Flow Based Programming
FBP
Applied to IoT Development

OpenIoT & ELC Europe 2016

Agenda

- Who am I?
- Challenge & Motivation
- Flow-based programming
- Soletta
- Pros & Cons

Who am I?

Gustavo Sverzut Barbieri
Computer Engineer
ProFUSION embedded systems

- Brazilian
- Software Developer since 9yo
- Working with Embedded since 2005
- Software development services
- Passionate about efficiency
- Years of experience with event loop based programming
- Soletta Architect & Lead Developer

IoT Challenge

- IoT differences to traditional embedded systems
- Solutions are focused on a single subset (just hardware, just network…)
- Solutions are platform specific, no scalable solutions
- Nothing is integrated
- Hard to reuse your knowledge
- Soletta: uniform API for platform tasks, sensors and networking, from MCU to Linux

http://github.com/solettaproject

creating an efficient & easy to use API requires you to understand your users

- How did we learn to program?
- What’s the IoT device workflow?
- Do they match?

Programming 101

```
int main(int argc, char *argv[]) {
    data = read_input();
    process_data(data);
    report(data);
    return 0;
}
```

Programming 101

- Procedural Batch Programming
- Single workflow
- Often not even error handling

Expected workflow of an IoT device

Workflow of an IoT Device

Continuous serving multiple simultaneous input:

- Network
- Sensors
- User
- Timers

IoT Device + Programming 101 ?

```
int main(int argc, char *argv[]) {
    data = read_input();
    process_data(data);
    report(data);
    return 0;
}
```

How to make it work?

IoT Device + Programming 101 (Try #1)

```
int main(int argc, char *argv[]) {
  while (1) { // there! I fixed it
      data = read_input();
      process_data(data);
      report(data);
   }
   return 0;
}
```

What about other inputs?

IoT Device + Programming 101 (Try #2)

```
int main(int argc, char *argv[]) {
   while (1) {
        net_data = read_network_input();
        process_network_data(net_data);
        report_network_data(net_data);

        sensor_data = read_sensor_input(); // there! I fixed it!
        process_sensor_data(sensor_data);
        report_sensor_data(sensor_data);
    }
    return 0;
}
```

What about no network input while new sensor input?

IoT Device + Programming 101 (Try #3)

```
int main(int argc, char *argv[]) {
   while (1) {
        if (has_network_input()) {  // there! I fixed it!
            net_data = read_network_input();
            process_network_data(net_data);
            report_network_data(net_data);
        }
        if (has_sensor_input()) {
            sensor_data = read_sensor_input();
            process_sensor_data(sensor_data);
            report_sensor_data(sensor_data);
        }
    }
    return 0;
}
```

1. What about simultaneous input?
2. Noticed Feedback LED stops blinking?
3. Busy wait = battery drain!

IoT Device + Programming 101 (Try #4)

```
void thread_network(void *data) {
    while (1) {
        net_data = read_network_input();
        process_network_data(net_data);
        report_network_data(net_data);
    }
}

int main(int argc, char *argv[]) {
    // there! I fixed it!
    pthread_create(&t_net, NULL, thread_network, NULL);
    pthread_create(&t_sensor, NULL, thread_sensor, NULL);
    pthread_create(&t_led, NULL, thread_led_blinking, NULL);
    pthread_join(t_net, NULL);
    pthread_join(t_sensor, NULL);
    pthread_join(t_led, NULL);
    return 0;
}
```

What about thread-unsafe resources?

Reporting sensors to the network?

GUI/UX updates?

widely known paradigm

Event-Driven Programming

- a.k.a. “Main Loop Programming”
- servers
- graphical user interfaces

Event Driven Programming

```
int main(int argc, char *argv[]) {
   while (wait_events(&events, &current)) {
         if (current->type == NETWORK) {
             net_data = read_network_input(current);
             process_network_data(net_data);
             report_network_data(net_data);
         } else if (current->type == SENSOR) {
             sensor_data = read_sensor_input(current);
             process_sensor_data(sensor_data);
             report_sensor_data(sensor_data);
         }
    }
    return 0;
}
```

Easy to understand, similar to 101 Try #3.
May use a dispatcher table

Event Driven Programming

```
void on_network_event(event) {
    net_data = read_network_input(event);
    process_network_data(net_data);
    report_network_data(net_data);
}

int main(int argc, char *argv[]) {
    register_event_handler(NETWORK, on_network_event);
    register_event_handler(SENSOR, on_sensor_event);
    wait_and_handle_events(); // blocks forever aka “main loop”
    return 0;
}
```

