Daisy Chain Protocol Development in Force-Guiding Particle Chains for Shape-Shifting Displays

Network Protocol Implementation

Raoul Rubien
rubienr@sbox.tugraz.at

Institute for Technical Informatics
Graz University of Technology

5th August 2016

What this session covers

the development tool chain
simulation
protocol implementation

Underlying work

Force-Guiding particle Chains for Shape-Shifting Displays[1]
development board
grid board

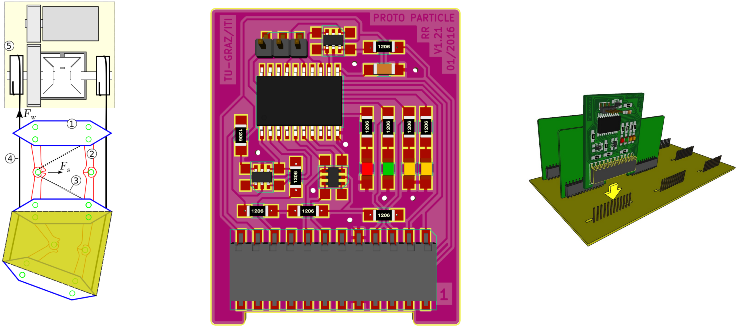

Figure 1: *underlying work*

Project extent

daisy chain protocol development (OSI)

Physical Layer, Data Layer, Network Layer

actuation scheduling and execution

time synchronization

runtime compensation of RC-oscillator discrepancy

Project constraints

exploit actuation wires for protocol communication

single communication entry point to the network

daisy chained network

Planning the development work flow

Aim	**Solution**
i) How to guarantee good **code quality**?	by continuous **testing**
ii) How to **test**?	by **simulation**
iii) How how to **speed up** development?	by using customized **tool chain**, **i)** and **ii)**

The Tool Chain

Tool chain overview

IDE independent
multiple projects can be integrated
CMake provides all necessary make targets such as
- deployment to real MCU and
- simulation targets

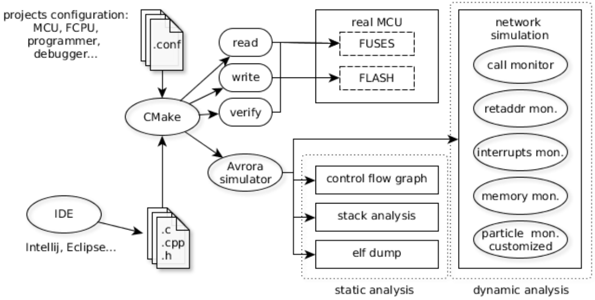

Figure 2: *tool chain overview*

CMake targets in detail

CMake implementation provides various targets

- dynamic analysis (monitoring) of:
  - wires
  - any SRAM register
  - I/O ports
  - interrupts
  - function calls
  - stack overflow
  - statistic reports (profiling): memory writes, function calls
- static analysis:
  - stack maximum size
  - control flow graph (code optimization)
  - inspecting CPU cycles per instruction (code opt.)

Simulation

How does a simulation framework help - I

the simulation framework simulates a whole network
each node is served by a dedicated thread
any node is defined by an abstract PCB model (platform)
each platform is associated with the firmware, whereas
the firmware is the same as for a real MCU

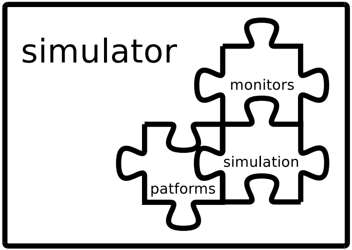

Figure 3: *simulator structure*

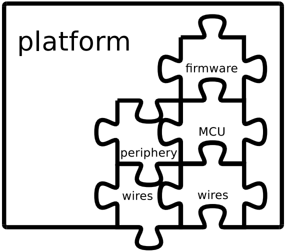

Figure 4: *platform structure*

How does a simulation framework help - II

the simulation output is

- inspected by JUnit tests
- interpreted by the developer
- used to visualize signals, wires, variables, state changes and much more

