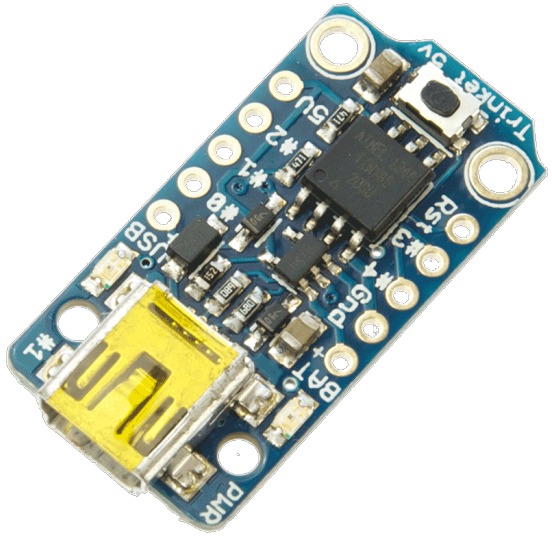
Rizzly: Event Driven Microcontroller Programming

Urs Fässler

Saturday 17:40, H.2215, FOSDEM 2015

31.01.2015

Introduction



[1]

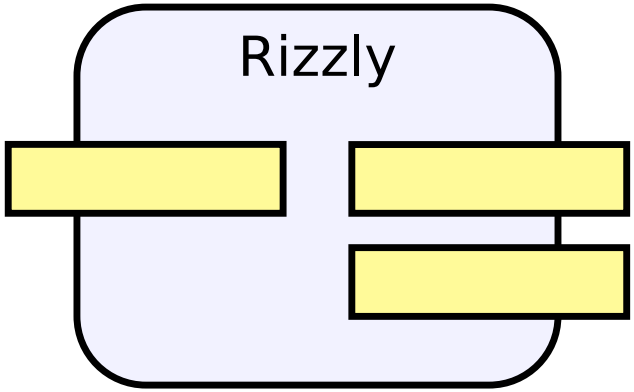
- Micro Controller
  - no OS
  - no dynamic memory
  - Interrupts
- Event Driven Programming
  - every software that interacts with the “real world”
  - Source of Event is Interrupt or Callback
  - Inversion of Control[2]
  - Frameworks, Graphical Tools, Code Generators
  - Qt (on big Machine)

- Micro Controller
  - no OS
  - no dynamic memory
  - Interrupts
- Event Driven Programming
  - every software that interacts with the "real world"
  - Source of Event is Interrupt or Callback
  - Inversion of Control[2]
  - Frameworks, Graphical Tools, Code Generators
  - Qt (on big Machine)

- 8 Bit
- 4-20 MHz
- >= 128 Byte RAM
- >= 2 KiB ROM
- no OS, no Memory Allocation
- whole Program and Environment is known at compile time

Rizzly

- Event Driven Programming Language
- for the smallest Micro Controllers
  - very static generated code
- high code re-usability
  - high machine abstraction
  - Templates
  - Compile Time Function Evaluation



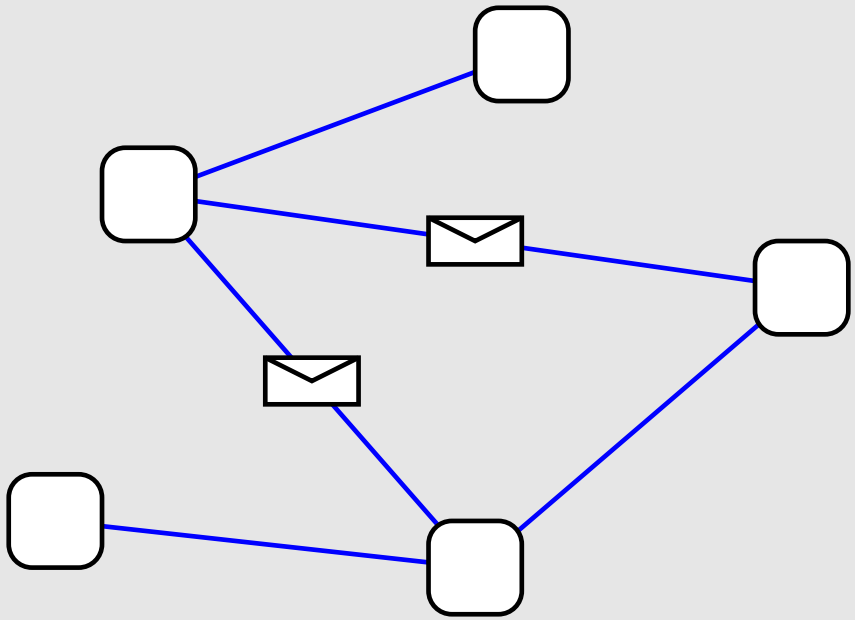
Rizzly3[width=8cm]

