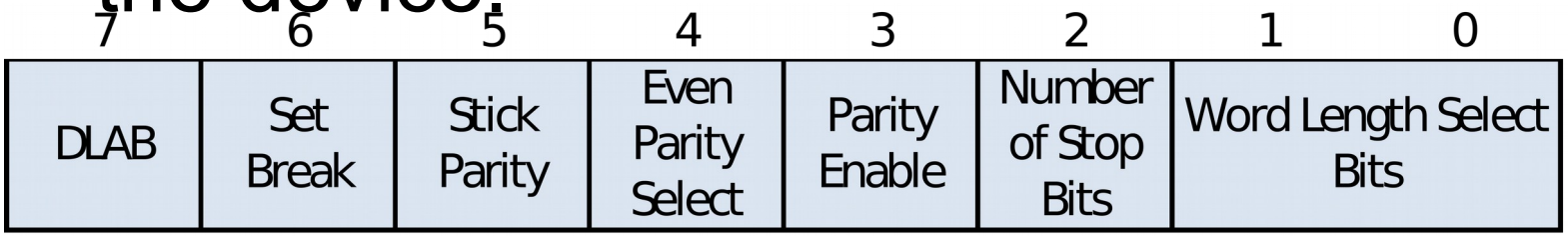
- Input Device: Status port contains a single bit that indicates when new data is available to be read from the data port.

7	6	5	4	3	2	1	0
Receiver Error	Trans. Empty	THRE	Break	Framing Error	Parity Error	Overrun Error	Data Ready

Data Ready will go to 1 when new data is available in the input data port; reading that data from the data port resets this bit to 0.

I/O Control Ports
(Example: UART Line Control Register)

- Used to control operational features of the device.



Accessing I/O Ports

I/O Mapped:
(Desktop Processors)

Processor has an I/O address bus separate from the memory address bus used by special instructions for transferring data to and from an I/O port.

Memory-Mapped:
(Embedded Processors)

I/O devices and memory both share parts of the same memory address space. I/O devices are accessed like regular memory locations.

Three Strategies

- Polled Waiting Loops
- Interrupt-driven I/O
- Direct Memory Access (DMA)

Polled Waiting Loop
(Example: Output data to UART)

```
//  THRE=1 if transmitter holding register empty
#define THRE   (1 << 5)

#define DATAPORT(base)    *(base + 0)
#define STATUSPORT(base) *(base + 24)

void  Output(uint8_t data, uint8_t *uart_base)
    {
    while (STATUSPORT(uart_base) & THRE) == 0)
        {
        // do nothing (wait for device to become
ready)
        }
    DATAPORT(uart_base) = data ;
    }
```

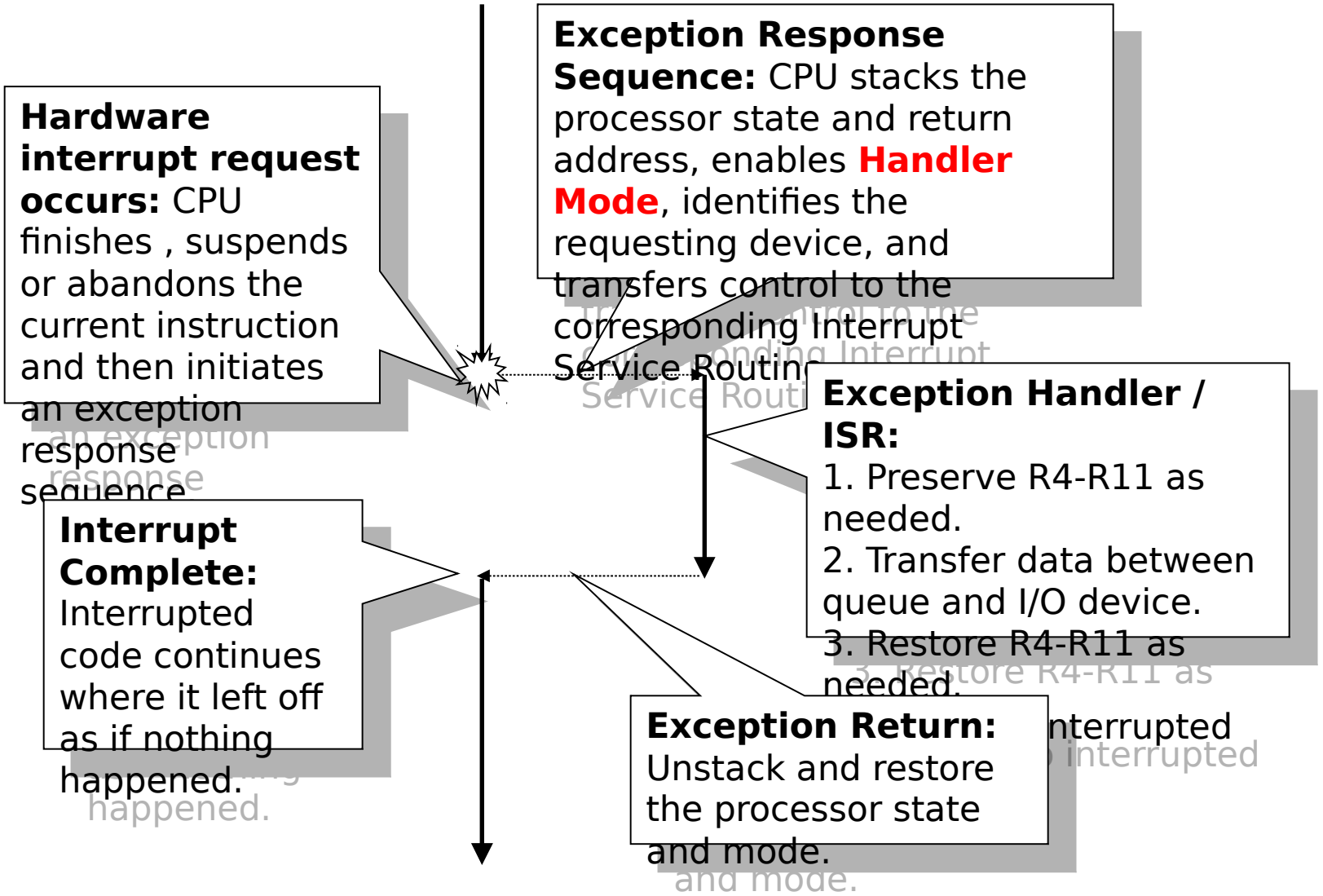
Polled Waiting Loop
(Example: Input data from UART)

```
//  DataReady = 1 if data available to be read
#define DATAREADY  (1 << 0)

#define DATAPORT(base)     *(base + 0
#define STATUSPORT(base) *(base + 24

uint8_t Input(uint8_t *uart_base)
    {
    while (STATUSPORT(uart_base) & DATAREADY) == 0)
        {
        // do nothing (wait for device to become
ready)
        }
    return DATAPORT(uart_base) ;
    }
```

