Lab 3: Sensing

victor.cionca@cit.ie

How does sensing work?

Sensor

analog

digital

Microcontroller and out

Physical process generates variation in

- Voltage
- Current
- Resistance

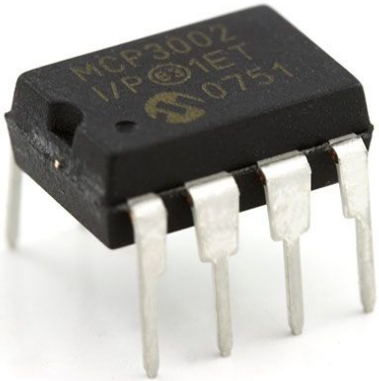
Process digital values, send over radio, output over serial or other interfaces

How does sensing work?

Sensor

analog

Analog to Digital converter (ADC)



digital

Microcontroller and out

Physical process generates variation in

- Voltage
- Current
- Resistance

Convert analog inputs to digital values

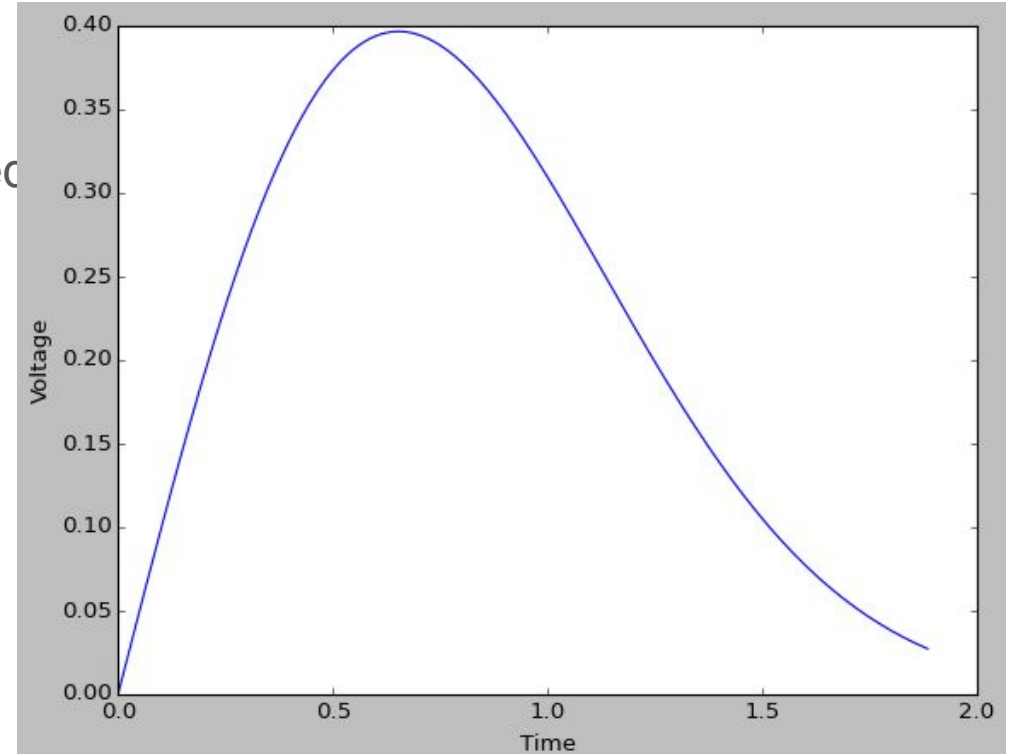
Process digital values, send over radio, output over serial or other interfaces

