The CPAL programming language

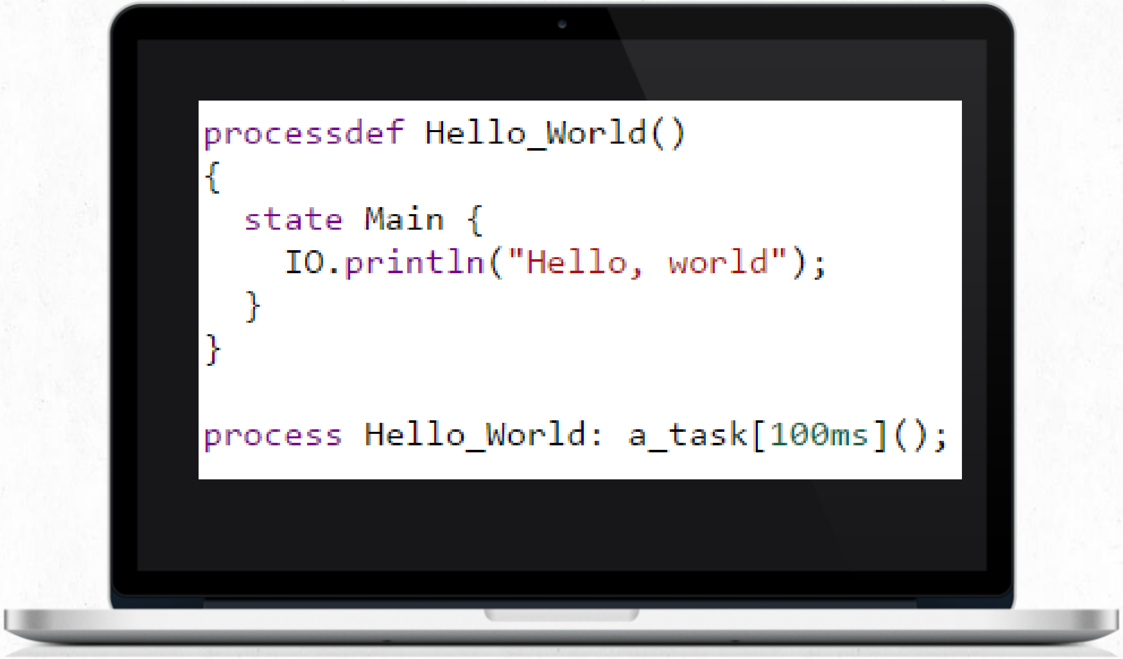
Design, Simulate, Execute
Embedded Systems

A tour of CPAL

Author: Nicolas.Navet@designcps.com

Version: 1.16 – October 21, 2016

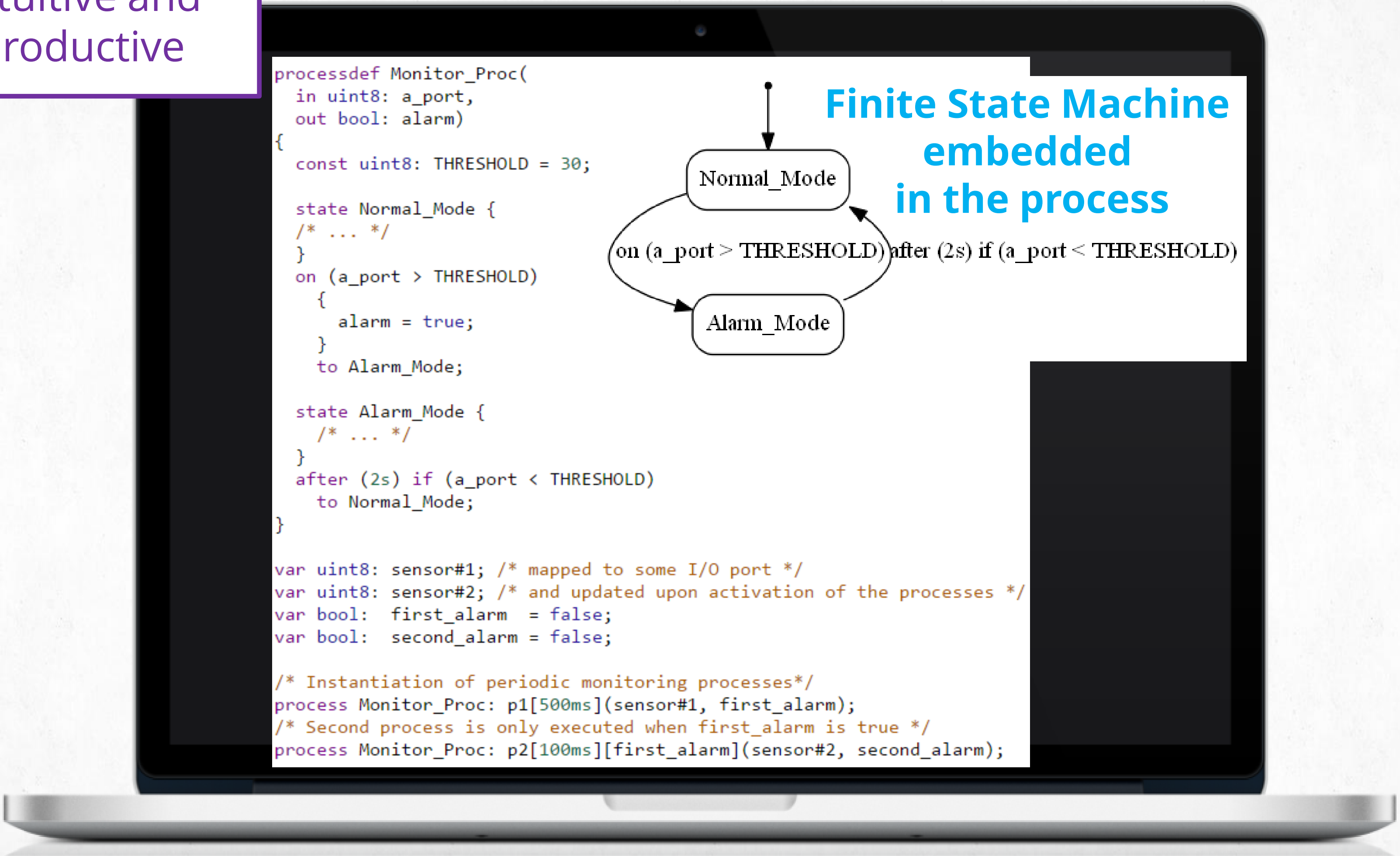
Hello, World



```
processdef Hello_World()
{
  state Main {
    IO.println("Hello, world");
  }
}

process Hello_World: a_task[100ms]();
```

Aim: be concise, intuitive and productive

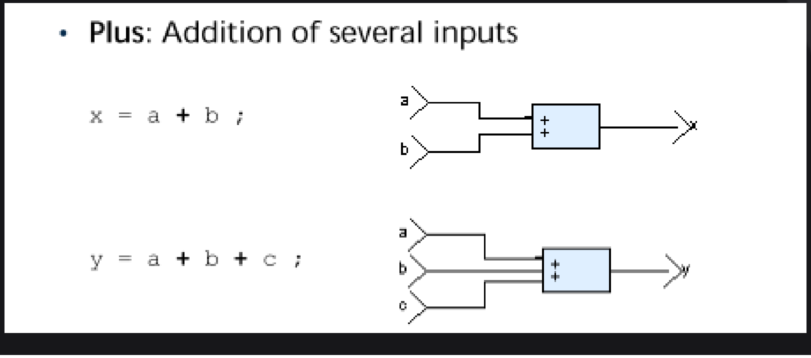
Hello, World



```
processdef Monitor_Proc(
  in uint8: a_port,
  out bool: alarm)
{
  const uint8: THRESHOLD = 30;

  state Normal_Mode {
  /* ... */
  }
  on (a_port > THRESHOLD)
    {
      alarm = true;
    }
    to Alarm_Mode;

  state Alarm_Mode {
    /* ... */
  }
  after (2s) if (a_port < THRESHOLD)
    to Normal_Mode;
}

var uint8: sensor#1; /* mapped to some I/O port */
var uint8: sensor#2; /* and updated upon activation of the processes */
var bool:  first_alarm  = false;
var bool:  second_alarm = false;

/* Instantiation of periodic monitoring processes*/
process Monitor_Proc: p1[500ms](sensor#1, first_alarm);
/* Second process is only executed when first_alarm is true */
process Monitor_Proc: p2[100ms][first_alarm](sensor#2, second_alarm);
```

Finite State Machine embedded in the process

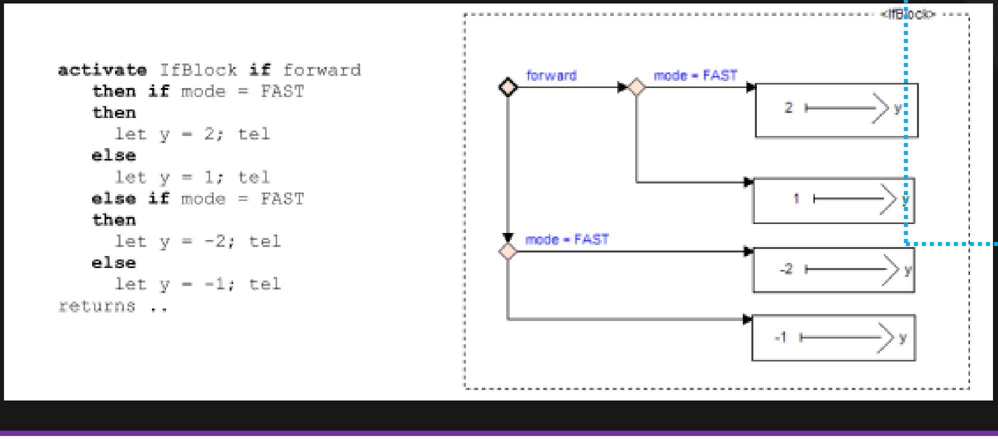
Preamble: a language can be textual, graphical or a mix of both



- Plus: Addition of several inputs

```
x = a + b ;
```

```
y = a + b + c ;
```



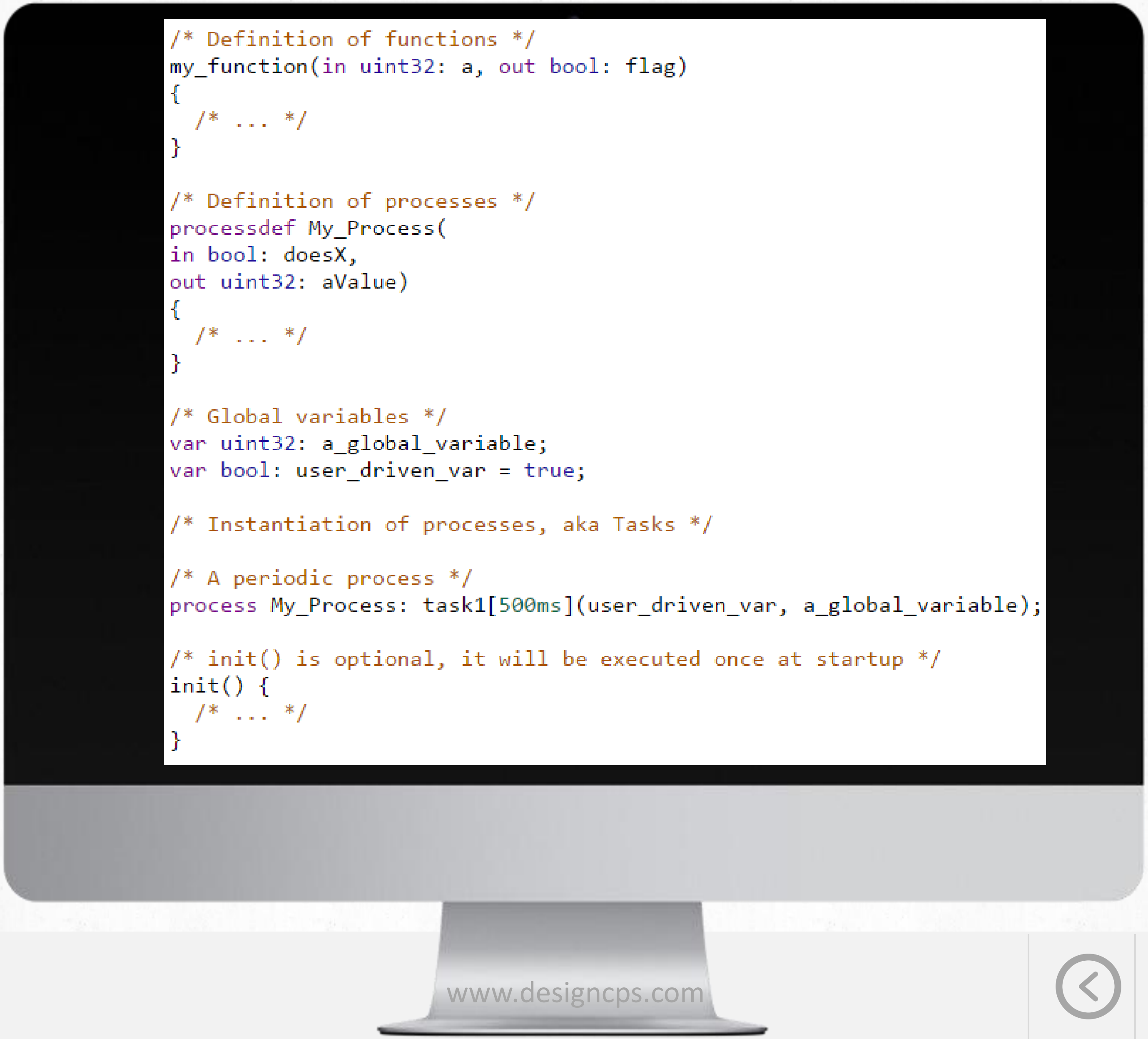
```
activate IfBlock if forward
   then if mode = FAST
   then
     let y = 2; tel
   else
     let y = 1; tel
   else if mode = FAST
   then
     let y = -2; tel
   else
     let y = -1; tel
returns ..
```

Examples from the Scade quick reference card

What do you think is the most efficient?

CPAL = textual programming with visual representation of facets out of the code: logic of the automata, data-flow between processes, task activation

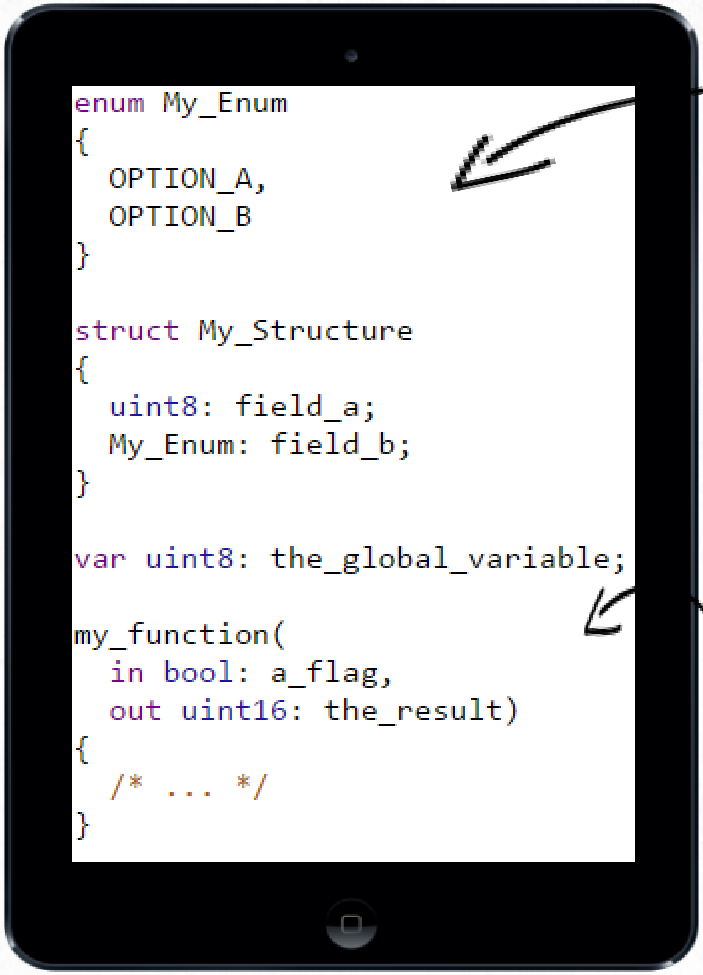
Structure of a program



```
/* Definition of functions */
my_function(in uint32: a, out bool: flag)
{
  /* ... */
}

/* Definition of processes */
processdef My_Process(
in bool: doesX,
out uint32: aValue)
{
  /* ... */
}

/* Global variables */
var uint32: a_global_variable;
var bool: user_driven_var = true;

/* Instantiation of processes, aka Tasks */

/* A periodic process */
process My_Process: task1[500ms](user_driven_var, a_global_variable);

/* init() is optional, it will be executed once at startup */
init() {
  /* ... */
}
```

CPAL Naming Convention

Names of user-defined process, structures, states, and enum. shall be **Mixed_Case_With_Underscores**



```
enum My_Enum
{
  OPTION_A,
  OPTION_B
}

struct My_Structure
{
  uint8: field_a;
  My_Enum: field_b;
}

var uint8: the_global_variable;

my_function(
  in bool: a_flag,
  out uint16: the_result)
{
  /* ... */
}
```

Names of enumerations values, and constant shall be **UPPER_CASE_WITH_UNDERSCORE**

Names of variables, arguments, functions, and tasks shall be **lower_case_with_underscores**

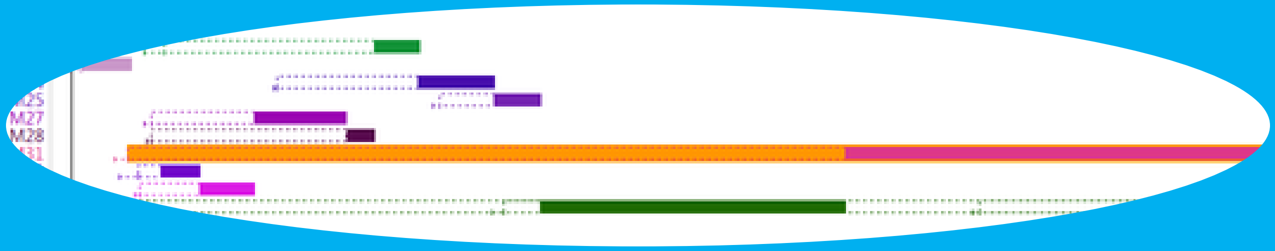
Use **cpal_lint** and **cpal2x** to resp. check and format code according to this naming convention

Why a programming language dedicated to Embedded Systems ?

