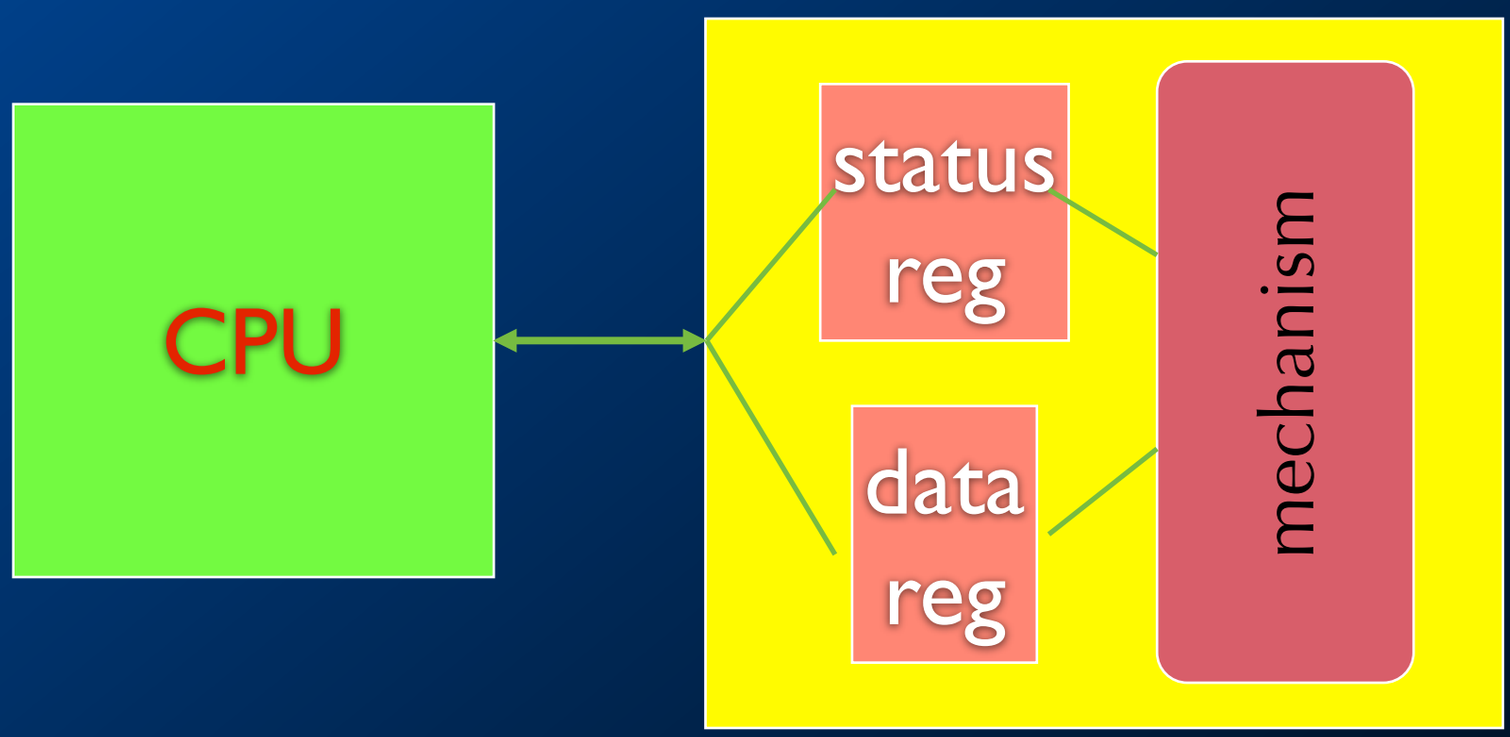
Bus-based computering systems

Agenda

- Input and output
- Bus

I/O devices

- Usually includes some non-digital component.
- Typical digital interface to CPU:



Application: 8251 UART

- Universal asynchronous receiver transmitter (UART) : provides serial communication.

- Allows many communication parameters to be programmed.

Serial communication

- Characters are transmitted separately:

Serial communication parameters

- Baud (bit) rate.
- Number of bits per character.
- Parity/no parity.
  - Even/odd parity.
- Length of stop bit (1, 1.5, 2 bits).

分类 — 从属关系

- 系统设备:操作系统启动时已经在系统中注册的标准设备。
  - 例如NOR/NAND闪存，触摸面板等。
  - 这些设备的操作系统中有设备驱动程序和管理程序。
  - 用户应用程序只需要调用操作系统提供的标准命令或函数就可以使用这些设备。
- 用户设备:操作系统启动时未在系统中注册的非标准设备。
  - 通常设备驱动程序是由用户提供的。用户必须以某种方式将这些设备的控制权转移到操作系统进行管理。
  - 典型的设备包括SD卡、u盘等。

分类 — 使用

- 专用设备:同一时间只能被一个进程使用的设备。对于多个并发进程，每个进程使用设备是互斥的。一旦操作系统将设备分配给一个特定的进程，它将被该进程独占，直到该进程使用后释放它。
- 共享设备:可被多个进程同时寻址的设备。共享设备必须是可寻址的，并且是随机寻址的。共享设备机制可以提高每个设备的利用率。
- 虚拟设备: 通过虚拟技术将一台独占设备虚拟成多台逻辑设备，供多个用户进程同时使用， 通常把这种经过虚拟的设备称为虚拟设备。

类别 — 特征

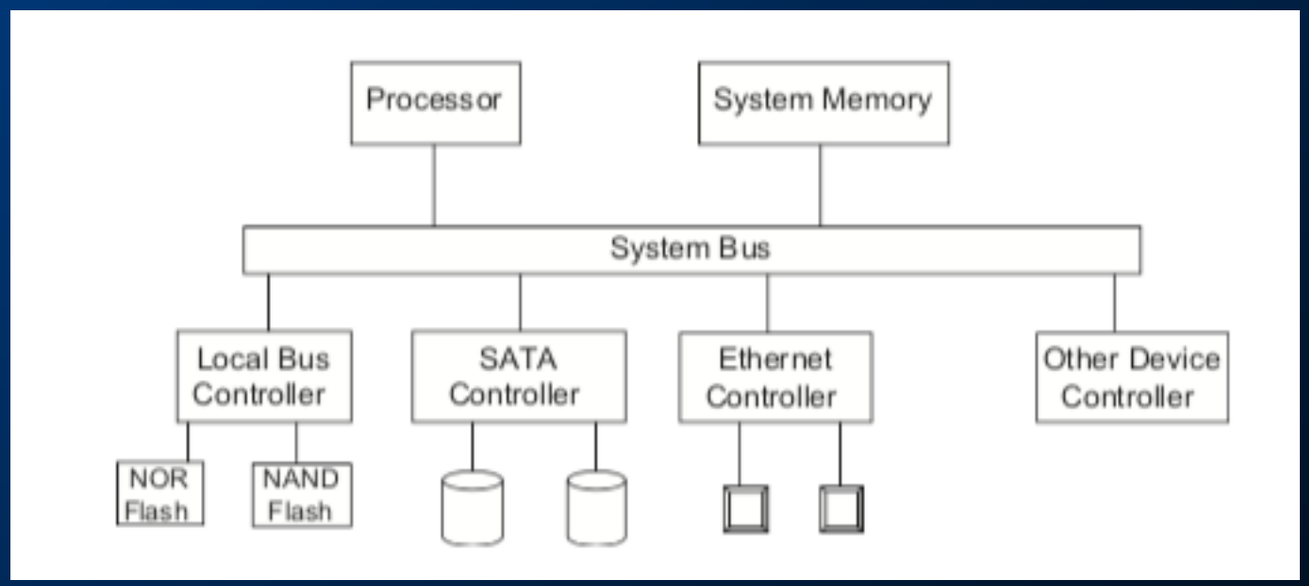
- 存储设备:用于存储信息的设备。嵌入式系统中的典型例子包括硬盘、固态硬盘、NOR/NAND闪存。
- I / O设备:
  - 输入设备:输入设备负责将信息从外部输入到内部系统，如触摸面板、条形码扫描仪等。
  - 输出设备:输出设备负责将嵌入式系统处理后的信息输出到外部世界，如液晶显示器、扬声器等。

类别 — 信息传输单元

- 块设备:这种类型的设备以数据块为单位组织和交换数据。
  - 它是一种结构化设备。
  - 典型的设备是硬盘。
  - 在I/O操作中，即使只是一个单字节的读/写，也应该读或写整个数据块。
- 字符设备:这类设备以字符单位组织和交换数据。
  - 它是一种非结构化设备。
  - 字符设备有很多种，如串口、触摸面板、打印机等。
  - 字符设备的基本特征是传输速率低且不可寻址。当字符设备执行I/O操作时，经常使用中断。

