



Introduction to Embedded Systems, Lecture 9

ES software design in Arduino IDE

Dmitry Zaitsev

<http://daze.ho.ua>

SKETCHBOOK

- Sketch_01.1_Blink
- Sketch_01.2_Blink
- Sketch_02.1_ButtonAndLed
- Sketch_02.2_TableLamp
- Sketch_03.1_FlowingLight
- Sketch_04.1_BreathingLight
- Sketch_04.2_FlowingLight2
- Sketch_05.1_RandomColorLight
- Sketch_05.2_GradientColorLight
- Sketch_06.1_LEDPixel
- Sketch_06.2_RainbowLight
- Sketch_07.1_Doorbell
- Sketch_07.2_Aleror
- Sketch_08.1_SerialPrinter
- Sketch_08.2_SerialRW
- Sketch_09.1_ADC
- Sketch_10.1_SoftLight

NEW SKETCH

Sketch_01.1_Blink.ino

```
1  /*****
2  * Filename   : Blink
3  * Description: Make an led blinking.
4  * Author    : www.freenove.com
5  * Modification: 2021/10/13
6  *****/
7  #define LED_BUILTIN 25
8
9  // the setup function runs once when you press reset or power the board
10 void setup() {
11   // initialize digital pin LED_BUILTIN as an output.
12   pinMode(LED_BUILTIN, OUTPUT);
13 }
14
15 // the loop function runs over and over again forever
16 void loop() {
17   digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage level)
18   delay(100);                      // wait for a second
19   digitalWrite(LED_BUILTIN, LOW);  // turn the LED off by making the voltage LOW
20   delay(100);                      // wait for a second
21 }
22
```

Output

```
In file included from C:\Users\dazai\AppData\Local\Arduino15\packages\arduino\hardware\avr\1.6.10\variants\arduino\pins_arduino.h:40,
| | | | | from C:\Users\dazai\AppData\Local\Arduino15\packages\arduino\hardware\avr\1.6.10\cores\arduino\Arduino.h:30,
| | | | | from C:\Users\dazai\AppData\Local\Temp\arduino\sketches\
c:\users\dazai\appdata\local\arduino15\packages\arduino\hardware\avr\1.6.10\cores\arduino\Arduino.h:30,
| | | | | 13 | #define LED_BUILTIN PIN_LED
```

Compiling sketch...

Downloading index: package_rp2040_index.json

Offline

Ln 22, Col 1 Raspberry Pi Pico W on COM5 [not connected]

7

Type here to search



54°F Cloudy

ENG

10:12 AM
3/20/2024

Arduino IDE

- Verify / Compile / Upload / Debug
- Sketchbook location / Sketch
- Additional board manager / Boards manager
- Library manager / Include library
- Tools: board, port, serial monitor

Open hardware specification

- Arduino IDE key to success – open hardware specification – architecture configuration
- Certain invariance with respect to hardware
- Support of various platforms of many manufacturers
- Add platform (board family) specification
- Choose definite type of board
- Design ES for chosen board

Architecture configurations

Each architecture must be configured through a set of configuration files:

- **platform.txt** contains definitions for the CPU architecture used (compiler, build process parameters, tools used for upload, etc.)
- **boards.txt** contains definitions for the boards (board name, parameters for building and uploading sketches, etc.)
- **programmers.txt** contains definitions for external programmers (typically used to burn bootloaders or sketches on a blank CPU/board)

RP Pico Description Example

```
...  
"platforms": [  
  {  
    "category": "Raspberry Pi Pico",  
    "name": "Raspberry Pi Pico/RP2040",  
    "url": "https://github.com/earlephilhower/arduino-  
pico/releases/download/3.7.2/rp2040-3.7.2.zip",  
    "version": "3.7.2",  
    "architecture": "rp2040",  
    "archiveFileName": "rp2040-3.7.2.zip",  
    "boards": [  
      {  
        "name": "Raspberry Pi Pico"  
      },  
    ],  
  },  
  ...  
]
```

Arduino IDE Sketch

- A sketch is the name of a program. Filename extension “**.ino**”. It's the unit of code to upload and run on an Arduino board.
- Two special functions are a part of every Arduino sketch: **setup()** and **loop()**.
- The **setup()** is called once, when the sketch starts.
- The **loop()** function is repeated implementing the basic loop of control.