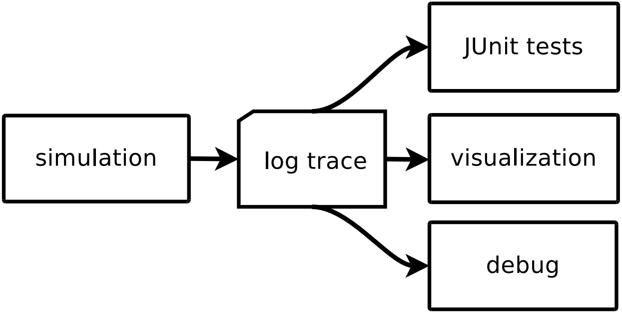

Figure 5: *work flow*

Avrora[1] simulation trace

may also be used in other tools such as:

- friction simulation
- network visualization

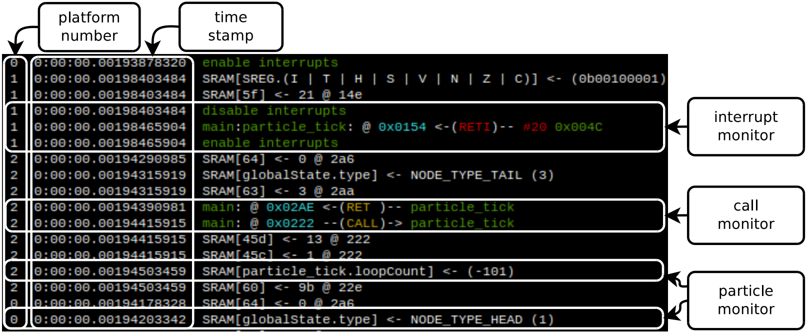

Figure 6: *simulation trace*

[1] http://compilers.cs.ucla.edu/avrora

Trace visualization

(2×2) network (interactive chart example [1] or [2]):
wire signals, arbitrary *uint8_t* debug output, internal global variables

