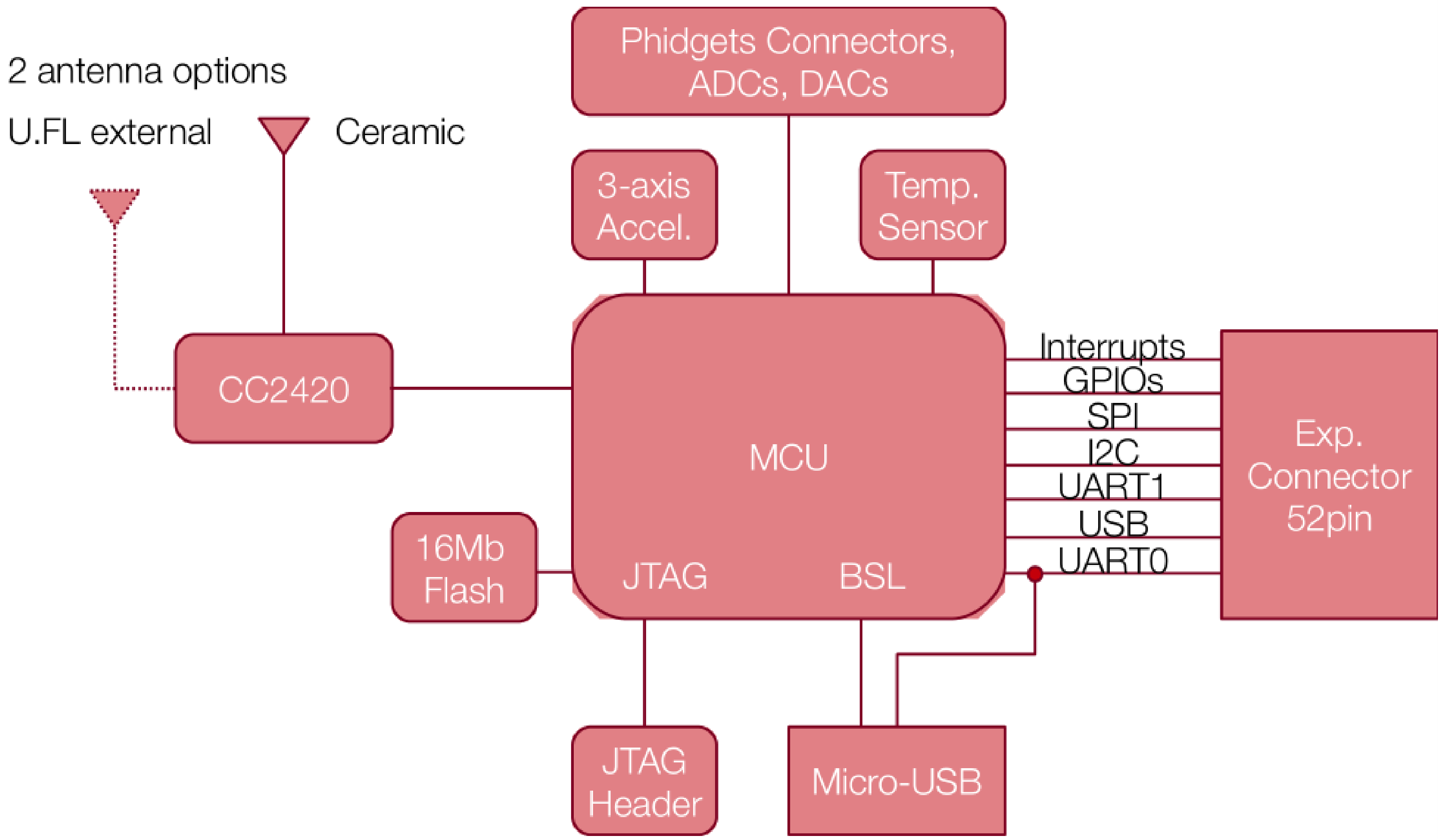
Operating Systems for
Small Embedded Devices and WSN

Typical hardware ("mote") for WSN/IoT

Z1 (by the company Zolertia)



Interaction with the physical world

- Out
  - Voltage on pins can be controlled from running program
- In
  - Buttons: read directly state of I/O pin
  - Analog (voltage): internal A/D converter connected to pins
  - Some sensors have their own microcontroller and can communicate through various protocols (I^2C,...)
- Two approaches to handle input: ***Polling*** vs ***Interrupts***

Polling

- Read status of input pin every few milliseconds.
- (Fictive) example:

```
while(true) {
    // normal program execution
    ...
    // check status of I/O pin
    bool res = readIOStatus(PIN_1);
    if(res) {
        // button was pressed. do something.
        ...
    }
}
```

- Easy to implement. However:
  - If not done frequently enough, there is a risk to miss a change of the pin state. Example: button quickly pressed and released
  - If done too frequently, waste of CPU time and energy!