Basic approaches to ES software design

- Bare hardware – manual organization of concurrent processes
 - pure bare hardware
 - using libraries
- Within OS (FreeRTOS) – using facilities of OS to control resources: processes, memory, devices

Polling vs Interrupt

- Polling – a basic eternal loop:
 - ✓ polling (reading) sensors,
 - ✓ compute (quasi) optimal control,
 - ✓ write control to actuators
- Interrupts – create a set of processes and run them in the even-driven mode, idling when there is no job

Arduino IDE Reference

- <https://www.arduino.cc/reference/en/>
- C/C++ language extended with library functions
- Digital/Analog GPIO I/O
- Serial I/O
- Interrupt control and processing
- Data transformation
- Peripheral device specific libraries

Basic routines

- **Digital I/O**

digitalRead()

digitalWrite()

pinMode()

- **Analog I/O**

analogRead()

analogWrite()

analogReadResolution()

analogWriteResolution()

analogReference()

- **Advanced I/O**

tone()

noTone()

pulseIn()

pulseInLong()

shiftIn()

shiftOut()

- **Time**

delay()

delayMicroseconds()

micros()

millis()

- **External Interrupts**

attachInterrupt()

detachInterrupt()

digitalPinToInterrupt()

- **Interrupts**

interrupts()

noInterrupts()

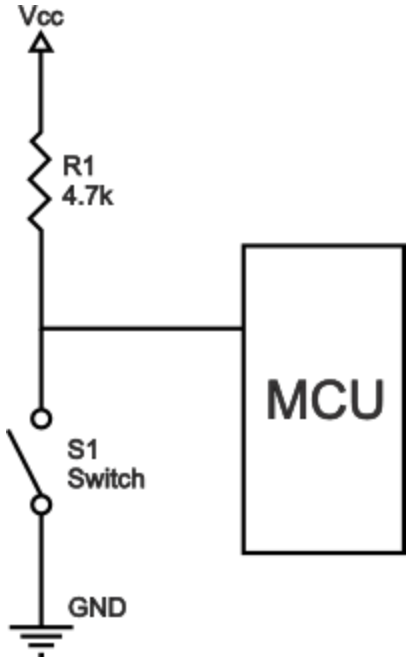
Digital I/O

- **digitalRead(pin)**: Reads the value from a specified digital pin, either HIGH or LOW.
- **digitalWrite(pin, value)**: Write a HIGH (5V) or a LOW (0V) value to a digital pin.
- **pinMode(pin, mode)**: Configures the specified pin to behave either as an input or an output; input pullup mode enables internal resistors.

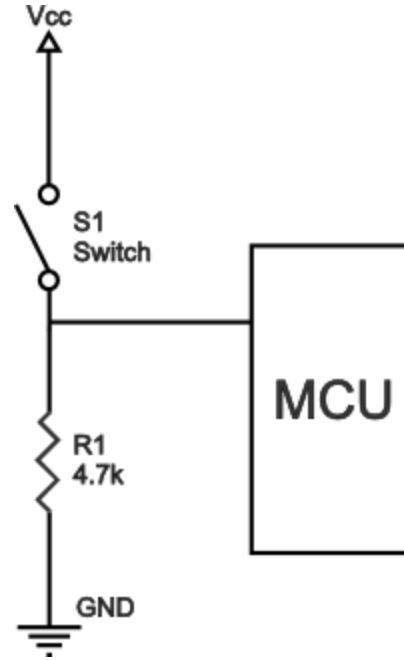
Pullup (Pulldown) Digital Input Modes

- Pull-up resistors are resistors used in logic circuits to ensure a well-defined logical level at a pin under all conditions. Digital logic circuits have three logic states: high, low and floating (or high impedance).
- When a pin is not connected from electrical circuit point of view (attached to a button), its value is rather indefinite
- Option PULLUP/PULLDOWN attaches an internal resistor to make the value certain: HIGH for PULLUP and LOW for PULLDOWN

Pullup and Pulldown Resistor Circuits



Pull-up resistor circuit



Pull-down resistor circuit

Analog I/O

- **analogRead(pin):** Reads the value from the specified analog pin in the range 0..1023 for 10 bit ADC.
- **analogWrite(pin, value):** Writes an analog value as a PWM wave to a pin. Can be used to light a LED at varying brightnesses or drive a motor at various speeds.
- **analogReadResolution(bits), analogWriteResolution(bits):** sets the corresponding resolution
- **analogReference(type):** Configures the reference voltage used for analog input.