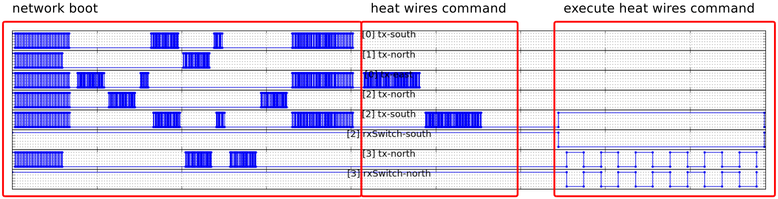

Figure 7: *node visualization*

Control flow graph

code otimizing helper

- inline vs. call
- prove expected inlining result

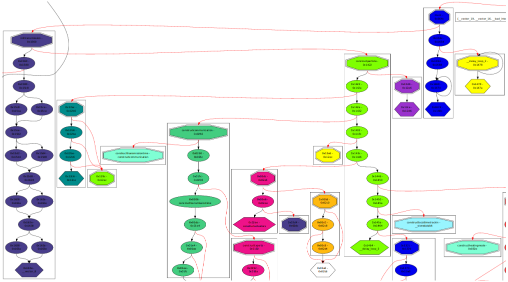

Figure 8: *control flow graph snippet*

square: interrupt
double octagon: procedure
hexagonal: blocks with return

edges: jumps, branches
red edges: calls
dotted: indirect calls or jumps

Profiling

customized memory profiling

interrupt profiling

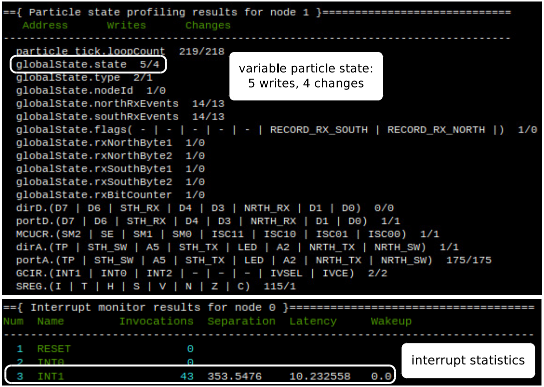

Figure 9: *Avrora profiling monitors example*

Code optimization helper

inspecting cycles per instruction example

```
    case STATE_TYPE_PREPARE_FOR_SLEEP:
        if (ParticleAttributes.directionOrientedPorts.north.txPort->isTransmitting ||
26ca:       e0 91 be 01     [[LDS -> 2]]    r30, 0x01BE
26ce:       f0 91 bf 01     [[LDS -> 2]]    r31, 0x01BF
26d2:       85 85           [[LDD -> 2|2]]  r24, Z+13       ; 0x0d
26d4:       80 fd           [[SBRC -> 1/2/3]]       r24, 0
26d6:       37 c0           [[RJMP -> 2]]   .+110           ; 0x2746 <particleTick+0x288>
            ParticleAttributes.directionOrientedPorts.east.txPort->isTransmitting ||
26d8:       e0 91 cc 01     [[LDS -> 2]]    r30, 0x01CC
26dc:       f0 91 cd 01     [[LDS -> 2]]    r31, 0x01CD
        break;
```

SBRC @ 26d4 takes 1, 2 or 3 cpu cycles

Figure 10: *cycles per instruction inspection*

Protocol Implementation

Recall objectives

What is needed?
A **daisy chain** protocol capable of **executing commands synchronously** at a **given time** using **actuator wires**.

That means roughly

1) Layer 0
2) Layer 1
3) Layer 2

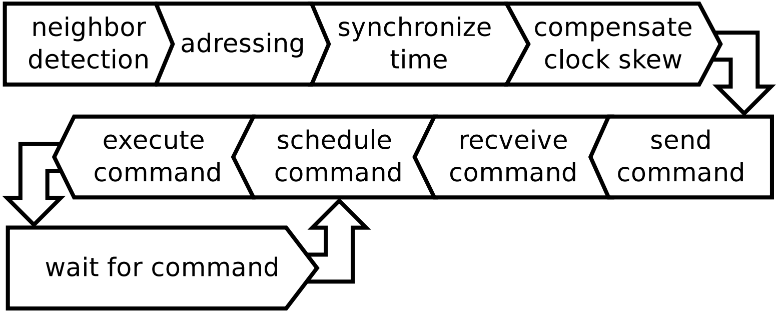

Figure 11: *Layer 2: protocol process*

State driven process implementation

```
void main() {
  // state machine call
  while(true) { process(); }
}
```

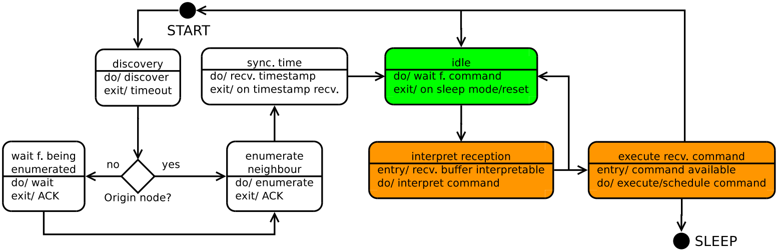

Figure 12: *protocol states*

Firmware sequence diagram

reception, transmission and processing are independent
may occur effectively concurrent

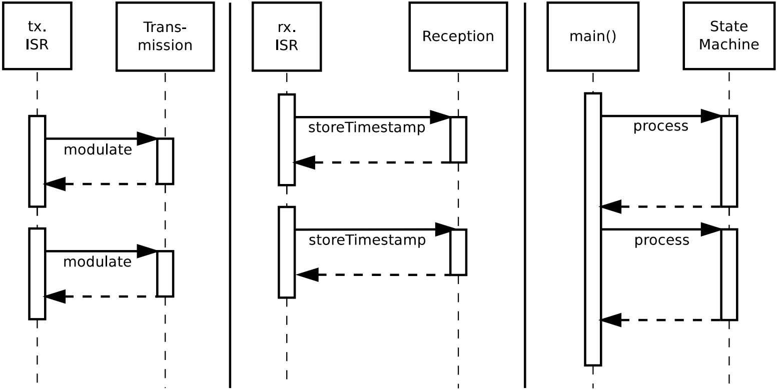

Figure 13: *firmware sequence diagram*

Encoding

opted for Manchester coding
no common tx/rx clock needed
can be exploited to calculate clock skew
simple to implement