What if urgent data arrives on another input port while executing this

EXCEPTIONS AND INTERRUPTS

Interrupt Processing



Exceptions

Definitions:

- Exception:
  Any condition that needs to halt the normal sequential flow of instruction execution.
- Reset
- SVC Supervisor Call
  (Software Interrupt)
- Fault
  E.g., undefined opcode
- Interrupts

Exceptions

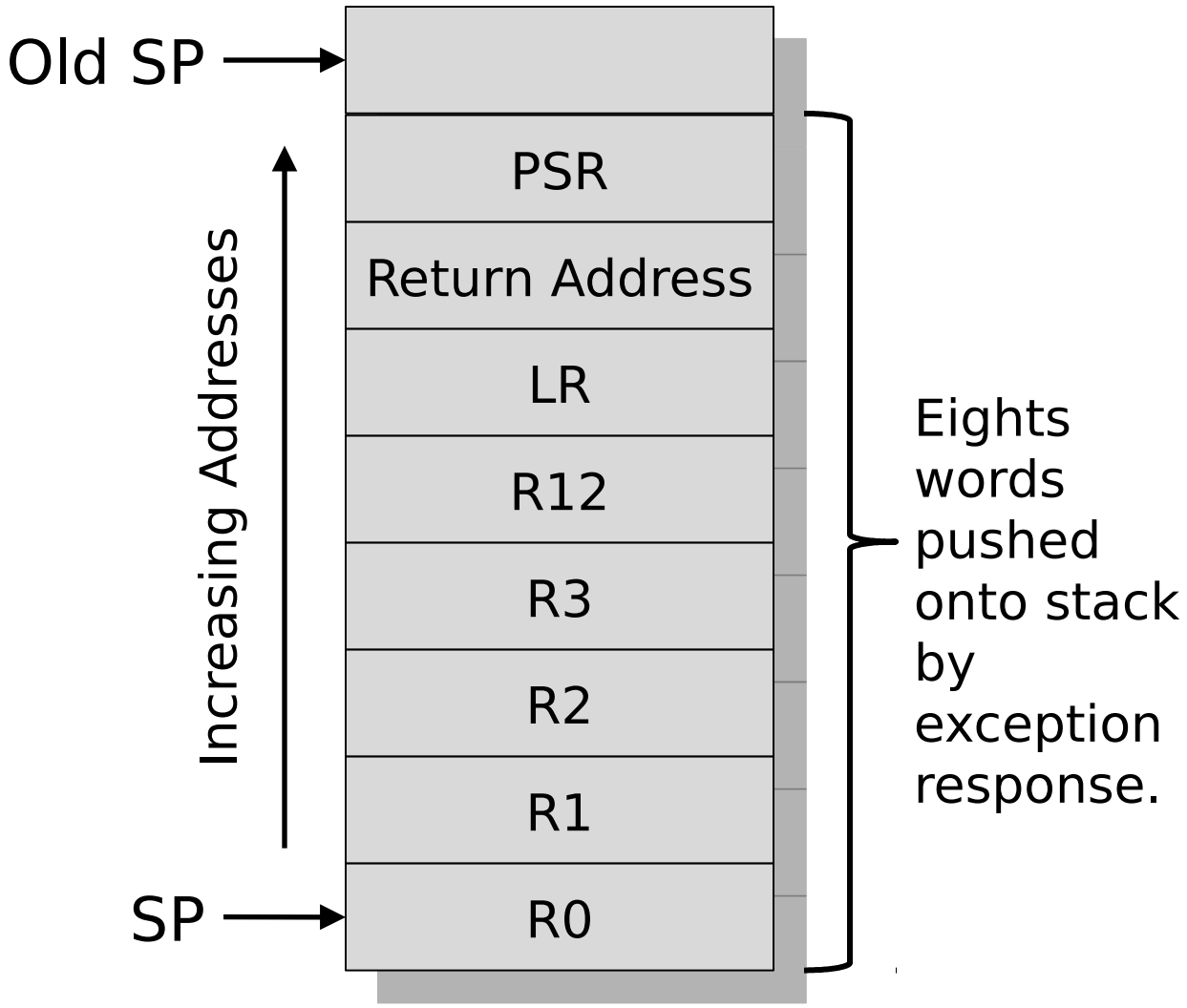
Each exception has:

1. An exception number
2. A priority level
3. An exception handler routine
4. An entry in the vector table
   (Address of associated handler routine)

Exception Response

1. Processor state (8 words) stored on stack:
   xPSR, Return Address, LR, R12, R3 - R0
   Allows a regular C function to be an ISR!
2. Processor switched (from Thread Mode) to Handler Mode
   recorded in “T” Bit (#24) of pushed xPSR.
3. PC ⩴ vector table[exception #]

Interrupt Stacking



Exception Handlers

- Definition:

  An exception handler is a software routine that is executed when a specific exception condition occurs.

  Note: Most, but not all, exception handlers return to the previous code.

Exception Return

Exception return occurs *when in Handler Mode* and one of the following instructions is executed:

1. POP/LDM includes the PC, or
2. LDR with PC as the destination, or
3. BX with any register as the source

"Unstacks" (pops and restores) registers and return address (into PC) from the stack.

- Restores T-bit (in xPSR) and thus previous processor mode.
- Resumes suspended multi-cycle instruction (if any), or restarts abandoned instruction (if any)

LATENCY

The passage of time from the moment an event occurs until the processor executes code to process the event.