- General purpose programming languages do not offer **the right abstractions** for:
  - Periodic activities and real-time scheduling
  - Time measurements and manipulation
  - Finite state machines
  - High-level interfaces to I/Os
  - etc

Both functional and non-functional concerns

- Design for facilitating the writing of **correct embedded code** (incl. restrictions)
- "Write once, Run Anywhere" of Java does not **guarantee** anything about **timing behaviour** on different platforms



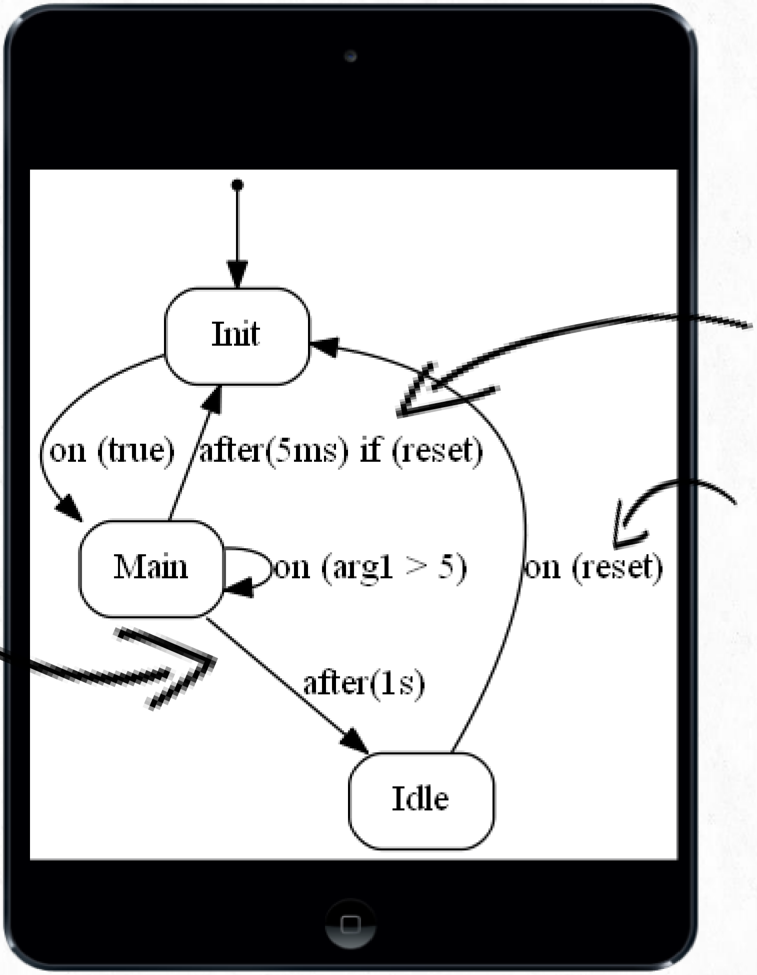
Processes: recurring activities whose logic is described as Finite State Machine

Finite-state Machines to describe the logic of processes

FSM =
states + transitions

Timed transition: after a certain time in a state, go to another state

Value that triggers a timed transition can change dynamically at run-time



Timed transition and condition

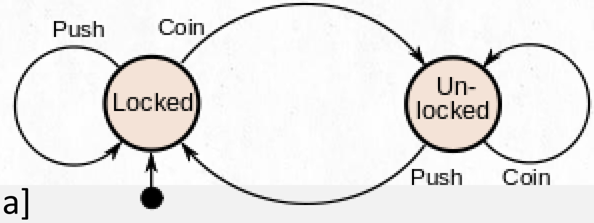
Boolean condition

Why using Finite State Machines ?

- Excellent way to describe the logic of programs that control “reactive” systems (=systems that react on external events)
- Non-ambiguous visual representation - one state at a time, transitions well defined
- Easy to execute, easy to simulate, properties can be verified by model-checking or simulation
- However, there is a variety of FSMs that may differ on when to trigger a transition, when leaving/entering a state, etc

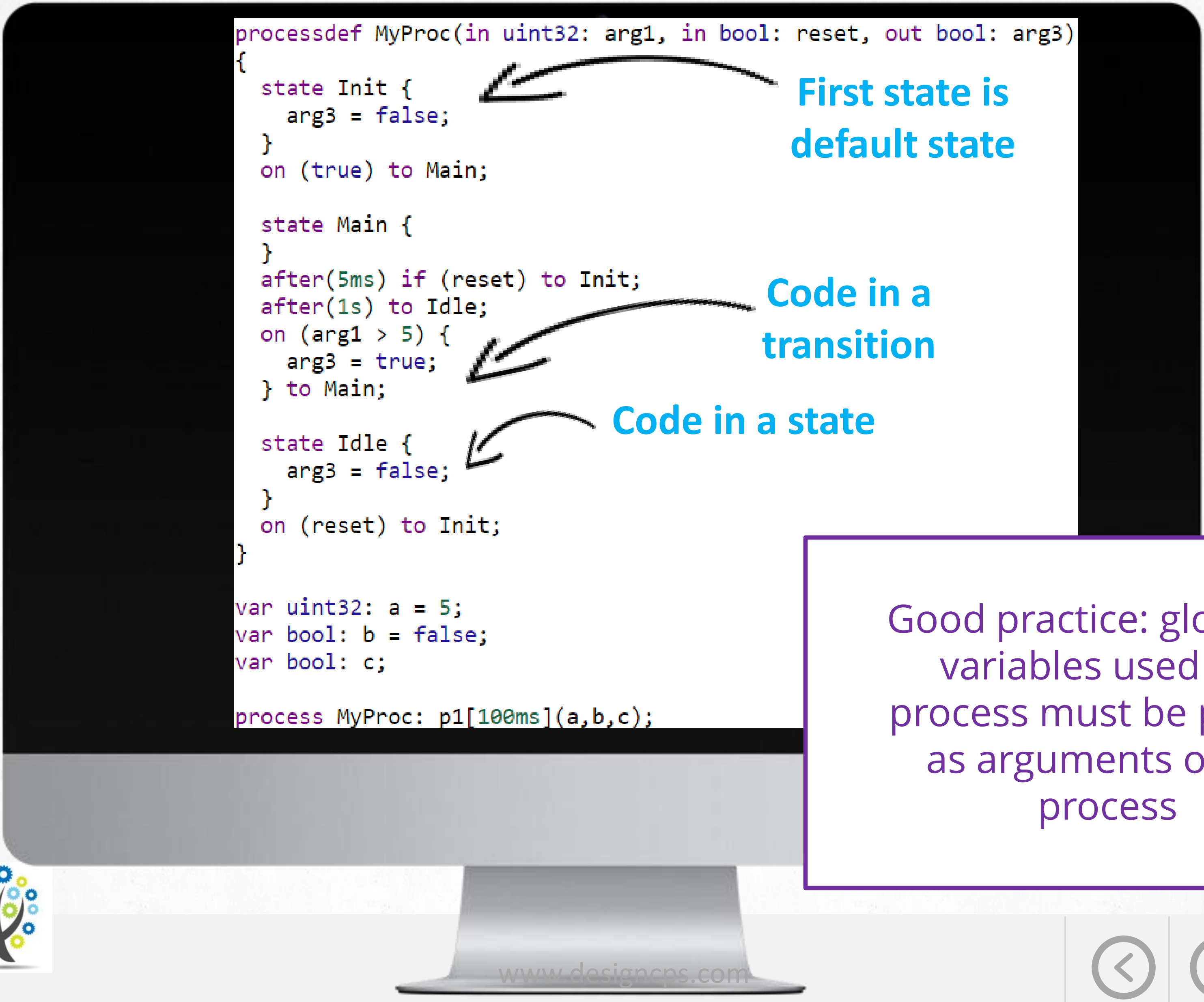
Question: draw the FSM that describes the functioning of a turnstile which allows someone to go through only after a coin has been inserted, discuss design choices

[wikipedia]



[wikipedia]

FSM in CPAL process



```
processdef MyProc(in uint32: arg1, in bool: reset, out bool: arg3)
{
  state Init {
    arg3 = false;
  }
  on (true) to Main;

  state Main {
  }
  after(5ms) if (reset) to Init;
  after(1s) to Idle;
  on (arg1 > 5) {
    arg3 = true;
  } to Main;

  state Idle {
    arg3 = false;
  }
  on (reset) to Init;
}

var uint32: a = 5;
var bool: b = false;
var bool: c;

process MyProc: p1[100ms](a,b,c);
```

First state is default state

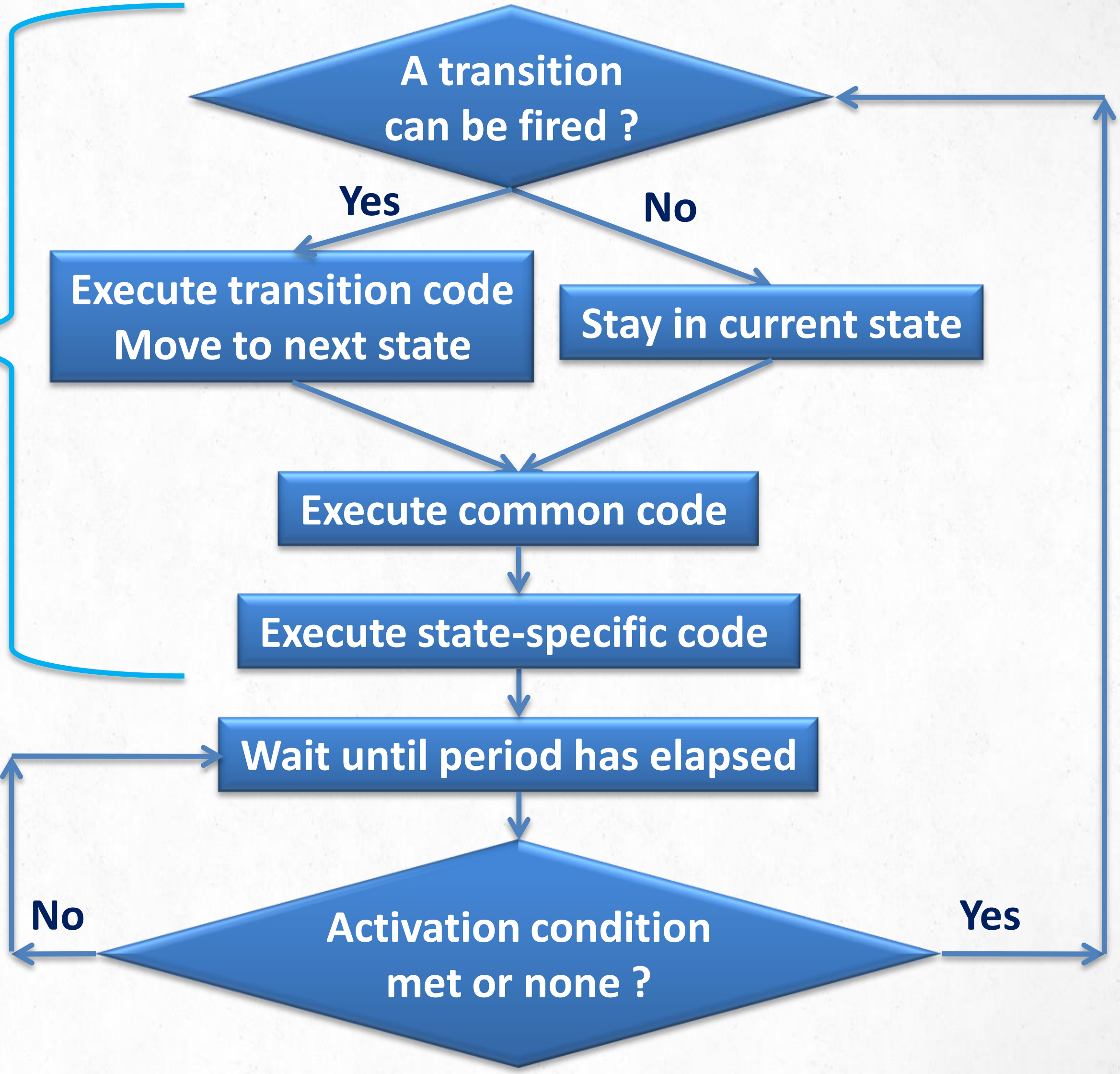
Code in a transition

Code in a state

Good practice: global variables used in a process must be passed as arguments of the process

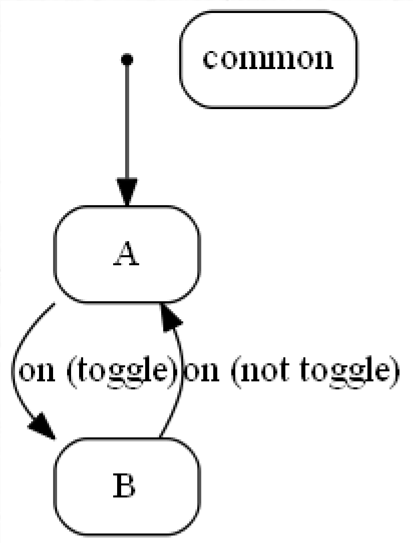
A process is periodically activated

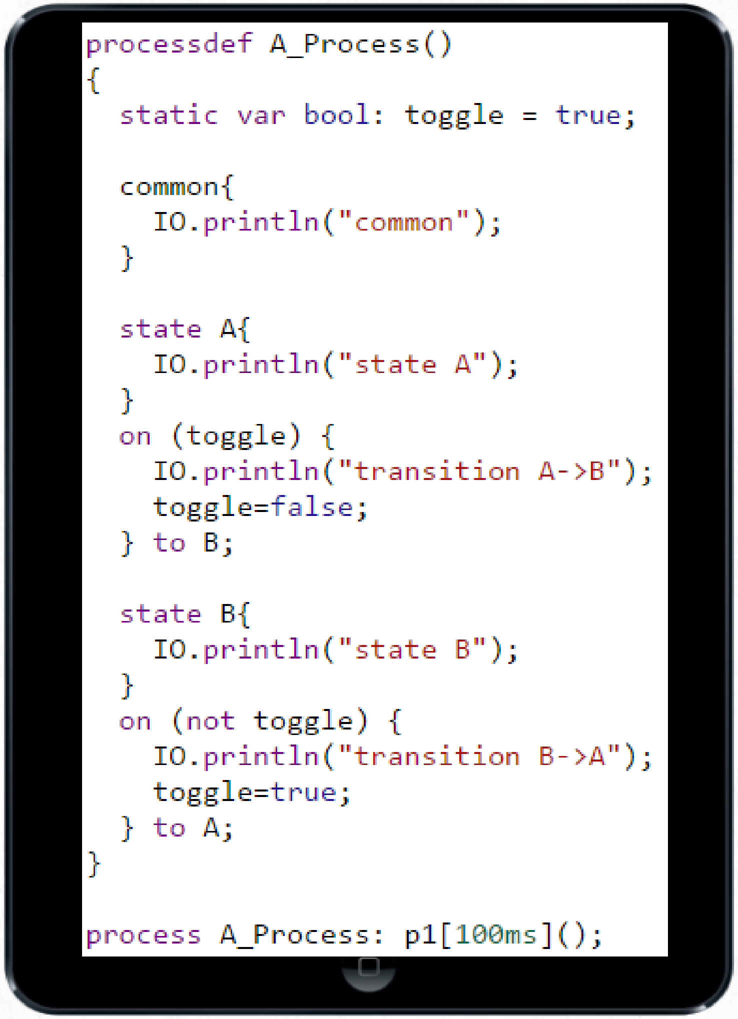
One “step” of execution of the FSM


A transition can be fired ?
Yes
No
Execute transition code
Move to next state
Stay in current state
Execute common code
Execute state-specific code
Wait until period has elapsed
No
Activation condition met or none ?
Yes

Execute a transition first (when possible) then the current state
→ best responsiveness to external events

Execution order

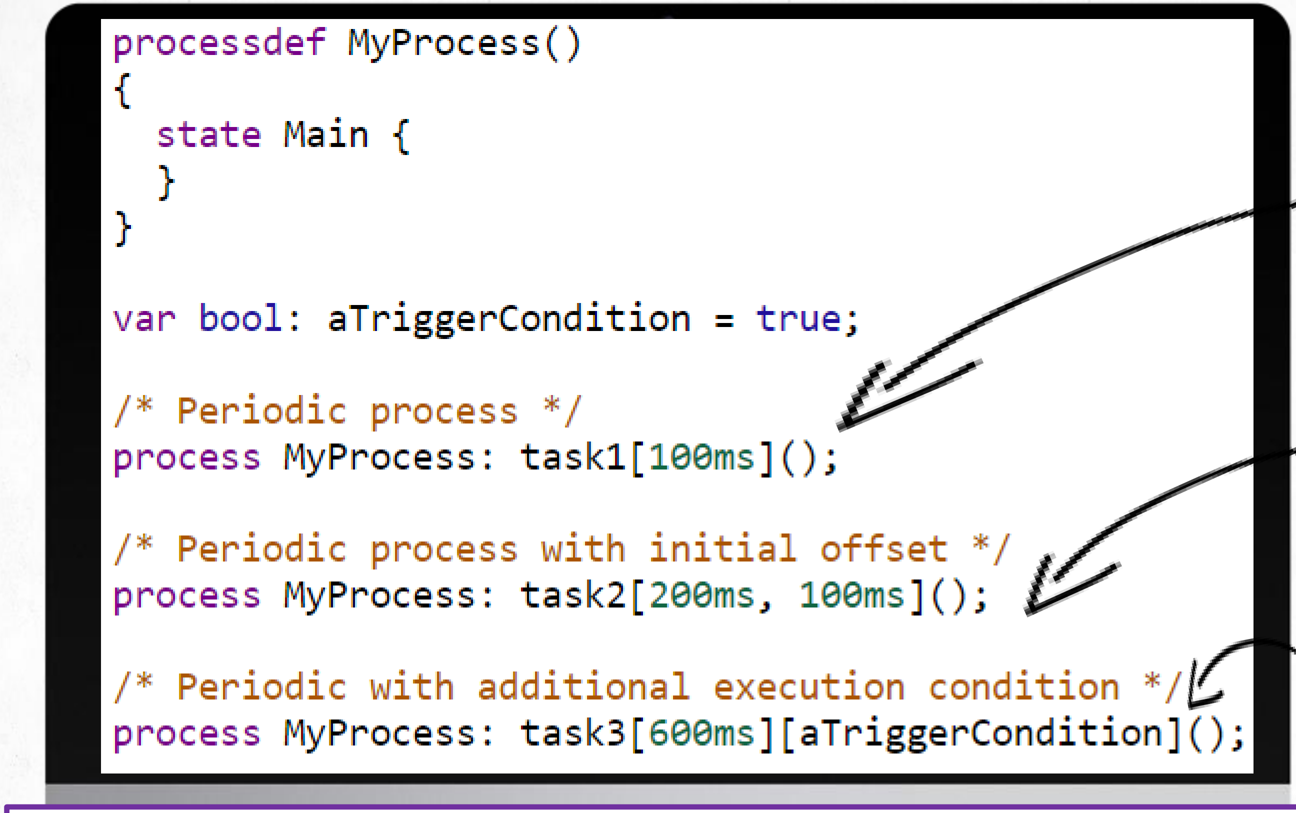




```
processdef A_Process()
{
  static var bool: toggle = true;

  common{
    IO.println("common");
  }

  state A{
    IO.println("state A");
  }
  on (toggle) {
    IO.println("transition A->B");
    toggle=false;
  } to B;

  state B{
    IO.println("state B");
  }
  on (not toggle) {
    IO.println("transition B->A");
    toggle=true;
  } to A;
}

process A_Process: p1[100ms]();
```