Event Driven Programming

- Similar to 101 Programming try #3
- `wait_events(list, current)` handles multiple input, once at time
- Single threaded usage, may contain multiple threads inside
- Easy to implement with POSIX `select()`, `poll()`, `epoll()`...
- Timeout is an event
- Suggests short cooperative coroutines, “idler” concept to help

Event Driven Programming: idler

time

LED LED LED LED LED LED LED LED LED LED LED LED LED LED LED LED LED

process_network_data()
4 seconds

process_network_data()
8 seconds

LED LED LED LED LED LED LED LED LED LED LED LED LED LED LED LED

⅓ second each

although it feels more responsive,
overall processing time is increased

Event Driven Programming: idler

```
void process_data(data, on_done_cb) {
	struct ctx *ctx = malloc(...);
	ctx->on_done_cb = on_done_cb;
	ctx->i = 0;
	ctx->data = data;
	ctx->idler = idler_start(
		process_data_idler, ctx);
}

void process_data_idler(void *d) {
	struct ctx *ctx = d;
	if (ctx->i == ctx->data->count) {
		idler_stop(ctx->idler);
		ctx->on_done_cb(ctx->data);
		free(ctx);
		return;
	}
	process_item(ctx->data->item[ctx->i]);
	ctx->i++;
}
```

Original code:

```
void process_data(data) {
	for (i = 0;
		i < data->count;
		i++)
	process_item(data->item[i]);
}
```

Blocks the main loop for
`COUNT * time(process_item)`

Blocks the main loop for
`time(process_item)`

Event Driven Programming: idler

Pros:

- no real concurrency: single threaded, no need for locks
- works everywhere, even on single task systems
- lean on memory, you manually save your “stack” in callback context

Cons:

- requires manual analysis and algorithm segmentation
- requires callbacks and extra context data
- cancellation and error handling must stop idler and free context data
- Painful to chain multiple stages (read, process, report…)

Soletta Project

initial design choices

http://github.com/solettaproject

- Focus on scalability
- Previous experience
- Object Oriented in C
- Main loop - Event Based Programming
- Network
- Sensors
- Actuators

as expected, the same design led to
the same problems...

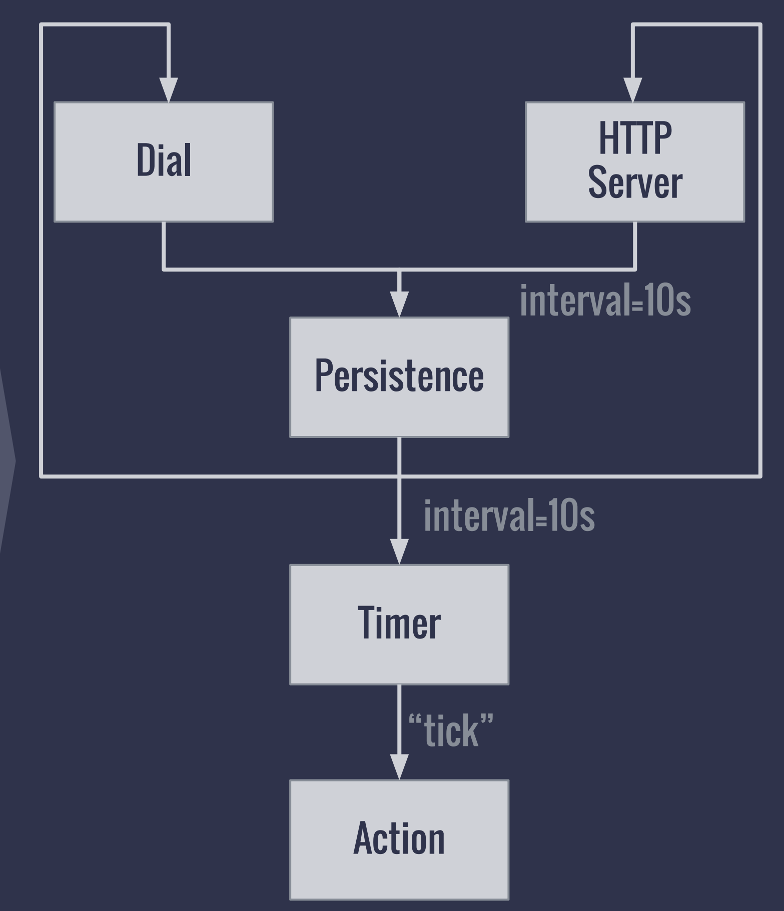
most users don’t get callbacks

Leaks & SEGV

boring pattern “on event, get data”