ARM Latency Reduction

1. Abandoning or Suspending the current instruction to start the exception response sooner.

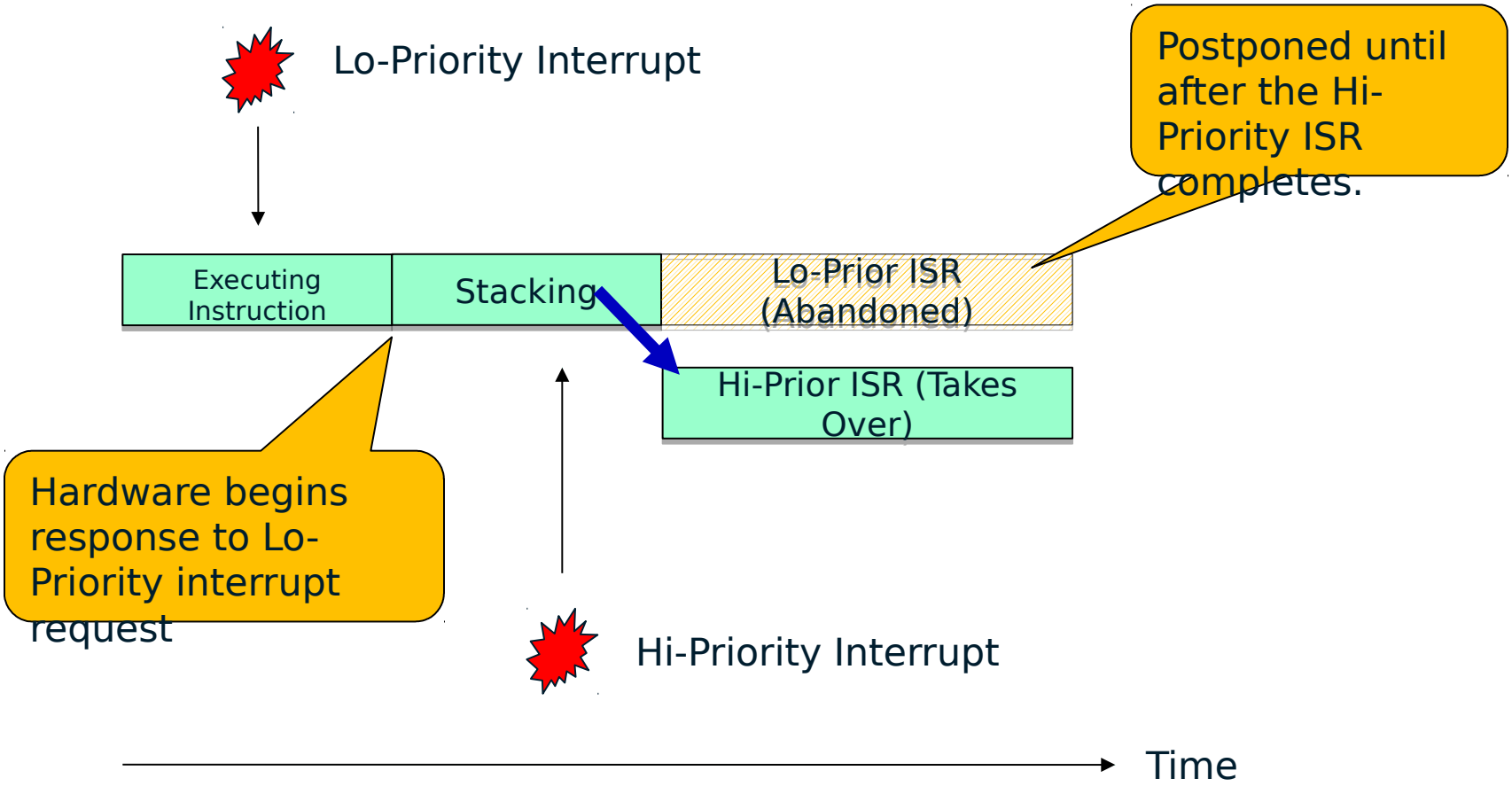
2. Late Arrival Processing: Allowing a high-priority event to take over the exception response of a low-priority event.

