The Embedded Computing Platform

I/O Interfaces and Service

José Costa

Software for Embedded Systems

Departamento de Engenharia Informática (DEI)
Instituto Superior Técnico

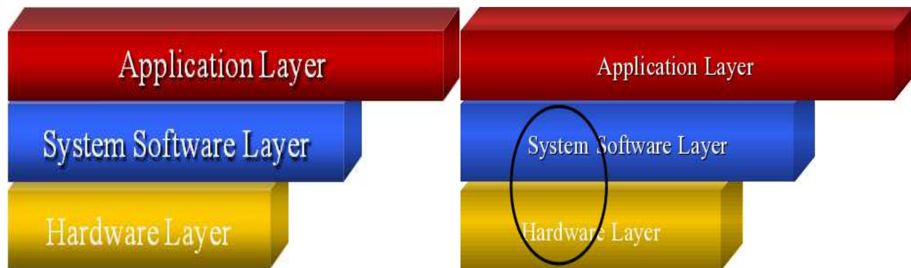
2015-09-29

Outline

TÉCNICO LISBOA

- I/O Interfaces and Service
- Serial Communication
- Programming I/O
- I/O Service
  - Busy/Wait Output
  - Interrupt I/O
  - I/O Coprocessor
- System Bus Configurations

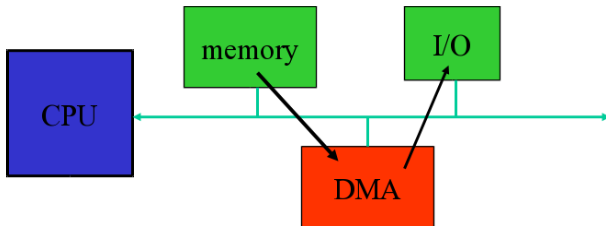
Embedded Systems Model



Input/Output (I/O) Interfaces

TÉCNICO LISBOA

- I/O is a fundamental function of the embedded systems platform
- I/O encapsulates the interaction with the real (physical) world
- I/O servicing is done by the cooperation of hardware + software



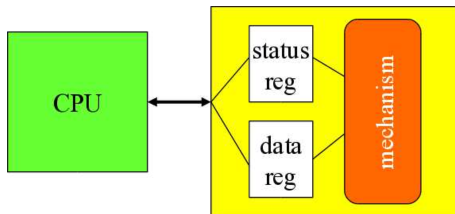
I/O Interface

TÉCNICO LISBOA

- I/O Device – Encapsulates physical and parts of the logical interface
- I/O Service – Interfaces the device with the run-time platform

I/O Devices

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



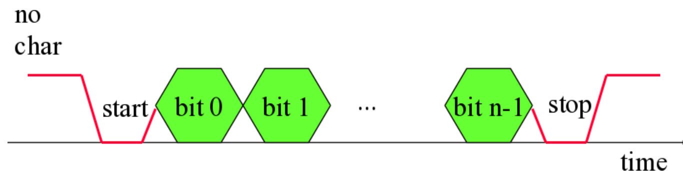
8251 UART

TÉCNICO LISBOA

- Universal asynchronous receiver transmitter (UART)
  - provides serial communication
- 8251 functions are integrated into standard PC interface chip
- Allows many communication parameters to be programmed

Serial Communication

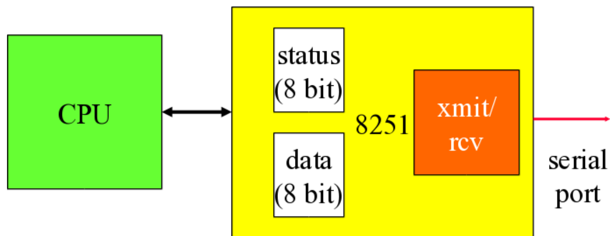
- Based on standard protocols to guarantee interoperability
- Characters are transmitted separately:



Serial Communication Parameters

- Baud (bit) rate
  - Obtained by dividing input frequency by a prescaler and a counter
- Number of bits per character
- Parity/no parity
- Even/odd parity
- Length of stop bit (1, 1.5, 2 bits)

8251 CPU Interface



Timers – Prescalers

TÉCNICO LISBOA

Assume that we have:

- An 8-bits timer and a prescaler with f, f/10, f/100 options.
- fin = 40 MHz.
- Timer drives UART which operates with transfers rates from 19200 to 1200 bps.
  1. Choose prescaler option.
  2. Choose baud rate for an application with an 850 Bps transfer rate with even parity and 2 stop bits.