└Rizzly

└Rizzly

- Event Driven Programming Language
- for the smallest Micro Controllers
  - very static generated code
- high code re-usability
  - high machine abstraction
  - Templates
  - Compile Time Function Evaluation

- Event handling is based on the Atom-Model
  - Communication only by Events (Messages)
  - Transmission needs time
  - Execution/Handling does not need time
  - Component/Node can only be activated by Event
  - Application is distributed system



- Event Driven Programming Language
- for the smallest Micro Controllers
  - very static generated code
- high code re-usability
  - high machine abstraction
  - Templates
  - Compile Time Function Evaluation

- basic number data type is an integer range type
  - R{from, to}
  - mathematical model: the type of the result of an operation is as big as needed $\rightarrow$ no overflow or implicit cast

Rizzly3[width=8cm]

Rizzly

Rizzly

- Event Driven Programming Language
- for the smallest Micro Controllers
  - very static generated code
- high code re-usability
  - high machine abstraction
  - Templates
  - Compile Time Function Evaluation

- strict Language
- Compiler understands Model
- Compiler sees whole program

$\rightarrow$ better optimization (e.g. eliminate not reachable states in hfsm)

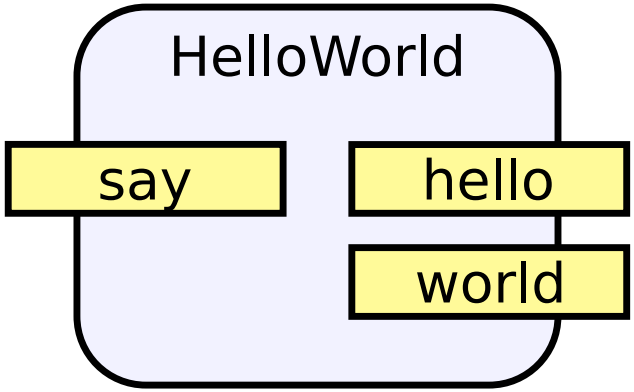
- static Code (no function pointer, no dynamic)

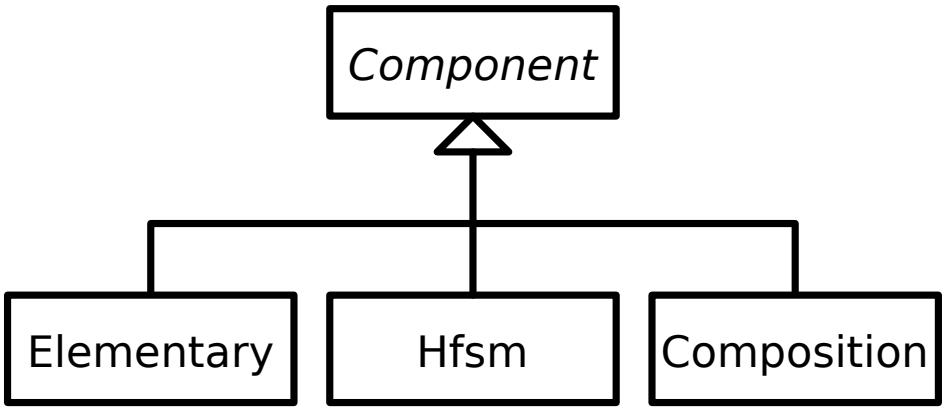
Component

- Slots are Input Interface
- Signals are Output Interface

→ only method to communicate

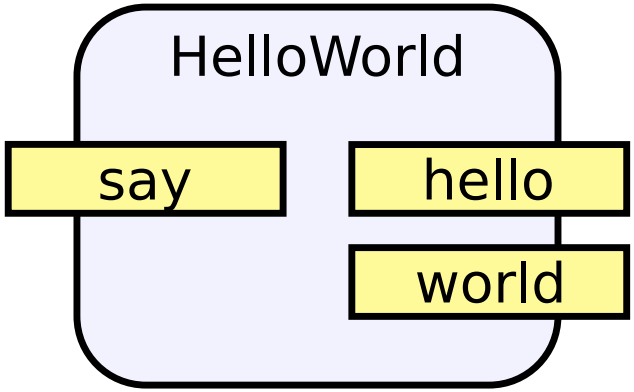
- contains a state
- there is no main . . .
- different implementations