3. Tail-Chaining: Eliminating unnecessary stacking and unstacking.

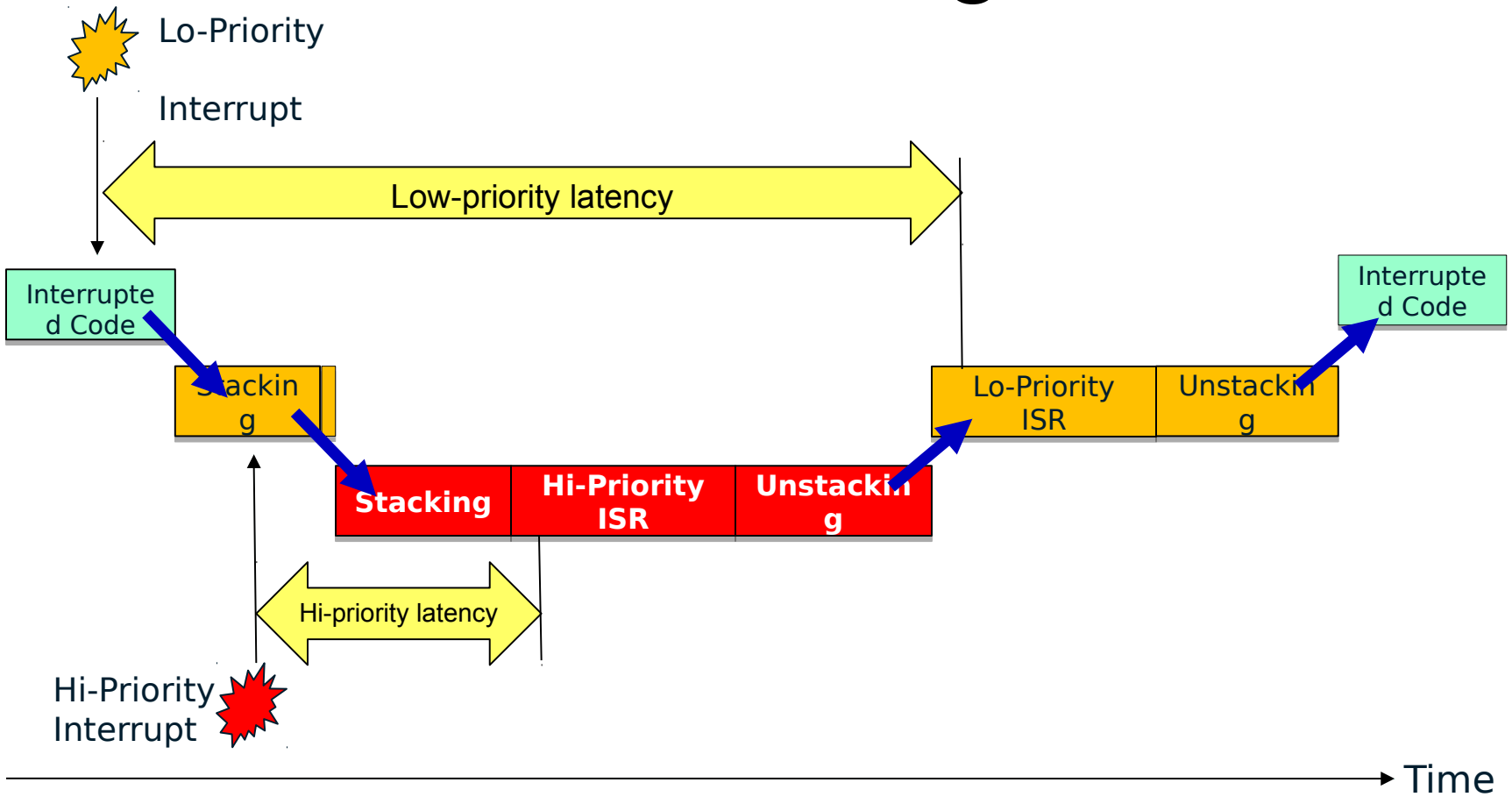
Suspending/Abandoning Instructions

1. Instructions are allowed to complete if only 1 clock cycle is required
2. Instructions that transfer multiple words to/from memory are suspended
   (LDM, STM, PUSH & POP)
3. All other instructions are abandoned and restarted after the interrupt.