Usually part of the same package

A bit on ADCs

Sampling rate

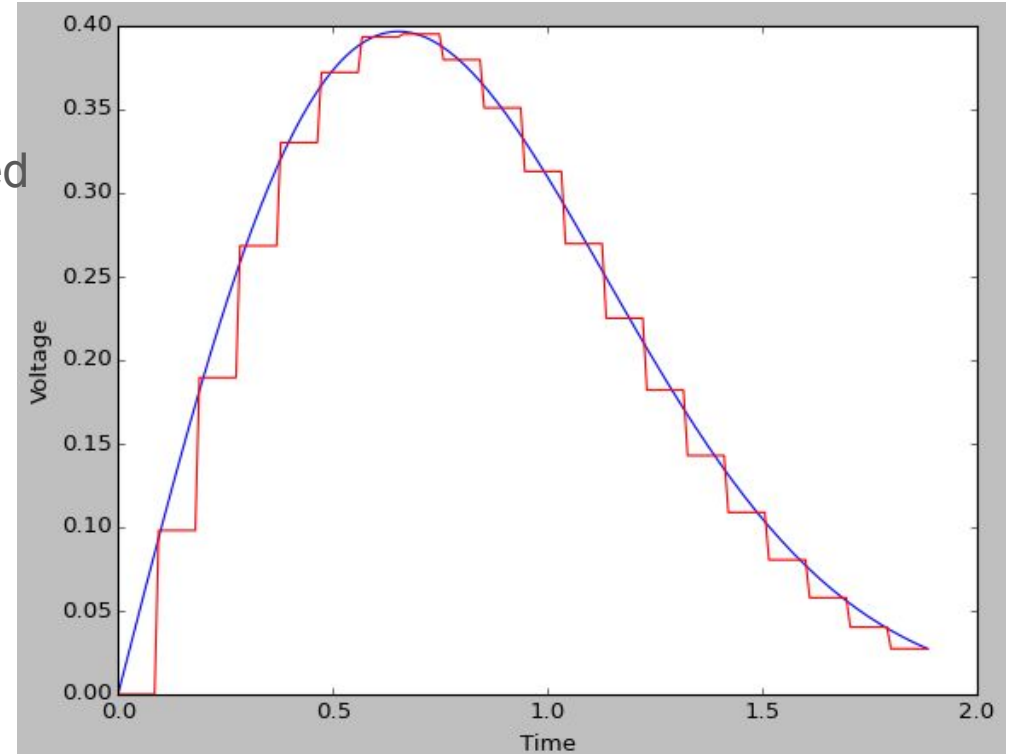
- How much information is captured



A bit on ADCs

Sampling rate

- How much information is captured
- Nyquist's theorem
  - Sample at 2x desired frequency



A bit on ADCs

Resolution

- Number of bits used for output
- Max input - reference voltage

Input max ----------> 11...11 (Output max)