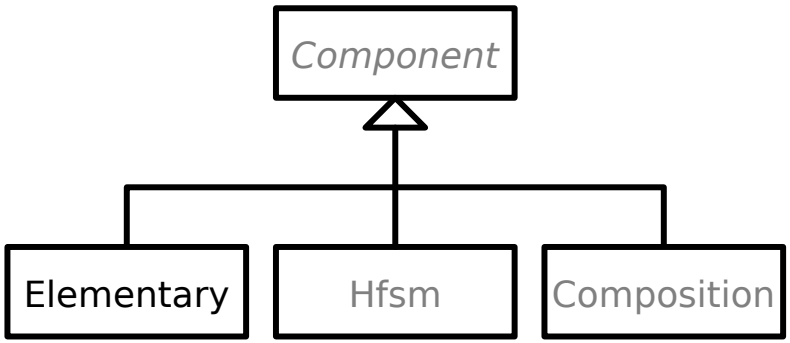




Elementary

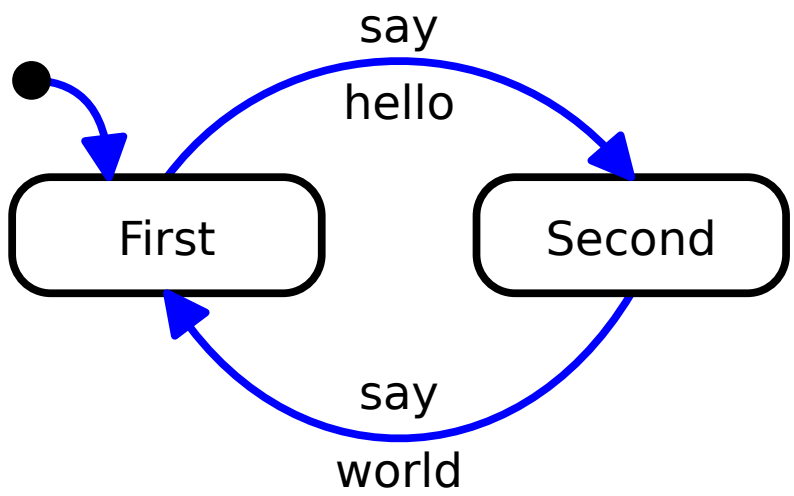
```
HelloWorld = Elementary
  first : Boolean = True;

  hello : signal();
  world : signal();

  say : slot()
    if first then
      hello();
    else
      world();
    end
    first := not first;
  end
end
```

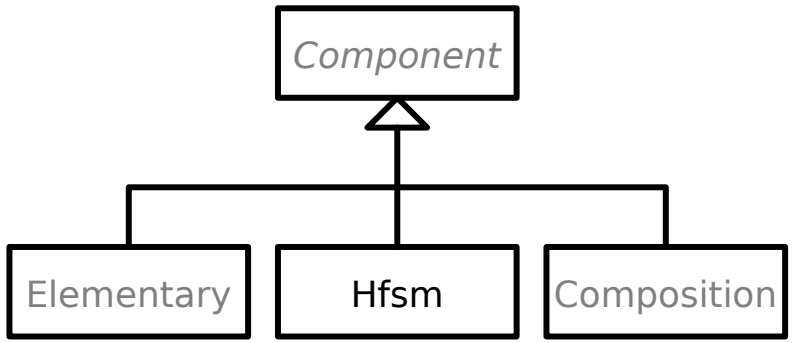




Hirarchical Finite State Machine

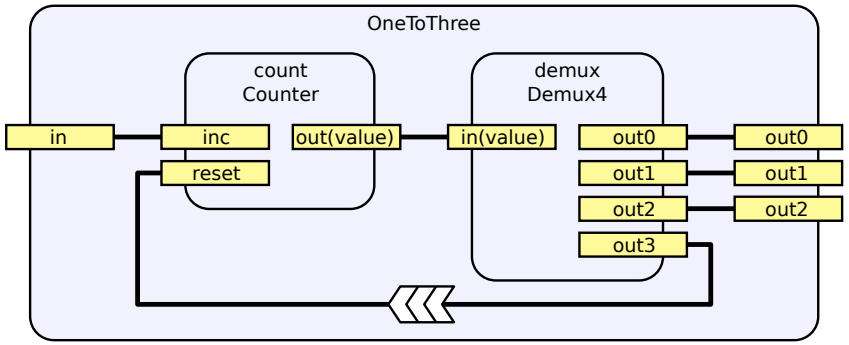
```
HelloWorld = Hfsm
  hello : signal();
  world : signal();
  say   : slot();

  state(First)
    First  : state;
    Second : state;

    First to Second by say() do
      hello();
    end
    Second to First by say() do
      world();
    end
  end
end
```