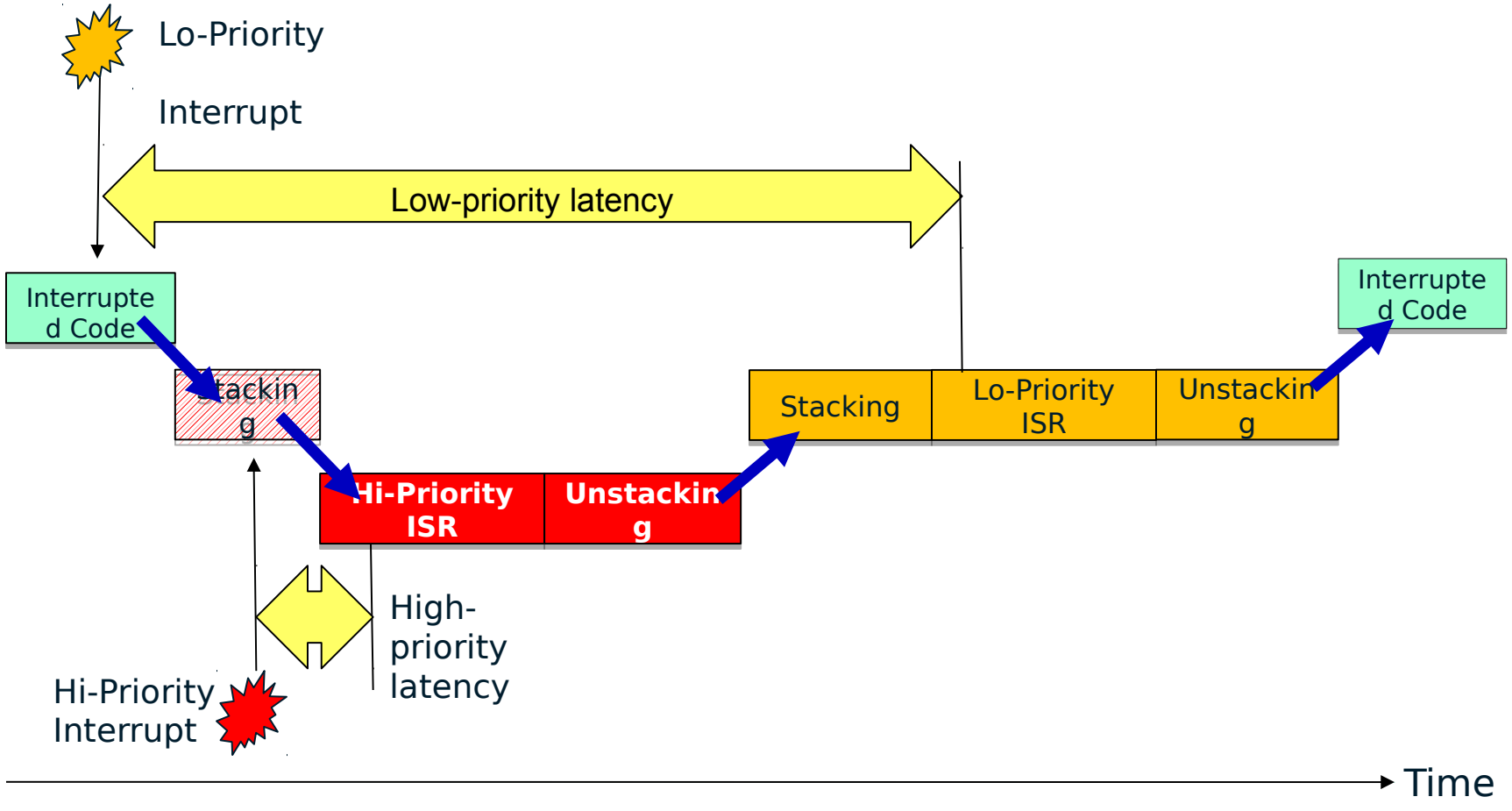
Late Arrival Processing



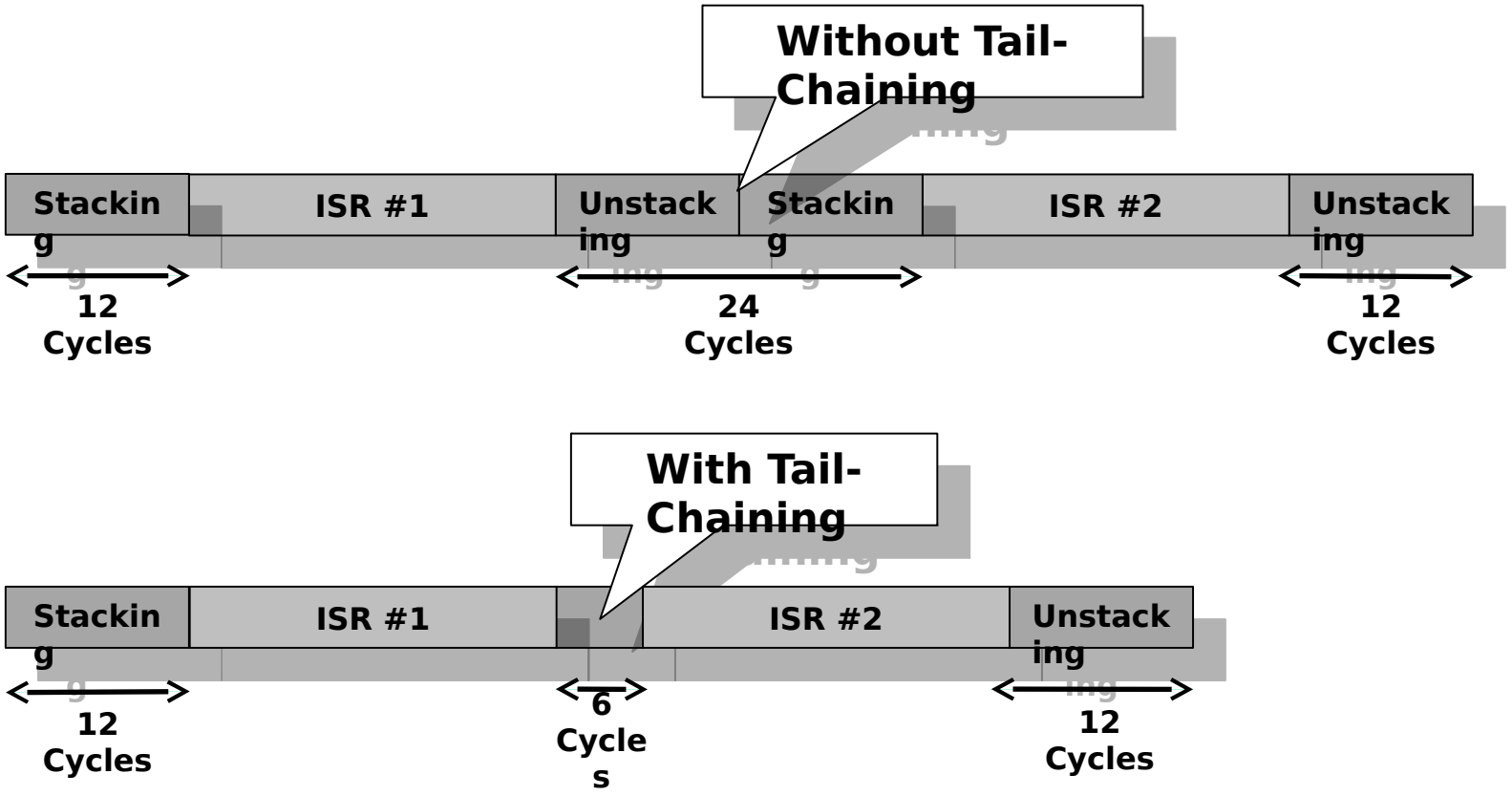
Without Late Arrival Processing



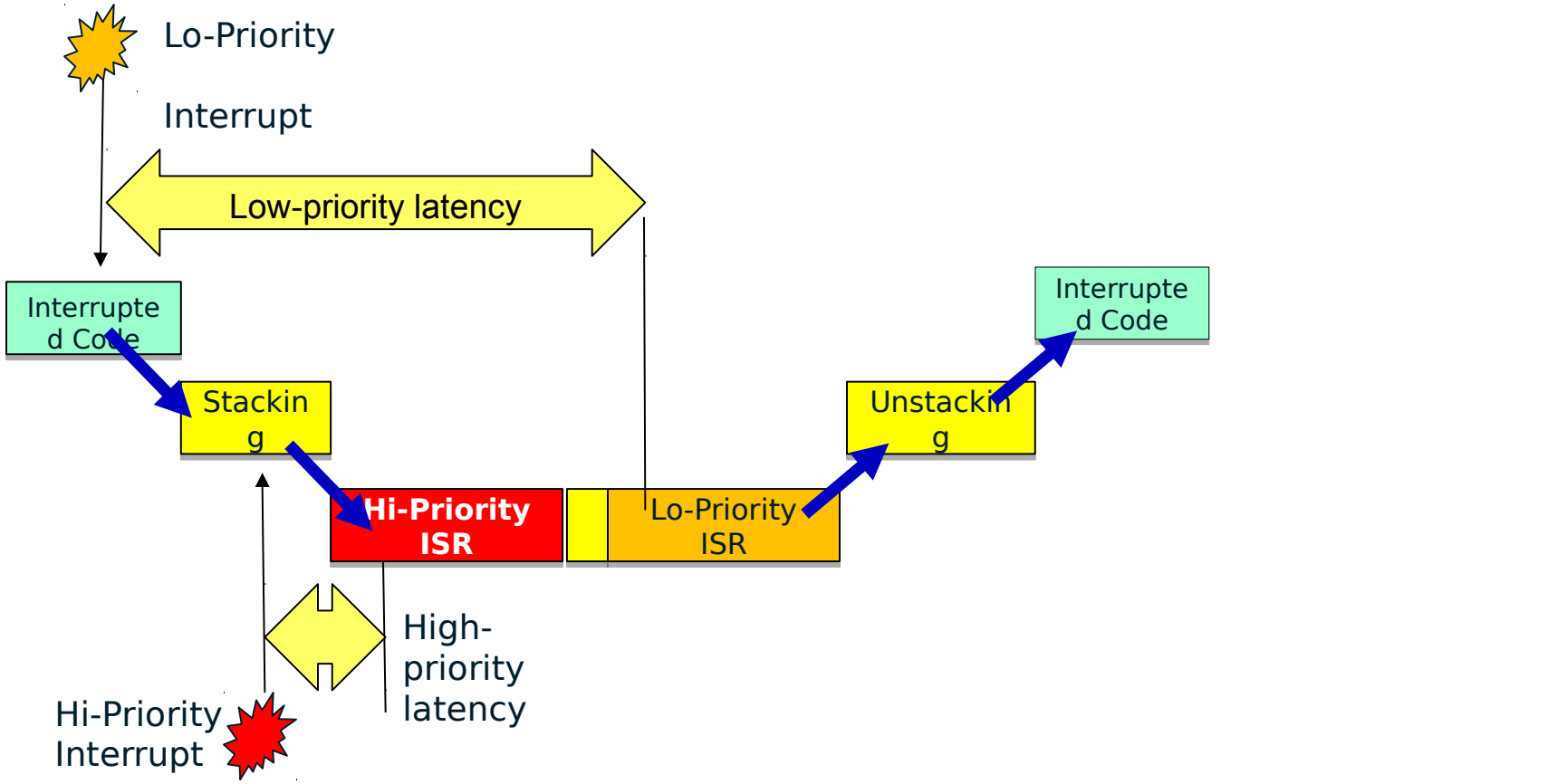
With Late Arrival Processing



Tail-Chaining



With LAP and Tail-Chaining



EXCEPTION AND INTERRUPT PRIORITIES

Nested Vectored Interrupt Controller

(Mapped to addresses $E000E100\text{-}E000ECFF_{16}$)

The NVIC provides the ability to:

1. Individually Enable/Disable interrupts from specific devices.

2. Establishes relative priorities among the various interrupts.

Exception Priorities and Vector Table Positions

Exception Type	Exception #[1]	IRQ #	Priority	Comment
		n/a		Initial SP value (loaded on reset)
Reset	1		-3 (fixed)	Power up and warm reset
NMI	2		-2 (fixed)	Non-Maskable Interrupt
Hard Fault	3		-1 (fixed)	
Memory Mgmt.	4		Configured through System Handler Priority Registers	
Bus Fault	5			Address/Memory-related faults
Usage Fault	6			Undefined instruction
Reserved	7-10			
SVCall	11			Software Interrupt (SVC instruction)
Debug Monitor	12			
Reserved	13			