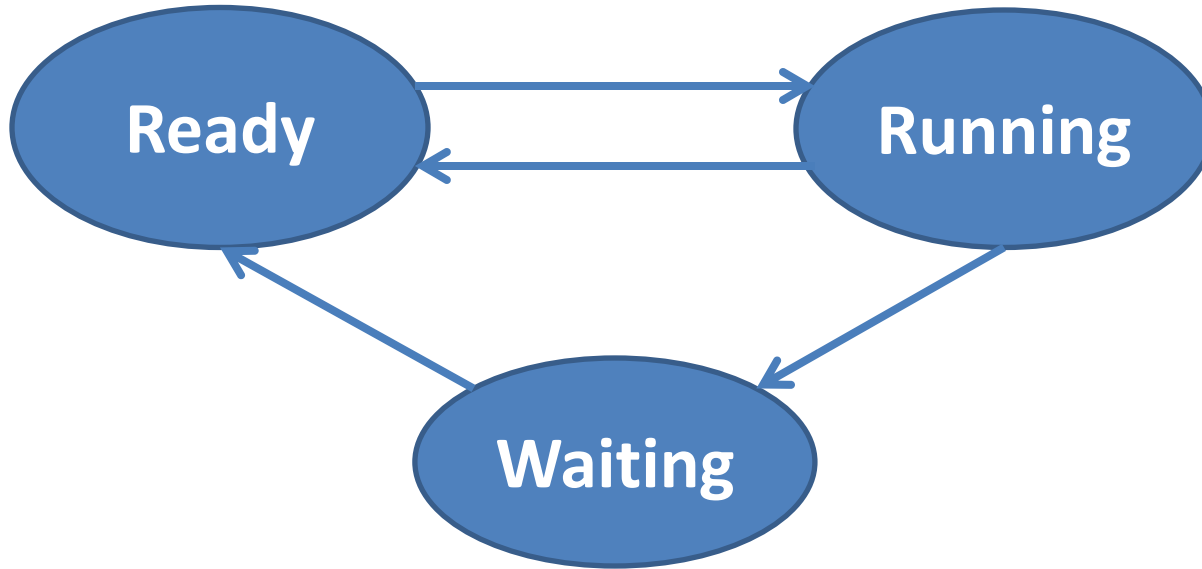
Time functions

- **delay(ms)**: Pauses the program for the specified amount of time in milliseconds.
- **delayMicroseconds(us)**: pause in microseconds
- **millis()**: Returns the number of milliseconds passed since the Arduino board began running the current program.
- **micros()**: the number of microseconds

Avoid long delays!

- Nothing useful is done when `delay()` function checks elapsing time
- Usually OS blocks a process requiring to wait for an event (time)
- After blocking a process, OS looks for other ready process to run
- Programming without OS, we are obliged to imitate OS functions running a few processes simultaneously

Typical process state diagram



Blink without delay

```
#define DT 1000
#define LEDpin 15
unsigned long t, t0=0;
int LEDstate=LOW;
void setup(){
  pinMode(LEDpin OUTPUT);
}
```