I/O device

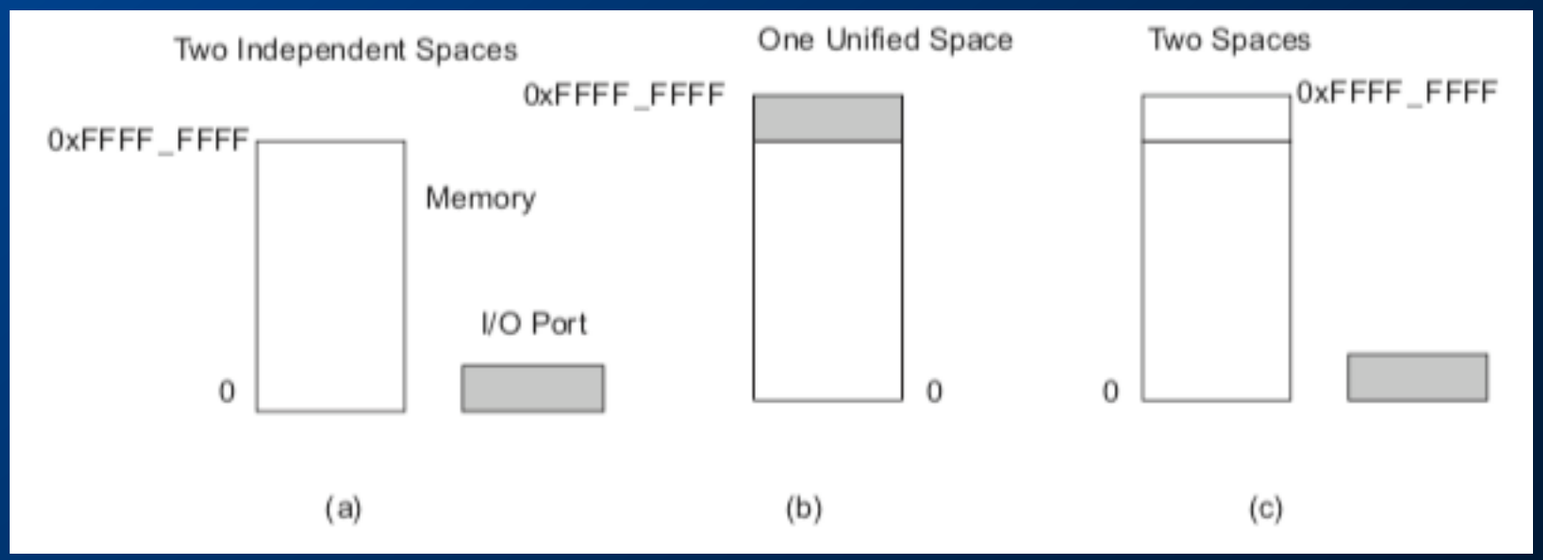
- Usually the I/O device is composed of two parts
  - a mechanical part
  - an electronic part.
    - the electronic part is called the device controller or adapter.



Programming I/O

- three methods to select the control registers or data buffer during the communication
  - Independent I/O port.
  - memory-mapped I/O.
  - Hybrid solution. the hybrid model include memory-mapped I/O data buffers and separate I/O ports for the control registers.

- Intel x86 provides in, out instructions. Most other CPUs use memory-mapped I/O.
- I/O instructions do not preclude memory-mapped I/O.



(a) Independent I/O and memory space. (b) Memory-mapped I/O. (c) Hybrid Solution.

memory-mapped I/O

- The strength of the memory-mapped I/O can be summarized as:
  - In a memory-mapped I/O mode, device control registers are just variables in memory and can be addressed in C the same way as any other variables. Therefore, an I/O device driver can be completely written in the C language.
  - In this mode, there is no special protection mechanism needed to keep user processes from performing I/O operations.
- The disadvantages of memory-mapped I/O mode can be summarized as:
  - Most current embedded processors support caching of memory. Caching a device control register would cause a disaster. In order to prevent this, the hardware has to be given the capability of selectively disabling caching. This would increase the complexity of both the hardware and software in the embedded system.
  - If there is only one address space, all memory references must be examined by all memory modules and all I/O devices in order to decide which ones to respond to. This significantly impacts the system performance.

ARM memory-mapped I/O

- Define location for device:

```
DEV1 EQU 0x1000
```

- Read/write code:

```
LDR r1,#DEV1 ; set up device adrs
LDR r0,[r1] ; read DEV1
LDR r0,#8 ; set up value to write
STR r0,[r1] ; write value to device
```

Peek and poke

- Traditional HLL interfaces:

```
int peek(char *location) {
    return *location; }
```

```
void poke(char *location, char newval) {
    (*location) = newval; }
```

Busy/wait output

- Simplest way to program device.
  - Use instructions to test when device is ready.

```
current_char = mystring;
while (*current_char != '\0') {
    poke(OUT_CHAR,*current_char);
    while (peek(OUT_STATUS) != 0);
    current_char++;
}
```

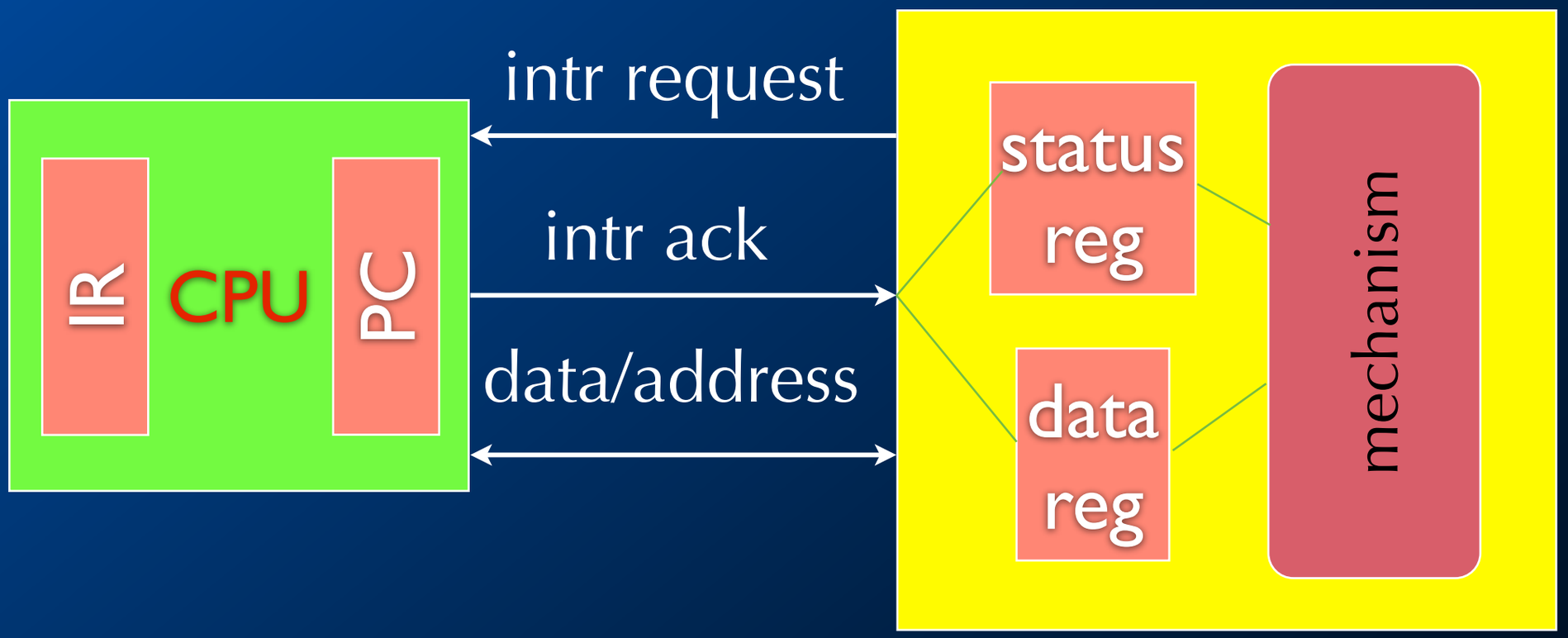
Simultaneous busy/wait input and output

```
while (TRUE) {
   /* read */
   while (peek(IN_STATUS) == 0);
   achar = (char)peek(IN_DATA);
   /* write */
   poke(OUT_DATA,achar);
   poke(OUT_STATUS,1);
   while (peek(OUT_STATUS) != 0);
}
```

Interrupt I/O

- Busy/wait is very inefficient.
  - CPU can't do other work while testing device.
  - Hard to do simultaneous I/O.
- Interrupts allow a device to change the flow of control in the CPU.
- Causes subroutine call to handle device.

Interrupt interface



Interrupt behavior

- Based on subroutine call mechanism.
- Interrupt forces next instruction to be a subroutine call to a predetermined location.
-

Interrupt physical interface

- CPU and device are connected by CPU bus.
- CPU and device handshake:
- device asserts interrupt request;
- CPU asserts interrupt acknowledge when it can handle the interrupt.

Example: character I/O handlers

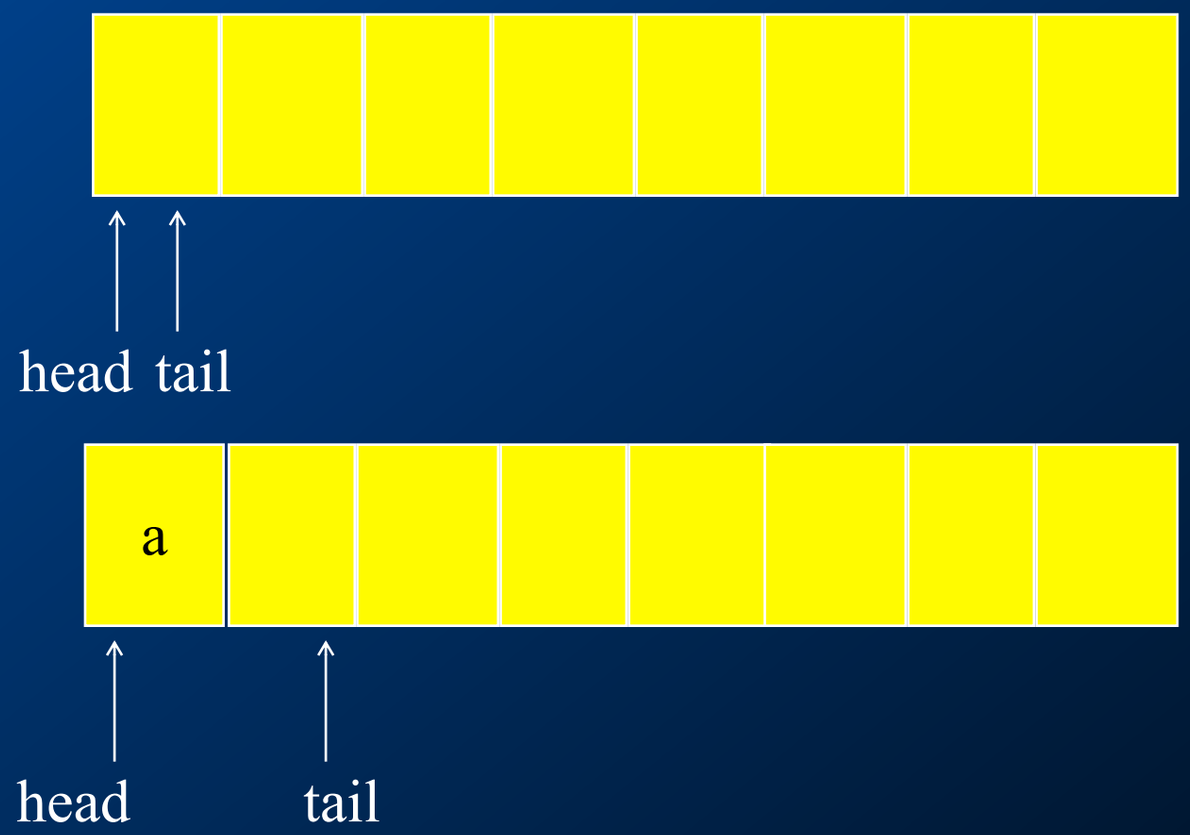
```
void input_handler() {
    achar = peek(IN_DATA);
    gotchar = TRUE;
    poke(IN_STATUS,0);
}
void output_handler() {
}
```