Timers – Prescaler

Prescaler option

- An 8-bits timer and a prescaler with f, f/10, f/100 options.
- fin = 40 MHz.
- Timer drives UART which operates with transfers rates from 19200 to 1200 bps.

- Prescaler as fixed setup; timer as programmable setup.
- Maximum rate = 16 $\times$ minimum rate, to be programmed in the timer.

- 40 MHz / 1200 = 33333,33
- = prescaler factor $\times$ maximum timer factor ($256 = 2^8$)
- prescaler factor = 33333,33/256 = 130,2 $\Rightarrow$ choose 100, the maximum, but
- **this system can not generate 1200 bps**.

- Setup for 19200 bps
- 40 MHz / 19200 = 2083,33
- = prescaler factor (100) $\times$ timer factor = 20,8 $\Rightarrow$ 21 .

Timers – Prescaler

Choose baud rate

- An 8-bits timer and a prescaler with f, f/10, f/100 options.
- fin = 40 MHz.
- Timer drives UART which operates with transfers rates from 19200 to 1200 bps.

We want to transmit at a rate of 850 Bps assuming 2 stop bit and 1 parity bit. What is the baud rate?

- 850 Bps of data $\Rightarrow$ 850 $\times$ (
  - 1 start bit +
  - 8 data bits +
  - 1 parity bit +
  - 2 stop bits) = 850 $\times$ 12 bits = 10.200 bits.
- Choose a rate $\geq$ 10200 baud.
- Note that the data rate seen at the application level is smaller than the data rate at the communication level.

Programming I/O

TÉCNICO LISBOA

- Two types of assembly instructions can support I/O:
  - special-purpose I/O instructions
  - memory-mapped load/store instructions
- Intel x86 provides in, out instructions
- Most other CPUs use memory-mapped I/O
- I/O instructions do not preclude memory-mapped I/O

ARM Memory-Mapped I/O

TÉCNICO LISBOA

- Define location for device:

```
DEV1 EQU 0x1000
```

- Read/write code (RISC architecture):

```
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
```

I/O in High-level Languages

- Traditional HLL interfaces:

```
/* Input */
int peek(char *location) {
  return *location;
}
/* Output */
void poke(char *location, char newval) {
  (*location) = newval;
}
```

I/O Service

- Polling
  - busy-wait I/O
- Interrupt
- I/O Coprocessor
  - Direct Memory Access (DMA)
  - I/O Processor

Busy/Wait Output

TÉCNICO LISBOA

- 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++;
}
```

- Fast way to service I/O

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);
}
```

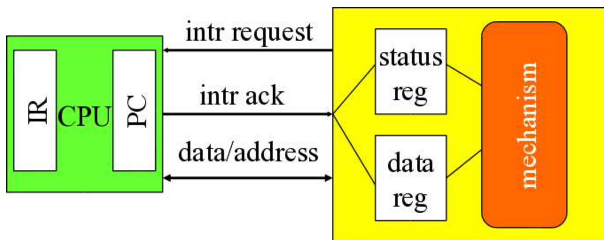
Interrupt I/O

TÉCNICO LISBOA

- 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

TÉCNICO LISBOA

- Based on subroutine call mechanism
- Interrupt forces next instruction to be a subroutine call to a predetermined location
- Return address is saved to resume executing foreground program

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

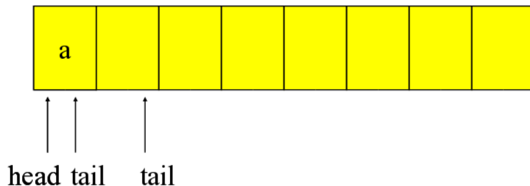
```
void input_handler() {
  achar = peek(IN_DATA);
  gotchar = TRUE;
  poke(IN_STATUS,0);
}
```