Try it out to check execution order at http://www.designcps.com/cpal-playground?path=talks/tutorial/samples/tut-execution-order.cpal

Process instantiations



```
processdef MyProcess()
{
  state Main {
  }
}

var bool: aTriggerCondition = true;

/* Periodic process */
process MyProcess: task1[100ms]();

/* Periodic process with initial offset */
process MyProcess: task2[200ms, 100ms]();

/* Periodic with additional execution condition */
process MyProcess: task3[600ms][aTriggerCondition]();
```

Periodic process

Periodic with an offsets: first instance is released at time `offset`

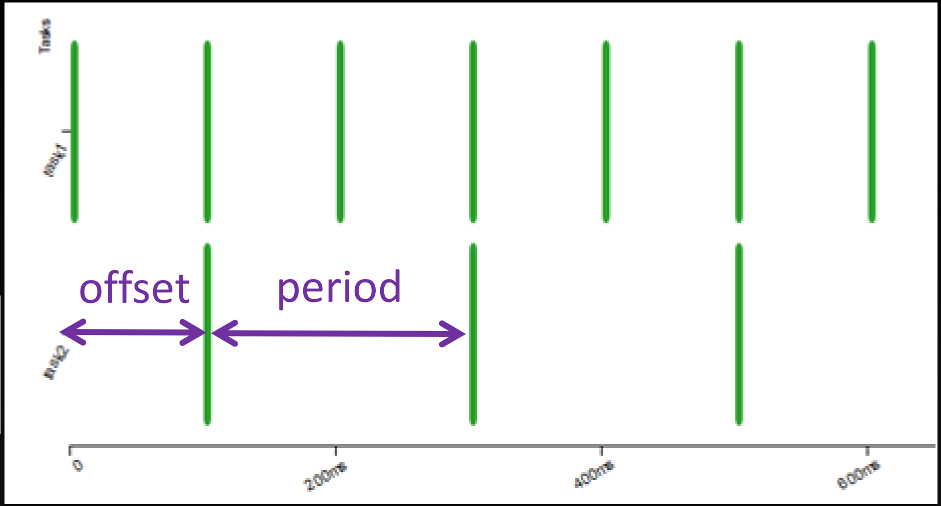
Periodic instance with activation condition

Activation conditions serve to implement **functioning modes** and execute activities only if specific conditions are met (e.g., event such as an alarm).

Process instantiations cont'd

```
/* Periodic process */
process MyProcess: task1[100ms]();
```

```
/* Periodic process with initial offset */
process MyProcess: task2[200ms, 100ms]();
```



Hands-on exercise #1

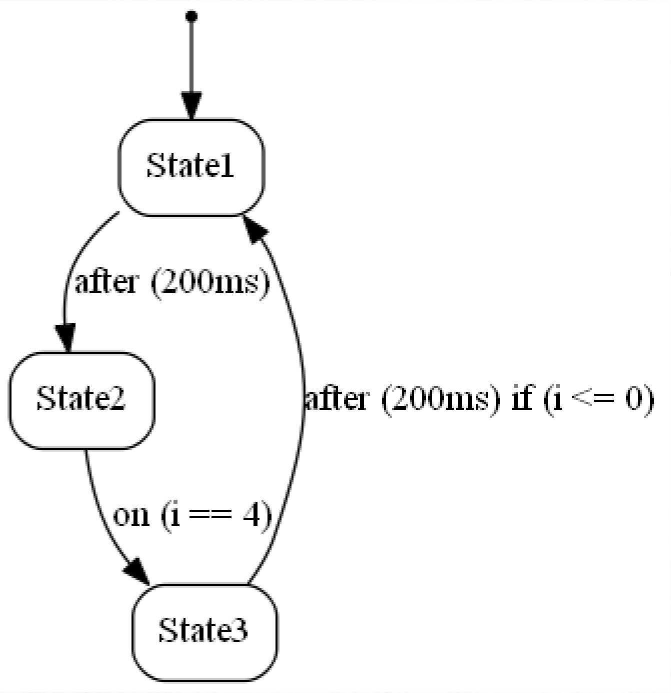
A] Write a process controlling the turnstile

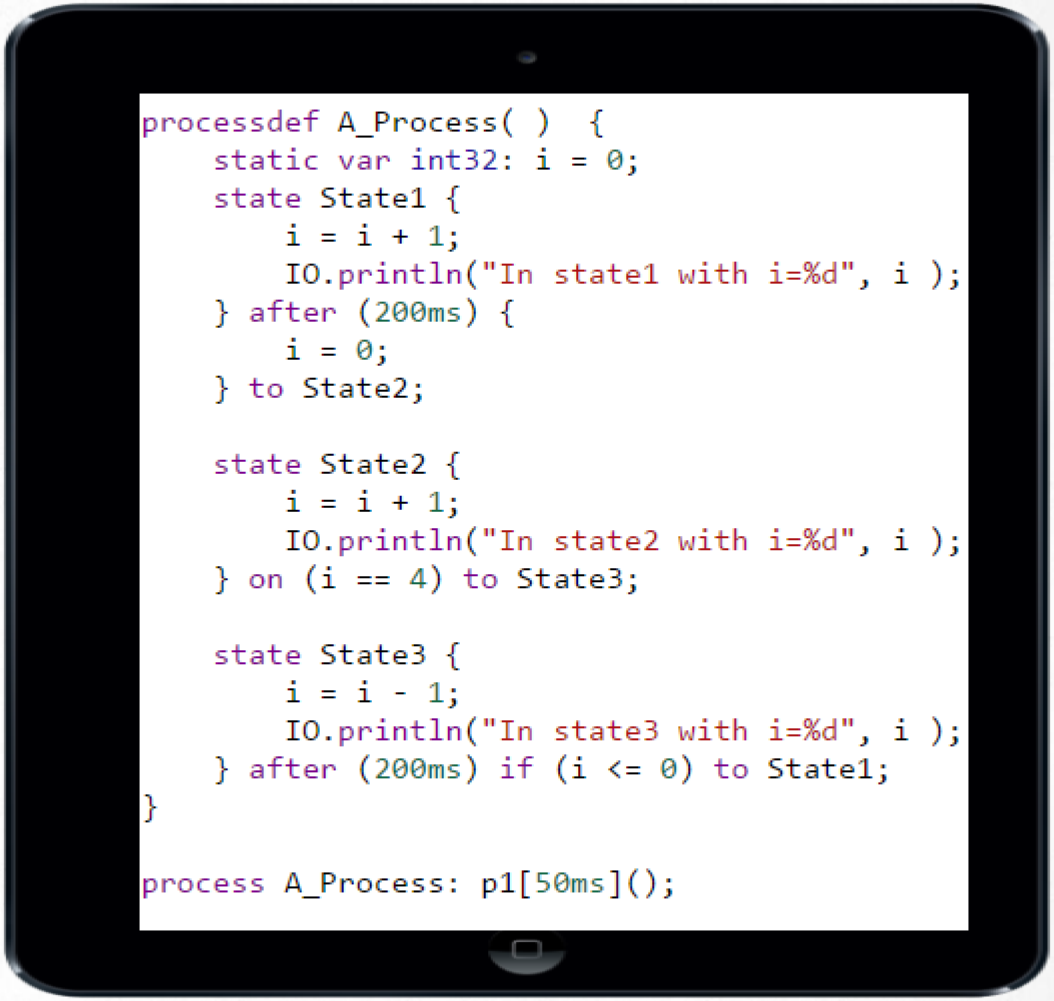
B] Write a process with period 50ms that:

- ✓**stay in `state1` during 200ms (where a variable *i* is incremented),**
- ✓**then goes to `state2` after having set *i* to 0 in transition. In state 2, *i* is incremented and the FSM goes to `state3` when *i* equals 4.**
- ✓**in `State3`, *i* is decremented and the FSM goes back to `state1` when it has stayed at least 200ms in `state3` and *i* is less than or equal to 0.**

C] Verify that the process runs as expected by executing the model and examining the changes of states and transitions triggered

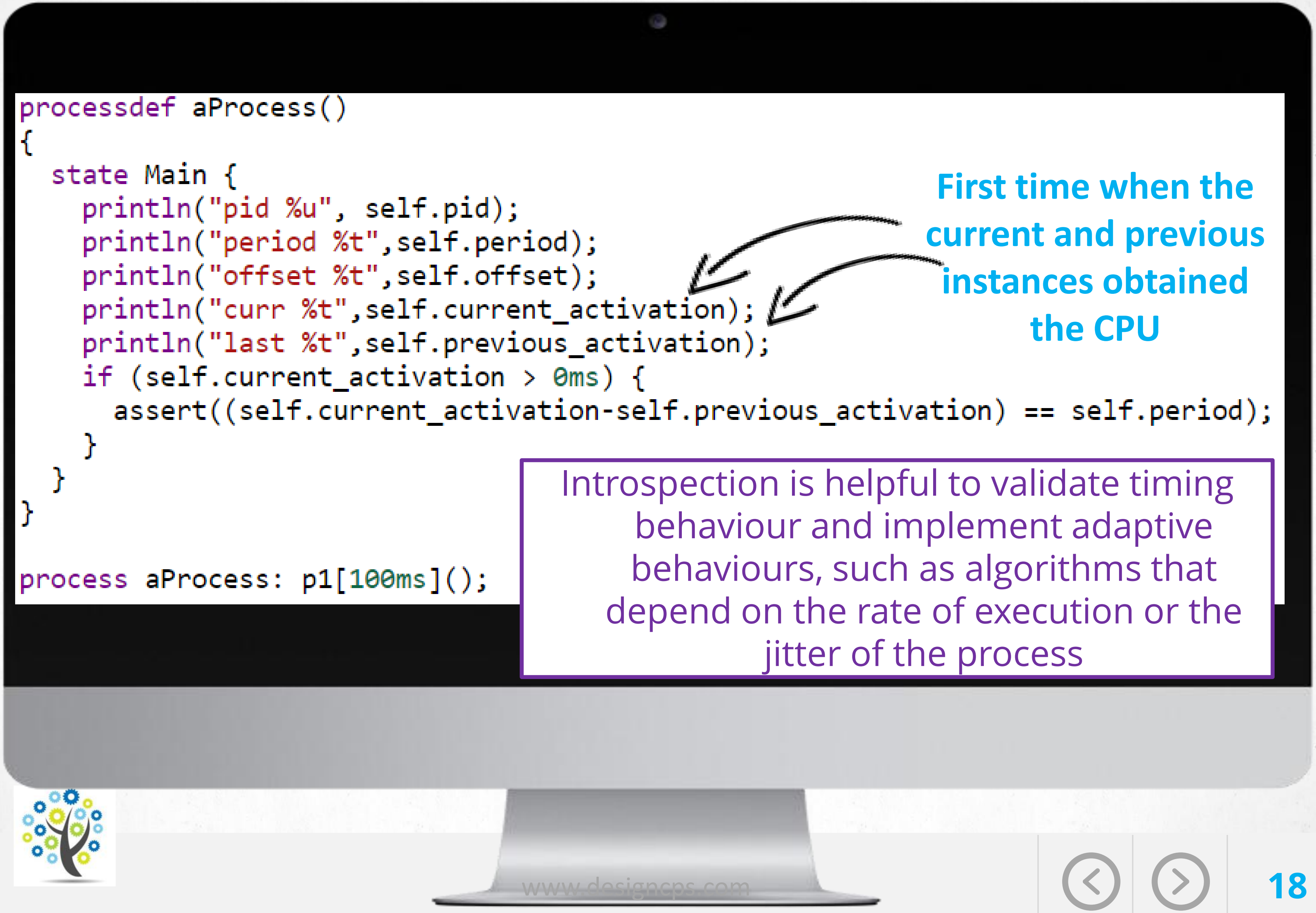
Solution to exercise #1-B)





```
processdef A_Process( )  {
    static var int32: i = 0;
    state State1 {
        i = i + 1;
        IO.println("In state1 with i=%d", i );
    } after (200ms) {
        i = 0;
    } to State2;

    state State2 {
        i = i + 1;
        IO.println("In state2 with i=%d", i );
    } on (i == 4) to State3;

    state State3 {
        i = i - 1;
        IO.println("In state3 with i=%d", i );
    } after (200ms) if (i <= 0) to State1;
}

process A_Process: p1[50ms]();
```

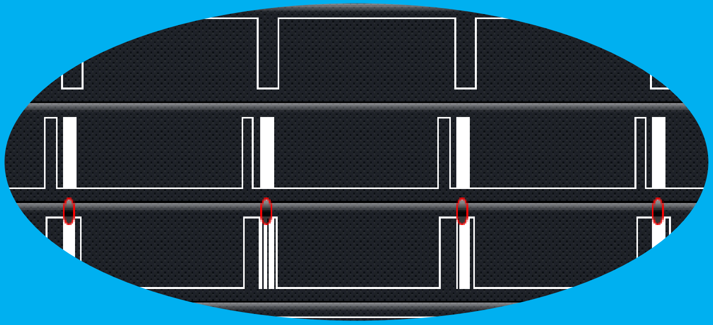
Process introspection



```
processdef aProcess()
{
  state Main {
    println("pid %u", self.pid);
    println("period %t",self.period);
    println("offset %t",self.offset);
    println("curr %t",self.current_activation);
    println("last %t",self.previous_activation);
    if (self.current_activation > 0ms) {
      assert((self.current_activation-self.previous_activation) == self.period);
    }
  }
}

process aProcess: p1[100ms]();
```

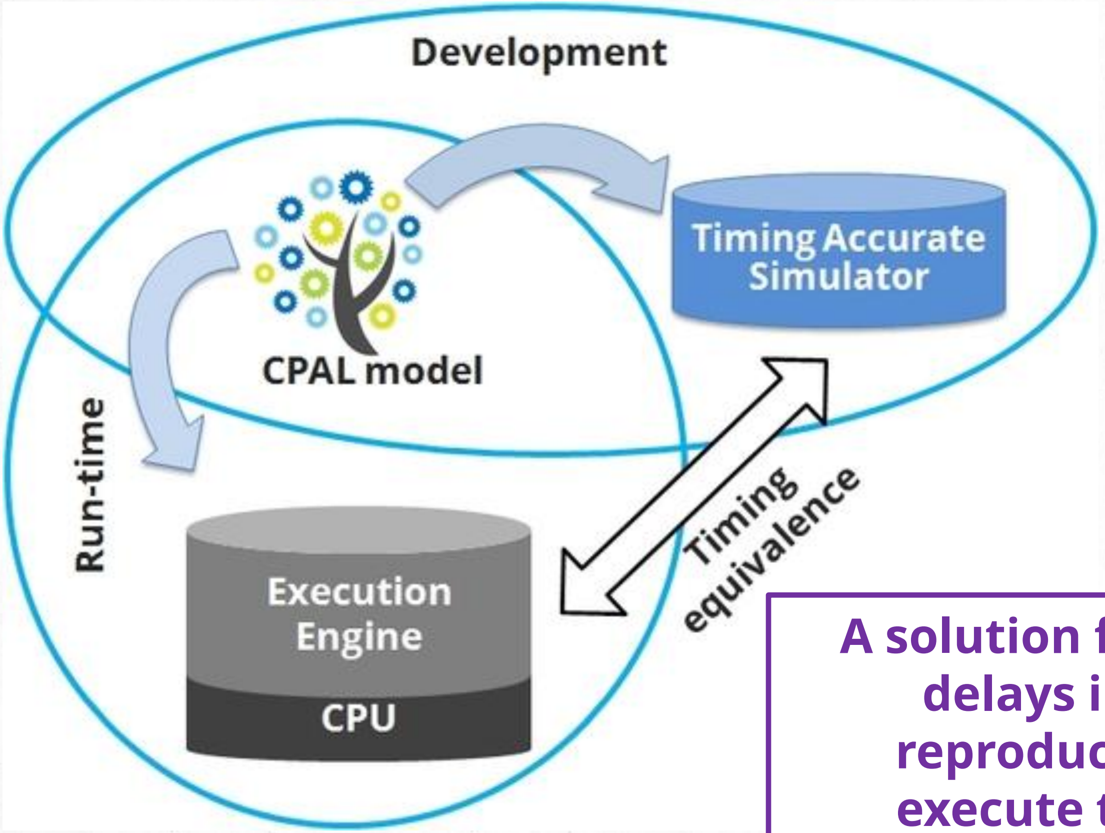
First time when the current and previous instances obtained the CPU

Introspection is helpful to validate timing behaviour and implement adaptive behaviours, such as algorithms that depend on the rate of execution or the jitter of the process



Simulation and Real-Time Execution Mode

Designer's objective: model behaves as the real-system



Buzzwords: "digital mockups", "digital twins"

A solution for timing is to inject delays in the model so as to reproduce the time it takes to execute the code on a specific platform

CPAL's 2 Execution Modes

Simulation mode

Development

- ✓ Execution is as fast as possible (e.g. periods are not respected)
- ✓ Code executed in zero time – except if stated otherwise with timing annotations
- ✓ CPAL interpreter is hosted by an OS
- ✓ No access to real I/Os

Real-Time mode

Deployment

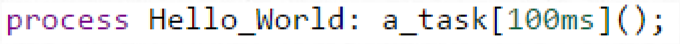
- ✓ Real-time execution
- ✓ Code (instructions, read/write I/Os) takes time to execute – depends on the platform
- ✓ CPAL can be executed on bare hardware or hosted by an OS

Overhead data on Freescale FRDM-K64F:

- ✓ max. activation jitter: 40us
- ✓ timer interrupt: 0.6us
- ✓ context switch overhead: 2us

Logical time vs physical time

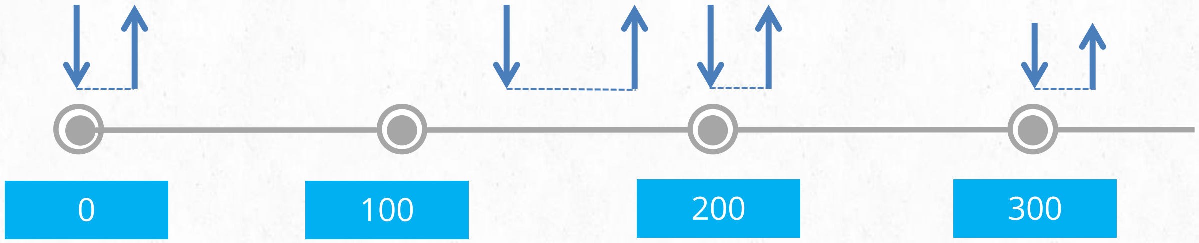
Let's consider this process



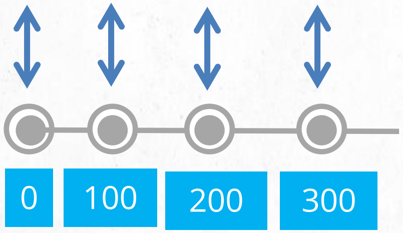
```
process Hello_World: a_task[100ms]();
```

Start of execution ↓ **End of execution** ↑

Question: where do the delays between start and end come from ?