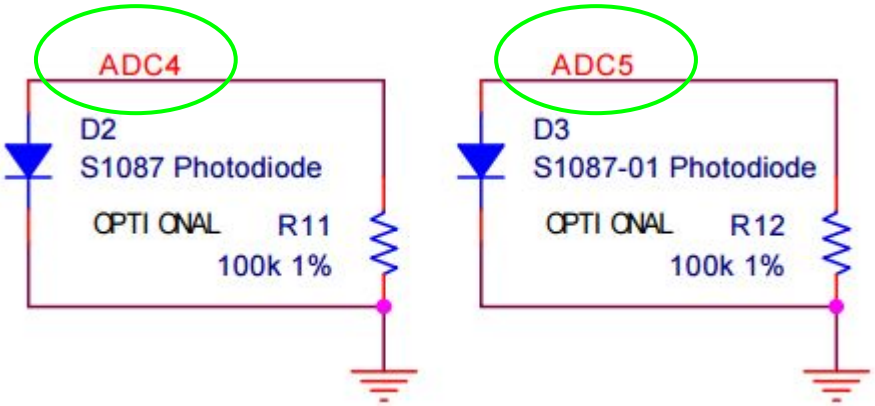
Input crt ----------> Output crt

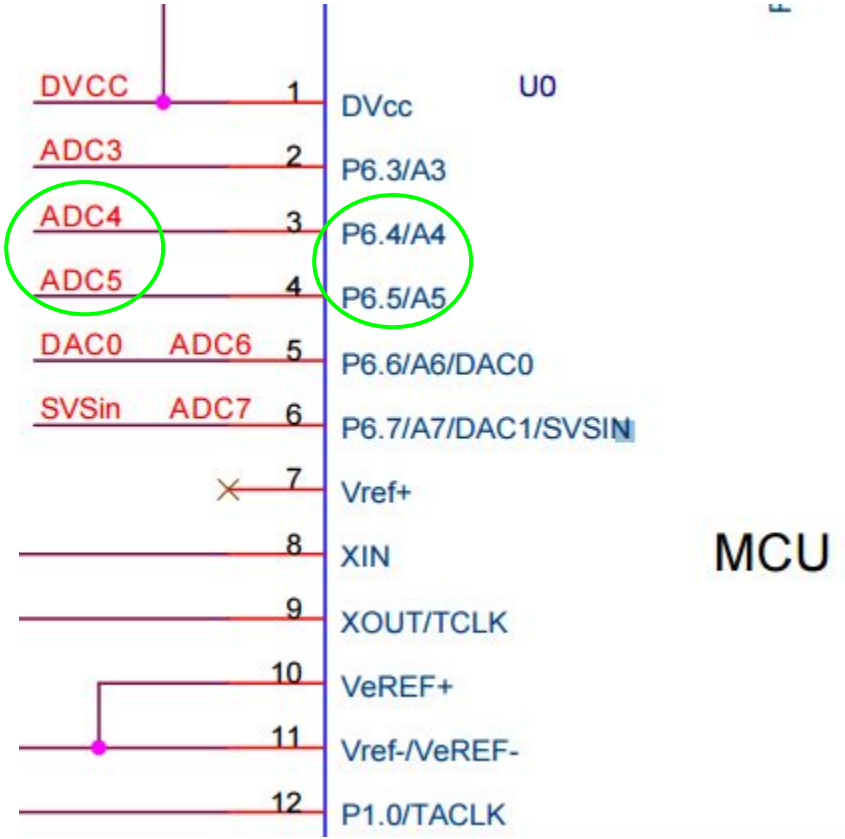
So: $\frac{In_{crt}}{In_{max}} = \frac{Out_{crt}}{Out_{max}}$, resolution is $\frac{Out_{max}}{In_{max}}$

So for every unit variation in input we have out_max/in_max change in output.

A bit on ADCs

- ADCs also have multiple input channels
- Can read things in parallel
- Must know to which channel the sensor is connected
  - Based on hardware design



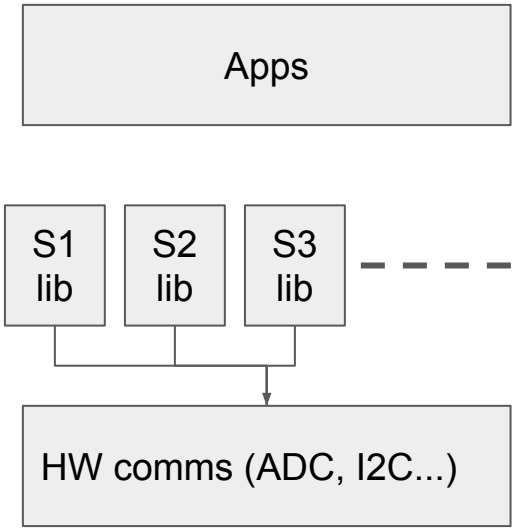


Sensing support from the OS

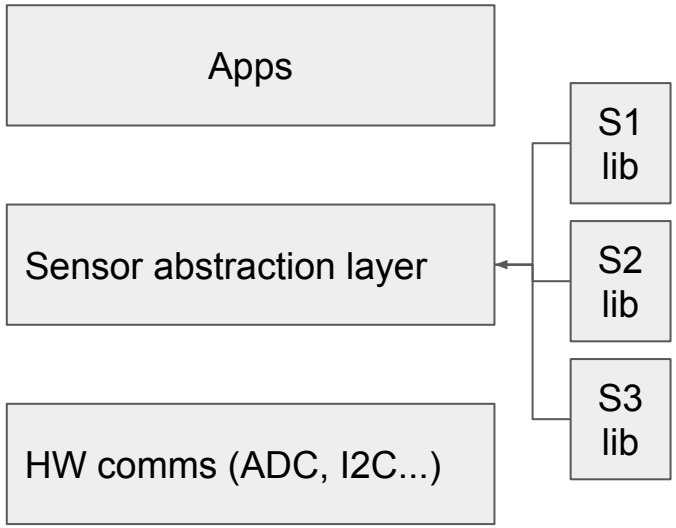
- Driver for the ADC
- Driver for the sensor
  - Which ADC channels it’s using
  - Access and timing information
  - Registers/commands to switch on/off state machines, etc
- Please note - there are sensors that don’t need ADC
  - Communicate over serial, I2C, SPI, etc