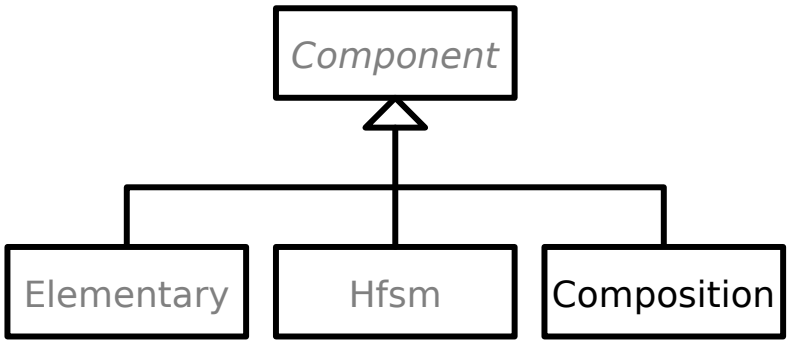




Composition

```
OneToThree = Composition
  out0 : signal();
  out1 : signal();
  out2 : signal();
  in   : slot();

  count  : Counter{0};
  demux  : Demux4;

  in -> count.inc;
  count.out -> demux.in;
  demux.out3 >> count.reset;

  demux.out0 -> out0;
  demux.out1 -> out1;
  demux.out2 -> out2;
end
```





TODO

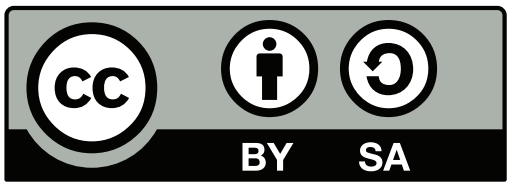
- method to access io, catch interrupts (see appendix)
- ~~llvm/gcc for Code generation~~
  - best is to stick with C code generation since there exists probably for every micro controller a C compiler
- IDE Integration
- . . . and a lot more

Thank you

→ Rizzly
- Event-Driven
- Micro Controller

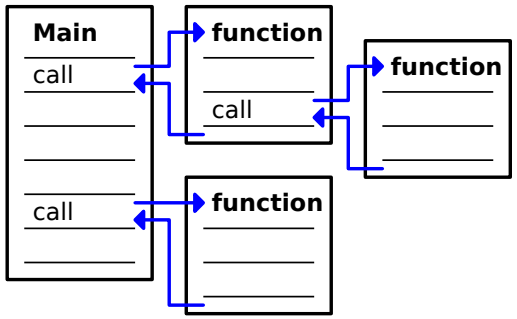
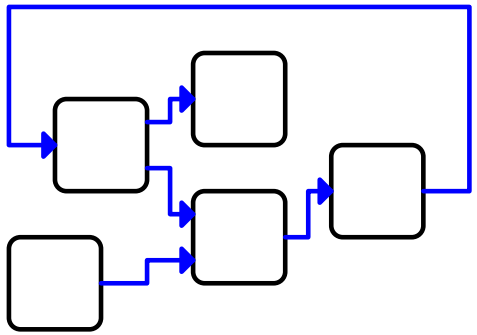
- urs@bitzgi.ch

- gitorious.org/rizzly



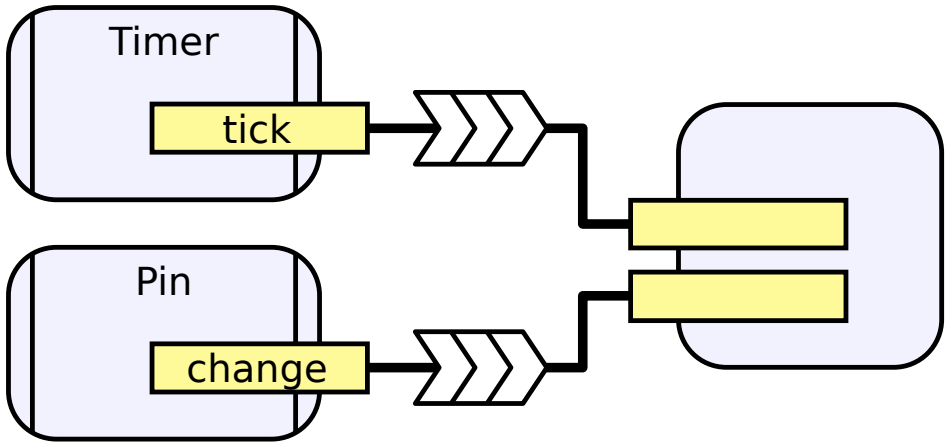
Exclusive external Content

Imperative ↔ Event Driven

	Imperative	Event Driven
Contrololle	Program	Environment
Program flow	linear	non-linear
program is	calculating	waiting
run time	short	long
parallel	no	yes
coupling	strong	weak
		