PendSV	14			
SysTick	15			System Timer Tick
Ext. Interrupts	16-255	0-239	Configurable	

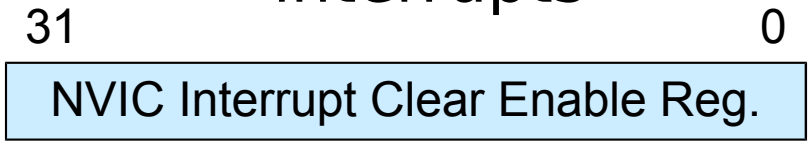
[1]**Vector table entries are ordered by exception number, with the first entry reserved for the initial SP value. IRQ # is the interrupt request number (same as the exception number minus 16).**

Interrupt Priority Registers

- Each interrupt is assigned an 8-bit priority number:
  - Highest Priority: 0
  - Lowest Priority: 255

Enabling/Disabling Interrupts

```
CPSIE ; Enable all external
interrupts
CPSID    ; Disable all external
interrupts
```



Set a bit to 1 in this register to disable interrupts from an external I/O device

Set a bit to 1 in this register to enable interrupts from an external I/O device

NVIC External Interrupts

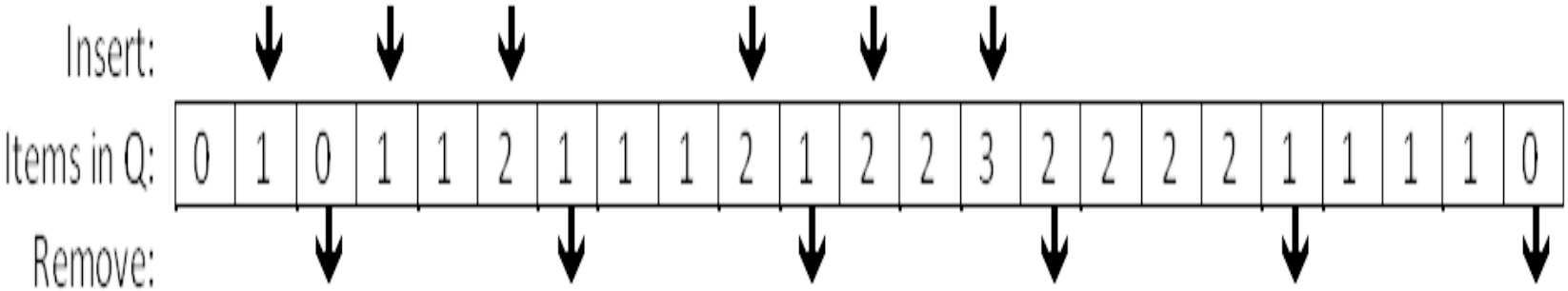
0-4 GPIO Ports A-E
5,6 UART 0 & 1
7 SSI
8 I2C
9 PWM Fault
10-12 PWM Generator 0-2
13 Reserved
14-17 ADC Sequence 0-3

18 Watchdog Timer
19-24 Timer 0a-2b
25 Analog Comparator
26-27 Reserved
28 System Control
29 Flash Control
30-31 Reserved