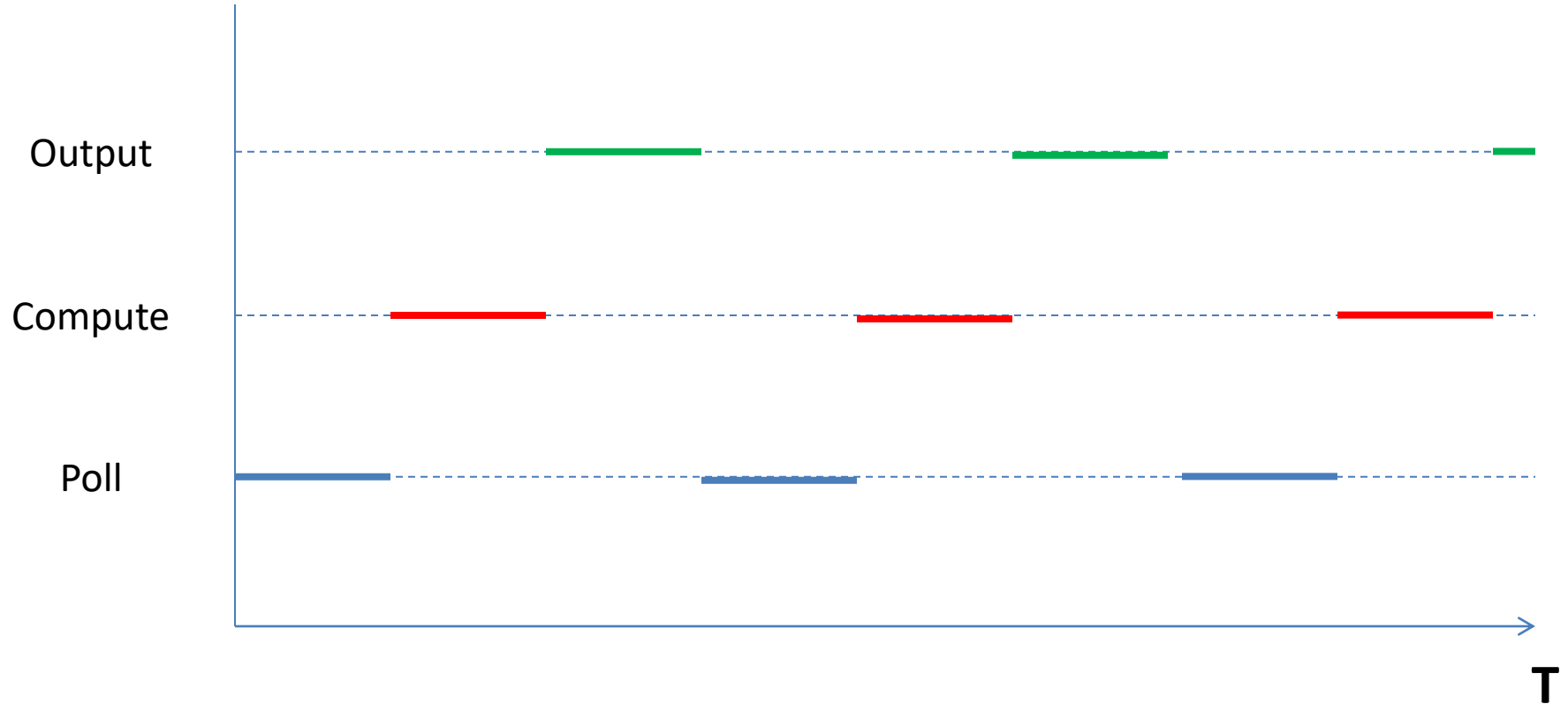
```
void loop(){
  t=millis();
  if( t-t0 >= DT ){
    t0=t;
    LEDstate=!LEDstate;
    digitalWrite(LEDpin, LEDstate);
  }
}
```

Polling

```
void setup() {  
    // initialize ES  
}  
  
void loop() {  
    // poll sensors  
    // calculate control  
    // output control  
}
```

- Advantages:
 - simplicity of concept
- Disadvantages:
 - it is difficult to run concurrent processes
 - limited reaction influenced by the loop processing time
 - it is difficult to implement priorities of processes

Polling time diagram



Partitioning processes

- Divide each process into parts
- Each part does not use `delay()` or similar functions
- For each part specify activation time with respect to the previous part
- Represent each part as a C language function
- Use global or dynamic memory for information exchange within a process between parts

Partitioned Process Scheduling Data

Processes					
Parts		0	1	...	np-1
	0
	1
	run_ij()	...
	np-1
Next				c, at	

- np – number of processes
- np-1 – number of process parts
- c – next current part
- at – next activation time
- run_ij() – i^{th} part of j^{th} process