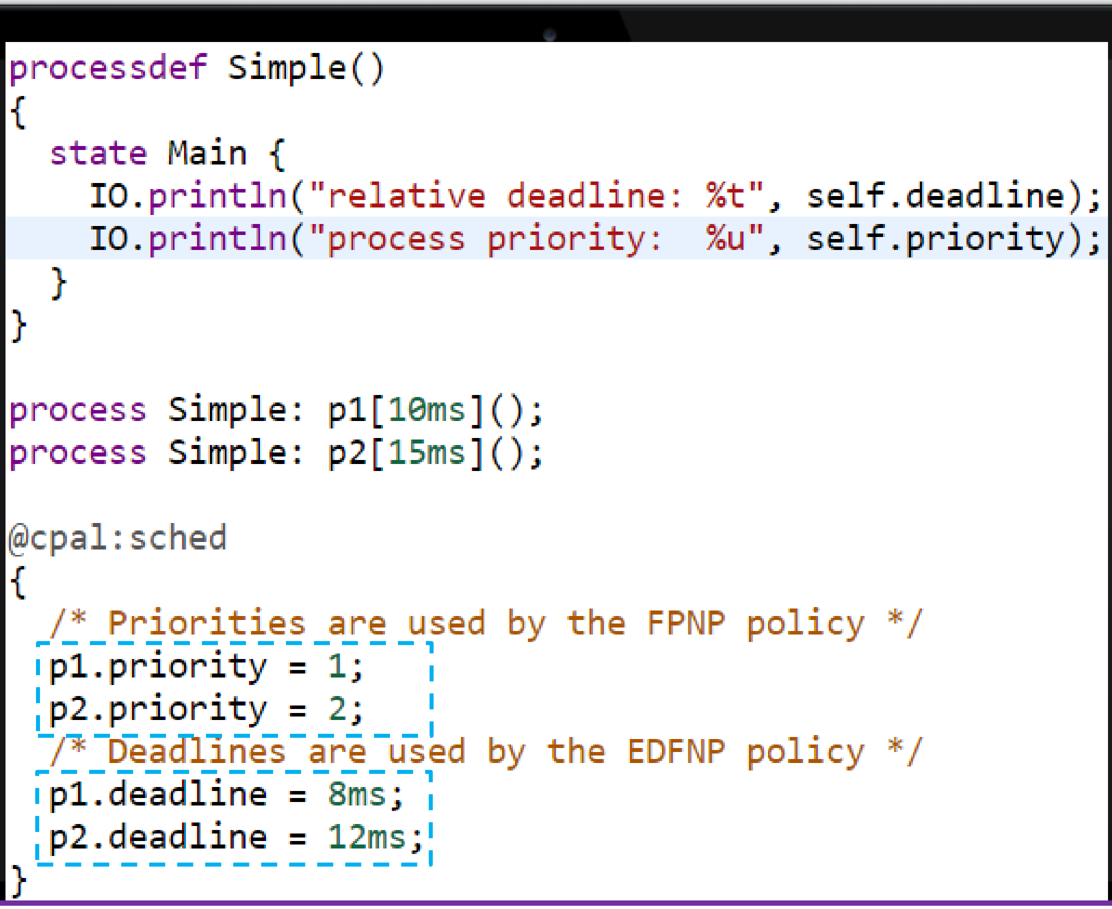
Real-time execution mode



Simulation mode

- ✓ In real-time mode, only physical time
- ✓ In simulation mode:
  - ○ order of events is ensured by a logical time
  - ○ Execution is as fast as possible (not in real-time) and code is executed in zero logical time

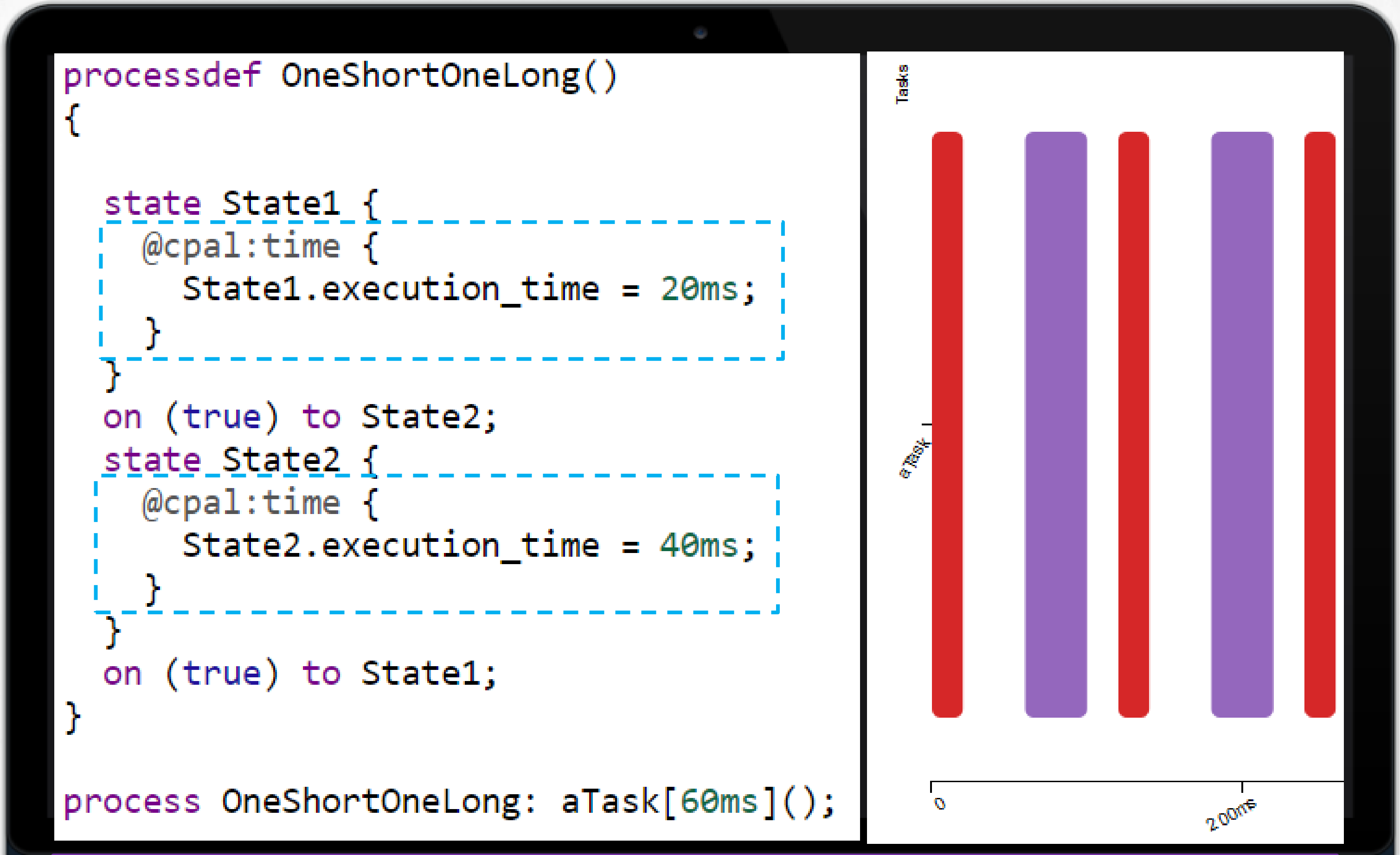
Real-time scheduling



```
processdef Simple()
{
  state Main {
    IO.println("relative deadline: %t", self.deadline);
    IO.println("process priority:  %u", self.priority);
  }
}

process Simple: p1[10ms]();
process Simple: p2[15ms]();

@cpal:sched
{
  /* Priorities are used by the FPNP policy */
  p1.priority = 1;
  p2.priority = 2;
  /* Deadlines are used by the EDFNP policy */
  p1.deadline = 8ms;
  p2.deadline = 12ms;
}
```

Scheduling policies: FIFO (by default), Fixed Priority non-preemptive (FPNP), Earliest Deadline First non-preemptive (EDFNP**)**

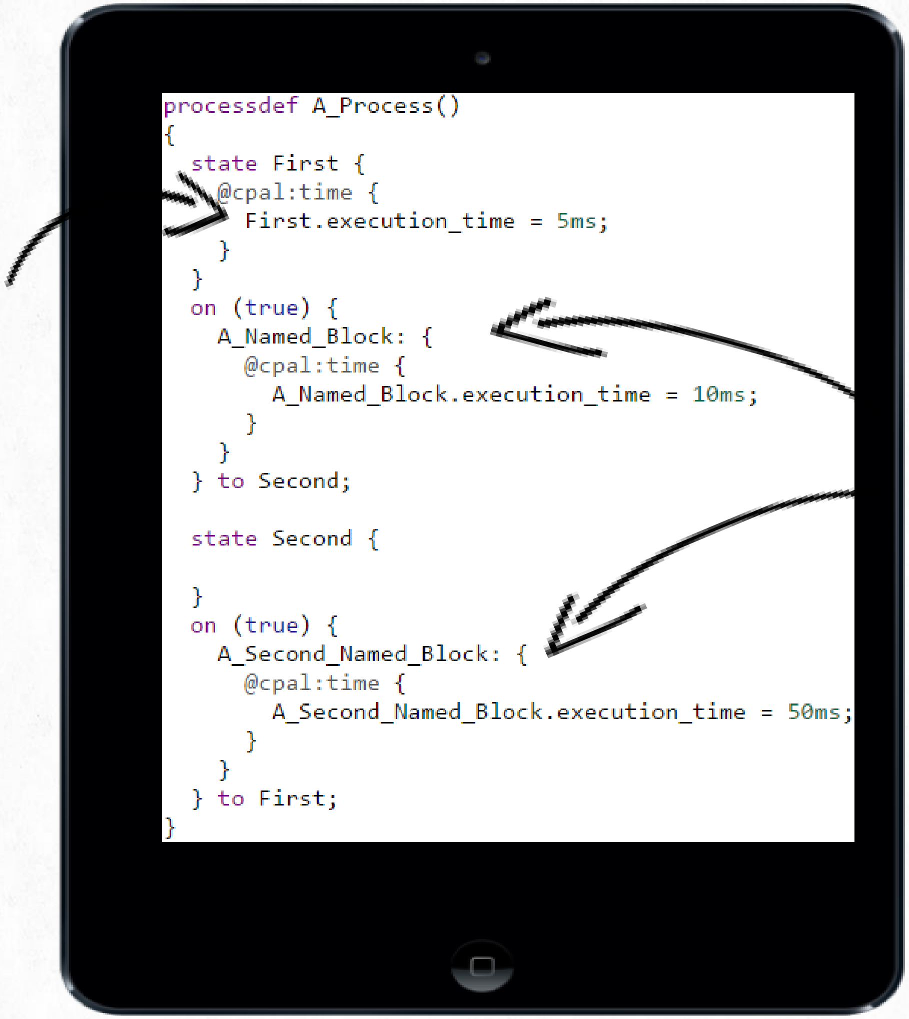
Simulating execution times



```
processdef OneShortOneLong()
{

  state State1 {
    @cpal:time {
      State1.execution_time = 20ms;
    }
  }
  on (true) to State2;
  state State2 {
    @cpal:time {
      State2.execution_time = 40ms;
    }
  }
  on (true) to State1;
}

process OneShortOneLong: aTask[60ms]();
```

Timing annotations can be derived by built-in monitoring facilities and are respected by the simulator

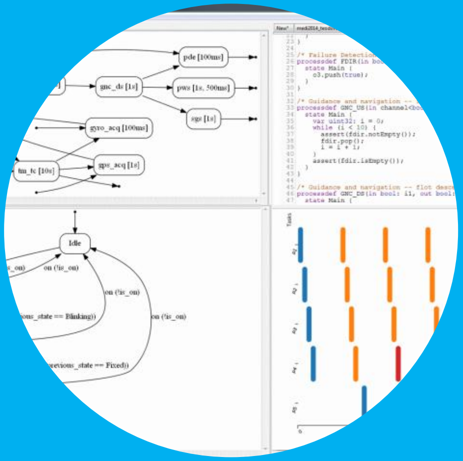
Execution time in transitions too



```
processdef A_Process()
{
  state First {
    @cpal:time {
      First.execution_time = 5ms;
    }
  }
  on (true) {
    A_Named_Block: {
      @cpal:time {
        A_Named_Block.execution_time = 10ms;
      }
    }
  } to Second;

  state Second {

  }
  on (true) {
    A_Second_Named_Block: {
      @cpal:time {
        A_Second_Named_Block.execution_time = 50ms;
      }
    }
  } to First;
}
```

Execution time of complete state

Execution time of the named blocks

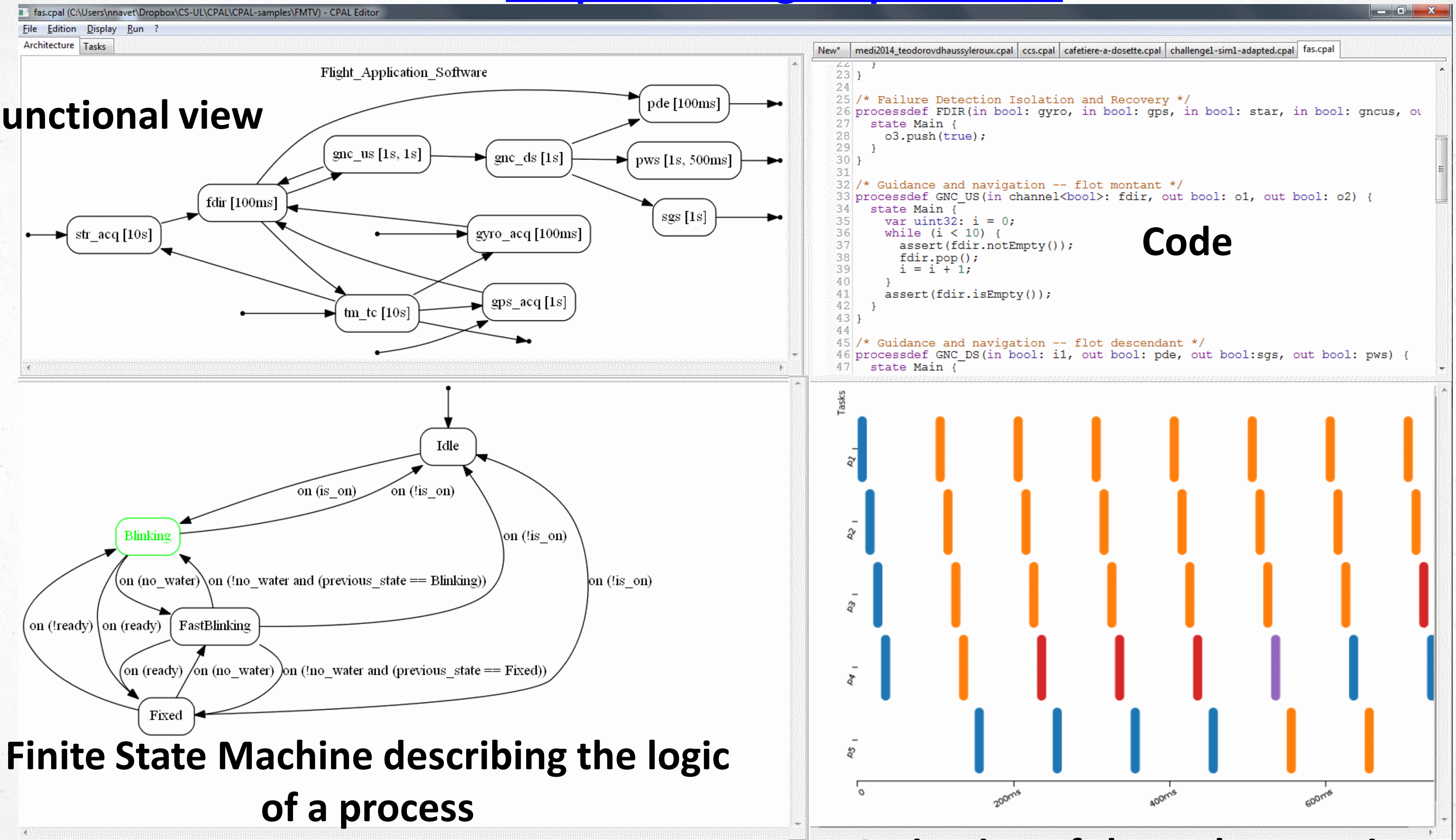


```
}

/* Failure Detection
processdef FDIR(in boo
  state Main {
    o3.push(true);
  }
}

/* Guidance and navigation -- 
processdef GNC_US(in channel<boo
  state Main {
    var uint32: i = 0;
    while (i < 10) {
      assert(fdir.notEmpty());
      fdir.pop();
      i = i + 1;
    }
    assert(fdir.isEmpty());
  }
}

/* Guidance and navigation -- flot desc
processdef GNC_DS(in bool: i1, out bool:
  state Main {
```

The CPAL development environment

Complete development environment

from http://designcps.com



Functional view

```
  }
}

/* Failure Detection Isolation and Recovery */
processdef FDIR(in bool: gyro, in bool: gps, in bool: star, in bool: gncus, ou
  state Main {
    o3.push(true);
  }
}

/* Guidance and navigation -- flot montant */
processdef GNC_US(in channel<bool>: fdir, out bool: o1, out bool: o2) {
  state Main {
    var uint32: i = 0;
    while (i < 10) {
      assert(fdir.notEmpty());
      fdir.pop();
      i = i + 1;
    }
    assert(fdir.isEmpty());
  }
}

/* Guidance and navigation -- flot descendant */
processdef GNC_DS(in bool: i1, out bool: pde, out bool:sgs, out bool: pws) {
  state Main {
```