Example: interrupt-driven main program

```
main() {
    while (TRUE) {
        if (gotchar) {
            poke(OUT_DATA,achar);
            poke(OUT_STATUS,1);
            gotchar = FALSE;
        }
    }
    other processing....
}
```

Example: interrupt I/O with buffers

- Queue for characters:

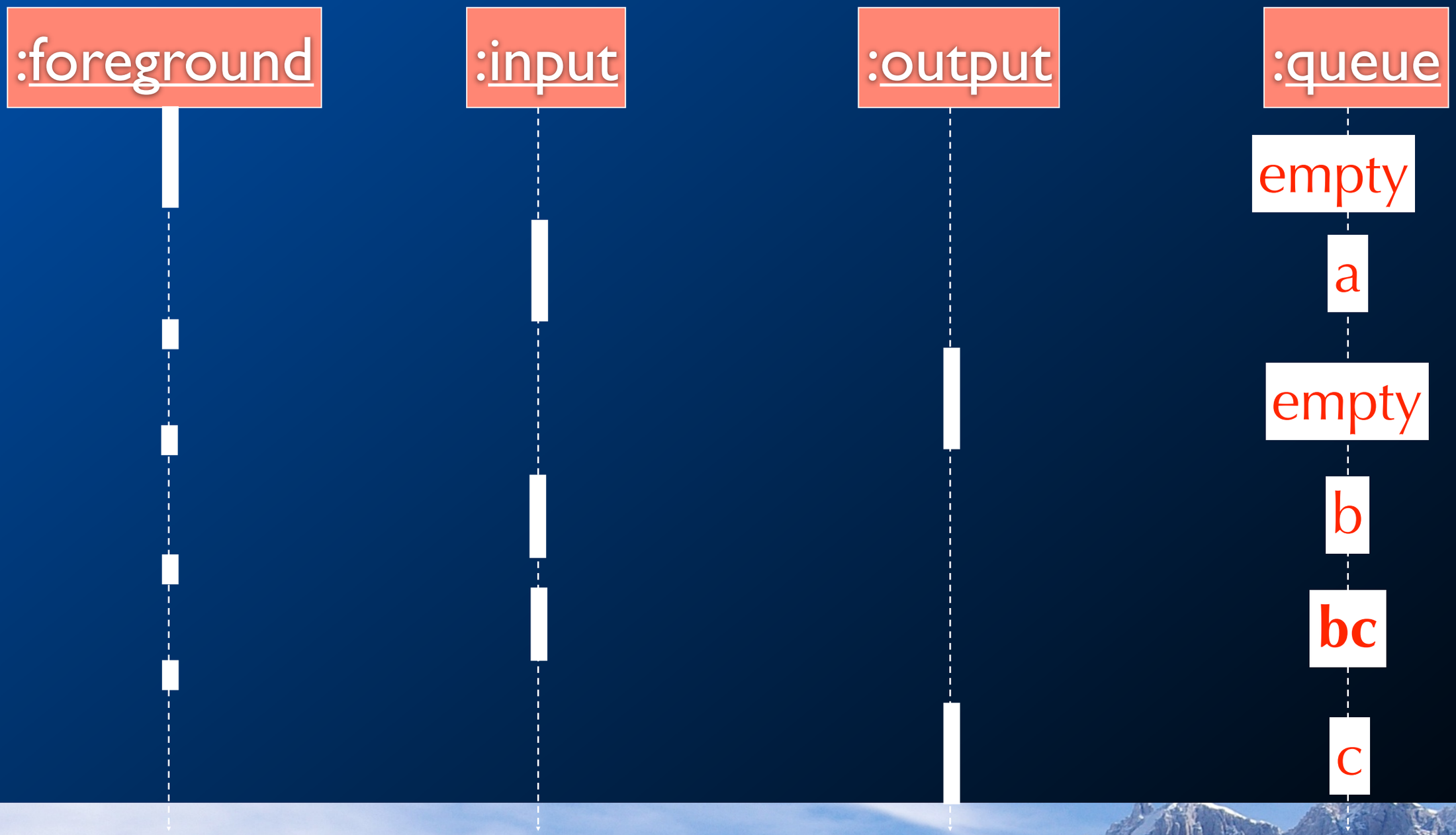


leave one empty slot to allow full buffer to be detected

Buffer-based input handler

```
void input_handler() {
    char achar;
    if (full_buffer()) error = 1;
    else {
        achar = peek(IN_DATA);
        add_char(achar); }
    poke(IN_STATUS,0);
    if (nchars == 1)
    {
        poke(OUT_DATA,remove_char());
         poke(OUT_STATUS,1); }
}
```

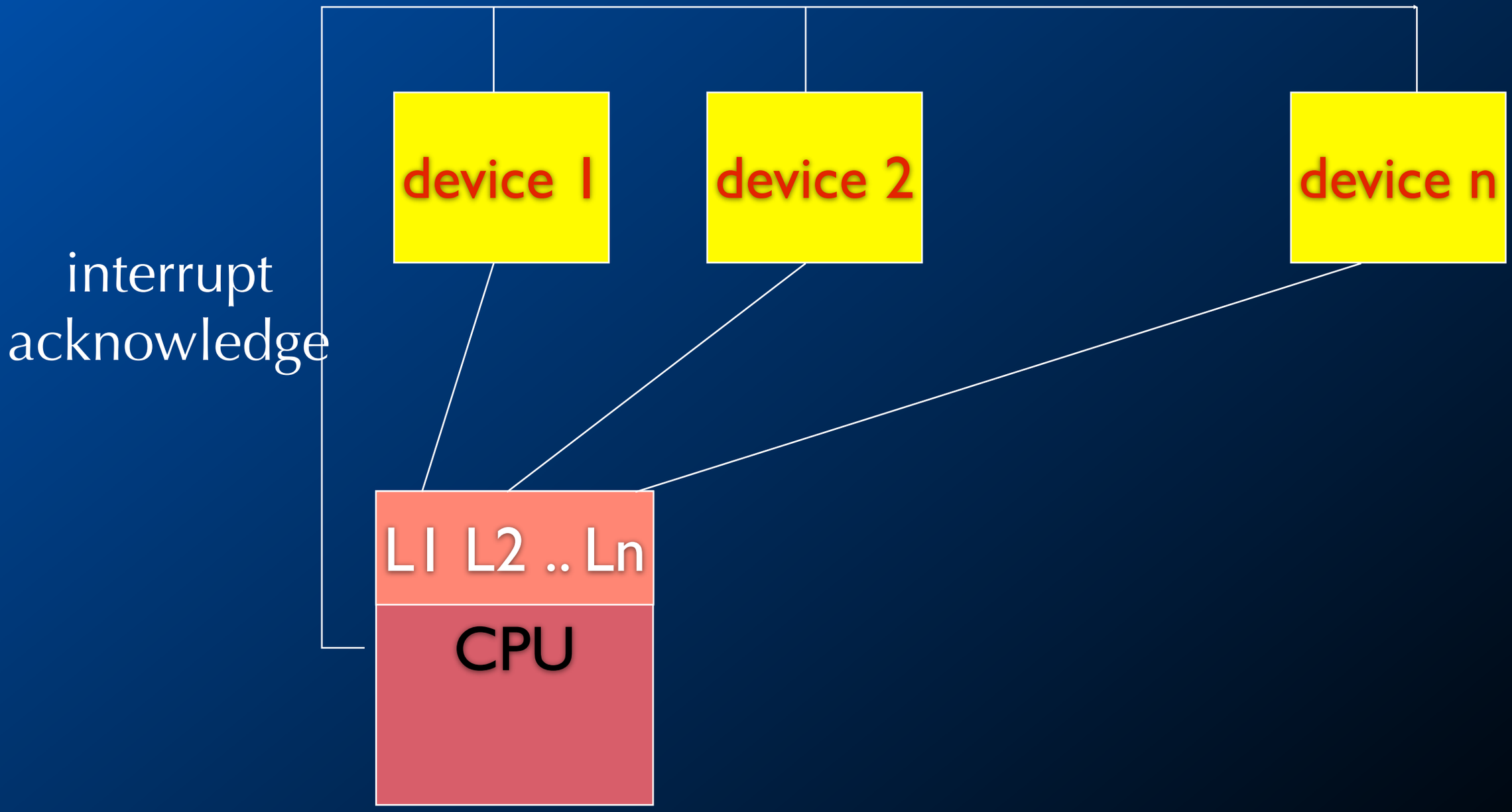
I/O sequence diagram



Priorities and vectors

- Two mechanisms allow us to make interrupts more specific:
  - Priorities determine what interrupt gets CPU first.
  - Vectors determine what code is called for each type of interrupt.
- Mechanisms are orthogonal: most CPUs provide both.