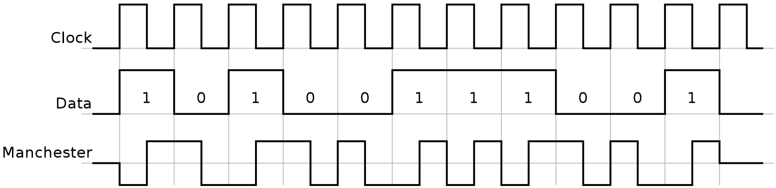

Figure 14: *Manchester coding: data = clock ⊕ manchester*

Decoding implementation

store timestamp of signal flank
to circular buffer
timestamp: 16*bit* timer-counter value
decoder removes from buffer, stores to decoded buffer
interpreter: interprets on interpret-able decoded buffer

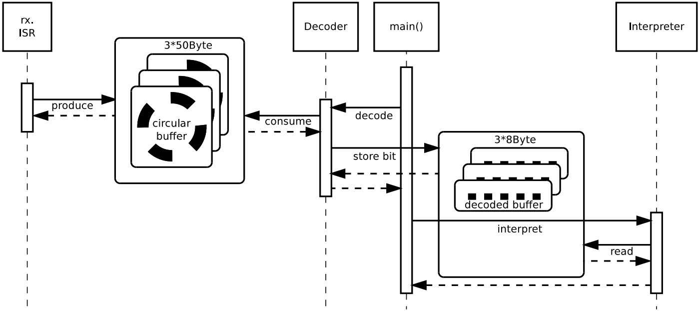

Figure 15: *decoding sequence diagram*

Network boot example

1^{st}	phase:	neighbor discovery
2^{nd}	phase:	address assignment
3^{rd}	phase:	enumeration finished
4^{th}	phase:	time synchronization

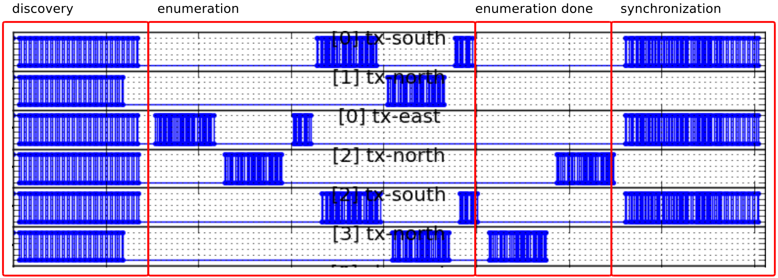

Figure 16: *network boot visualization*

Addressing example

1^{st}	phase:	network discovery and address assignment
2^{nd}	phase:	send "heat wires at specific time" command
3^{rd}	phase:	execute command

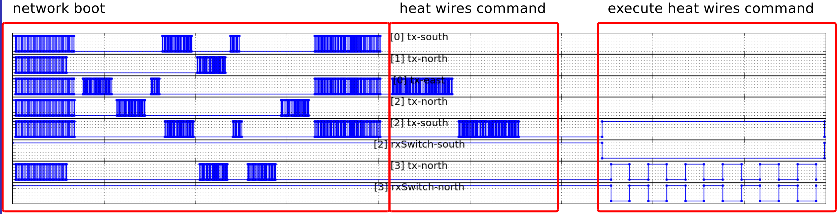

Figure 17: *"execute command" network visualization*

Results

i)	An extend-able **daisy chain network protocol** that
ii)	**executes** actuation **commands**
	at a given time **synchronously**, and
iii)	a development **tool chain** that sustains:
iii.a)	**simulation**,
iii.b)	**debugging** and
iii.c)	JUnit **testing**.

Because programmable matter matters!

Future work

Phy. Layer: considering partially implemented parity bit
Network Layer: fault detection, fault tolerance
actuation: power adjustment (PWM tuning)
time compensation: evaluation of calibration accuracy
firmware replication: customize boot loader
forward/backward shaping of 1^{st} row

M. Lasagni and K. Römer, "Force-guiding particle chains for shape-shifting displays," *CoRR*, vol. abs/1402.2507, 2014.