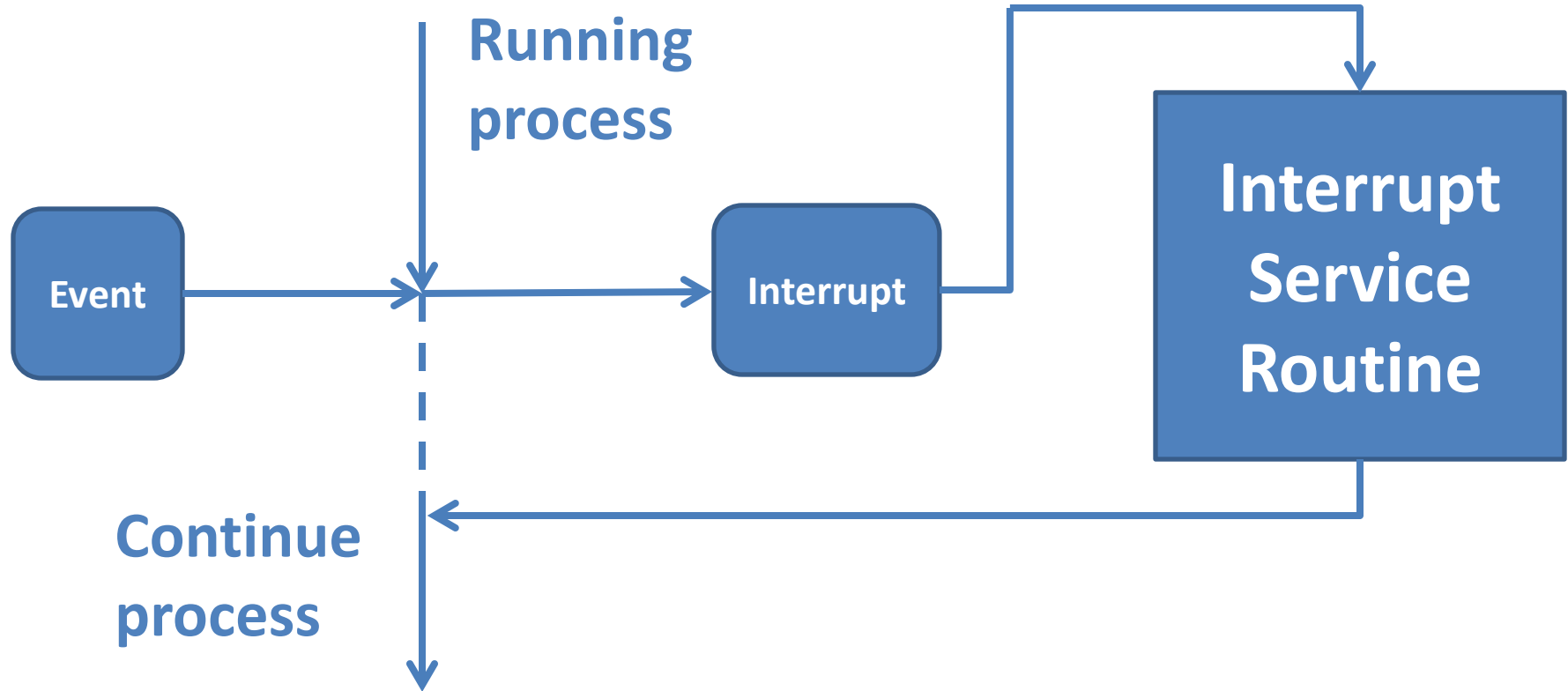
Partitioned Process Scheduling Algorithm

```
void loop() {  
    t = millis();  
    dt = t-pre_t;  
    for(p=0; p<np; p++)  
        if(next[p].at <= t){  
            proc[p][c].run(t, dt, p, c, &nc, &nat);  
            next[p].c = nc;  
            next[p].at = nat;  
        }  
    pre_t=t;  
}
```