Code

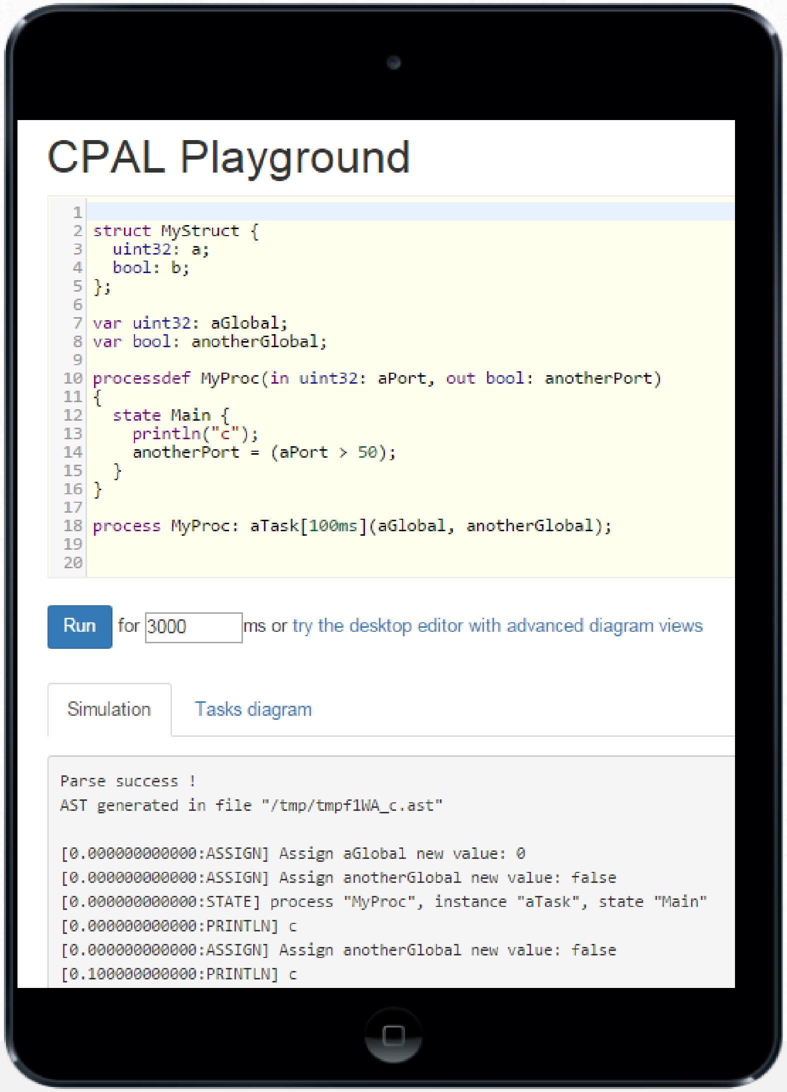
Finite State Machine describing the logic of a process

Activation of the tasks over time

Zero install with the CPAL- Playground

http://designcps.com/cpal-playground

- + no install, run from everywhere
- + nice to experiment with the example programs available on-line



CPAL Playground

```
struct MyStruct {
  uint32: a;
  bool: b;
};

var uint32: aGlobal;
var bool: anotherGlobal;

processdef MyProc(in uint32: aPort, out bool: anotherPort)
{
  state Main {
    println("c");
    anotherPort = (aPort > 50);
  }
}

process MyProc: aTask[100ms](aGlobal, anotherGlobal);

```

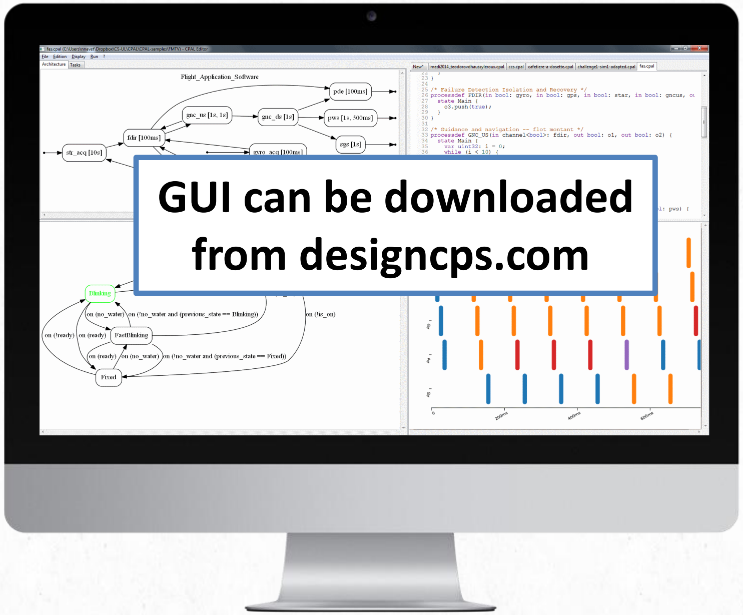
Run
for 3000 ms or try the desktop editor with advanced diagram views
Simulation
Tasks diagram

```
Parse success !
AST generated in file "/tmp/tmpf1WA_c.ast"

[0.000000000000:ASSIGN] Assign aGlobal new value: 0
[0.000000000000:ASSIGN] Assign anotherGlobal new value: false
[0.000000000000:STATE] process "MyProc", instance "aTask", state "Main"
[0.000000000000:PRINTLN] c
[0.000000000000:ASSIGN] Assign anotherGlobal new value: false
[0.100000000000:PRINTLN] c
```

- – no way to change variable values at run-time or run scenarios
- – no graphical representation of FSMs and functional architecture
- – No real-time mode
- – Not embedded programming !

CPAL-Editor on all platforms with Java Web Start - https://www.designcps.com/binaries/

+ the graphical editor on all platforms with Java (Raspberry, MacOS, etc)


fas.cpal (C:\Users\nnavet\Dropbox\CS-UL\CPAL\CPAL-samples\FMTV) - CPAL Editor
File Edition Display Run ?
Architecture Tasks
New* medi2014_teodorovdhaussyleroux.cpal ccs.cpal cafetiere-a-dosette.cpal challenge1-siml-adapted.cpal fas.cpal
Flight_Application_Software
pde [100ms]
gnc_us [1s, 1s]
gnc_ds [1s]
pws [1s, 500ms]
fdir [100ms]
sgs [1s]
str_acq [10s]
gyro_acq [100ms]
Blinking
FastBlinking
Fixed
on (no_water)
on (!no_water and (previous_state == Blinking))
on (!is_on)
on (!ready)
on (ready)
on (no_water)
on (!no_water and (previous_state == Fixed))
P3
P4
P5
0
200ms
400ms
600ms

```
}

/* Failure Detection Isolation and Recovery */
processdef FDIR(in bool: gyro, in bool: gps, in bool: star, in bool: gncus, ou
  state Main {
    o3.push(true);
  }
}

/* Guidance and navigation -- flot montant */
processdef GNC_US(in channel<bool>: fdir, out bool: o1, out bool: o2) {
  state Main {
    var uint32: i = 0;
    while (i < 10) {
```

GUI can be downloaded from designcps.com

- Have to add security exception to Java

- Have to manually install Graphwiz and command-line tools

Command-line tools overview

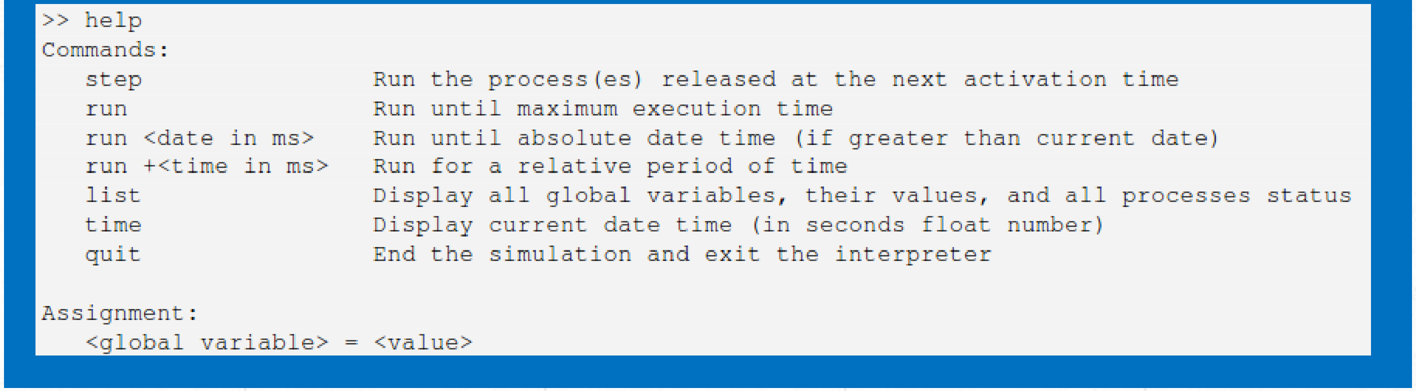
1 Parse

```
$ cpal_parser input.cpal output.ast
```

→ .ast file created on success, parse errors listed otherwise

2 Execute

Interactive mode within the interpreter



```
>> help
Commands:
   step                   Run the process(es) released at the next activation time
   run                    Run until maximum execution time
   run <date in ms>       Run until absolute date time (if greater than current date)
   run +<time in ms>      Run for a relative period of time
   list                   Display all global variables, their values, and all processes status
   time                   Display current date time (in seconds float number)
   quit                   End the simulation and exit the interpreter

Assignment:
   <global variable> = <value>
```

Non-interactive mode, e.g. on embedded Linux or Raspberry

```
$ cpal_interpreter -i -q input.ast
```

-i : interactive mode toggled on, -q : quiet mode (less verbose)

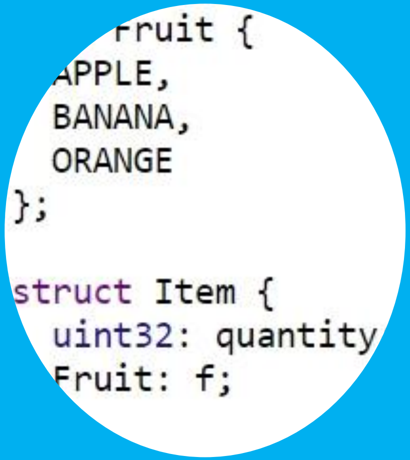
Available CPAL ports

PLATFORMS	EXECUTION MODE	HOSTED BY AN OS ?	ACCESS TO HARDWARE ?	EXECUTABLE
Windows 32/64bit	Simulation	Yes	No	*cpal_interpreter*
Windows 32/64bit	Real-time	Yes	No	*cpal_interpreter_winmbed*
Linux 64 bit	Simulation	Yes	No	*cpal_interpreter*
Linux 64 bit	Real-Time	Yes	Yes	*cpal_interpreter_linuxmbed*
Mac OS X	Simulation	Yes	No	*cpal_interpreter*
Freescale FRDM-K64F	Real-Time	**No**	Yes	NA, an image is uploaded
Raspberry Pi (Raspbian)	Real-Time and Simulation	Yes	Yes	*cpal_interpreter_raspberry*

Best real-time performance

Ex. of interpreter command lines

- `cpal_interpreter my_program.ast`: execute indefinitely in **simulation mode** and non-interactive mode
- `cpal_interpreter -i -q my_program.ast`: execute in simulation mode and **interactive mode** and **quiet mode**
- `cpal_interpreter -p NPFP my_program.ast`: execute with processes scheduled under the **non-preemptive Fixed Priority policy** instead of FIFO, **and non-interactive mode**
- `cpal_interpreter_linuxmbded -q my_program.ast`: execute in Linux, indefinitely in **real-time mode**, non-interactive mode and quiet mode
- `cpal_interpreter_winmbded -r -i -s scenario.sce my_program.ast` : execute on Windows in real-time mode the **scenario defined in file** scenario.sce then remain in the interpreter in interactive mode
- `cpal_interpreter --silent --time 5000 my_program.ast`: execution in simulation and non-interactive mode **during 5000ms**, with no outputs to the console
- `cpal_interpreter_raspberry -r -v --stats --time 5000 my_program.ast`: execute on Raspberry in real-time, non-verbose and non-interactive mode during 5000ms with the **monitoring of the Worst-Case Execution Times (WCET)** of the processes



```
     Fruit {
  APPLE,
  BANANA,
  ORANGE
};

struct Item {
  uint32: quantity
  Fruit: f;
```

Data types in CPAL

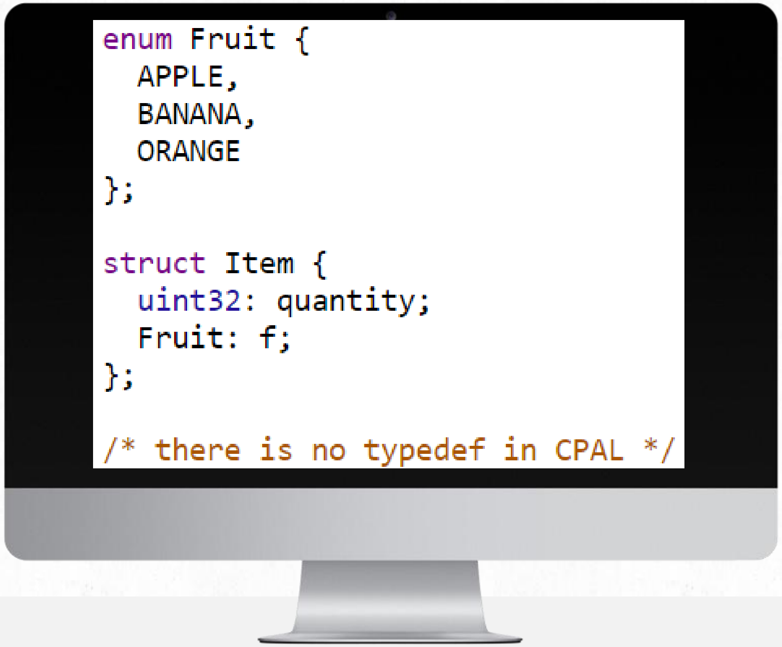
Overview on types

- ✓ No untyped data and no pointer in CPAL
- ✓ No memory is dynamically allocated / freed at run-time
- ✓ **Basic types**: `bool`, `uint8`, `int64`, `float32`, `time64`, etc
- ✓ **User-defined types**: array, enum and structure
- ✓ **Collections**: stacks and queues
- ✓ **Process** is a built-in type for an activity of the system (similar to threads or tasks in other contexts)

CPAL is a strongly typed language – conversions between types have to be explicit: `uint8.as(x)`, `uint16.as(x)`, `uint32.as(x)`, `uint64.as(x)`, `int8.as(x)`, `int16.as(x)`, `int32.as(x)`, `int64.as(x)`, `time64.as(x)`, `bool.as(x)`. **Binary reinterpretation** through `type.cast(x)`

Overview on types cont'd

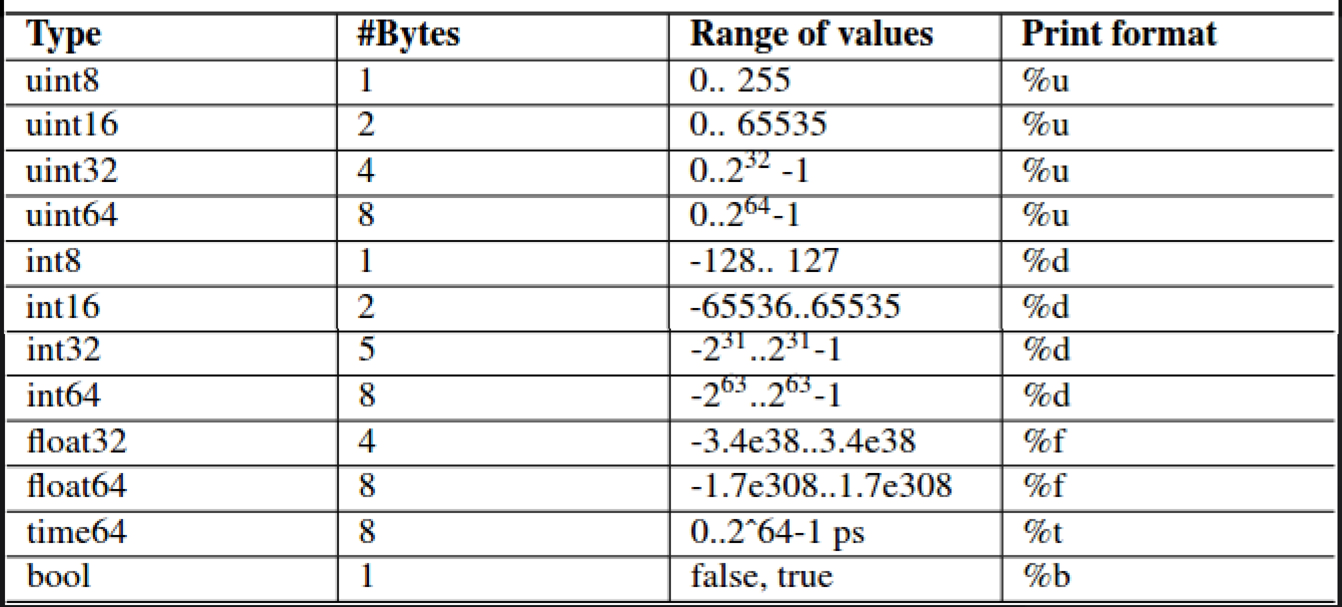
- Variables of basic types and user-defined types are all initialized to zero at creation (i.e., all bits are set to zero)
- Arrays are uni-dimensional
- No char or string type but writing to terminal is possible with `IO.print()` and `IO.println()` functions
- Integers can be specified in decimal or hexadecimal (0xA1E = 2590)



```
enum Fruit {
  APPLE,
  BANANA,
  ORANGE
};

struct Item {
  uint32: quantity;
  Fruit: f;
};

/* there is no typedef in CPAL */
```