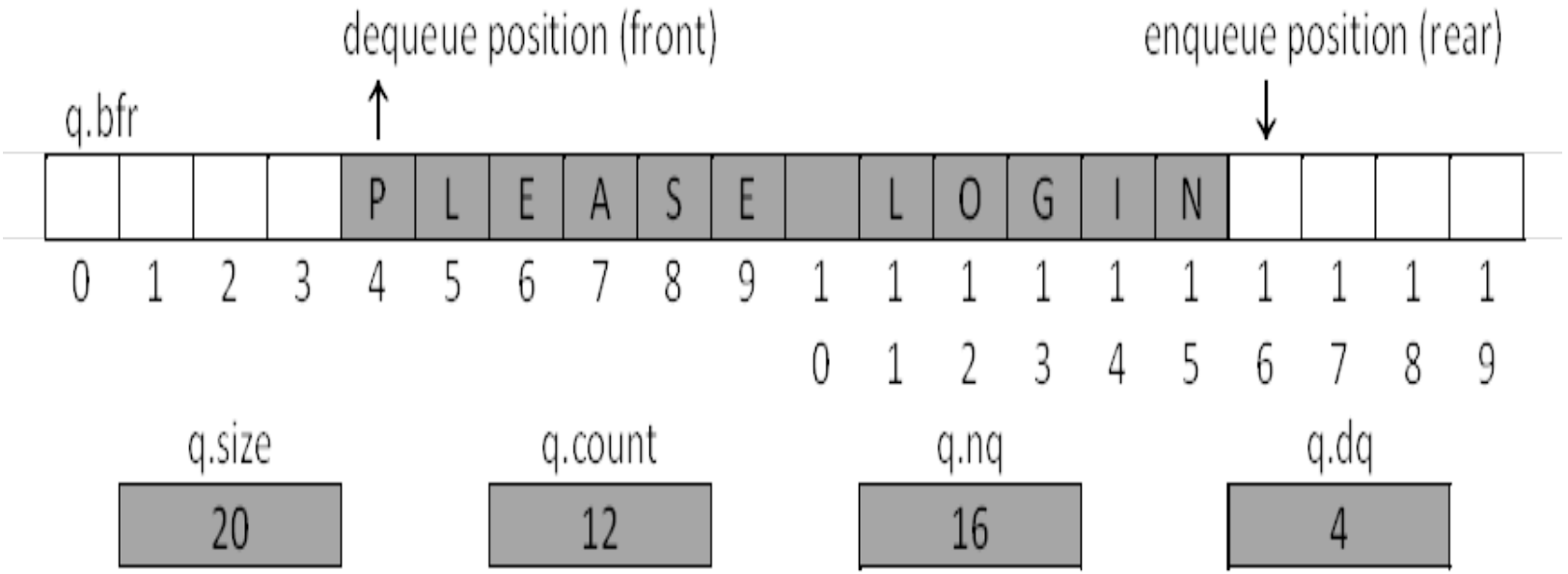
Bit in Interrupt Registers

BUFFERING

Using a Queue to Decouple Data Arrival from Data Processing



Interrupt Buffering

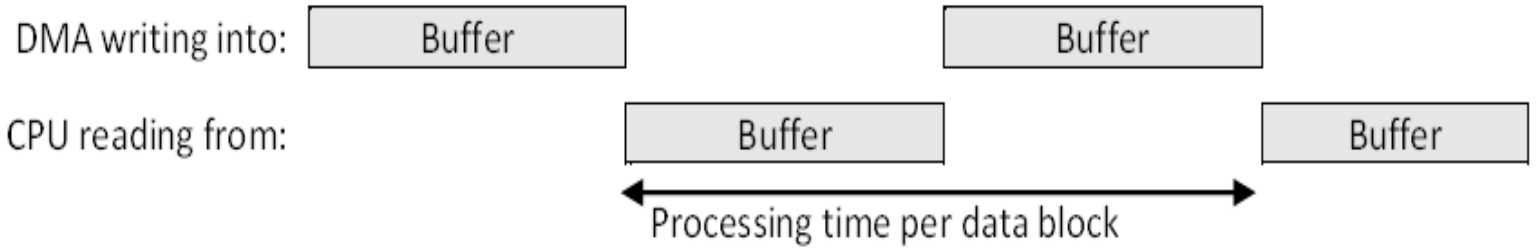


Implementing a Queue in C

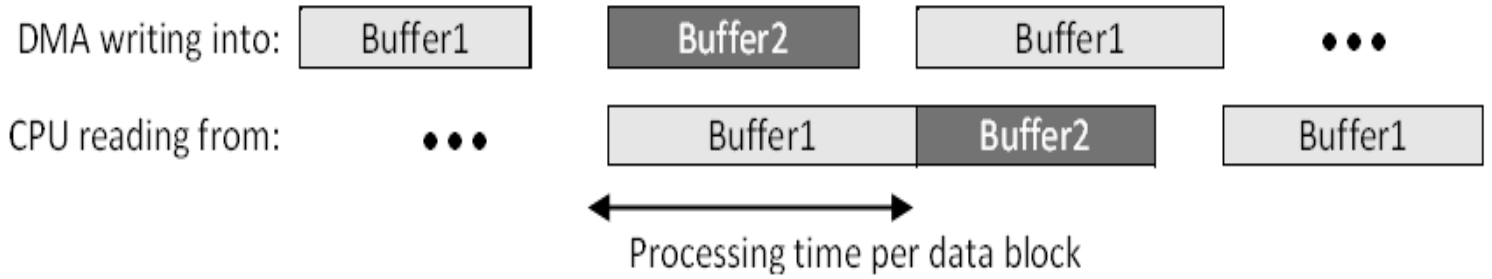
```
BOOL Enqueue(QUEUE *q, uint8_t data)
    {
    BOOL full ;

    disable() ;
    full = q->count == q->size ;
    if (!full)
        {
        q->bfr[q->nq] = data ;
        if (++q->nq == q->size) q->nq = 0 ;
        q->count++ ;
        }
    enable() ;

    return !full ;
    }
```

```
BOOL Dequeue(QUEUE *q, uint8_t *ptr2data)
                {
                BOOL empty ;

                disable() ;
                empty = q->count == 0 ;
                if (!empty)
                        {
                        *ptr2data = q->bfr[q->dq] ;
                        if (++q->dq == q->size) q->dq
= 0 ;
                        q->count-- ;
                        }
                enable() ;

                return !empty ;
                }
```

Single Buffered DMA



Double Buffered DMA



SYNCHRONIZATION, TRANSFER RATE AND LATENCY

Polled Waiting Loops

- Test device status in a ***waiting loop*** before transferring each data byte.
- Maximum data rate: Determined by the time required to execute one iteration of the waiting loop plus the transfer.
- Latency: Unpredictable - no guarantee when the program will arrive at the waiting loop.