Sensing support from the OS

Option 1



Option 2



Contiki sensors

Contiki sensing infrastructure

Contiki sensing infrastructure

- core/lib/sensors.[ch]
- Sensors for tmote:
  - Light sensor: platform/sky/dev/light-sensor.[ch]
  - Temp-hum sensor: dev/sht11/
- Macros:
  - SENSORS_ACTIVATE(<sensor>)
  - SENSORS_DEACTIVATE(<sensor>)
- Functions:
  - <sensor>.value(<sub-sensor>)

Problem 1

- Periodically sample temperature (every 1s)
  - Remember the **etimer** and **ctimer** from last week
  - Temperature is provided by the SHT11 sensor - you need to find the name of that sensor
  - Sensor names are defined as “const struct sensors_sensor <name>”
    - Handy command line for searching for text in files:
    - grep <text> <path> will search for <text> in all the files in <path> NOT RECURSIVE
    - grep <text> -R <path> does the same but recursively
    - So, “grep <text> -R .” will search recursively in the current folder
- Average every 5 samples
  - If you want to use an array, use a static one, embedded development doesn’t like malloc
  - So, “int samples[5]; int crt_sample;” one variable to hold the current sample index
- If average is greater than a threshold, print “It sure is nice today!”

Processes and events

Contiki processes

- Declared with PROCESS(<handle>, <description string>)
- Start on boot by including in AUTOSTART_PROCESSES(<p1>, <p2>, …)
- Defined with PROCESS_THREAD(<handle>, <event>, <event_data>)

Contiki process management

```
PROCESS_BEGIN();
while (1) {
  ……
  PROCESS_YIELD();
  … …
}
PROCESS_END();
```

```
PROCESS_BEGIN();
while (1) {
  ……
  PROCESS_WAIT_EVENT();
  … …
}
PROCESS_END();
```

```
PROCESS_BEGIN();
while (1) {
  ……
PROCESS_WAIT_EVENT_UNTIL(...);
  … …
}
PROCESS_END();
```

process functions (PROCESS_THREAD…)