Example of Complex types

Primitive data types

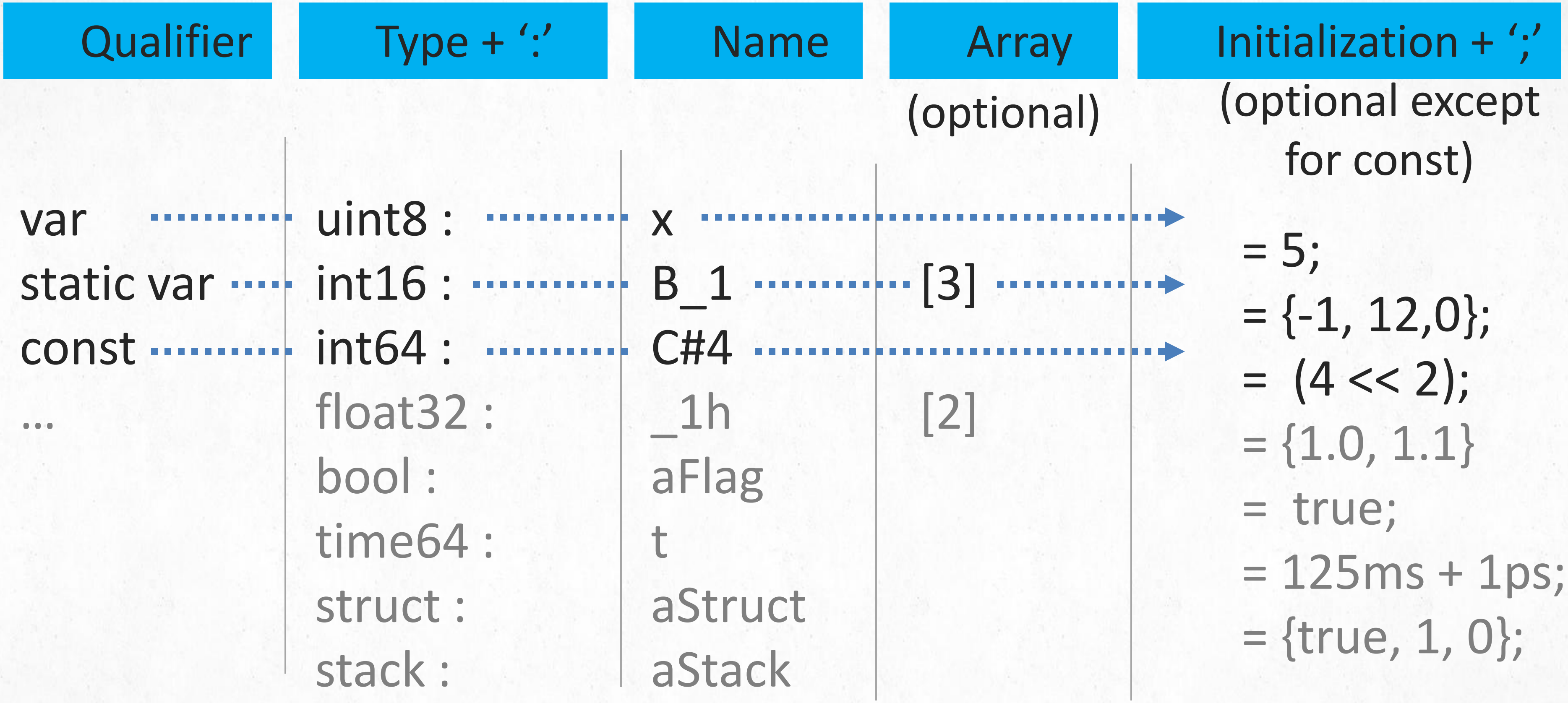


Type	#Bytes	Range of values	Print format
uint8	1	0.. 255	%u
uint16	2	0.. 65535	%u
uint32	4	$0..2^{32}$ -1	%u
uint64	8	$0..2^{64}$-1	%u
int8	1	-128.. 127	%d
int16	2	-65536..65535	%d
int32	5	$-2^{31}..2^{31}$-1	%d
int64	8	$-2^{63}..2^{63}$-1	%d
float32	4	-3.4e38..3.4e38	%f
float64	8	-1.7e308..1.7e308	%f
time64	8	0..2ˆ64-1 ps	%t
bool	1	false, true	%b

Min and max value of each type:
`type.FIRST` **and** `type.LAST`

Min and max between two variables:
`type.min(a,b)` **and** `type.max(a,b)`

Declaring a data



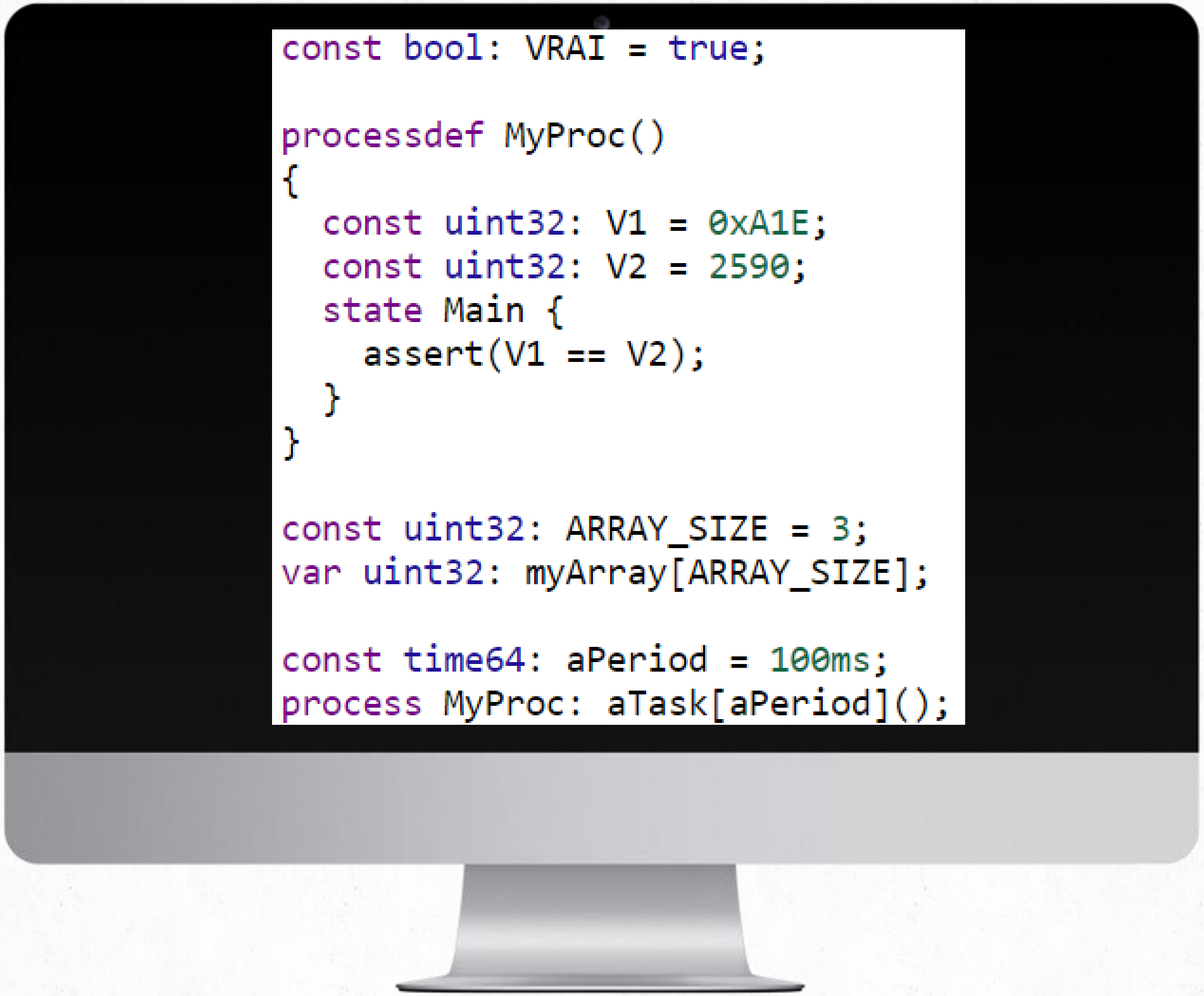
Qualifier	Type + ‘:’	Name	Array	Initialization + ‘;’
			(optional)	(optional except for const)
var	uint8 :	x		= 5;
static var	int16 :	B_1	[3]	= {-1, 12,0};
const	int64 :	C#4		= (4 << 2);
…	float32 :	_1h	[2]	= {1.0, 1.1}
	bool :	aFlag		= true;
	time64 :	t		= 125ms + 1ps;
	struct :	aStruct		= {true, 1, 0};
	stack :	aStack		

Scientific notations for `float32`: 3.43e5, 3.43e+5, 3.43E+5, 3.43e-5 and 3.43E-5.

Declaration statements

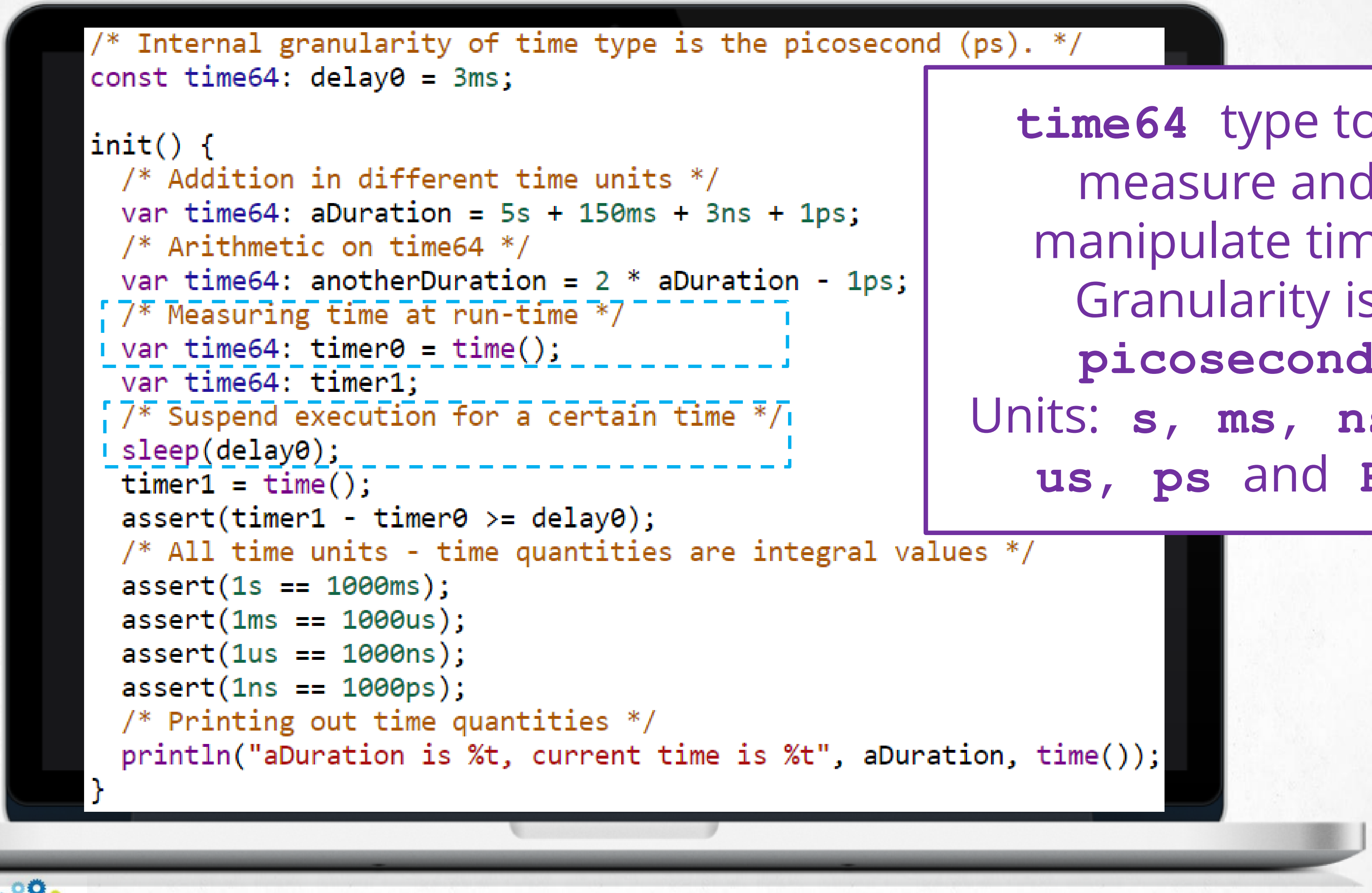
- Scope of declaration
  - global variable
  - local to a process
  - local to the code of a state or local the code of transition
  - local to the `init()` function
  - Local to a named block
- But always at the beginning of the scope!
- The visibility of a variable extends throughout the scope (e.g. a process-level variable is known in the code of all the states and transitions of the process)
- In addition to normal variables, there are constants and static variables – similar as in C (static var. only allowed at process-level)
- What holds for basic types, holds for structures, enums and collections

A focus on constants



```
const bool: VRAI = true;

processdef MyProc()
{
  const uint32: V1 = 0xA1E;
  const uint32: V2 = 2590;
  state Main {
    assert(V1 == V2);
  }
}

const uint32: ARRAY_SIZE = 3;
var uint32: myArray[ARRAY_SIZE];

const time64: aPeriod = 100ms;
process MyProc: aTask[aPeriod]();
```

Working with time



```
/* Internal granularity of time type is the picosecond (ps). */
const time64: delay0 = 3ms;

init() {
  /* Addition in different time units */
  var time64: aDuration = 5s + 150ms + 3ns + 1ps;
  /* Arithmetic on time64 */
  var time64: anotherDuration = 2 * aDuration - 1ps;
  /* Measuring time at run-time */
  var time64: timer0 = time();
  var time64: timer1;
  /* Suspend execution for a certain time */
  sleep(delay0);
  timer1 = time();
  assert(timer1 - timer0 >= delay0);
  /* All time units - time quantities are integral values */
  assert(1s == 1000ms);
  assert(1ms == 1000us);
  assert(1us == 1000ns);
  assert(1ns == 1000ps);
  /* Printing out time quantities */
  println("aDuration is %t, current time is %t", aDuration, time());
}
```

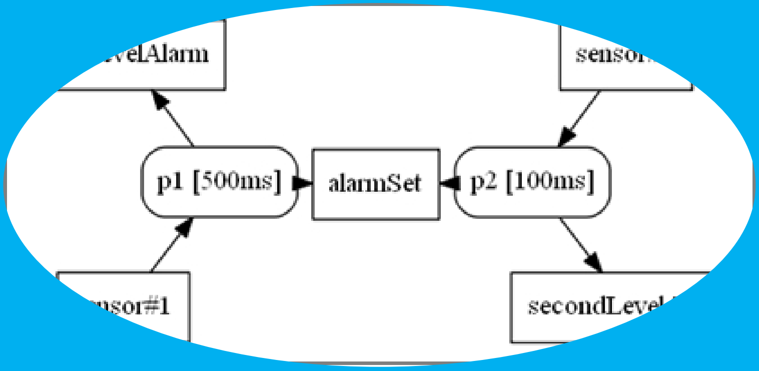
time64 type to measure and manipulate time. Granularity is **picosecond**

Units: **s, ms, ns, us, ps** and **Hz**

CPAL facilitates the writing of correct code

- ✓ Strongly typed language: conversions must be explicit
- ✓ Designed with simplicity in mind: no convoluted constructs
- ✓ No dynamic memory
- ✓ No pointers
- ✓ All processes are known before run-time - workload is bounded
- ✓ Built-in code execution time monitoring support
- ✓ Built-in loop over construct to prevent “off-by-one” errors when iterating over collections
- ✓ Testing the equality of floating-point numbers is forbidden
- ✓ Etc…

