Zephyr® Project
Developer Summit 2022

Combining Zephyr & C++20 to build safe, low-footprint, event-based applications

Ioannis Papamanoglou
ioannis.papamanoglou@zonneplan.nl

Zonneplan

9th June 2022

Outline

- Introduction
- Motivation
- Demonstration
- Implementation
- Conclusions
- Appendix

Introduction

About me

- Ioannis Papamanoglou
- Rotterdam - The Netherlands
- lead embedded developer @Zonneplan since 2021

About me

- working with Zephyr since 2019
- currently running 2.7.0 (LTS2)
- Zephyr code contributions
  - first version of DT_INST_FOREACH
  - drivers (gpio expander)
  - bugfixes (flash)

About me

- mobility
  - car sharing
  - bicycle lock for bicycle sharing
  - smart cabins
- energy
  - smart grid (solar panels, batteries, . . .)
  - ev charger

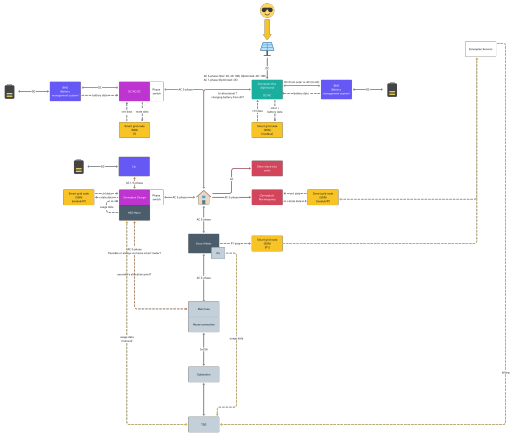
About Zonneplan

- founded 2012 as solar panel expert focused on b2c
- expanded to market leader in the Netherlands ($>$ 100k customers)
- transitioning to smart energy business
- growing energy tech department
- all firmware running on event-based, Zephyr-powered architecture
- $\backsim$ 200 employees $\rightarrow$ 22 developers $\rightarrow$ 5 embedded

zonneplan

About Zonneplan

- currently designing and building a smart-grid platform
- Airnode
  - IoT device that connects energy assets
  - plug-and-play for user
- goal: coordinate the huge amount of energy consumers/producers/storage



Abstract

- motivate event-based embedded applications
- compare to main-loop & threaded-loops
- demonstrate potential implementation

Preview

```
1  struct FooEventData {
2      int foo;
3  };
4  using FooEvent = GenEvent<BaseEvent, FooEventData>;
5
6  class FooGenerator
7      : public EventGeneratorFull<FooEvent>
8  {
9  public:
10     FooGenerator() {}
11
12     void do_something(int val) {
13         FooEvent event;
14         event.foo = val+1;
15         NOTIFY(event);
16     }
17 };
```

Listing: event generator example

Preview

```
19  class FooListener
20      : public EventListenerWithAsyncStreamQueue<FooEvent, 5>
21  {
22  public:
23      FooListener(FooEvent::GeneratorType& generator)
24        : EventListenerWithAsyncStreamQueue<FooEvent, 5>(
25            generator, RingBufferPolicy::KEEP_LAST, WorkerQSys)
26      { }
27
28      void handle(FooEvent& event, FooEvent::GeneratorType& generator)
29      override {
30          LOG_DBG("Got foo event: %d", event.foo)
31      }
32  };
```

Listing: event listener example

Motivation

Typical embedded example

- power: mains
- interfaces: modbus rtu, pwm, adc
- network: none
- memory: 16k
- flash: 128k
- application: smart electric vehicle charger

Typical embedded example

- communicates to EV via PWM & ADC
  - $\rightarrow$ maximum charge current
  - $\leftarrow$ EV status
- communicates to host via Modbus RTU
  - $\rightarrow$ current state, ...
  - $\leftarrow$ charging current, charging on/off
- control power relays (GPIO)
- check safety systems
  - leakage current (GPIO)
  - relay weldcheck (GPIO)
  - temperature (I^2C sensor)