Interrupts

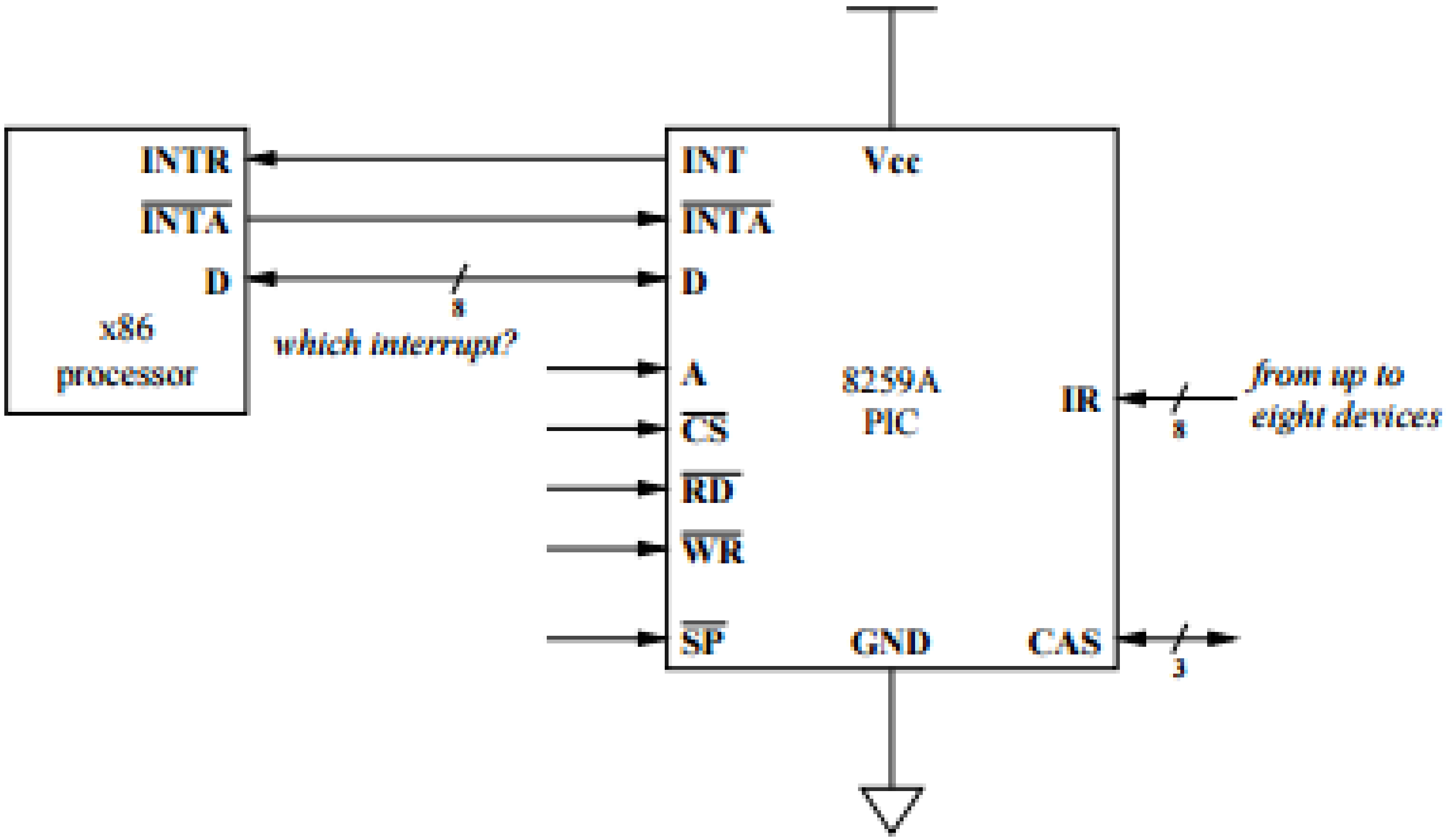
- Interrupt = signal to the processor that current code execution should be interrupted because “something important” has happened
- Typically handled by OS and device drivers, not by the user application
- Interrupts exist in all CPUs (not just embedded systems)