Inspired from Misra C and CERT C coding standards

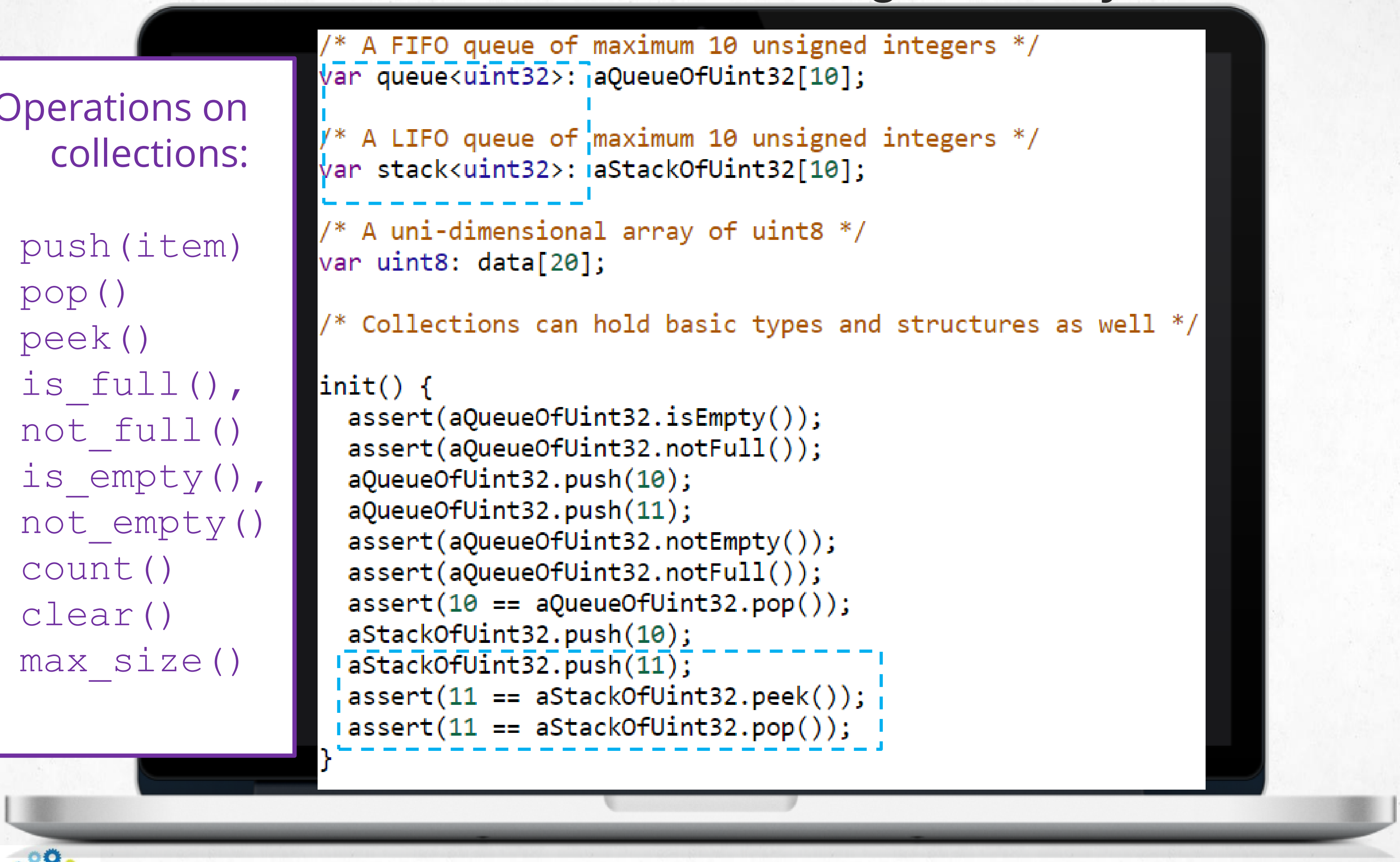


Collections and inter-process communication

Overview on collections
FIFO vs LIFO buffering vs arrays

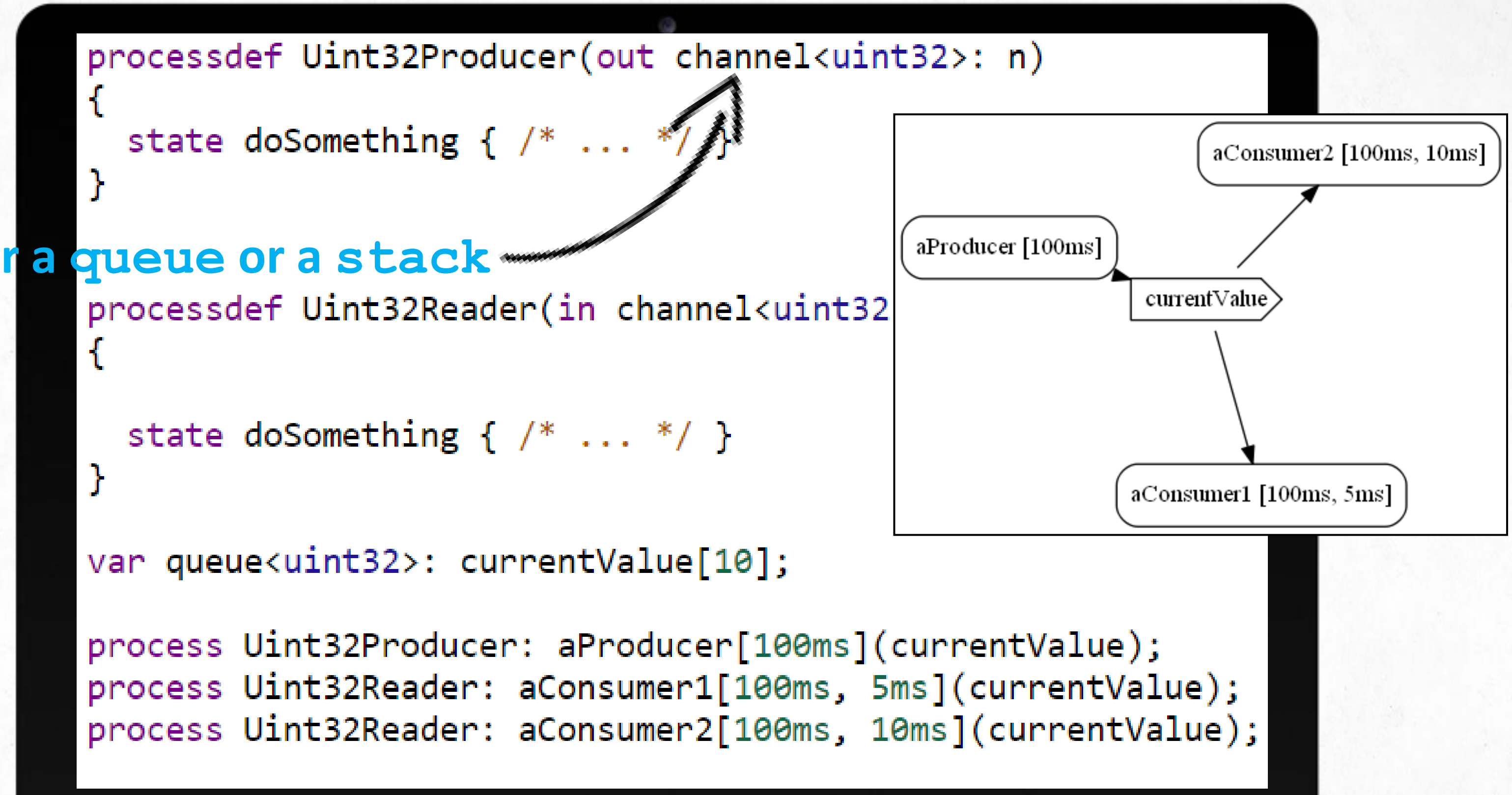
Operations on collections:

- ✓ push(item)
- ✓ pop()
- ✓ peek()
- ✓ is_full(),
- ✓ not_full()
- ✓ is_empty(),
- ✓ not_empty()
- ✓ count()
- ✓ clear()
- ✓ max_size()



```
/* A FIFO queue of maximum 10 unsigned integers */
var queue<uint32>: aQueueOfUint32[10];

/* A LIFO queue of maximum 10 unsigned integers */
var stack<uint32>: aStackOfUint32[10];

/* A uni-dimensional array of uint8 */
var uint8: data[20];

/* Collections can hold basic types and structures as well */

init() {
  assert(aQueueOfUint32.isEmpty());
  assert(aQueueOfUint32.notFull());
  aQueueOfUint32.push(10);
  aQueueOfUint32.push(11);
  assert(aQueueOfUint32.notEmpty());
  assert(aQueueOfUint32.notFull());
  assert(10 == aQueueOfUint32.pop());
  aStackOfUint32.push(10);
  aStackOfUint32.push(11);
  assert(11 == aStackOfUint32.peek());
  assert(11 == aStackOfUint32.pop());
}
```

Communication channels



```
processdef Uint32Producer(out channel<uint32>: n)
{
  state doSomething { /* ... */ }
}

processdef Uint32Reader(in channel<uint32
{

  state doSomething { /* ... */ }
}

var queue<uint32>: currentValue[10];

process Uint32Producer: aProducer[100ms](currentValue);
process Uint32Reader: aConsumer1[100ms, 5ms](currentValue);
process Uint32Reader: aConsumer2[100ms, 10ms](currentValue);
```

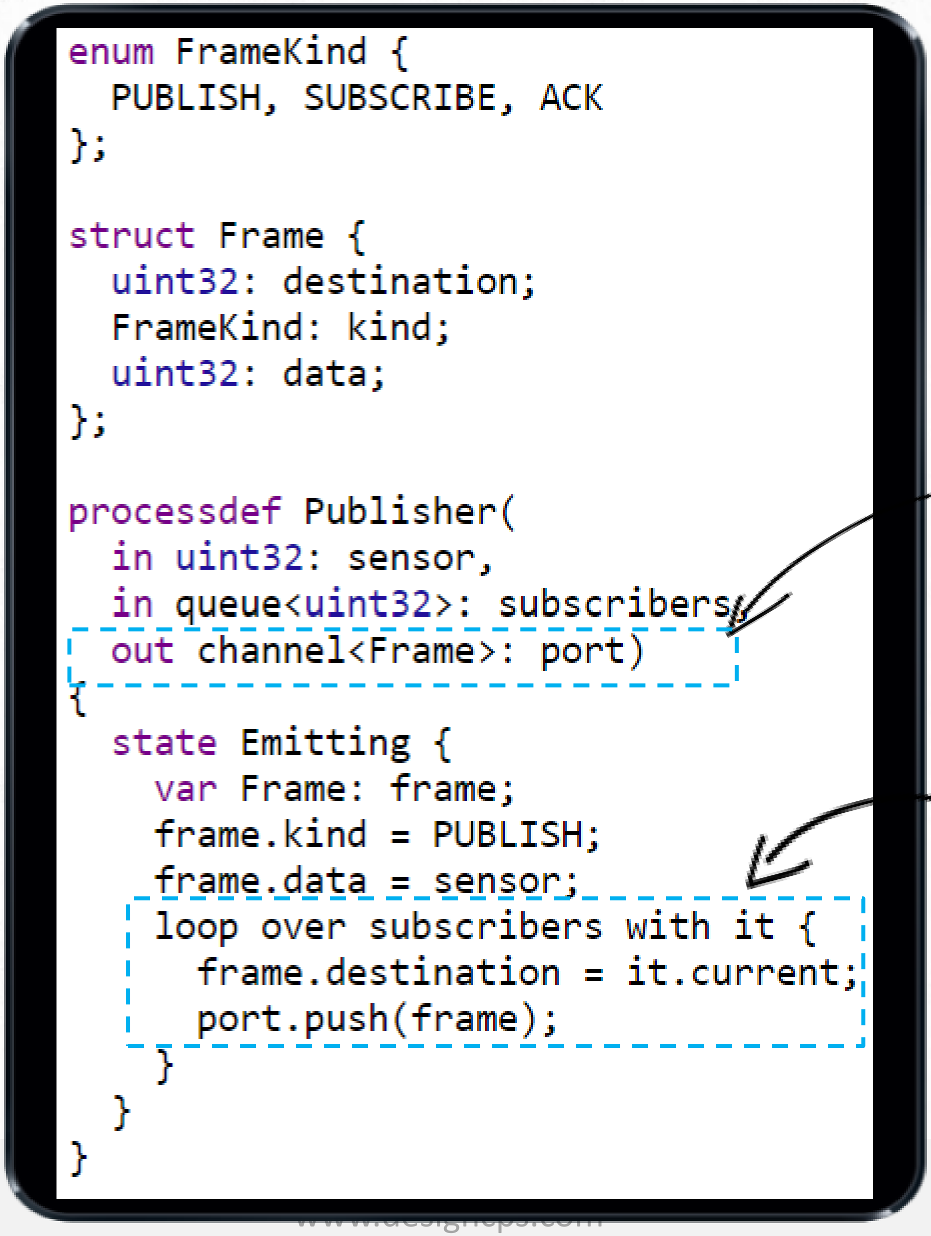
Can be either a queue or a stack

Inter-process communication can be done through normal global variables as well (*i.e.*, overwrite semantics)

Iterating on collections (1/2)

Constructs for iterators:

- ✓ it.index
- ✓ it.current
- ✓ it.is_last
- ✓ remove_current (continue|restart|break)
- ✓ continue
- ✓ break



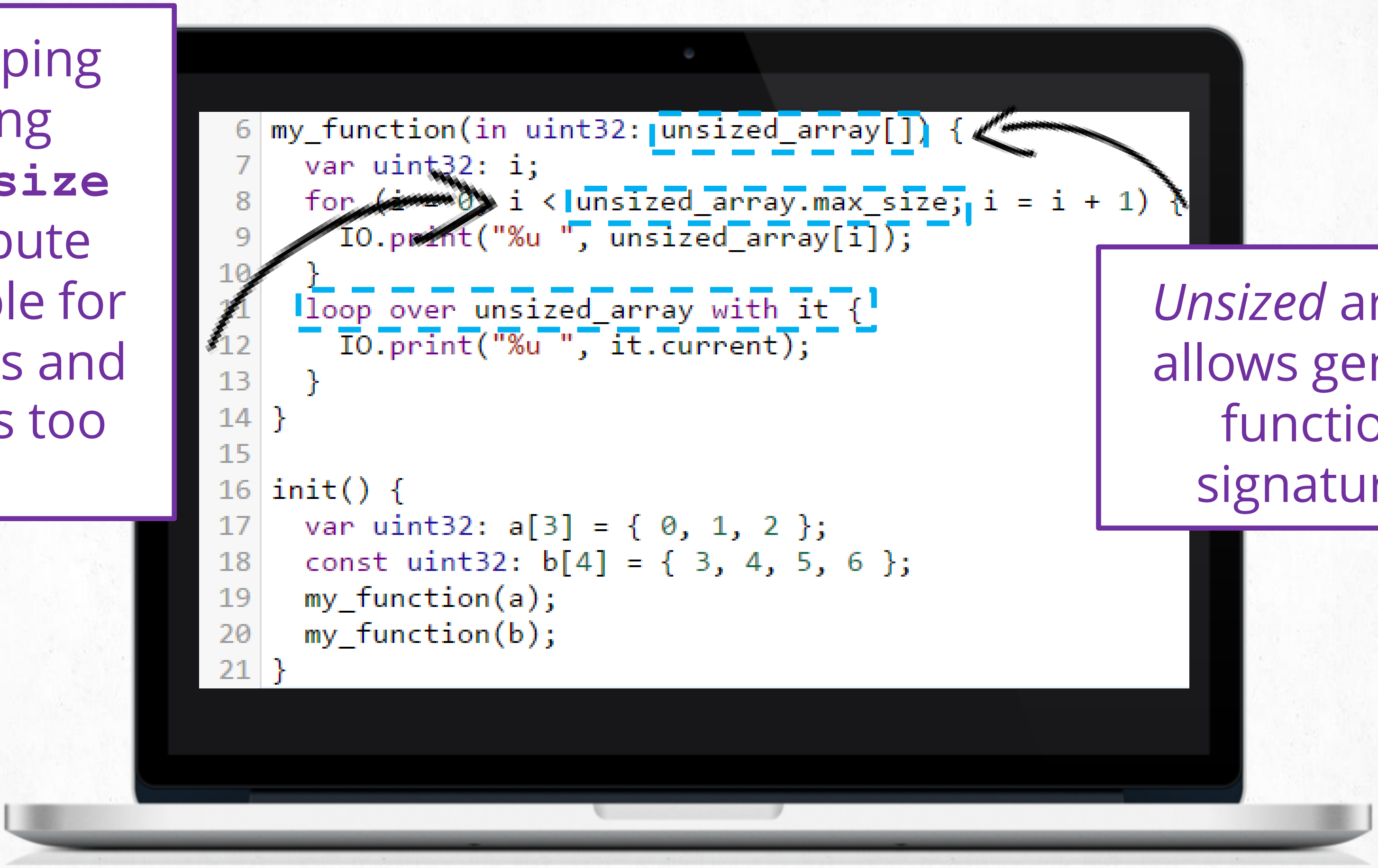
```
enum FrameKind {
  PUBLISH, SUBSCRIBE, ACK
};

struct Frame {
  uint32: destination;
  FrameKind: kind;
  uint32: data;
};

processdef Publisher(
  in uint32: sensor,
  in queue<uint32>: subscribers,
  out channel<Frame>: port)
{
  state Emitting {
    var Frame: frame;
    frame.kind = PUBLISH;
    frame.data = sensor;
    loop over subscribers with it {
      frame.destination = it.current;
      port.push(frame);
    }
  }
}
```

Works whatever the collections used for the communication channel

Goes through the entire collection, iterator it does not need to be declared

Iterating on (unsized) arrays

Sweeping using **`max_size`** attribute possible for queues and stacks too



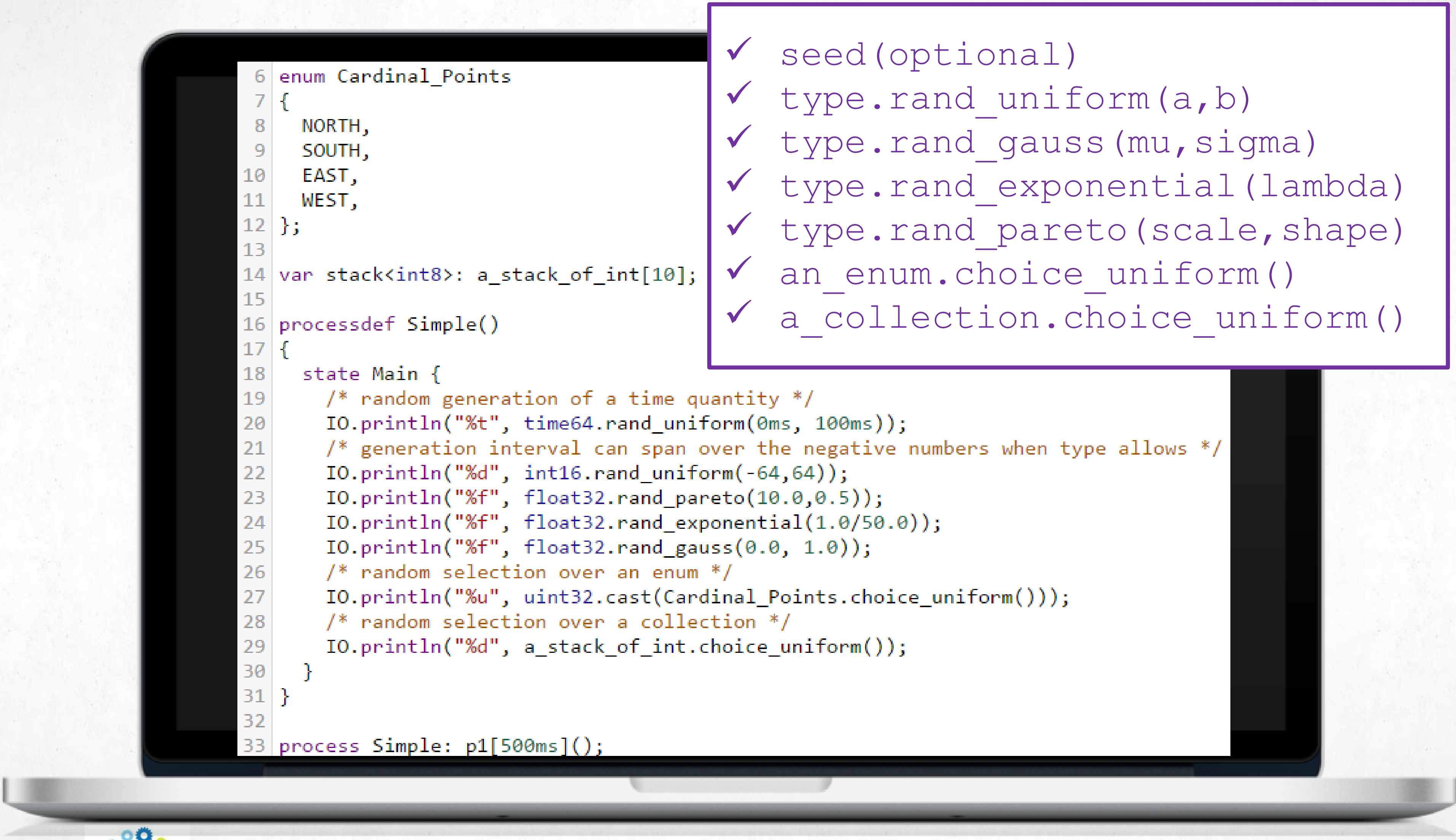
```
my_function(in uint32: unsized_array[]) {
  var uint32: i;
  for (i = 0; i < unsized_array.max_size; i = i + 1) {
    IO.print("%u ", unsized_array[i]);
  }
  loop over unsized_array with it {
    IO.print("%u ", it.current);
  }
}

init() {
  var uint32: a[3] = { 0, 1, 2 };
  const uint32: b[4] = { 3, 4, 5, 6 };
  my_function(a);
  my_function(b);
}
```