Synchronizing data transfers using polled waiting loops.

```
//  TXFE (bit 7 of status port) =1 if transmitter holding register empty

#define TXFE          (1 << 7)

// RS232 Data Port  = rs232 base address + 0
// RS232 Status Port          = rs232 base address + 24

void  rs232Output(uint8_t *ptr2data, int bytes, volatile uint32_t *rs232base)
        {
        for (int byte = 0; byte < bytes; byte++)
                {
                while ((*(rs232base + 6) & TXFE) == 0)
                        {
                        // do nothing (wait for data to arrive)
                        }
                *rs232base = (uint32_t) *ptr2data++ ;
                }
        }
```

Polled waiting loop: Optimized in assembly.

```
        EXPORT rs232Output
rs232Output:

        ;  R0 contains the parameter ‘ptr2data’
        ;  R1 contains the parameter ‘bytes’
        ;  R2 contains base address of RS232 device

Repeat:   CBZ     R1,Return       ; more data to send?
OutWait:  LDR     R3,[R2,#0x18]   ; get status word
          TST     R3,#0x80        ; TXFE = 1?
          BEQ     OutWait         ; loop until ready
          LDRB    R3,[R0],#1      ; get next byte & update pointer
          STR     R3,[R2]         ; send data to device
          SUB     R1,R1,#1        ; decrement byte count
          B       Repeat          ; do it all again
Return:   BX      LR              ; return
```

Cortex-M3 Instruction Clock Cycles[1]

Instructions		Clock Cycles
PUSH, POP, LDM, STM		1 + #regs
SDIV, UDIV		2 - 12
SMLAL, UMLAL		4 - 7
SMULL, UMULL		3 - 5
Unconditional Branch (B, BL, BX)		2 - 4
Conditional Branch	Successful	2 - 4
	Failed	1
LDRD, STRD		3
ADR, MLA, MLS, & all LDR's and STR's		2
All other instructions		1

[1]Assumes memory and processor run at the same speed.

Data Rate: Polled Waiting Loop

```
                EXPORT rs232Output          ;  Clock
rs232Output:                                ;  Cycles

Repeat:   CBZ     R1,Return          ;  1 (Fail)
OutWait:  LDR     R3,[R2,#0x18]      ;  2
          TST     R3,#0x80           ;  1
          BEQ     OutWait            ;  1 (Fail)
          LDRB    R3,[R0],#1         ;  2
          STR     R3,[R2]            ;  2
          SUB     R1,R1,#1           ;  1
          B       Repeat             ;  4
```

Clock cycles: 14

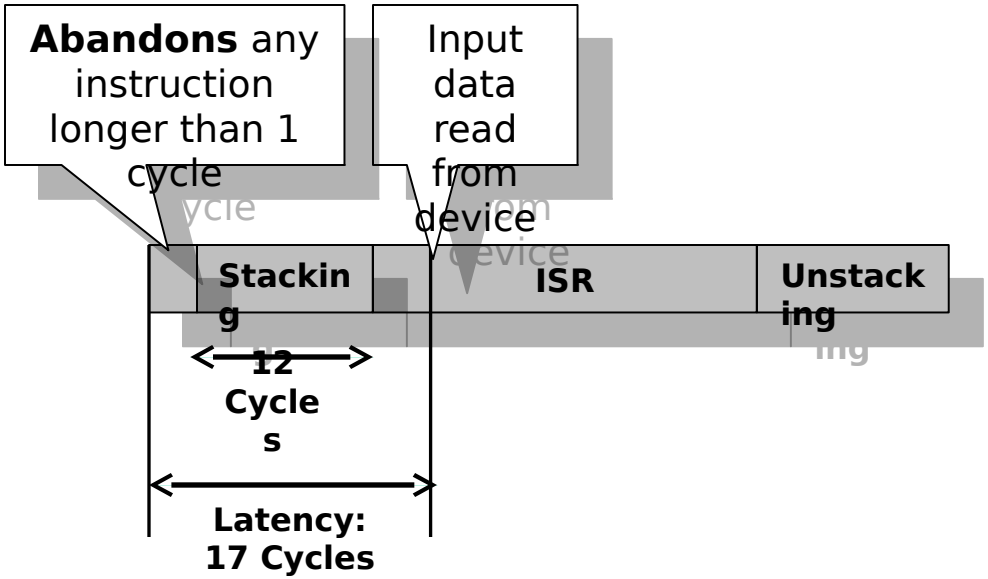
Execution time: 14 × 20 nsec = 280 nsec

Maximum Data Rate: $10^9 \div 280$ = **3.6 MB/sec**

Data Rate:Interrupt-Driven I/O

```
                                                                    Clock
Rs232InputHandler:                                                  Cycles

        ADR     R0,rs232InpDataPort     ; R0 ≔ rs232 device address     2
        LDR     R0,[R0]                 ; R0 ≔ rs232 input data byte    2

        LDR     R2,rs232InputNQIndex    ; update enqueue index          2
        ADD     R2,R2,#1                                                1
        AND     R2,R2,#0x3F             ; increment & wrap (0 to 63)    1
        STR     R2,rs232InputNQIndex                                    2

        LDR     R3,rs232InputQueue      ; R3 ≔ address of buffer        2
        STRB    R0,[R3,R2]              ; store data into buffer        2

        LDR     R2,rs232InputCount      ; update #items in queue        2
        ADD     R2,R2,#1                                                1
        STR     R2.rs232InputCount                                      2

        BX      LR                      ; return                        4

        Tail-chaining time between successive ISRs:                     6

        Clock cycles:                                                   29
        Execution time:                                                 29 × 20 nsec = 580 nsec
        Maximum Data Rate:                                              10^9÷580 = 1.7 MB/sec
```

Interrupt Latency



Total latency: 17 Cycles × 20 nsec = **340 nsec**

Direct Memory Access

- Requires additional hardware to control data transfers independent of CPU.
- Competes with CPU for control of memory.
- Does not have to wait for current instruction to complete – only the current memory cycle.
  - Latency is thus 1 memory cycle.
  - If memory and I/O are on separate chips: 1 cycle = **60 nsec**
- Data Rate determined by memory speed.
  - 32-bit data bus: 4 bytes / (60×10^{-9} sec) = **66 MB/Sec**

Performance Estimates and Relative Cost

	Polled Waiting Loop	Interrupt-Driven I/O	Direct Memory Access
Maximum Transfer Rate	~3.6 MB/sec	~1.7 MB/sec	~66 MB/sec
Best-Case Latency	unpredictable	340 nsec	~60 nsec
Hardware Cost	Least	Low	Moderate
Software Complexity	Low	Moderate	Moderate