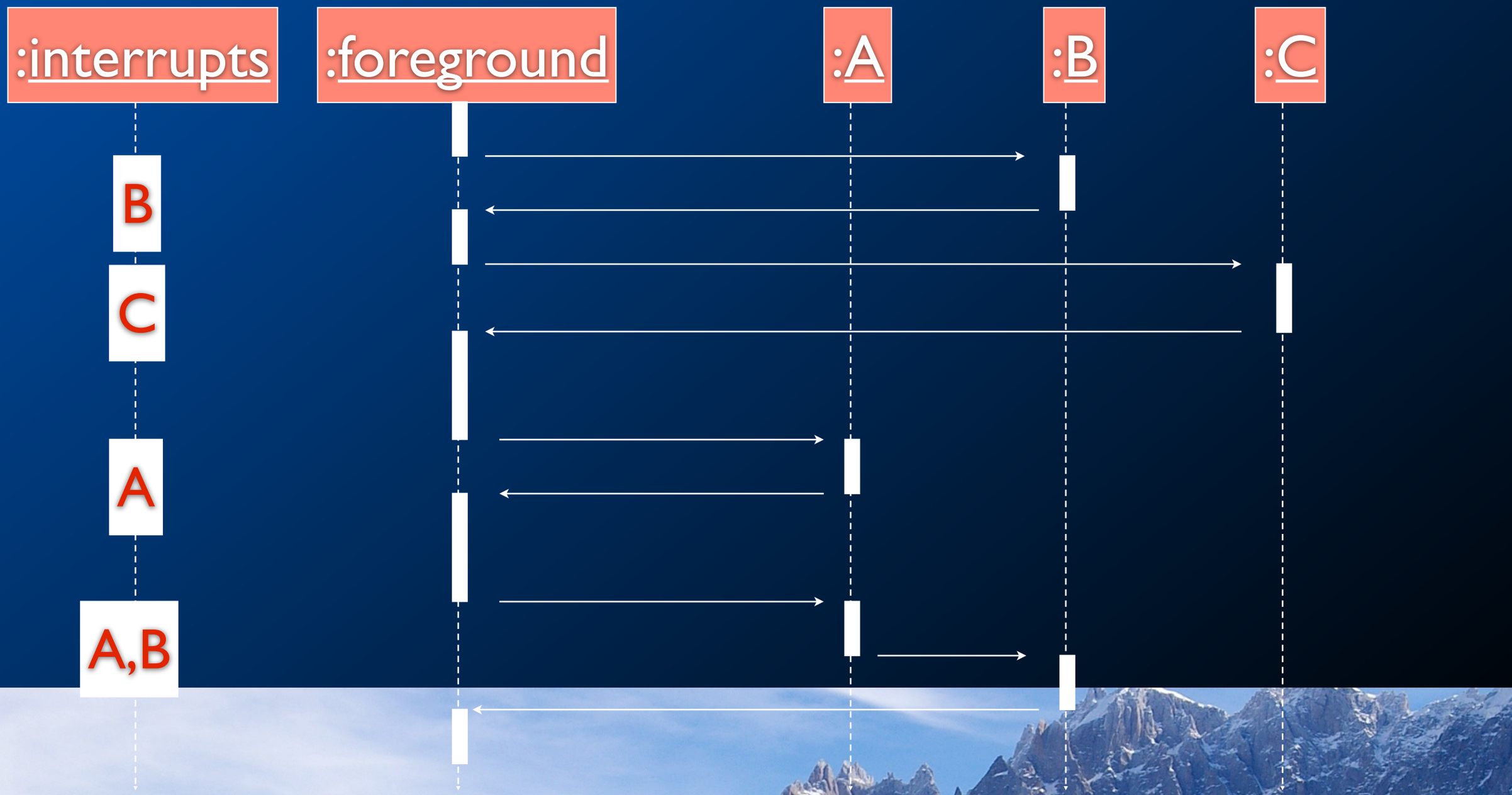
Prioritized interrupts



Interrupt prioritization

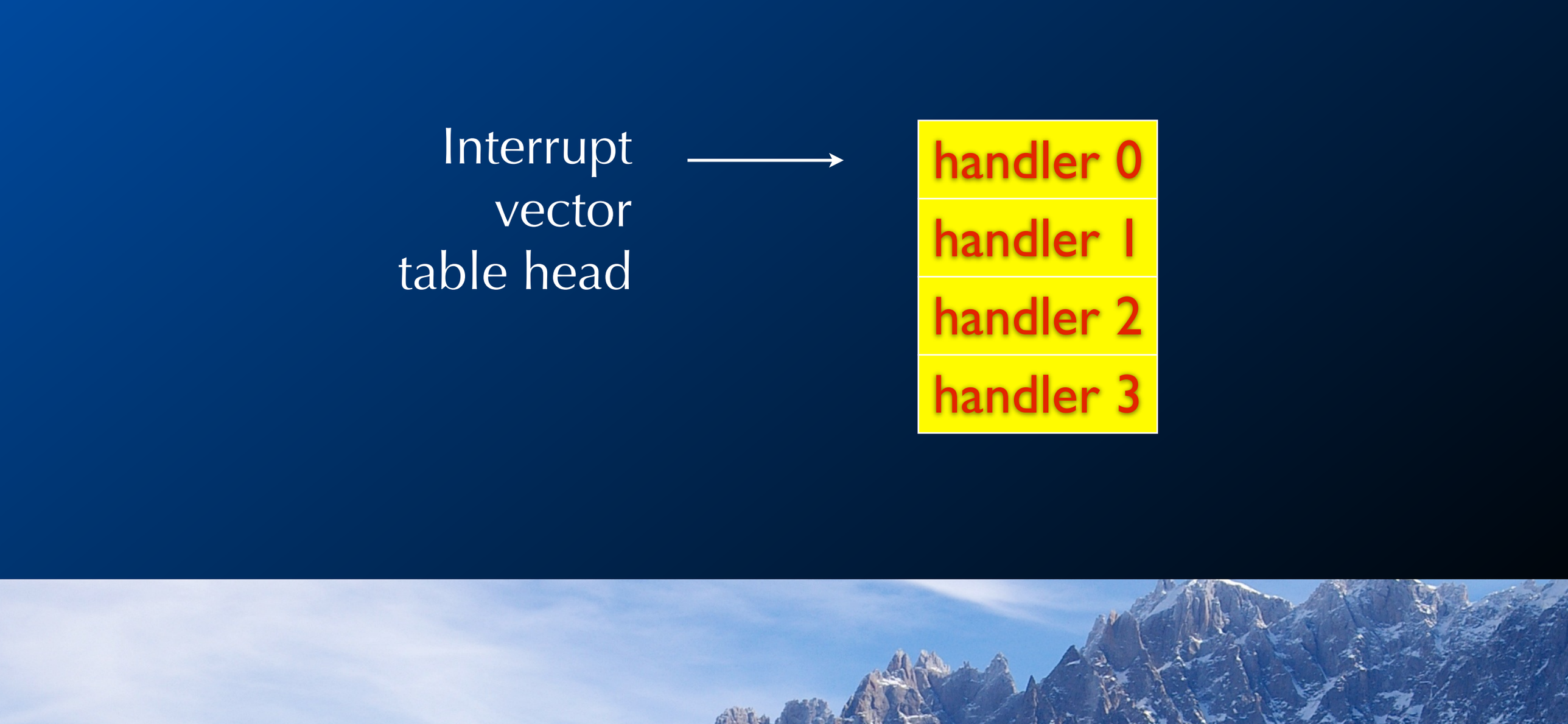
- Masking: interrupt with priority lower than current priority is not recognized until pending interrupt is complete.
- Non-maskable interrupt (NMI): highest-priority, never masked.
- Often used for power-down.