- Two types of interrupts:
  Hardware interrupts vs. ***Software interrupts***

Hardware interrupts

- triggered by external hardware, for example
  - keyboards ("a key has been pressed")
  - disks ("the data you requested is ready")
  - clocks ("you wanted me to notify you at 14:00")
  - ...
- Each hardware interrupt requires a pin at the CPU
- To reduce the number of pins used for interrupt handling, modern CPUs have a **Programmable Interrupt Controller (PIC)**
- PIC collects interrupt requests (IRQ) from external sources, sorts them by priority, and passes them to the CPU one by one → only a few pins needed
- Priorities can be modified by the CPU, hence the name "Programmable"

Example: 8259A PIC



Source: http://mohsaad.com/2017/08/16/PIC/

Interrupt handling

- Most CPUs have special registers or locations in memory that store the addresses of the interrupt handling functions: The **Interrupt Vector Table**

- When an interrupt happens, the CPU...
  1. stops the current program execution
  2. saves the address of the next instruction to execute at a special place (or the stack)
  3. (optional: save program state, register values, etc. Expensive!)
  4. jumps to the address of the interrupt handler
  5. after the interrupt handler is finished, program execution is resumed at the saved address

OS for embedded systems

- Do we need an OS?
- Actually, no!
  - That’s how people developed software for embedded devices for many years
- But not very nice: you have to do everything by yourself
  - Handle interrupts
  - Manage memory
  - Write your own process scheduler if you need multi-processing

Linux for embedded systems?

- Possible, provided you have enough memory and a memory management unit (for memory protection)
- But difficult if you have real-time requirements, i.e., when you need guaranteed upper bounds to reaction times to events or exact timing (motor control etc.)
  - In Linux, kernel code (system calls, interrupts, device drivers) cannot be interrupted by user-mode code. If something important happens (button pressed), the user application cannot react immediately
    $\rightarrow$ in the worst case, latency $>$ hundreds of milliseconds
- There is a RT extension for Linux:
  https://rt.wiki.kernel.org

Real-Time OS

- There are OS specifically designed for real-time applications
  - Typically, much smaller than desktop/server OS
  - Reduced set of features (may not support virtual memory, etc.)
  - Fast context switch
  - Guaranteed time-bounded response to interrupts
  - Tries to avoid code that cannot be preempted

Some real-time OS for embedded systems

- pOSEK
  - Commercial real-time OS for embedded devices, presented first for 32-bit PowerPC CPUs in 1998
  - 2 KB size, rather static, designed for motor control etc.
- VxWorks
  - Commercial real-time OS for x86, MIPS, PowerPC and ARM architectures
  - Big and complete: ≥100 KB size, supports multi-core CPUs, memory protection, POSIX threads, etc.
  - Made for fast CPUs: a context switch takes 2400 cycles
- Similarly powerful: pSOSystem
- Not really useful for small low-power devices!

OS for small embedded devices

- OS especially designed for WSN/IoT
  - Provide thin abstraction layer to hardware (I/O, timers, etc.)
  - Very limited multiprocessing (preemptive priority scheduling, no time-slice scheduler)
  - Specific network protocol implementations
  - Small CPU and memory footprint
  - Power saving
- Often, the OS is just “a set of libraries”
  - Statically linked to your program. Result is uploaded to device.
  - Efficient! Allows compiler to make global optimizations and remove unused functions.
- Several free and open-source OS available for various hardware platforms: TinyOS, Contiki, RIOT

TinyOS

- Yet another UC Berkeley output… (started in 1999)
- Modular, OS only needs 400 bytes!
- Special programming language (C without dynamic memory allocation and function pointers) to simplify program optimization by the compiler
- Preemptive priority scheduler
  - Low priority: background tasks, tasks for data processing, etc.
  - High priority: handling of events (interrupts)
- No time-slice scheduler: tasks wait in a FIFO queue for execution
- The CPU sleeps when there is no event and no task waiting in the queue

TinyOS' Execution Model

- Applications are event-driven:
  - When an event has been triggered the OS calls the application function that is responsible for handling the event
    - Event = timer, a pressed button, incoming network packet,...
  - The event handling function can create new tasks for data processing etc.

```
implementation {
        event void Boot.booted() {
                call MyTimer.startPeriodic(500);
        }
        event void MyTimer.fired() {
                call Leds.led0Toggle();
                post myLongTask();
        }
        task void myLongTask() {
     ...
    }
}
```