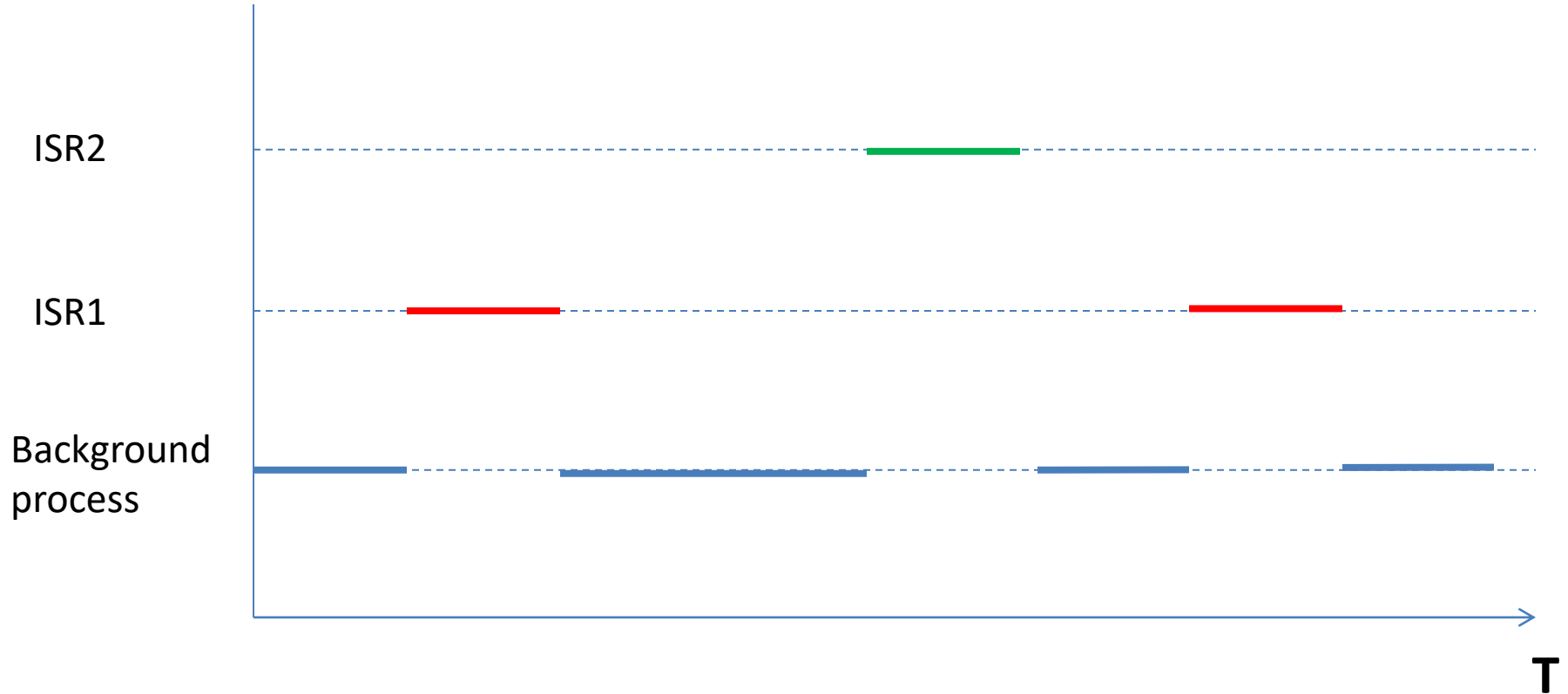
or
“greener”

```
void loop() {  
    t = mint{0<i<np}{ next[i].at };  
    dt = t-pre_t;  
    delay( dt );  
    for(p=0; p<np; p++)  
        if(next[p].at <= t){  
            proc[p][c].run(t, dt, p, c, &nc, &nat);  
            next[p].c = nc;  
            next[p].at = nat;  
        }  
    pre_t=t;  
}
```


Interrupt Service Routine – Event Driven Processing of Information



Interrupt time diagram



External Interrupts

- **attachInterrupt(digitalPinToInterrupt(pin), ISR, mode):** attach interrupt to Interrupt Service Routine (ISR) with mode: LOW, CHANGE, RISING, FALLING, HIGH
- **detachInterrupt(digitalPinToInterrupt(pin)):** turns off the given interrupt
- **digitalPinToInterrupt(pin):** checks whether a pin can be used for interrupt (-1 if not available)

Interrupts

- **interrupts()**: Re-enables interrupts (after they've been disabled by `noInterrupts()`). Interrupts are enabled by default.
- **noInterrupts()**: Disables interrupts. For critical sections of code.