Flow-Based Programming

technology from 1970
that came to rescue
the web…
… and IoT



Flow Based Programming

- Invented by J. Paul Morrison in the early 1970s http://www.jpaulmorrison.com/fbp
- Components are Black Boxes with well defined interfaces (Ports)
- Focus on Information Packets (IP)
- Started to gain traction in Web:

  NoFlo | Facebook Flux | Google TensorFlow | Microsoft Azure Event Hubs
- Also on Embedded Systems:

  ROS | MicroFlo | NodeRED
- Also on Multimedia:

  V4L | Gstreamer | Apple Quartz

FBP Concepts & Terms

Node1(Type1) OUT -> IN Node2(Type2)

FBP is easy to read, write and visualize

FBP: Nodes as Black Boxes

- Simple interface
- Low (no?!) coupling, allows replacing components
- Easy to optimize code size by removing unused ports
- Parallelization
- Isolation (including processes)
- Internally can use Event-Driven Programming (Main Loop), Threads...

If an FBP program ever crashes it's guaranteed that it's the node type provider fault!

FBP: It's all about Information Packets

- "What goes where"
- Clear data ownership
- Memory management hidden in the core
- Callbacks hidden in the core
- Packet delivery can be delayed - reduced power consumption!
- Packet memory can be recycled - reduced memory fragmentation!
- Ports and Packets can be typed - compile & runtime safety

Leaks or SEGV are impossible

Soletta’s FBP

What’s specific & Why?

- Scalability - MCU and up
- Extensibility
- Configurations

more details and a comparison with classical FBP at:
https://github.com/solettaproject/soletta/wiki/Flow-Based-Programming-Study

Soletta FBP: Statically Typed Packets & Ports

- More information allows more optimization possibilities
- Type checking at both compile and runtime
- Pre-defined basic packet types (boolean, integer, string, direction-vector…)
- Composed packet types, similar to structures
- Extensible via user-defined types for domain specific data

Soletta FBP: Packet Delivery & Ownership

- Packets are immutable aka “read-only”
- Packets are created by nodes and sent on its output ports
- Once sent, flow core owns the packets
- Packets are queued for delivery
- Each delivery happens from different main loop iteration
- Multiple connections are allowed to/from ports
- Ports know of connections using `connect()` and `disconnect()`
- Packets are delivered by calling port’s `process()`

Soletta Usage Workflow

source.c

Compiler & Linker

Binary

sol-fbp-generator

generated source.c

source.fbp

+

sol-flow-board.json

sol-fbp-runner

Soletta FBP: Configuration

- Unique feature!
- Single FBP handling multiple hardware configurations
- `sol-flow-${APP_NAME}-${BOARD_NAME}.json`
- Board name from libsoletta.so, envvar or autodetected
- Fallback `sol-flow.json` allows easily testing on PC with console or GTK…
- `sol-fbp-generator -c file.json`…

Soletta FBP: Node Types (Components)

- Pointer to C structure with `open()`, `close()` and ports
- Built-in to libsoletta.so, application or external "module.so"
- Descriptions (meta-information) can be compiled out
- `sol-fbp-generator` uses JSON descriptions to output "resolved" code
- `sol-fbp-runner` uses compiled in descriptions
- Can be auto-generated by meta-types using `DECLARE=` (FBP, Composed, JS...)

http://solettaproject.github.io/docs/nodetypes/

Soletta FBP: Node Type Options

gpio1(gpio/reader)

'1' -> PIN gpio1

'true' -> ACTIVE_LOW gpio1

gpio1(gpio/reader:pin=1,active_low=true)

or

gpio1(my_gpio1)

sol-flow-myboard1.json:

```
"name": "my_gpio1",
"type": "gpio/reader",
"options": {
    "pin": 1,
    "active_low": true
}
```

- Simplifies setup
- Efficient memory usage
- Allows external configuration

FBP - Pros & Cons

Cons:

- Paradigm shift
- Although small, still adds overhead compared to carefully written C code
- Requires “bindings” (node type module) to use 3rd party libraries
- Needs balance on what to write as FBP and what to create custom node types

Pros:

- No leaks or SEGV, reduced blaming!
- Simple interface (nodes & ports) eases team collaboration
- Easy to read, write and visualize, aids communication with customers & designers
- Super fast prototyping & testing

Thank You!

Questions?

Gustavo Sverzut Barbieri
<barbieri@profusion.mobi>

github.com/solettaproject/soletta/blob/master/
doc/tutorials/ostro-oic-tutorial/step0/tutorial.md