Contiki process management

Exit function for now,
until event happens

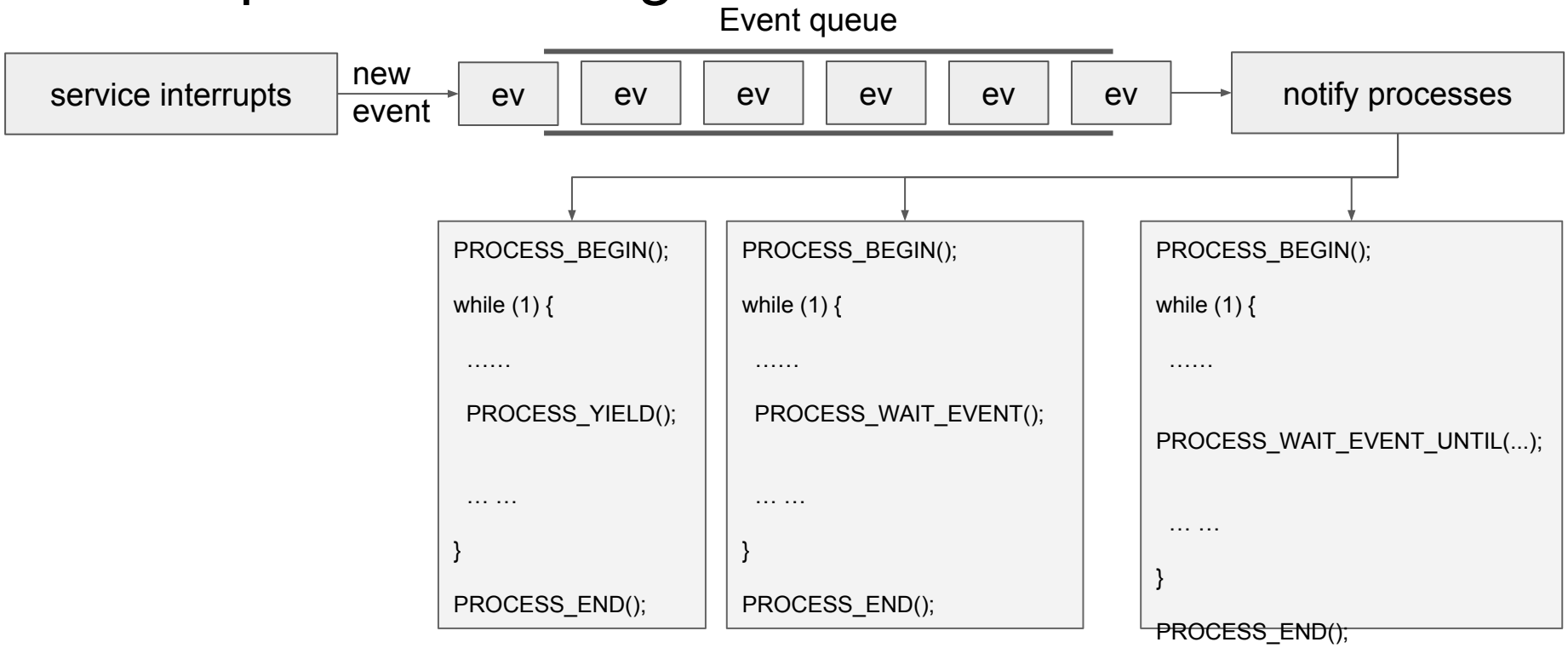
```
PROCESS_BEGIN();
while (1) {
  ……
  PROCESS_YIELD();
  … …
}
PROCESS_END();
```

```
PROCESS_BEGIN();
while (1) {
  ……
  PROCESS_WAIT_EVENT();
  … …
}
PROCESS_END();
```

```
PROCESS_BEGIN();
while (1) {
  ……
PROCESS_WAIT_EVENT_UNTIL(...);
  … …
}
PROCESS_END();
```