Example: Prioritized I/O

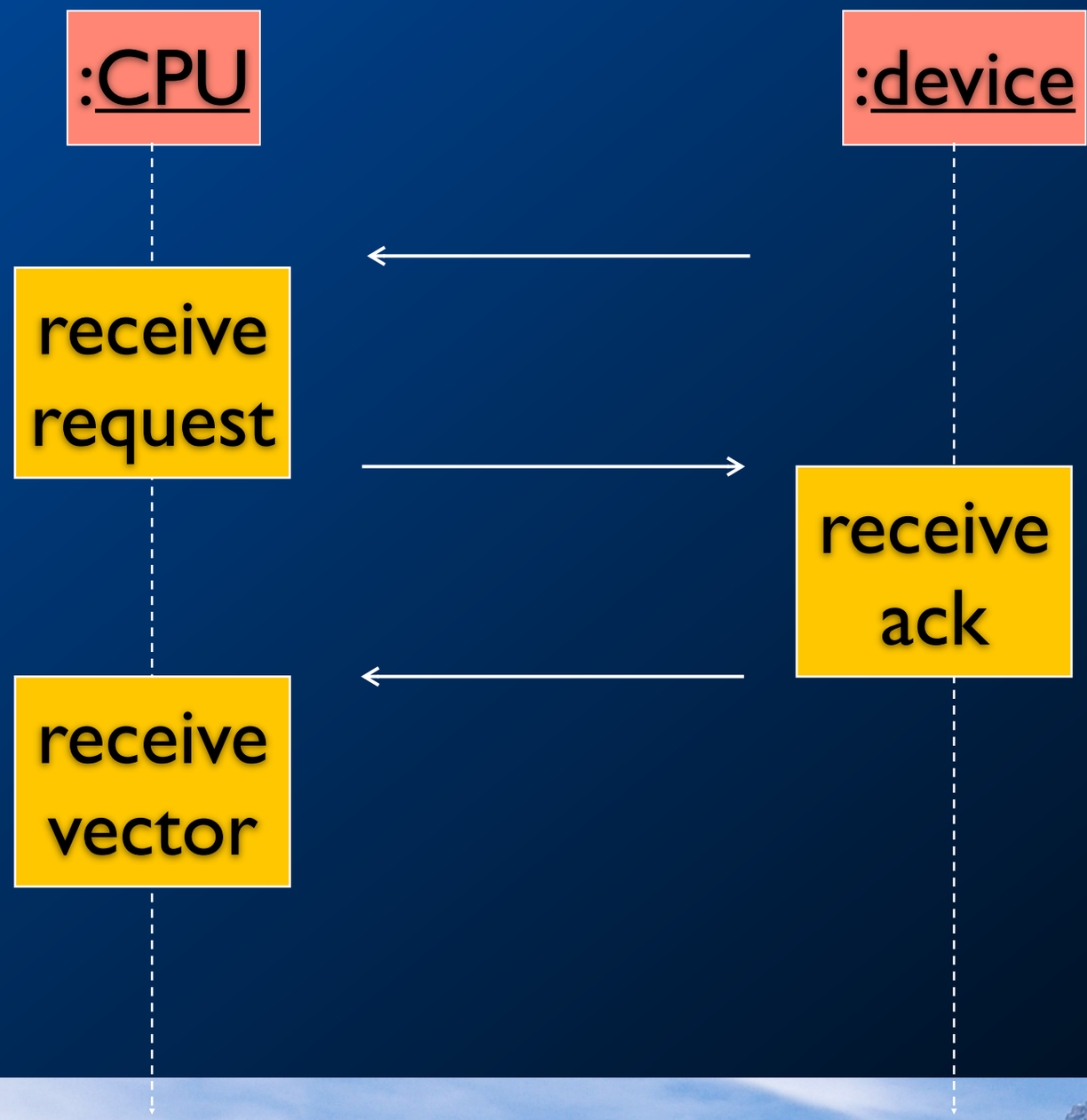


Interrupt vectors

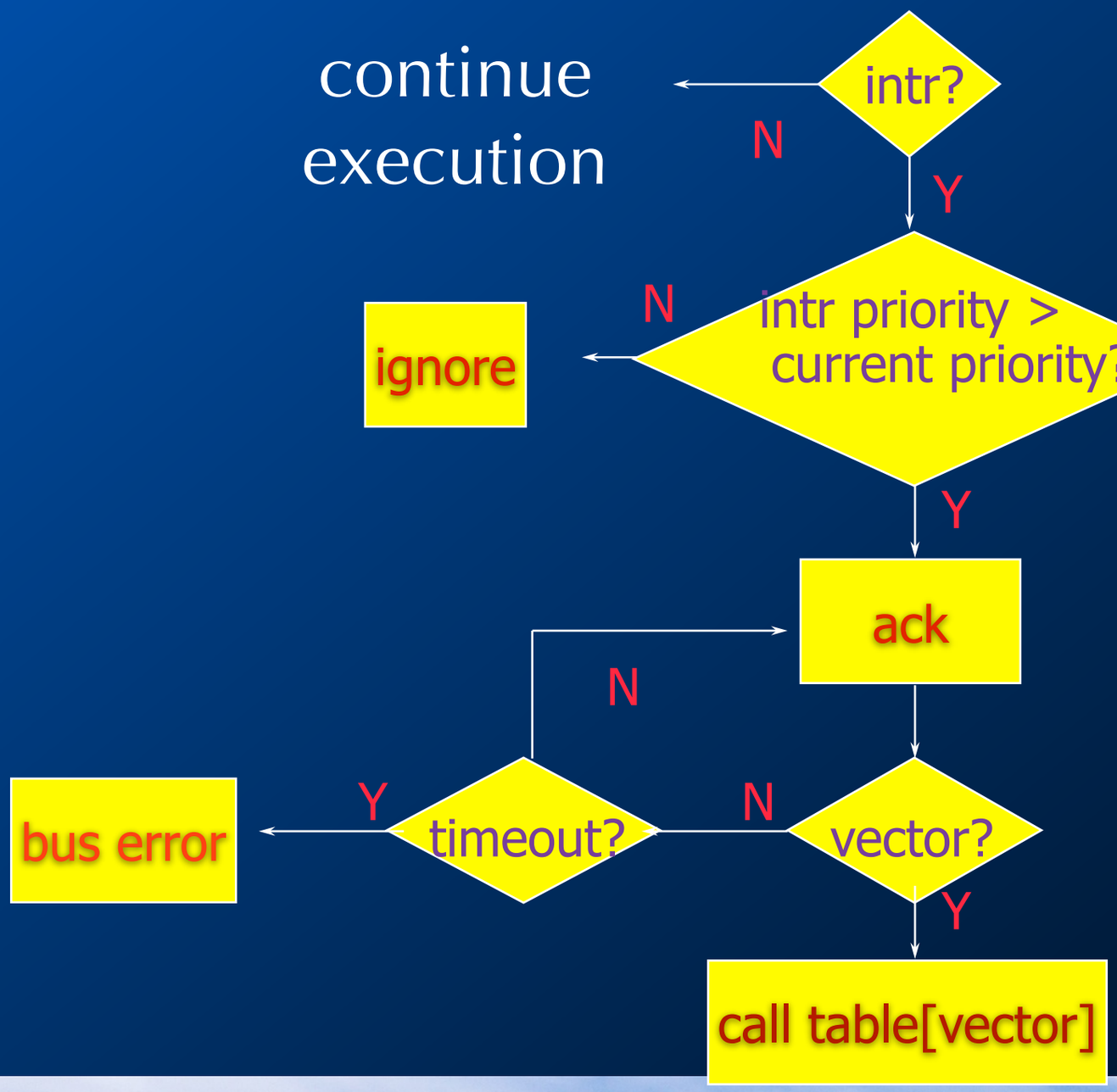
- Allow different devices to be handled by different code.
- Interrupt vector table:



Interrupt vector acquisition



Generic interrupt mechanism



Assume priority selection is handled before this point.

Interrupt sequence

- CPU acknowledges request.
- Device sends vector.
- CPU calls handler.
- Software processes request.
- CPU restores state to foreground program.

Sources of interrupt overhead

- Handler execution time.
- Interrupt mechanism overhead.
- Register save/restore.
- Pipeline-related penalties.
- Cache-related penalties.

Interrupt design guidelines

- While crummy code is just hard to debug, crummy ISRs are virtually undebuggable.
- how?
  - First, don't even consider writing a line of code for your new embedded system until you lay out an interrupt map. List each interrupt, and give an English description of what the routine should do.
  - The map is a budget. It gives you an assessment of where interrupting time will be spent.
  - Approximate the complexity of each ISR.
  - The cardinal rule of ISRs is to keep the handlers short.
  - Short, of course, is measured in time, not in code size. Avoid loops. Avoid long complex instructions (repeating moves, hideous math, and the like).
  - Re-enable interrupts as soon as practical in the ISR. Do the hardware-critical and non-reentrant things up front, then execute the interrupt enable instruction. Give other ISRs a fighting chance to do their thing.

- Figure An interrupt map
-

	Latency	Max-time	Freq	Description
INT1		1000 µsec	1000 µsec	timer
INT2		100 µsec	100 µsec	send data
INT3		250 µsec	250 µsec	Serial data in
INT4		15 µsec	100 µsec	write tape
NMI	200 µsec	500 µs	once!	System crash

- Fill all of your unused interrupt vectors with a pointer to a null routine.
-

```
Vect_table:
            dl    start_up            ; power up vector
            dl    null_isr            ; unused vector
            dl    null_isr            ; unused vector
            dl    timer_isr           ; main tick timer ISR
            dl    serial_in_isr       ; serial receive ISR
            dl    serial_out_isr      ; serial transmit ISR
            dl    null_isr            ; unused vector
            dl    null_isr            ; unused vector

null_isr:                             ; spurious intr routine
            jmp   null_isr            ; set BP here!
```

C or assembly?

- If the routine will be in assembly language, convert the time to a rough number of instructions. If an average instruction takes x microseconds (depending on clock rate, wait states, and the like), then it's easy to get this critical estimate of the code's allowable complexity.
- C is more problematic. In fact, there's no way to scientifically write an interrupt handler in C! You have no idea how long a line of C will take. You can't even develop an estimate as each line's time varies wildly.
  - A string compare may result in a run-time library call with totally unpredictable results.
  - A FOR loop may require a few simple integer comparisons or a vast amount of processing overhead.
- You may find one compiler pitifully slow at interrupt handling. Either try another or switch to assembly.

Debugging interrupt code

- What if you forget to change registers?
  - Foreground program can exhibit mysterious bugs.
  - Bugs will be hard to repeat---depend on interrupt timing.

Debugging INT/INTA Cycles

- The device hardware generates the interrupt pulse.
- The interrupt controller (if any) prioritizes multiple simultaneous requests and issues a single interrupt to the processor.
- The CPU responds with an interrupt acknowledge cycle.
- The controller drops an interrupt vector on the data bus.
- The CPU reads the vector and computes the address of the user-stored vector in memory. It then fetches this value.
- The CPU pushes the current context, disables interrupts, and jumps to the ISR.

Finding Missing Interrupts

- You can build a little circuit using a single up/down counter that counts every interrupt, and that decrements the count on each interrupt acknowledge. If the counter always shows a value of zero or one, everything is fine.
- One design rule of thumb will help minimize missing interrupts: re-enable interrupts in ISRs at the earliest safe spot.

Avoid NMI

- Reserve NMI—the non-maskable interrupt—for a catastrophe like the apocalypse. Power-fail, system shutdown, and imminent disaster all good things to monitor with NMI. Timer or UART interrupts are not.
- NMI will break even well-coded interrupt handlers, since most ISRs are non-reentrant during the first few lines of code where the hardware is serviced. NMI will thwart your stack management efforts as well.
- NMI mixes poorly with most tools. Debugging any ISR—NMI or otherwise—is exasperating at best. Few tools do well with single stepping and setting breakpoints inside of the ISR.

Breakpoint Problems

- Though breakpoints are truly wonderful debugging aids, they are like Heisenberg's uncertainty principle: the act of looking at the system changes it. You can cheat Heisenberg —at least in debugging embedded code!—by using real-time trace, a feature available on all emulators and some smart logic analyzers.