Example: Interrupt-Driven Main Program

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

Example: Interrupt I/O With Buffers

- Queue for characters:

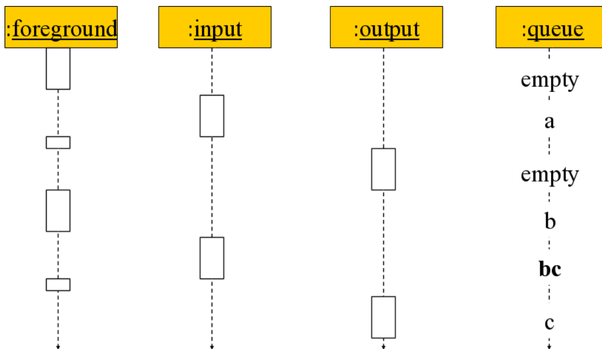


Buffer-Based Input/Output Handlers

```
void input_handler() {
  char achar;
  if (empty_buffer()) error = 1;
  else {
    achar = peek(IN_DATA);
    add_char(achar);
  }
  poke(IN_STATUS,0);
}

void output_handler() {
  if (!empty_buffer()) {
    poke(OUT_DATA,remove_char());
    poke(OUT_STATUS,1);
  }
}
```

I/O Sequence Diagram



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

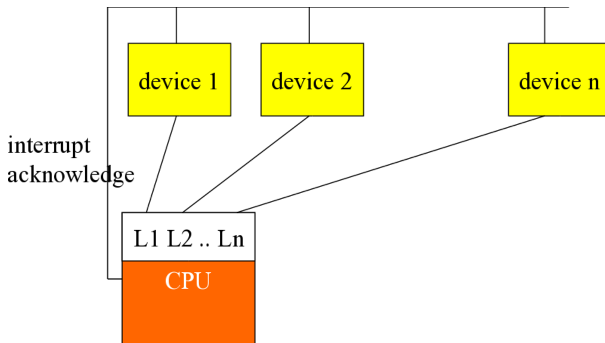
- Main program and I/O drivers are asynchronous

Priorities and Vectors

- Two mechanisms allow us to make interrupts more specific
  - priorities determine which interrupt gets CPU first
  - vectors determine what code is called for each type of interrupt

- Mechanisms are orthogonal
  - most CPUs provide both

Prioritized Interrupts

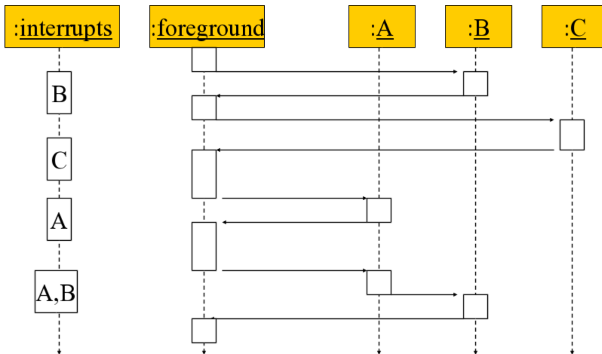


Interrupt Prioritization

- Masking: interrupt with priority lower than current priority is not recognized until pending interrupt is complete

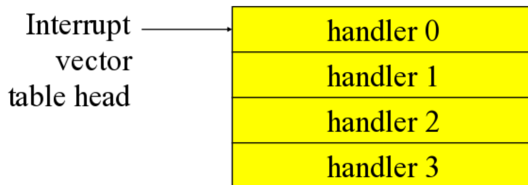
- Non-maskable interrupt (NMI): highest-priority, never masked
    - Often used for power-down