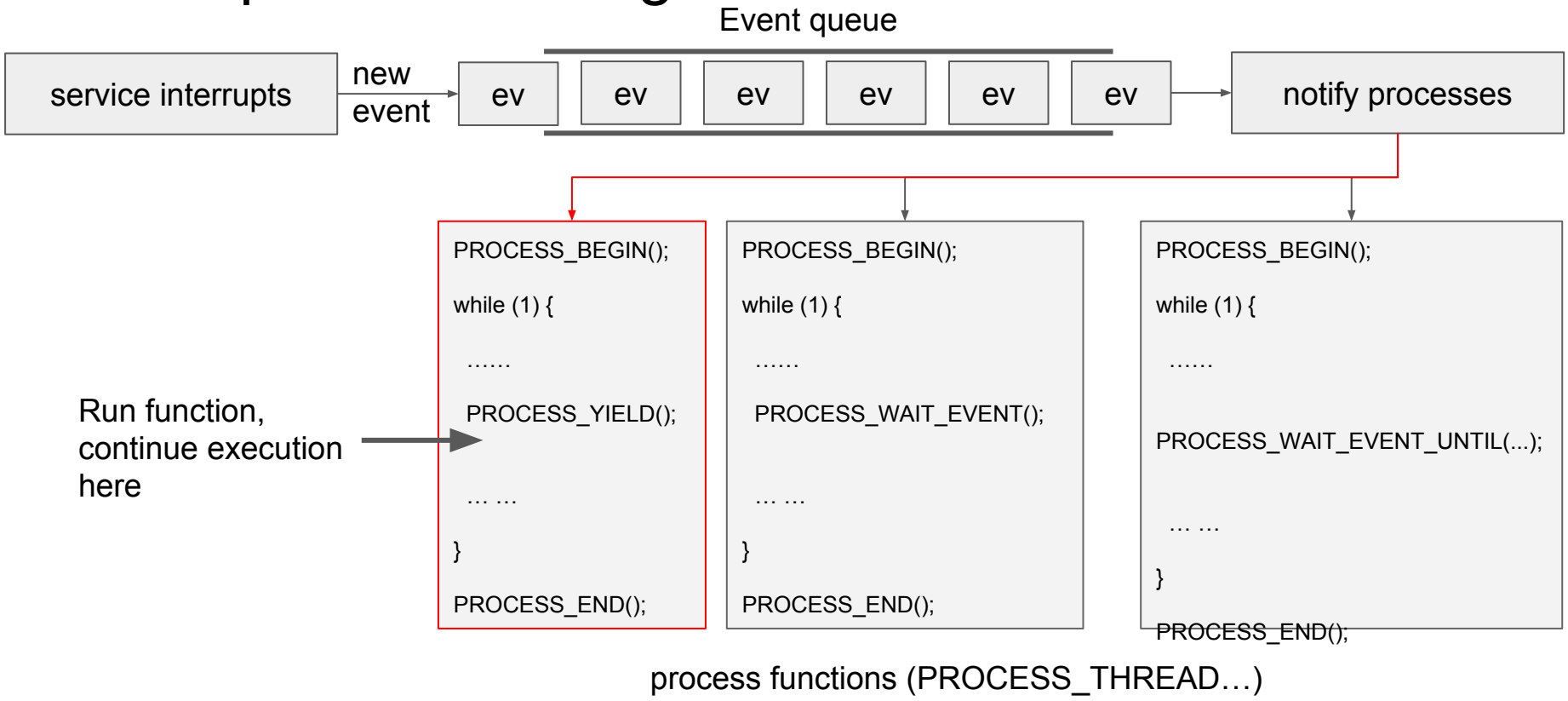
process functions (PROCESS_THREAD…)