Easy ISR Debugging

- What's the fastest way to debug an ISR?
  - Don't.
  - If your ISR is only 10 or 20 lines of code, debug by inspection. Don't fire up all kinds of complex and unpredictable tools.
  - Keep the handler simple and short.

Reentrancy

- In the embedded world a routine must satisfy the following conditions to be reentrant:
  - It uses all shared variables in an atomic way, unless each is allocated to a specific instance of the function.
  - It does not call non-reentrant functions.
  - It does not use the hardware in a non-atomic way.

http://www.ganssle.com/articles/begincornerent.htm

Atomic Variables

- Both the first and last rules use the word "atomic," which comes from the Greek word meaning "indivisible." In the computer world "atomic" means an operation that cannot be interrupted.
  - mov ax,bx
  - temp=foobar; temp+=1; foobar=temp;
  - foobar+=1;

```
mov ax,[foobar]
inc ax
mov [foobar],ax
```

```
inc [foobar]
```

```
lock inc [foobar]
```

- Rule 2 tells us a calling function inherits the reentrancy problems of the callee.
- Rule 3 is a uniquely embedded caveat. Hardware looks a lot like a variable; if it takes more than a single I/O operation to handle a device, reentrancy problems can develop.

Keeping Code Reentrant

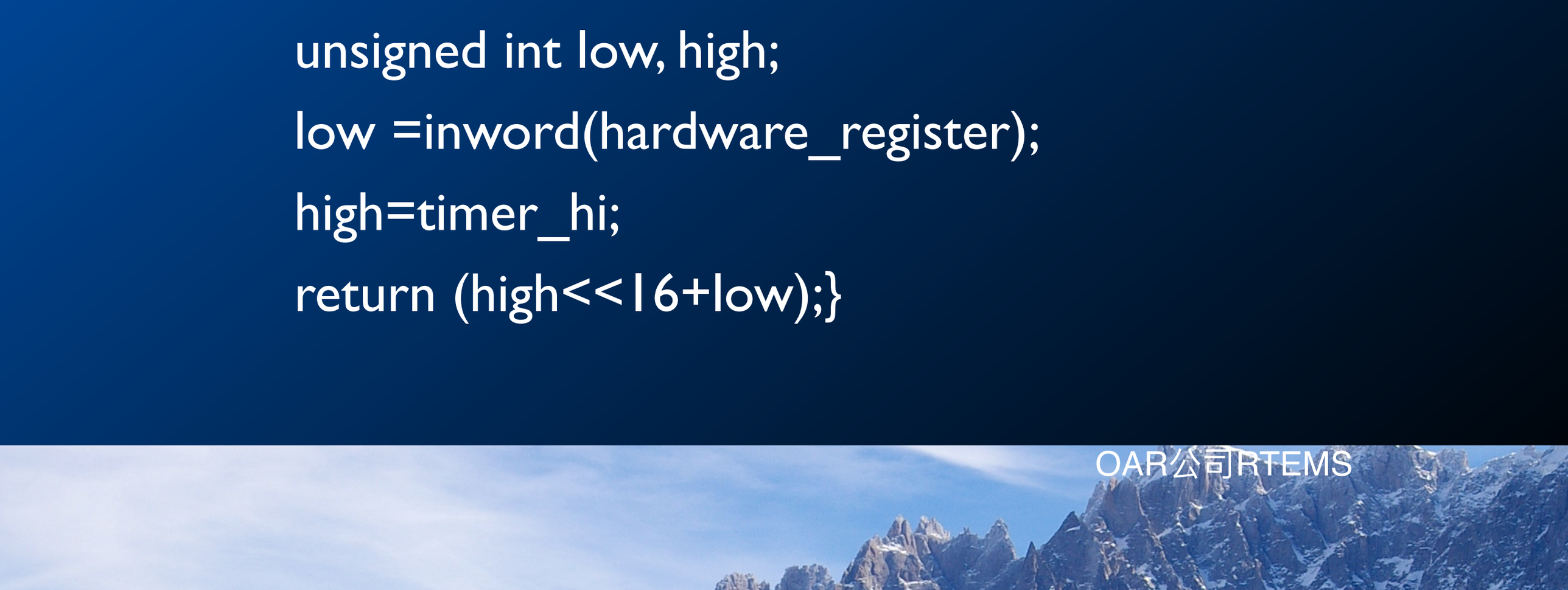
- What are our best options for eliminating non-reentrant code?
  - The first rule of thumb is to avoid shared variables. Globals are the source of no end of debugging woes and failed code. Use automatic variables or dynamically allocated memory.
  - The most common approach is to disable interrupts during non-reentrant code.
  - semaphores
-

Recursion

- A function is recursive if it calls itself.
- So all recursive functions must be reentrant ... but not all reentrant functions are recursive.

Asynchronous Hardware/Firmware

```
int timer_hi;
interrupt timer(){
        ++timer_hi;}
long timer_read(void){
unsigned int low, high;
low =inword(hardware_register);
high=timer_hi;
return (high<<16+low);}
```



Race Conditions

- One of timer_read's race conditions might be:
  - It reads the hardware and gets, let's say, a value of 0Xffff.
  - Before having a chance to retrieve the high part of the time from variable timer_hi, the hardware increments again to 0x0000.
  - The overflow triggers an interrupt. The ISR runs. timer_hi is now 0
  - 0001, not 0 as it was just nanoseconds before.
  - The ISR returns; our fearless timer_read routine, with no idea an interrupt occurred, blithely concatenates the new 0
  - 0001 with the previously read timer value of 0Xffff, and returns 0X1ffff—a hugely incorrect value.

Options

- The easiest is to stop the timer before attempting to read it.
  - lose time. Turning interrupts off during this period will eliminate unwanted tasking, but increases both system latency and complexity.
- Another solution is to read the timer_hi variable, then the hardware timer, and then reread timer_hi. An interrupt occurred if both variable values aren't identical. Iterate until the two variable reads are equal.
  - The upside: correct data, interrupts stay on, and the system doesn't lose counts.
  - The downside: in a heavily loaded, multitasking environment, it's possible that the routine could loop for rather a long time before getting two identical reads. The function's execution time is non-deterministic.

- Another alternative might be to simply disable interrupts around the reads.

```
long timer_read(void){
        unsigned int low, high;
        push_interrupt_state;
       disable_interrupts;
        low=inword(Timer_register);
       high=timer_hi;
       if(inword(timer_overflow))
       {       ++high;
               low=inword(timer_register);}
       pop_interrupt_state;
       return (((ulong)high)<<16+(ulong)low);
}
```

Agenda

- Input and output
- Busses

The CPU bus

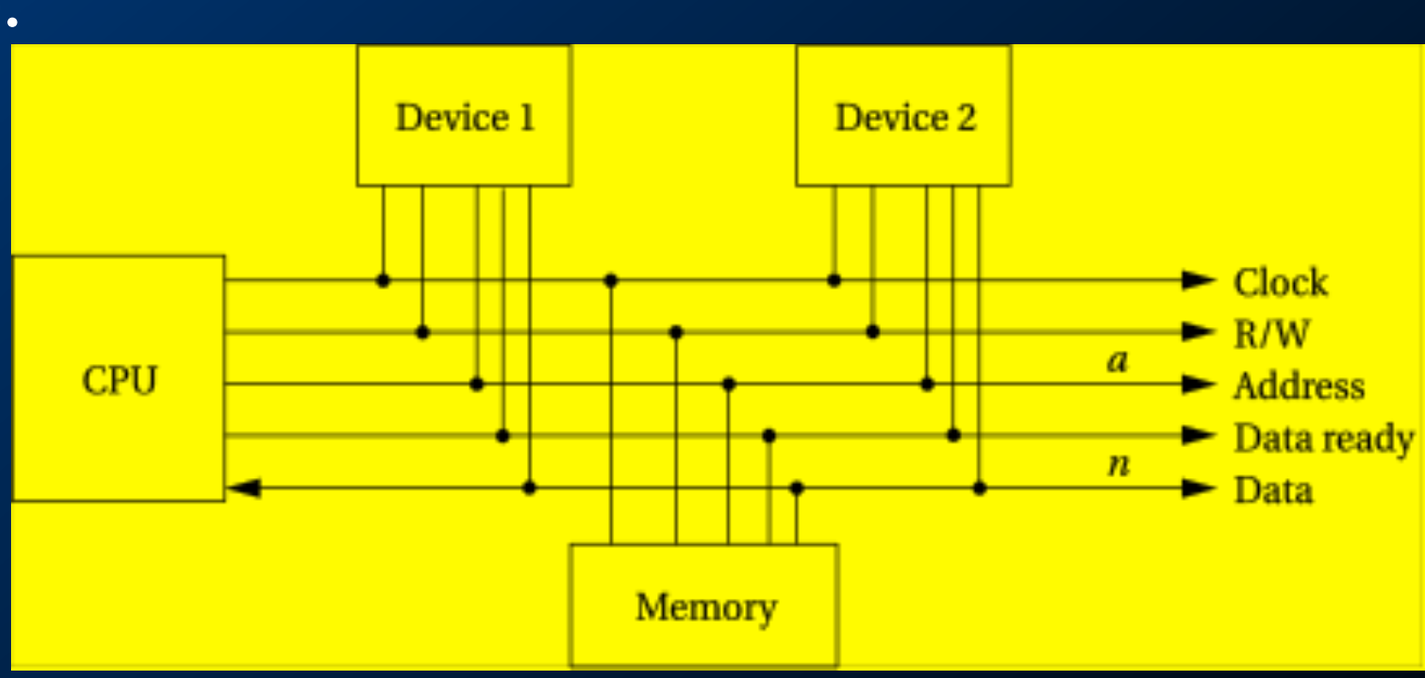
- Bus allows CPU, memory, devices to communicate.
  - Shared communication medium.
- A bus is:
  - A set of wires.
  - A communications protocol.