Example: Prioritized I/O

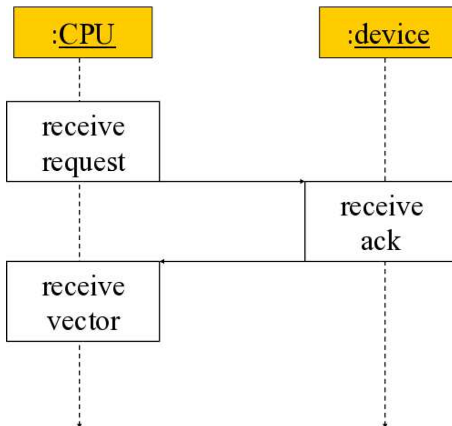


Interrupt Vectors

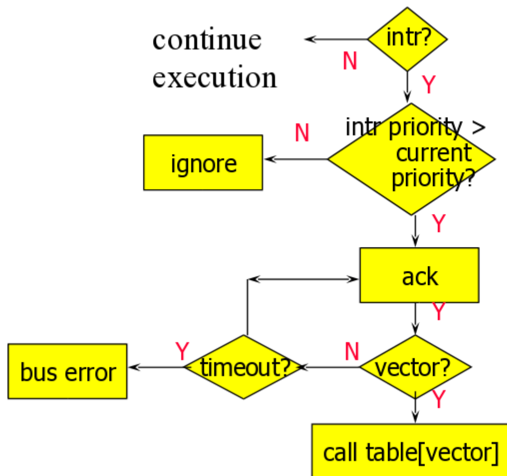
- Allow different devices to be handled by different code
- Interrupt vector table:



Interrupt Vector Acquisition



Generic Interrupt Mechanism



Assume priority selection is handled before this point.

Interrupt Sequence

TÉCNICO LISBOA

- 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
- ARM7 interrupt latency
  - Worst-case latency to respond to interrupt is 27 cycles
  - Best-case latency is 4 cycles

I/O Coprocessor

TÉCNICO LISBOA

- Applied in complex or high bandwidth interfaces
- I/O Coprocessors implement protocols and data transfer operations
- Still needs a mechanism to synchronize the coprocessor with the CPU
  - usually interrupts
- Examples
  - Direct Memory Access (DMA) controller
  - LAN controller

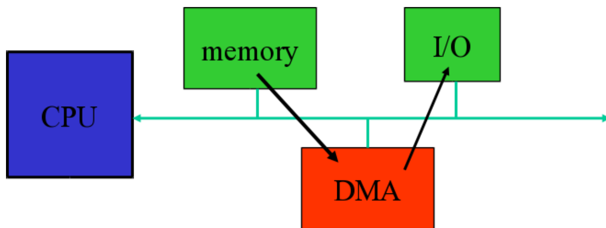
Communication Between CPU and I/O Coprocessor

- Synchronization
  - Busy/wait
  - Interrupts

- Communication
  - I/O Coprocessor registers, like in most (simple) peripheral controllers
  - Shared memory – Memory accessible by both the CPU and the controller
    - memory queues (ex. FIFO), dual-ported memories or common memory

Direct Memory Access (DMA)

- DMA provides parallelism on bus by controlling transfers without CPU



DMA Operation

TÉCNICO LISBOA

- CPU sets up DMA transfer:
  - Start address
  - Length
  - Transfer block length

  The DMA controller behaves like a “slave” peripheral

- DMA controller performs transfer, signals when done
  - Interruption

  The DMA acts as a master controller

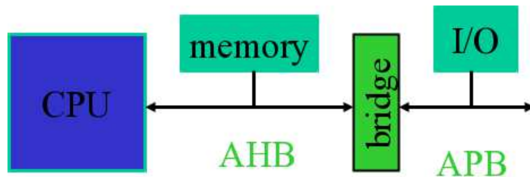
System Bus Configurations

TÉCNICO LISBOA

- Usually more than one bus
  - high-speed deviccs connected to high-performance bus
  - low-speed devices connected to different bus
- High speed buses may provide wider data connections
- High-speed bus requires more expensive circuits and connectors
- Bridge separates buses
- Allow parallelism in I/O operations