Contiki process management

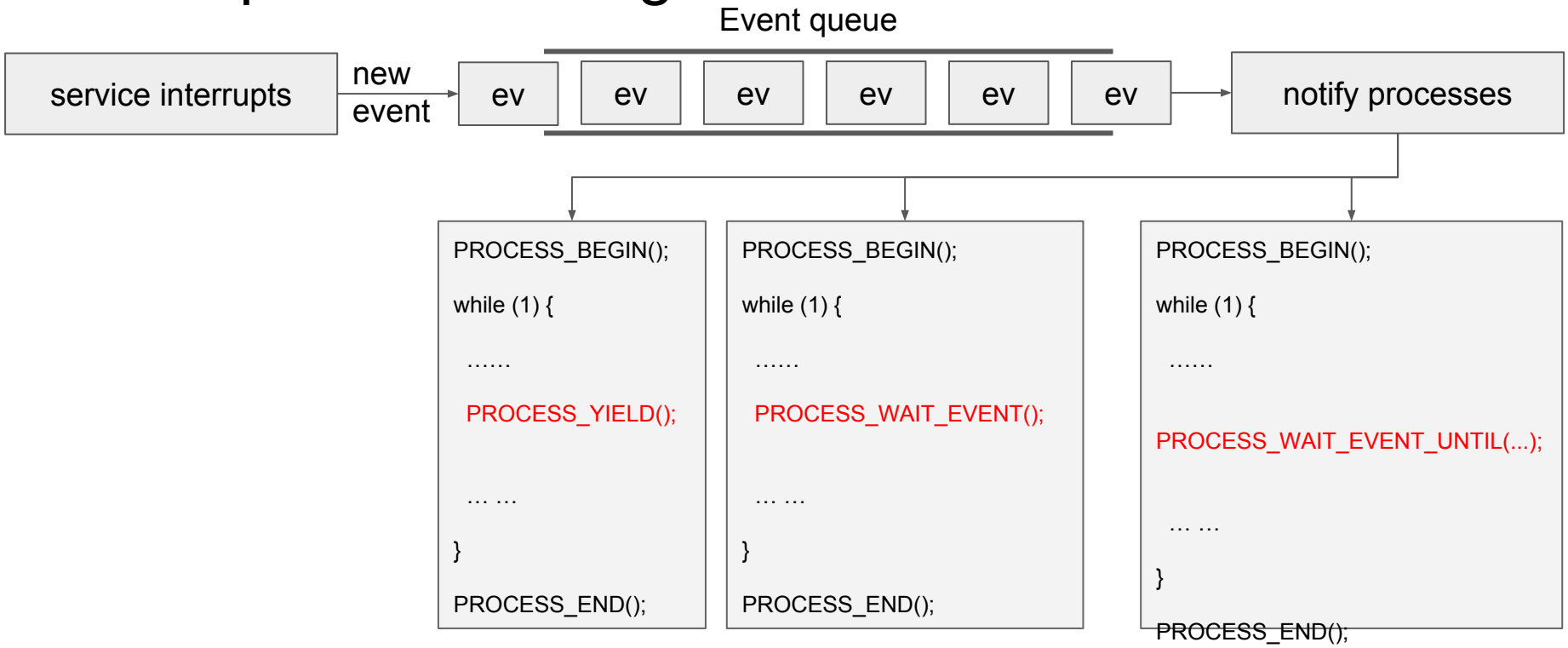


process functions (PROCESS_THREAD…)

Contiki process management



Contiki process management



process functions (PROCESS_THREAD…)

Contiki process function (PROCESS_THREAD)

```
PROCESS_THREAD(handle, ev, data)
{
        // pre-begin work

        PROCESS_BEGIN();

        // pre-loop setup work

        while (1) { // or any other loop

                // in-loop setup

                PROCESS_YIELD(); // or similar

                // processing
        }
        PROCESS_END();
}
```

- Regular C function
- Initially called at boot time with null ev and data

Contiki process function (PROCESS_THREAD)

```
PROCESS_THREAD(handle, ev, data)
{
        // pre-begin work

        PROCESS_BEGIN();

        // pre-loop setup work

        while (1) { // or any other loop

                // in-loop setup

                PROCESS_YIELD(); // or similar

                // processing
        }
        PROCESS_END();
}
```

- Regular C function
- Initially called at boot time with null ev and data
- The macros are used for jumping in the code
  - Actually implemented with switch-case statement

Contiki process function (PROCESS_THREAD)

```
PROCESS_THREAD(handle, ev, data)
{
        // pre-begin work
                                        //static int flag;
        PROCESS_BEGIN(); // switch(flag)

        // pre-loop setup work

        while (1) { // or any other loop

                // in-loop setup
                                // flag=V1;return;
                PROCESS_YIELD(); // case V1:

                // processing
        }
        PROCESS_END();
}
```

- Regular C function
- Initially called at boot time with null ev and data
- The macros are used for jumping in the code
  - Actually implemented with switch-case statement

Contiki process function (PROCESS_THREAD)

```
PROCESS_THREAD(handle, ev, data)
{
        // pre-begin work
                                        //static int flag;
        PROCESS_BEGIN(); // switch(flag)

        // pre-loop setup work

        while (1) { // or any other loop

                // in-loop setup
                                // flag=V1;return;
                PROCESS_YIELD(); // case V1:

                // processing
        }
        PROCESS_END();
}
```

- Regular C function
- Initially called at boot time with null ev and data
- The macros are used for jumping in the code
  - Actually implemented with switch-case statement
  - Also known as “Duff’s device”

Contiki process function (PROCESS_THREAD)

```
PROCESS_THREAD(handle, ev, data)
{
        // pre-begin work
                                        //static int flag;
        PROCESS_BEGIN(); // switch(flag)

        // pre-loop setup work

        while (1) { // or any other loop

                // in-loop setup
                                // flag=V1;return;
                PROCESS_YIELD(); // case V1:

                // processing
        }
        PROCESS_END();
}
```

- Regular C function
- Initially called at boot time with null ev and data
- The macros are used for jumping in the code
  - Actually implemented with switch-case statement
  - Also known as “Duff’s device”
  - Side-effect - can’t use switch inside while!
- pre-begin work is executed for every event received
  - Variables declared here: static, uninitialised