Typical embedded example

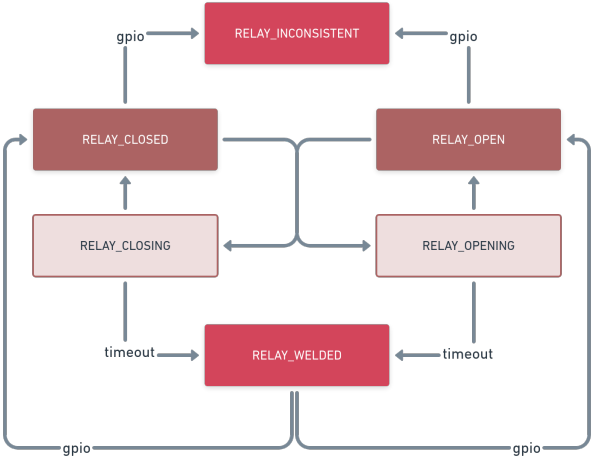


Figure: states single component (relay)

Typical embedded example

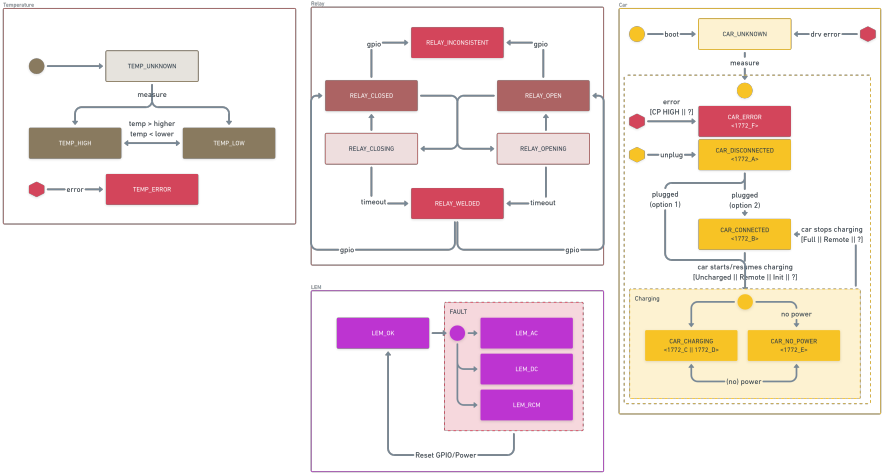


Figure: states components

Typical embedded example

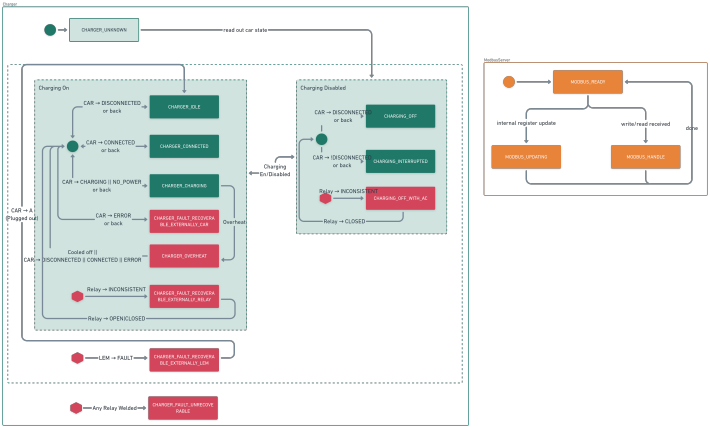


Figure: states app modules

Typical embedded example

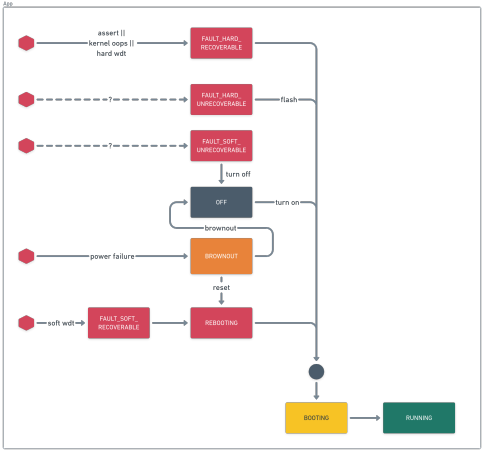


Figure: states app

Analysis

- several components with their own states
- state transitions triggered by external events or internal calls
- state changes themselves are events for higher-level modules
- events/calls can happen at the same time
- certain event/call handlers require higher priorities than others