Contiki

- Swedish Institute of Computer Science (SICS), 2004
- Differences to TinyOS:
  - No special programming language. Apps are written in C
  - Cooperative processes instead of event/task model
  - Core (kernel+important libraries) is uploaded once
  - Applications can be uploaded/updated at run-time

Hello World

Applications consist of one or multiple processes

```
// declares a process that starts when device powers up:
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);

PROCESS_THREAD(hello_world_process, ev, data)
{
  PROCESS_BEGIN();
  printf("Hello, world\n");
  PROCESS_END();
}
```

Process Scheduling

- Processes are *cooperative*
- Other processes can only run if current process stops

```
PROCESS_THREAD(hello_world_process, ev, data)
{
  PROCESS_BEGIN();
  …
  // process stops and waits for event.
  // other processes can run.
  PROCESS_WAIT_EVENT_UNTIL((ev==sensors_event) && (data==&button_sensor));
  // when an event has happend execution continues here
  …
  PROCESS_END();
}
```

Processes

- Processes communicate with each other by posting events
  - Synchronous events: delivered immediately
  - Asynchronous events: delivered later (queued)

- Processes are implemented as *Proto-Threads*
  - Like threads, but much cheaper
  - No separate stack per process → Local variables lose their value after a context switch!

Interrupt handling by the OS

- Button connected to port 2 of MCU triggers interrupt,:

```
ISR(PORT2, irq_p2) {
   // this notifies one of the processes of the OS that
   // is responsible for sensor events
   process_poll(&sensors_process);
}
```

- Sensors_process notifies waiting application processes:

```
PROCESS_THREAD(sensors_process, ev, data) {
   ...
   // wait for event from interrupt
   PROCESS_WAIT_EVENT();
   // notify application processes
   process_post(PROCESS_BROADCAST, sensors_event, ...);
   ...
}
```

RIOT

- Motivation/claims of the authors:
  - TinyOS and Contiki were designed for WSN: event-driven and, therefore, efficient on resource-restricted devices
  - IoT devices need more powerful networking applications. Multi-threading (instead of event-driven) makes implementation of server/client applications easier.
- RIOT applications can be programmed in C or C++
- Again, preemptive priority scheduling:
  - Every task gets a priority
  - Tasks run until completion, but can be preempted by tasks with higher priority
  - Tasks with same priority must cooperate, no time-slice scheduler

RIOT example

```
#include <stdio.h>
#include "xtimer.h"
#include "timex.h"

int main(void) {
    xtimer_ticks32_t last_wakeup = xtimer_now();

    while(1) {
        xtimer_periodic_wakeup(&last_wakeup, 1U*US_PER_SEC);
        printf("slept until %" PRIu32 "\n",
                                 xtimer_usec_from_ticks(xtimer_now()));
    }
    return 0;
}
```

Networking Stacks for small embedded devices

- uIP
  - IPv4 stack
  - 4-5 KBytes code, few hundred bytes RAM
- How is that possible?
- Design choices: Only implement what is necessary for standard compliance
  - Compatible but not necessarily good performance!

uIP Memory Management

- One single packet buffer
  - Easier to manage (compiler optimizations)
  - Less memory consumption
- Incoming packets:
  - Application has to process packet immediately (or copy it)
  - While processing, other incoming packets are (hopefully!) queued by radio interface or driver
- Outgoing packets:
  - Directly created in buffer

TCP implementation in uIP

- Retransmissions: up to the application to recreate lost packet segments
  - Not too bad. If the packet contained sensor data, we can just send new data.
- Only one single TCP segment “on the air”, i.e., next packet has to wait until ACK is received
  - Saves memory
  - Simplifies implementation: no congestion control, no sliding window
  - Affects performance, of course

IP simplifications

- One single buffer for IP fragment reassembly
  - Only one reassembling at the same time
  - Fragment reassembling is rare, anyway

Programming with uIP in Contiki

- Event-based API as everything else in Contiki

```
while(1) {
  PROCESS_YIELD();
  if(ev == tcpip_event) {
    // program has to check now what the reason
    // for the event was.
    // incoming packet, incoming connection,
    // retransmission, etc.
    if(uip_newdata()) {...}
  }
}
```

Other networking stacks

- uIP
  - IPv4 stack
  - 4-5 KBytes code, few hundred bytes RAM
- lwIP: better performance than uIP but needs more resources
  - 40kB RAM, 20kB code
- uIPv6
  - IPv6 stack
  - 16kB code, 2kB RAM (1300 bytes packet buffer)

- And many more free and commercial ones: uC/IP, CMX, NetX, NicheStack, ARC RTCS, RTXC Quadnet, TargetTCP,…