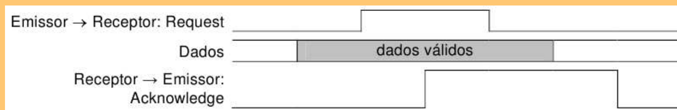
ARM Buses

- AMBA:
  - Open standard
  - Many external devices
- Two varieties:
  - AMBA High-Performance Bus (AHB)
  - AMBA Peripherals Bus (APB)



Exercise

A fleet management system consists of tracking systems installed on board vehicles and a service center where you can track the status of the fleet. On the system installed in each vehicle communication between the console and the driver onboard computer uses a parallel interface through which data and the control signals are transferred, using a request/acknowledge protocol.



Data is transferred in nibbles. The emitter activates and samples the signals in the channel with a period of 5μs. In each access to the channel a write access or a sample (read) of the port are made but never both simultaneously.

Resolution (1)

- **Calculate the maximum bandwidth of the channel assuming that the response of the receptor is instantaneous. (Ignore the overhead with the headers and terminations of the data packets.)**

- The sender performs the following steps:
  1. Places the data
  2. Activates the request
  3. Samples Acknowledge ($0 \to$ l)
  4. Withdraws request
  5. Removes the data
  6. Samples Acknowledge ($1 \to 0$)
- The total execution time of a cycle is $6 \times 5\mu s = 30\mu s$.
- The transfer of an octet takes ...
- $60\mu s$, and the transfer rate is ...
- $1/60\mu s =$ **16.7 KB/s**.

Resolution (2)

- **Optimize the protocol assuming that the response time of the receiver is negligible but not instantaneous. Calculate the maximum bandwidth of the channel.**

- What can you do to increase the bandwidth?
- Under the above conditions you can activate the request and the data simultaneously and remove the request and the data simultaneously thus reducing the cycle to 4 steps.
- The total execution time of a cycle is $4 \times 5\mu$s $= 20\mu$s.
- The transfer of an octet takes 40μs thus the rate of transfer is $1/40\mu$s $=$ **25 KB/s**.

Resolution (3)

TÉCNICO LISBOA

- **How is this handshaking protocol for data transfer compared with a synchronous protocol between systems?**

- The handshaking protocol presented seamlessly adapts to the characteristics of performance of each system
- However, if conditions permit, the synchronous protocol can achieve higher transfer rates.

Resolution (4)

- **The sampling rate is marked by an interruption, and it is inside the service routine that are made the direct accesses to the communication channel. (The channel is controlled directly by the processor.) Considering the execution times shown in the table, calculate the occupancy rate of the processor in a prolonged period of data transfer.**

	Time (μs)
Change of context on entering the interrupt routine	1.5
Change of context when leaving the interrupt routine	1.1
Channel service	1.0

- Processor occupation = (1.5 + 1.0 + 1.1) / 5 = **72%**.

Resolution (5)

- **Estimate the maximum transfer rates attainable with the service channel by interruptions or by direct sampling (polling) of the processor.**

- In both cases, the full rate is achieved with the processor occupied 100%.
- With interruptions each step takes: $(1.5 + 1.0 + 1.1) = 3.6\ \mu s$.
- With sampling each step takes about $1.0\ \mu s$. (There is no change of context, although the duty cycle of the channel is slightly more complex).
- Using the initial conditions we would have:

	Transfer cycle (nibble)	Transfer rate
Interrupts	$6 \times 3.6\ \mu s = 21{,}6 \mu s$	**23.1 KB/s**
Polling	$6 \times 1.0\ \mu s = 6.0 \mu s$	**83.3 KB/s**

Outline

TÉCNICO LISBOA

- I/O Interfaces and Service
- Serial Communication
- Programming I/O
- I/O Service
  - Busy/Wait Output
  - Interrupt I/O
  - I/O Coprocessor
- System Bus Configurations

References

Computers as Components: Principles of Embedded Computing System Design , Marilyn Wolf. Morgan Kaufman, Chs. 3.2, 4.2.2, 4.2.3, 4.2.4, 4.2.5

Next class

TÉCNICO LISBOA

- Software architectures