Attaching ISR

```
#include "IR.h"
attachInterrupt(digitalPinToInterrupt(irPin),
               IR_Read, CHANGE);
void IR_Read() {
    // Interrupt Service Routine (handler)
    // Should be fast! No parameters, no return value!
}
```

Peculiarities of Interrupt Service Routines

- ISRs are special kinds of functions that have some unique limitations. An ISR cannot have any parameters, and they shouldn't return anything.
- Typically global variables are used to pass data between an ISR and the main program. To make sure variables shared between an ISR and the main program are updated correctly, declare them as volatile.
- Generally, an ISR should be as short and fast as possible. If your sketch uses multiple ISRs, only one can run at a time, other interrupts will be executed after the current one finishes in an order that depends on the priority they have.
- `millis()` relies on interrupts to count, so it will never increment inside an ISR. Since `delay()` requires interrupts to work, it will not work if called inside an ISR.
- `micros()` works initially but will start behaving erratically after 1-2 ms. `delayMicroseconds()` does not use any counter, so it will work as normal.

Interrupt Control Button-LED Lamp

```
int buttonState0 = HIGH;
int buttonState1 = LOW;
int LEDState = LOW;
void setup() {
    pinMode(ledPin, OUTPUT);
    digitalWrite(ledPin, LEDState);
    IR_Init(buttonPin);
}
void IR_Init(int pin) {
    int irPin = pin;
    pinMode(irPin, INPUT_PULLUP);
    attachInterrupt(digitalPinToInterrupt(irPin),
                    IR_Read, CHANGE);
}
```

```
void IR_Read() {
    buttonState1 = digitalRead(buttonPin);
    if( buttonState1 != buttonState0 ) {
        if ( buttonState1 == HIGH ) {
            LEDState = ! LEDState;
            digitalWrite(ledPin, LEDState);
        }
        buttonState0=buttonState1;
    }
}

void loop() {
    delay(10000);
}
```

Advanced I/O (1)

- **tone(pin, frequency, duration)** : Generates a square wave of the specified frequency (and 50% duty cycle) on a pin. A duration can be specified, otherwise the wave continues until a call to noTone()
- **noTone(pin)** : Stops the generation of a square wave triggered by tone(). Has no effect if no tone is being generated.

Advanced I/O (2)

- **pulseIn(pin, value, timeout)**: Reads a pulse (either HIGH or LOW) on a pin. For example, if value is HIGH, pulseIn() waits for the pin to go from LOW to HIGH, starts timing, then waits for the pin to go LOW and stops timing. Returns the length of the pulse in microseconds or gives up and returns 0 if no complete pulse was received within the timeout.
- **pulseInLong(pin, value, timeout)**: is an alternative to pulseIn() which is better at handling long pulse and interrupt affected scenarios.

Advanced I/O (3)

- **shiftIn(dataPin, clockPin, bitOrder):** Shifts in a byte of data one bit at a time. Starts from either the leftmost or rightmost significant bit. For each bit, the clock pin is pulled high, the next bit is read from the data line, and then the clock pin is taken low.
- **shiftOut(dataPin, clockPin, bitOrder, value):** Shifts out a byte of data one bit at a time.

Overview of other Arduino IDE functions

- Math functions: `abs()`, `min()`, `sqrt()`, `sin()`,...
- Working with bits and bytes: `bitWrite()`,...
- Working with characters: `isAlpha()`, `isSpace()`,...
- Random numbers: `random()`, `randomSeed()`
- Communication: `print()`, `read()`, `parseInt()`,...

Arduino IDE Libraries

- [Communication](#) (1456)
- [Data Processing](#) (415)
- [Data Storage](#) (177)
- [Device Control](#) (1187)
- [Display](#) (580)
- [Sensors](#) (1356)
- [Signal Input/Output](#) (519)
- [Timing](#) (254)