Bus protocols

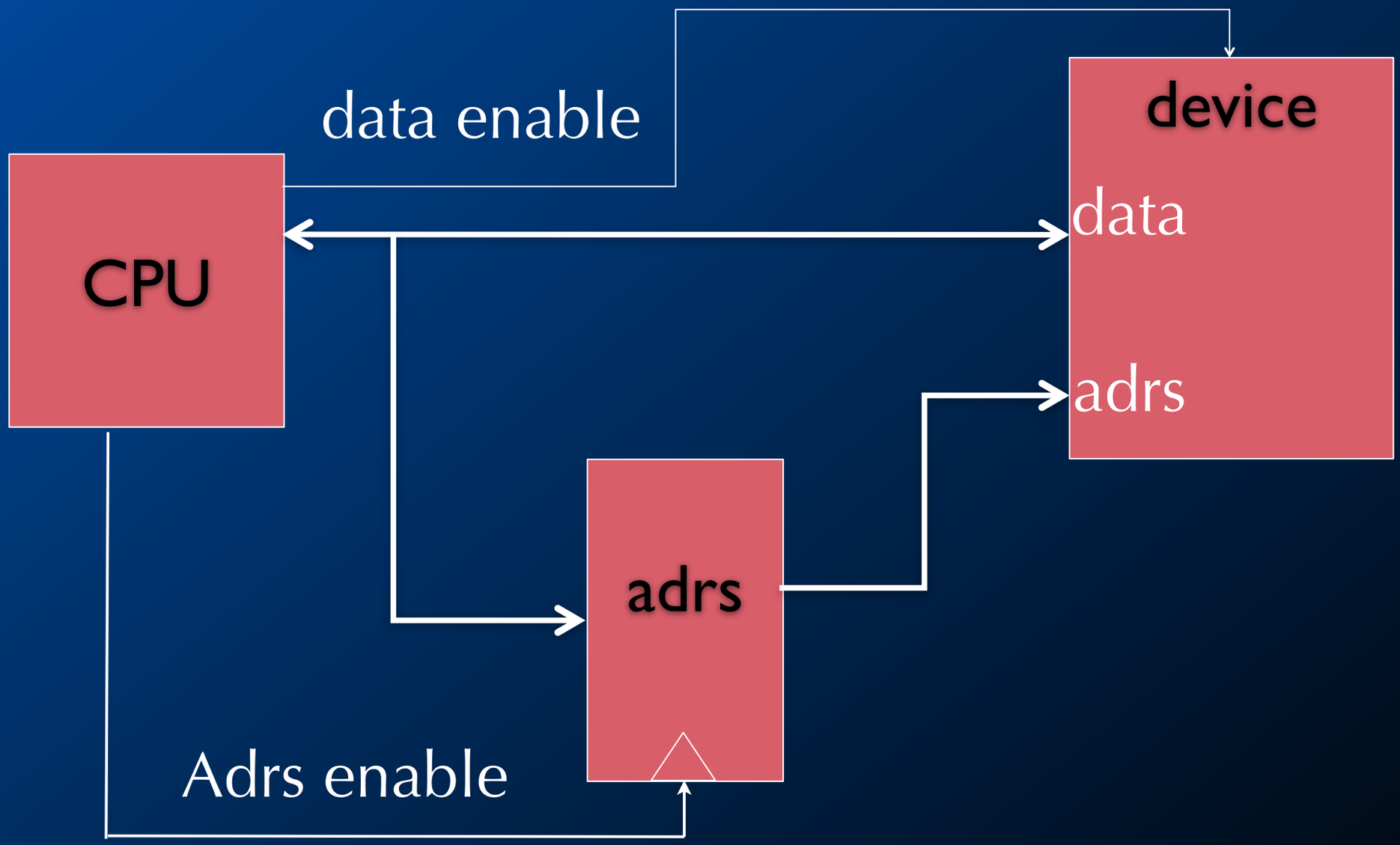
- Bus protocol determines how devices communicate.
- Devices on the bus go through sequences of states.
  - Protocols are specified by state machines, one state machine per actor in the protocol.
- May contain asynchronous logic behavior.

Microprocessor busses

- Clock provides synchronization.
- R/W is true when reading (R/W' is false when reading).
- Address is a-bit bundle of address lines.
- Data is n-bit bundle of data lines.
- Data ready signals when n-bit data is ready.