Contiki process function (PROCESS_THREAD)

```
PROCESS_THREAD(handle, ev, data)
{
        // pre-begin work
                                        //static int flag;
        PROCESS_BEGIN(); // switch(flag)

        // pre-loop setup work

        while (1) { // or any other loop

                // in-loop setup
                                // flag=V1;return;
                PROCESS_YIELD(); // case V1:

                // processing
        }
        PROCESS_END();
}
```

- Regular C function
- Initially called at boot time with null ev and data
- The macros are used for jumping in the code
  - Actually implemented with switch-case statement
  - Also known as “Duff’s device”
  - Side-effect - can’t use switch inside while!
- pre-begin work is executed for every event received
  - Variables declared here: static, uninitialised
- pre-loop setup is only executed on first run
  - Commonly used for initialisation

Contiki process function (PROCESS_THREAD)

```
PROCESS_THREAD(handle, ev, data)
{
        // pre-begin work
                                        //static int flag;
        PROCESS_BEGIN(); // switch(flag)

        // pre-loop setup work

        while (1) { // or any other loop

                // in-loop setup
                                // flag=V1;return;
                PROCESS_YIELD(); // case V1:

                // processing
        }
        PROCESS_END();
}
```

- Regular C function
- Initially called at boot time with null ev and data
- The macros are used for jumping in the code
  - Actually implemented with switch-case statement
  - Also known as “Duff’s device”
  - Side-effect - can’t use switch inside while!
- pre-begin work is executed for every event received
  - Variables declared here: static, uninitialised
- pre-loop setup is only executed on first run
  - Commonly used for initialisation
- execution continues here on new event

Creating and raising events

- We’ve seen: PROCESS_EVENT_TIMER, sensors_event
- Anyone can declare an event:
  - process_event_t <name of event>;
  - Then after PROCESS_BEGIN(), and before the while loop,
    - <name of event> = process_alloc_event();
- Why?
  - Custom events for inter-process communication
  - Example:
    - one process for churning data, one process for reporting it
    - Data process notifies reporting process when data is ready
- Event posting (sending events)
  - process_post(PROCESS_BROADCAST, <name of event>, <event data>)

Handling events

First of all - need to know about the event

- Usually define event (process_event_t…) in a header file that can be included by anyone interested

The PROCESS_THREAD function takes as parameters:

- event and event data (pointer to!)

After a PROCESS_YIELD, PROCESS_WAIT_EVENT, etc

- Check event type: if (ev == <name of shared event>) { // great, process data}

Problem 2: multiple processes

- Process 1 prints out “hello world” repeatedly
- Process 2 prints out “this is pseudo-multi-threading” repeatedly
  - Timers are not important

Instructions:

- Can have both processes in same file, or in separate files
- AUTOSTART_PROCESSES is important
- If you use two files
  - Makefile must be updated - specify both files
  - One process must declare its process in an include file for the other to access for AUTOSTART_PROCESSES
    - In the include file, use PROCESS_NAME(<name of process>)

Problem 3: inter-process communication

Problem 1 split over two processes: one for sensing, the other for printing.

- Process 1
  - Periodically senses temperature
  - Averages every 5 samples
  - Defines an event that indicates when an average is ready
  - After averaging, sends the event to Process 2, setting the data to the average
- Process 2
  - Waits for the event from process 1
  - Reads the data and prints it out

Can be done in a single file or separately

Additional info

- Tmote Sky HW datasheet and schematic:
  - http://www2.ece.ohio-state.edu/~bibyk/ee582/telosMote.pdf
- SHT11 datasheet, including formulas for converting ADC values to temp/hum
  - https://cdn.sparkfun.com/datasheets/Sensors/Pressure/Sensirion_Humidity_SHT1x_Datasheet_V5.pdf
- MSP430 (microcontroller on Tmote Sky) user guide
  - Pins, timers, adc, etc
  - In case you want to program the motes in assembly ;)
  - http://www.ti.com/lit/ug/slau049f/slau049f.pdf