Unsized arrays allows generic function signatures

CPAL for simulation

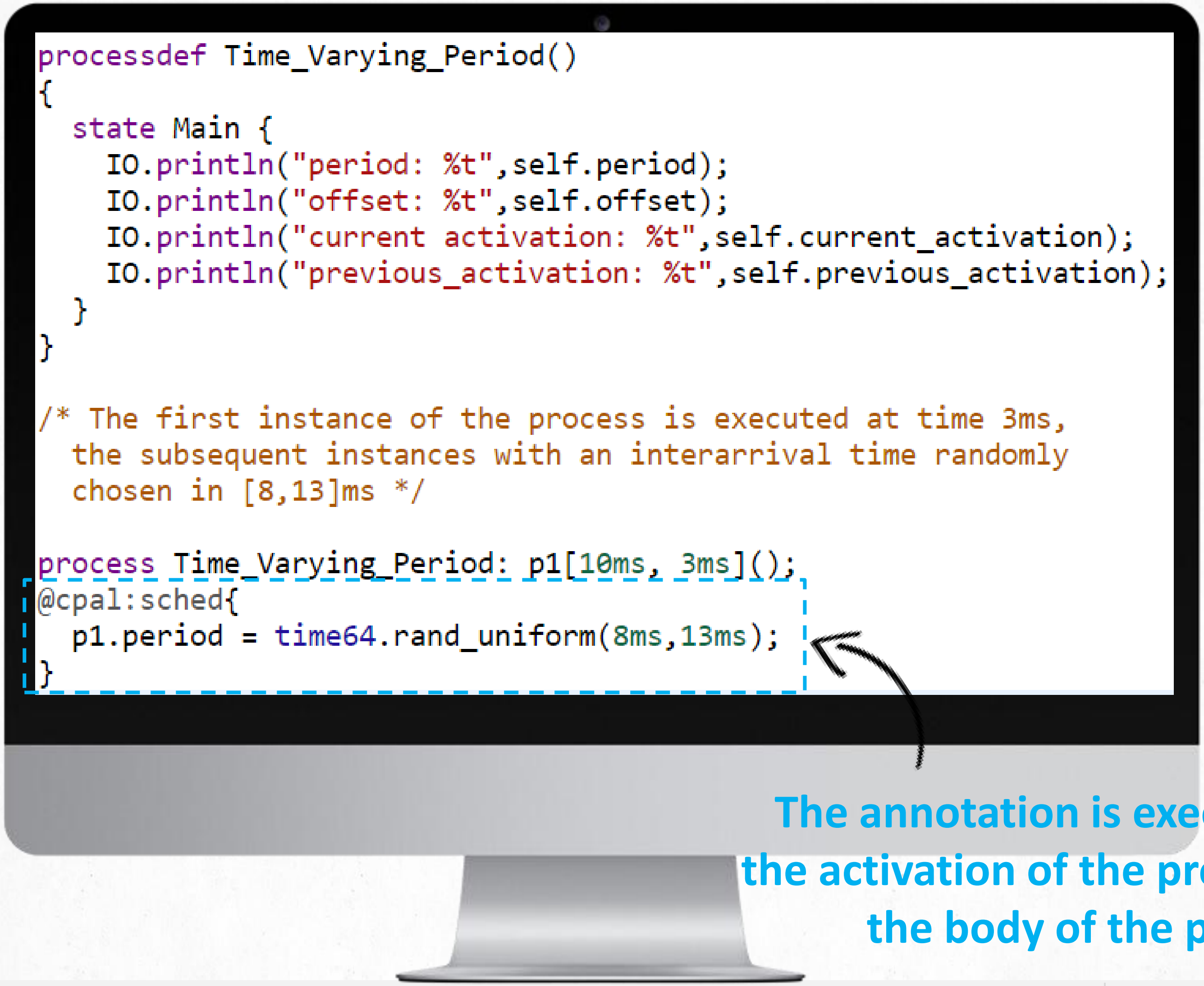
Pseudo-random numbers

- ✓ seed(optional)
- ✓ type.rand_uniform(a,b)
- ✓ type.rand_gauss(mu,sigma)
- ✓ type.rand_exponential(lambda)
- ✓ type.rand_pareto(scale,shape)
- ✓ an_enum.choice_uniform()
- ✓ a_collection.choice_uniform()



```
enum Cardinal_Points
{
  NORTH,
  SOUTH,
  EAST,
  WEST,
};

var stack<int8>: a_stack_of_int[10];

processdef Simple()
{
  state Main {
    /* random generation of a time quantity */
    IO.println("%t", time64.rand_uniform(0ms, 100ms));
    /* generation interval can span over the negative numbers when type allows */
    IO.println("%d", int16.rand_uniform(-64,64));
    IO.println("%f", float32.rand_pareto(10.0,0.5));
    IO.println("%f", float32.rand_exponential(1.0/50.0));
    IO.println("%f", float32.rand_gauss(0.0, 1.0));
    /* random selection over an enum */
    IO.println("%u", uint32.cast(Cardinal_Points.choice_uniform()));
    /* random selection over a collection */
    IO.println("%d", a_stack_of_int.choice_uniform());
  }
}

process Simple: p1[500ms]();
```

Varying process inter-arrival times

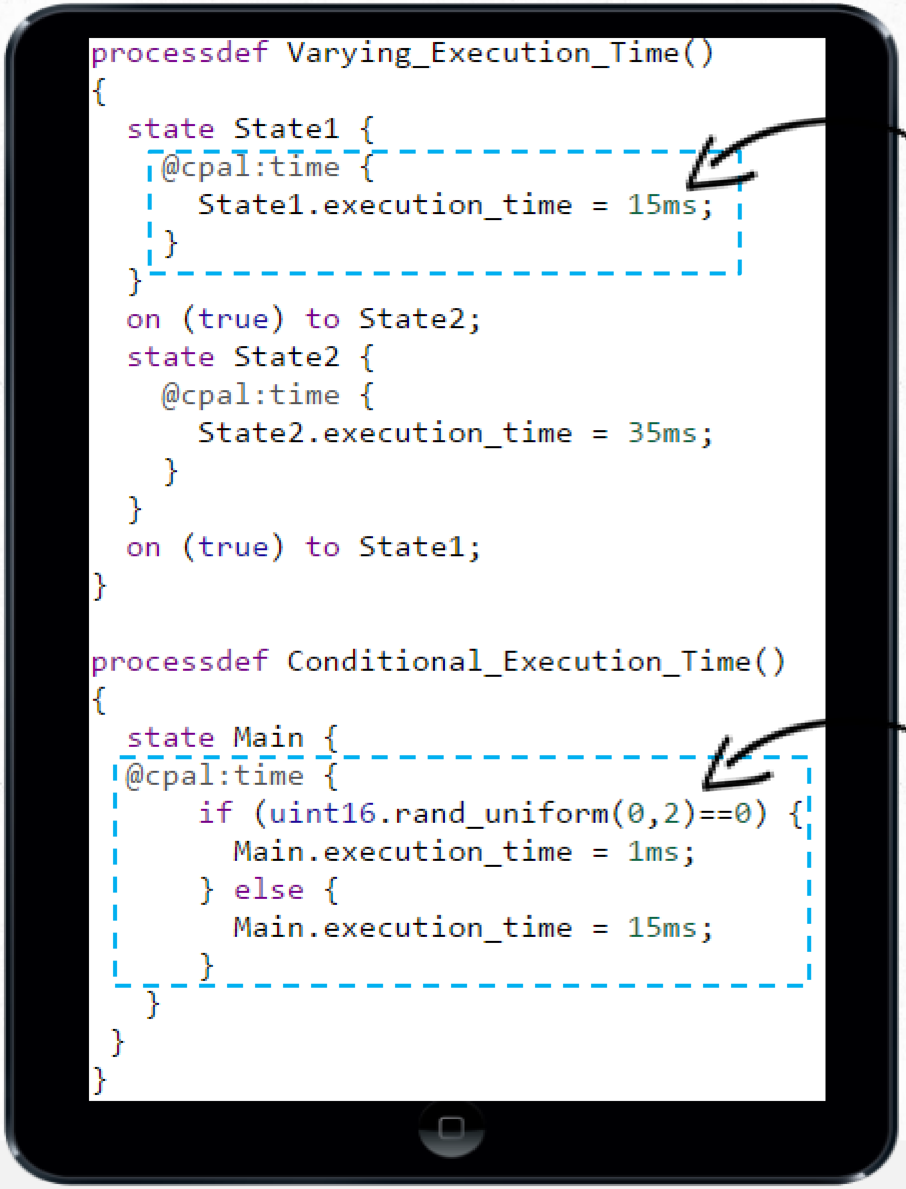


```
processdef Time_Varying_Period()
{
  state Main {
    IO.println("period: %t",self.period);
    IO.println("offset: %t",self.offset);
    IO.println("current activation: %t",self.current_activation);
    IO.println("previous_activation: %t",self.previous_activation);
  }
}

/* The first instance of the process is executed at time 3ms,
  the subsequent instances with an interarrival time randomly
  chosen in [8,13]ms */

process Time_Varying_Period: p1[10ms, 3ms]();
@cpal:sched{
  p1.period = time64.rand_uniform(8ms,13ms);
}
```

The annotation is executed upon the activation of the process, before the body of the process

Varying execution times

`@cpal:time` annotations respected in simulation mode but ignored in real-time mode



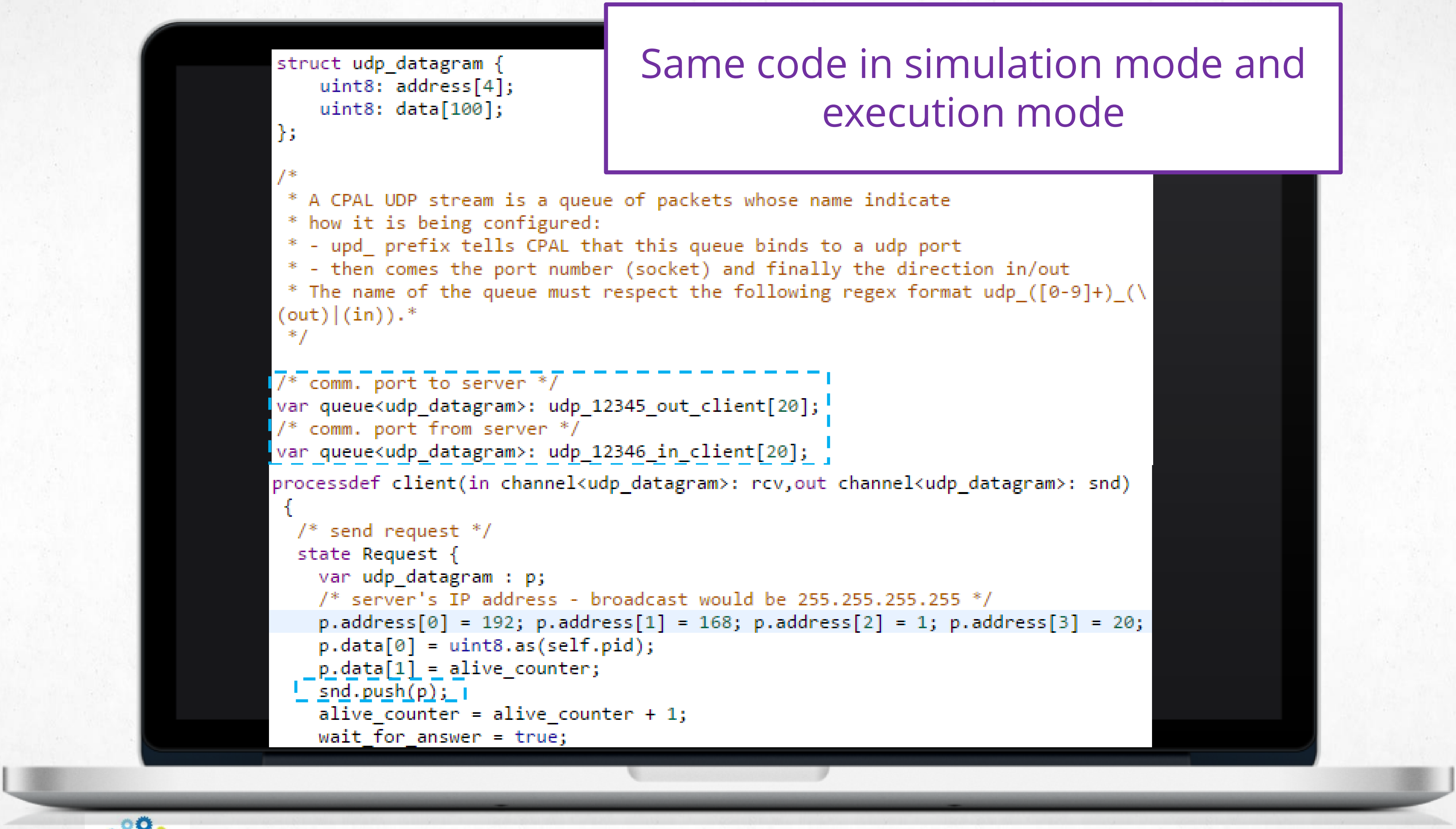
```
processdef Varying_Execution_Time()
{
  state State1 {
    @cpal:time {
      State1.execution_time = 15ms;
    }
  }
  on (true) to State2;
  state State2 {
    @cpal:time {
      State2.execution_time = 35ms;
    }
  }
  on (true) to State1;
}

processdef Conditional_Execution_Time()
{
  state Main {
  @cpal:time {
      if (uint16.rand_uniform(0,2)==0) {
        Main.execution_time = 1ms;
      } else {
        Main.execution_time = 15ms;
      }
   }
 }
}
```

Can be derived by monitoring at run-time with –stats interpreter option

Execution times can dynamically change over time

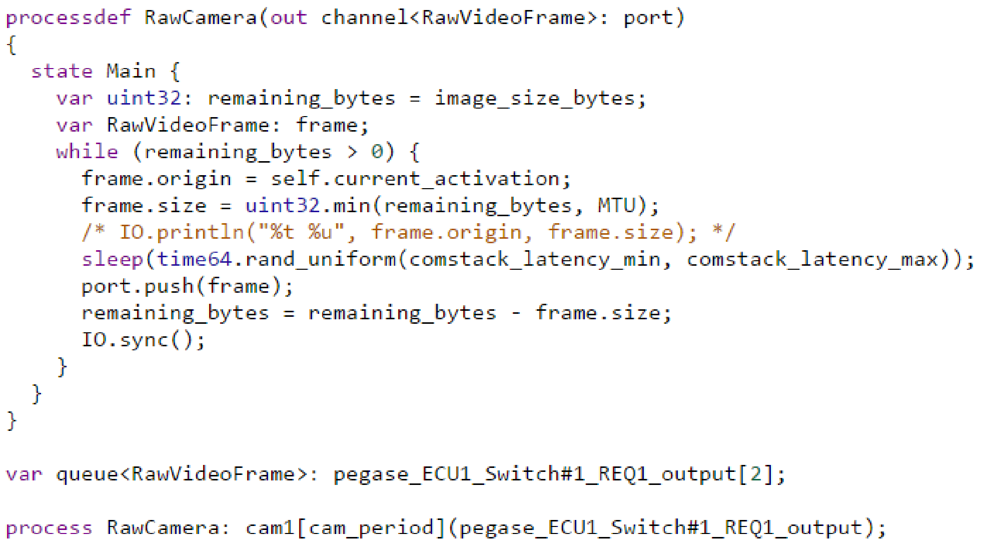
Distributed applications: e.g. UDP or CAN

Same code in simulation mode and execution mode



```
struct udp_datagram {
    uint8: address[4];
    uint8: data[100];
};

/*
 * A CPAL UDP stream is a queue of packets whose name indicate
 * how it is being configured:
 * - upd_ prefix tells CPAL that this queue binds to a udp port
 * - then comes the port number (socket) and finally the direction in/out
 * The name of the queue must respect the following regex format udp_([0-9]+)_(\
(out)|(in)).*
 */

/* comm. port to server */
var queue<udp_datagram>: udp_12345_out_client[20];
/* comm. port from server */
var queue<udp_datagram>: udp_12346_in_client[20];
processdef client(in channel<udp_datagram>: rcv,out channel<udp_datagram>: snd)
 {
  /* send request */
  state Request {
    var udp_datagram : p;
    /* server's IP address - broadcast would be 255.255.255.255 */
    p.address[0] = 192; p.address[1] = 168; p.address[2] = 1; p.address[3] = 20;
    p.data[0] = uint8.as(self.pid);
    p.data[1] = alive_counter;
    snd.push(p);
    alive_counter = alive_counter + 1;
    wait_for_answer = true;
```

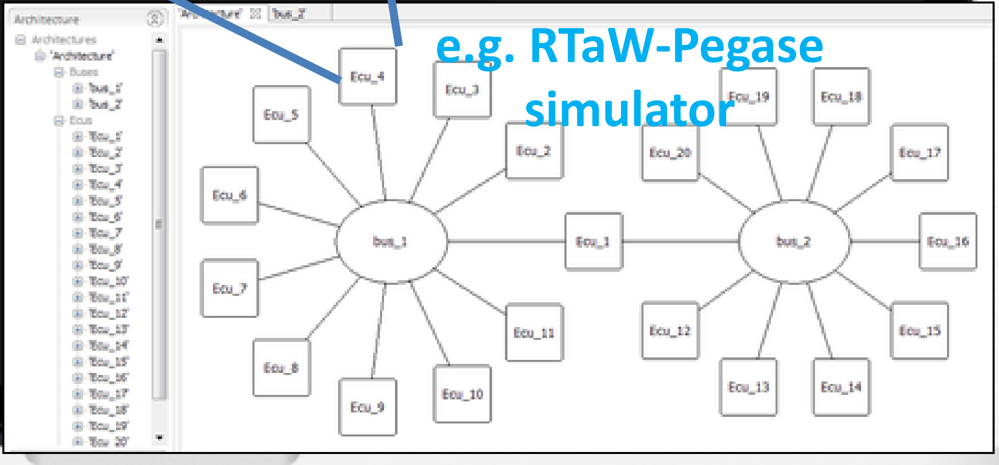
System-level simulation



```
processdef RawCamera(out channel<RawVideoFrame>: port)
{
  state Main {
    var uint32: remaining_bytes = image_size_bytes;
    var RawVideoFrame: frame;
    while (remaining_bytes > 0) {
      frame.origin = self.current_activation;
      frame.size = uint32.min(remaining_bytes, MTU);
      /* IO.println("%t %u", frame.origin, frame.size); */
      sleep(time64.rand_uniform(comstack_latency_min, comstack_latency_max));
      port.push(frame);
      remaining_bytes = remaining_bytes - frame.size;
      IO.sync();
    }
  }
}

var queue<RawVideoFrame>: pegase_ECU1_Switch#1_REQ1_output[2];

process RawCamera: cam1[cam_period](pegase_ECU1_Switch#1_REQ1_output);
```

CPAL to describe the behavior of a station, an application or a protocol layer

The simulation model can later be executed with no changes on a testbed or a prototype of the system.



e.g. RTaW-Pegase simulator

Further information

- ✓ The CPAL programming language: an introduction, 2015.
- ✓ Resources such as technical papers to learn CPAL at https://www.designcps.com/resources-to-learn-cpal/
- ✓ Code examples that can be run in the CPAL-Playground at https://www.designcps.com/cpal-code-examples-index/
- ✓ Download binaries from https://www.designcps.com/binaries/

designCPS

email: contact@designcps.com

https://twitter.com/DesignCPS

☁ http://www.designcps.com

Thank You !