TODO: access Hardware / Interrupts

```
Timer = Elementary
  tick = signal();

  reload       : Register{R{0,255}, 2048, 0, 8, MAP11};

  interrupt{INTERRUPT_TIMER}
    // clear interrupt flag
    reload := ...;
    tick();
  end
end
```



```
Timer = Elementary
  tick = signal();

  reload      : Register{R{0,255}, 2048, 0, 8, MAP11};

  interrupt{INTERRUPT_TIMER}
    // clear interrupt flag
    reload := ...;
    tick();
  end
end
```

reload is a register of the CPU. It is defined by the built in Register Template. The arguments are the type as seen from the Rizzly program, the address of the register, the first and last bit, and the mapping from the Rizzly data type to the physical representation.

TODO: access Hardware / Interrupts

```
Timer = Elementary
  tick = signal();

  reload      : Register{R{0,255}, 2048, 0, 8, MAP11};

  interrupt{INTERRUPT_TIMER}
    // clear interrupt flag
    reload := ...;
    tick();
  end
end
```

interrupt is a (special) event handler for interrupts. The template argument could be the interrupt number or the address where the interrupt function has to be placed.

TODO: access Hardware / Interrupts

```
Timer = Elementary
  tick = signal();

  reload      : Register{R{0,255}, 2048, 0, 8, MAP11};

  interrupt{INTERRUPT_TIMER}
    // clear interrupt flag
    reload := ...;
    tick();
  end
end
```

The component is no longer pure since she can send event without an activation as also since she changes the environment without sending an event. As this violates the model, we need queues to connect such components. To keep an application “clean”, impure components should be used carefully and only on the top layer.

Rizzly2[width=8cm]

└TODO

└TODO: access Hardware / Interrupts

TODO: access Hardware / Interrupts

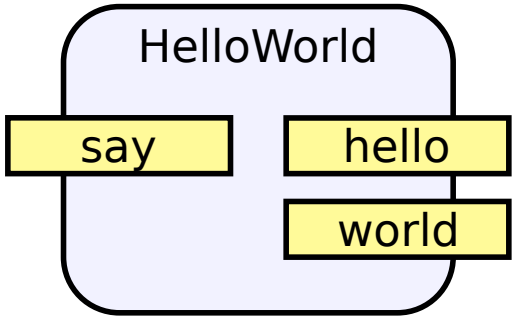
```
Timer = Elementary
  tick = signal();

  reload      : Register{R{0,255}, 2048, 0, 8, MAP11};

  interrupt{INTERRUPT_TIMER}
    // clear interrupt flag
    reload := ...;
    tick();
  end
end
```

As it is not yet possible to access hardware, some glue code is needed.

Usage

Main (Glue Code)

```
ISR(INT0_vect){
  inst_say();
}
void inst_hello(){
  LED_HELLO = 1;
}
void inst_world(){
  LED_WORLD = 1;
}
void main(){
  ...
  inst__construct();
  ...
```



Generated Header

```
extern void inst__construct();
extern void inst__destruct();
extern void inst_say();
// void inst_hello();
// void inst_world();
```

Templates

```
Point{T: Type{Integer}} = Record
  x : T;
  y : T;
end

max{N: R{0,100}} = function(x: R{0,100}):R{0,N}
  if x > N then
    return N;
  else
    return x;
  end
end

a : Point{R{-10,10}};
y := max{80}( 42 );
```

Compile-time function evaluation

```
lookuptable : const = calcTable(57);

calcTable = function(n: R{0,100}):Array{10,R{0,100}}
  res : Array{10,R{0,100}};
  i   : R{0,11} = 0;
  while[1] i < 11 do
    idx : R{0,10} = R{0,10}(i);
    res[idx] := idx * n / 10;
    i := idx + 1;
  end
  return res;
end
```

[1] foreach loop is coming

Literature I

[1] Adafruit trinket.
URL: http://www.tandyonline.co.uk/adafruit-trinket-mini-microcontroller-5v.html.

[2] Wikipedia.
Inversion of control — wikipedia, die freie enzyklopädie, 2013.
[Online; Stand 11. November 2014].
URL: http://de.wikipedia.org/w/index.php?title=Inversion_of_Control&oldid=125646165.