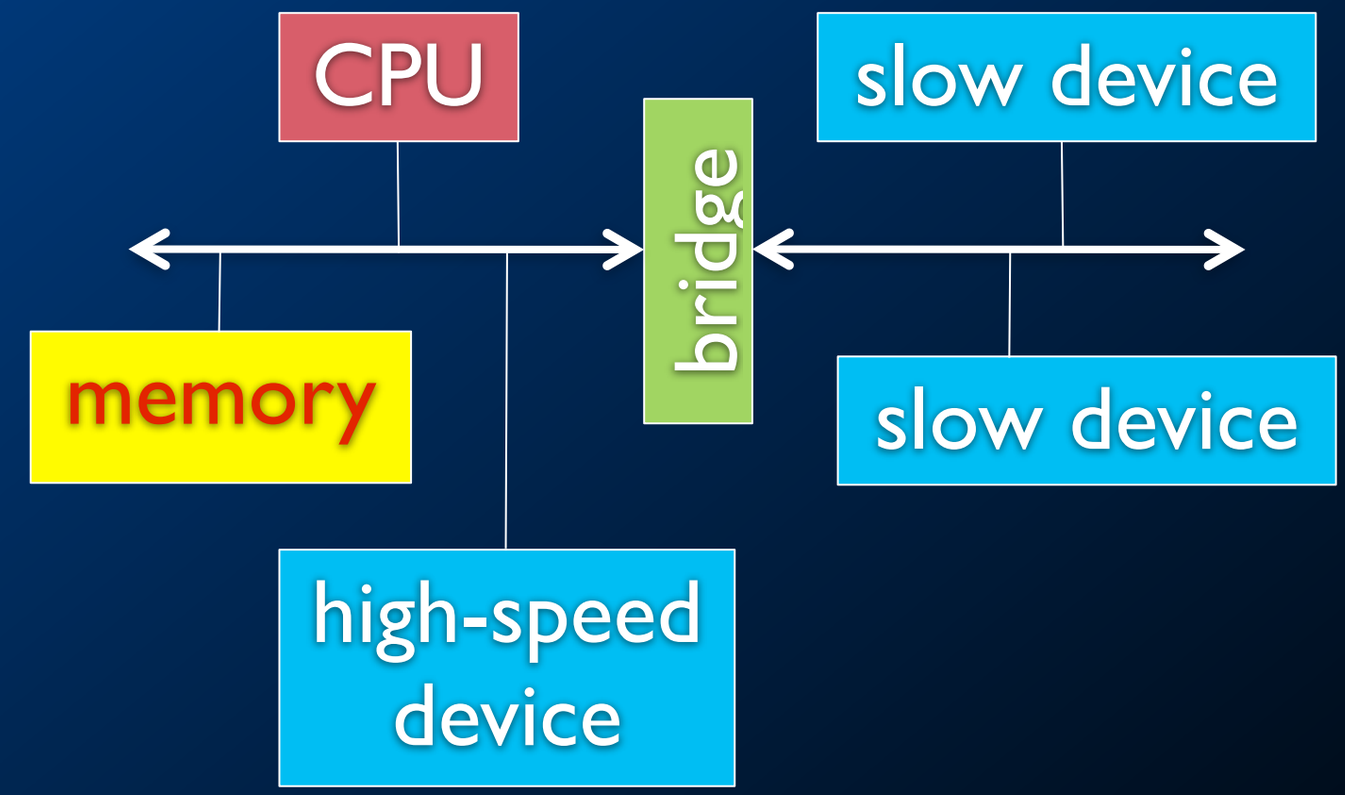


Bus multiplexing

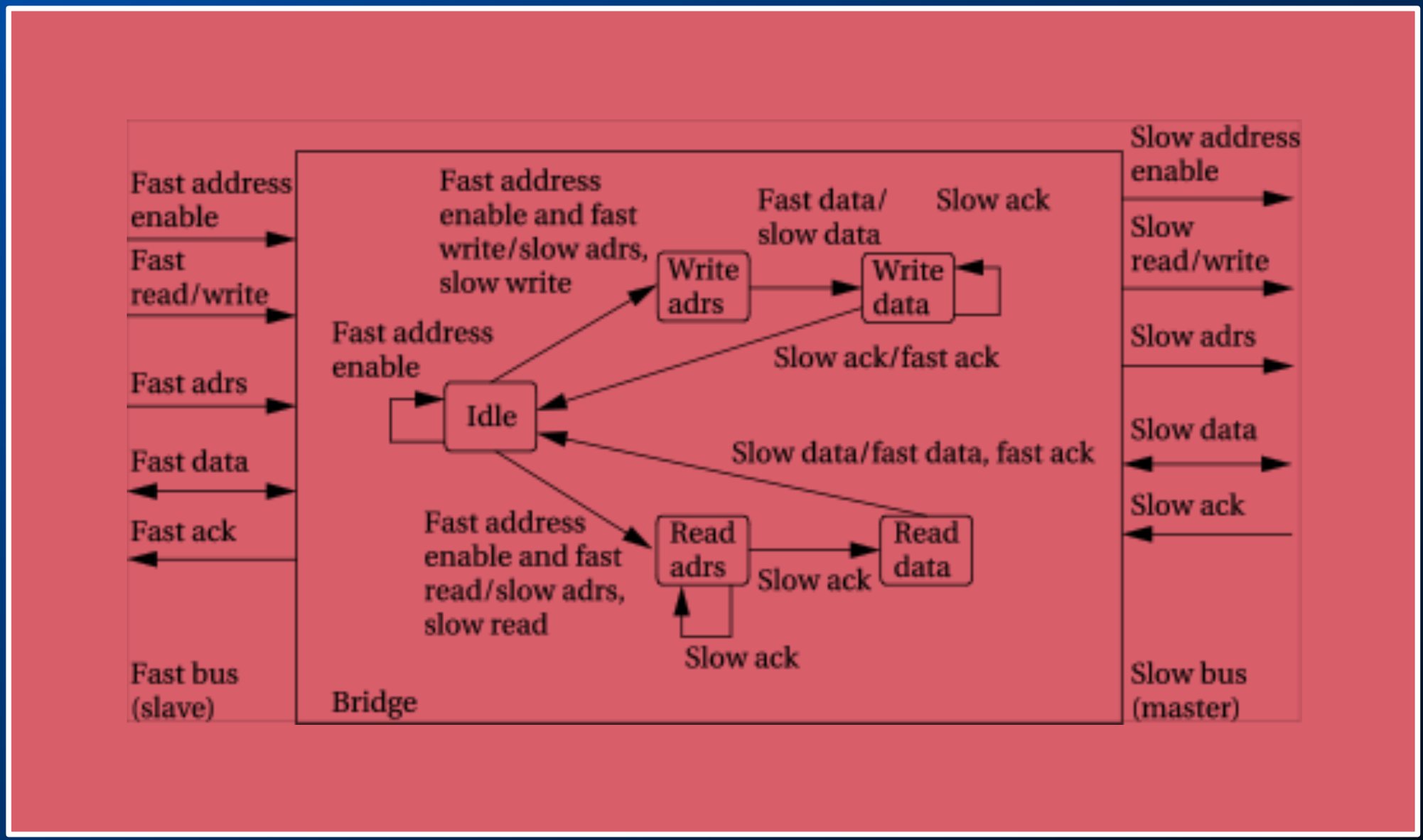


System bus configurations

- Multiple busses allow parallelism:
  - Slow devices on one bus.
  - Fast devices on separate bus.
- A bridge connects two busses.

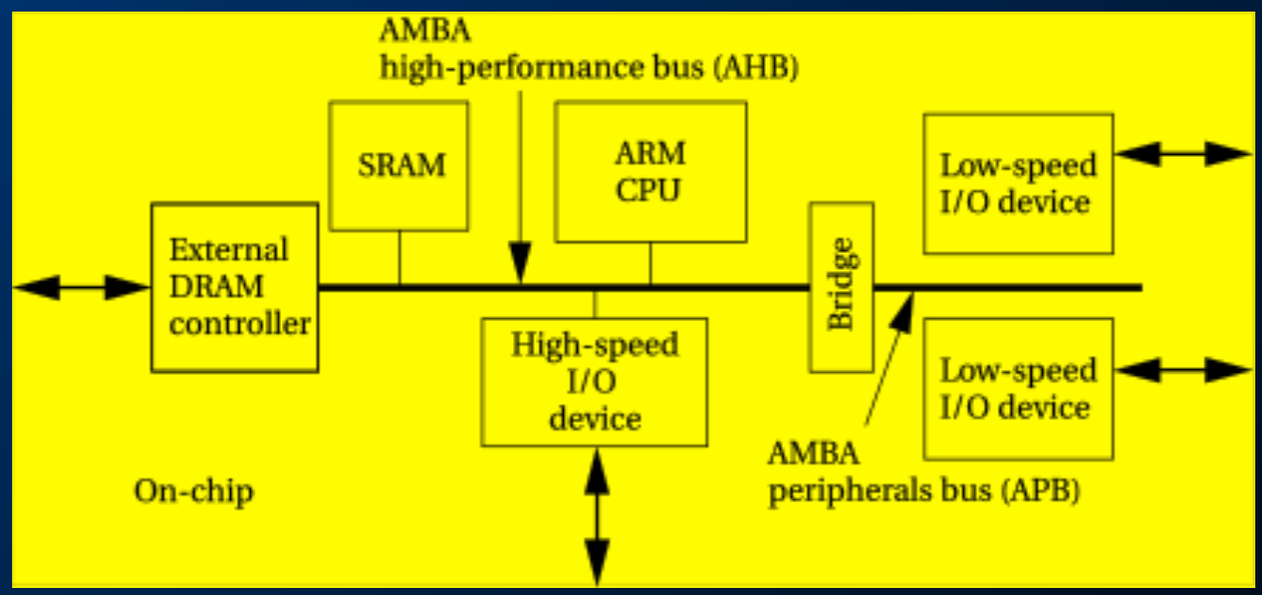


Bridge state diagram



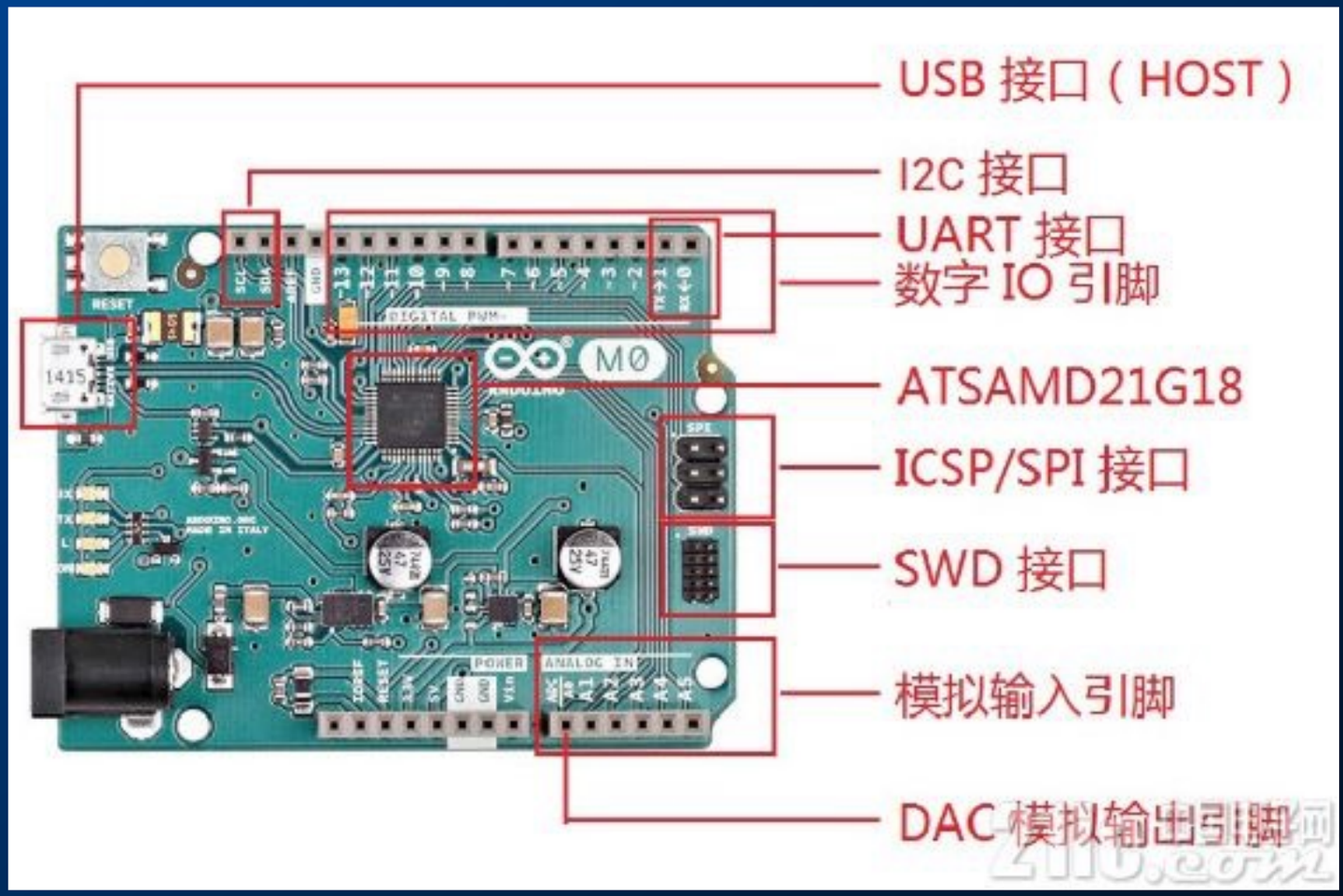
ARM AMBA bus

- Two varieties:
  - AHB is high-performance.
  - APB is lower-speed, lower cost.
- AHB supports pipelining, burst transfers, split transactions, multiple bus masters.
- All devices are slaves on APB.



UART	CAN	USB	SPI	I²C
认知度高 经济有效 简单	安全 快速	快速 即插即用硬件 简单 低成本	快速 广发接受 低成本 大型系列	简单 认知度高 广泛接受 即插即用 大型系列 经济有效
功能有限 点对点	复杂 源自汽车 固件昂贵	需要强大主机 无即插即用软件，需指定驱动	无即插即用硬件 没有固定标准	有限速率

Arduino Uno R3



Raspberry Pi 3 GPIO Header

Pin#	NAME		NAME	Pin#
01	3.3v DC Power		DC Power 5v	02
03	GPIO02 (SDA1 , I²C)		DC Power 5v	04
05	GPIO03 (SCL1 , I²C)		Ground	06
07	GPIO04 (GPIO_GCLK)		(TXD0) GPIO14	08
09	Ground		(RXD0) GPIO15	10
11	GPIO17 (GPIO_GEN0)		(GPIO_GEN1) GPIO18	12
13	GPIO27 (GPIO_GEN2)		Ground	14
15	GPIO22 (GPIO_GEN3)		(GPIO_GEN4) GPIO23	16
17	3.3v DC Power		(GPIO_GEN5) GPIO24	18
19	GPIO10 (SPI_MOSI)		Ground	20
21	GPIO09 (SPI_MISO)		(GPIO_GEN6) GPIO25	22
23	GPIO11 (SPI_CLK)		(SPI_CE0_N) GPIO08	24
25	Ground		(SPI_CE1_N) GPIO07	26
27	ID_SD (I²C ID EEPROM)		(I²C ID EEPROM) ID_SC	28
29	GPIO05		Ground	30
31	GPIO06		GPIO12	32
33	GPIO13		Ground	34
35	GPIO19		GPIO16	36
37	GPIO26		GPIO20	38
39	Ground		GP[illegible]1	40

Thank you!