Design pattern comparison

- main-loop
- threaded-loops
- event-based

main-loop

- single user thread
- loop
  - checks current state
  - checks current pending events
  - acts
  - switches state

main-loop

```
21  K_SEM_DEFINE(timing, 1, 1);
22  //define timer that gives timing sem every 100ms
23  //...
24
25  while(k_sem_take(&timing, K_FOREVER) == 0) {
26      // check lem events
27      LemState lem_state = lem.get_state();
28      if (lem_state != LEM_OK) {
29          this->try_switch_state(...);
30      }
31      else if (state == RECOVERABLE_LEM) {
32          this->try_switch_state(...);
33      }
34      // check relay events ...
35      // check ev events ...
36  }
```

main-loop

```
21  K_SEM_DEFINE(timing, 1, 1);
22  //define timer that gives timing sem every 100ms
23  //...
24
25  while(k_sem_take(&timing, K_FOREVER) == 0) {
26      // check lem events
27      LemState lem_state = lem.get_state();
28      if (lem_state != LEM_OK) {
29          this->try_switch_state(...);
30      }
31      else if (state == RECOVERABLE_LEM) {
32          this->try_switch_state(...);
33      }
34      // check relay events ...
35      // check ev events ...
36  }
```

- Only deterministic if execution paths take less than 100ms

main-loop

- conclusion
  - lots of intertwined logic (e.g down calls)
  - lots of state to keep track of
  - priorities implicit
  - difficult to read, extend and maintain
  - data passing difficult

threaded-loops

- multiple threads (for each module/priority)
- per-module state
- loop
  - check module state
  - check pending events for module
  - acts
  - switches state
  - communicate to other threads/modules

threaded-loops

```
1  //LEM THREAD
2  K_SEM_DEFINE(timing, 1, 1);
3  //define timer that gives timing sem every 100ms
4  //...
5
6  while(k_sem_take(&timing, K_FOREVER) == 0) {
7      // check lem events
8      LemState lem_state = this->get_state();
9      if (lem_state != LEM_OK) {
10         this->try_switch_state(...);
11     }
12     else if (state == RECOVERABLE_LEM) {
13         this->try_switch_state(...);
14     }
15 }
16
17 void LEM::try_switch_state(LemState new_state) {
18     // ... do state machine stuff
19
20     this->parent.notify_lem_state_change(new_state);
21 }
```

threaded-loops

```
1  //LEM THREAD
2  K_SEM_DEFINE(timing, 1, 1);
3  //define timer that gives timing sem every 100ms
4  //...
5
6  while(k_sem_take(&timing, K_FOREVER) == 0) {
7      // check lem events
8      LemState lem_state = this->get_state();
9      if (lem_state != LEM_OK) {
10         this->try_switch_state(...);
11     }
12     else if (state == RECOVERABLE_LEM) {
13         this->try_switch_state(...);
14     }
15 }
16
17 void LEM::try_switch_state(LemState new_state) {
18     // ... do state machine stuff
19
20     this->parent.notify_lem_state_change(new_state);
21 }
```

- explicit upcall
- synchronization in parent module needed

threaded-loops

```
27  void LEM::try_switch_state(LemState state) {
28      // do state machine stuff
29
30      k_msgq_put(&this->parent_queue,
31          &state, K_FOREVER);
32  }
33
34
35  //PARENT THREAD
36  int main() {
37      // check all queues
38      int rc;
39      rc = k_msgq_get(&this->lem_queue,
40          &lem_state, K_NO_WAIT);
41      if (rc == 0) {
42          //handle lem event
43      }
44      rc = k_msgq_get(&this->relay_queue,
45          &relay_state, K_NO_WAIT);
46      //...
47  }
```

threaded-loops

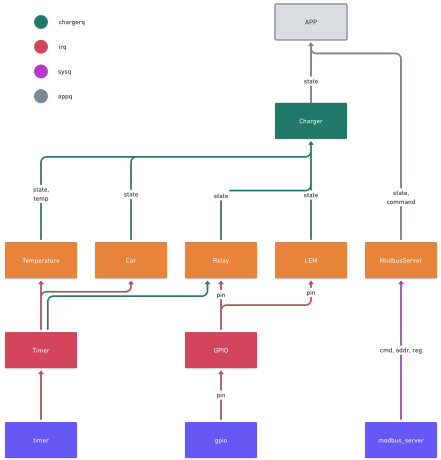
```
27  void LEM::try_switch_state(LemState state) {
28      // do state machine stuff
29
30      k_msgq_put(&this->parent_queue,
31          &state, K_FOREVER);
32  }
33
34
35  //PARENT THREAD
36  int main() {
37      // check all queues
38      int rc;
39      rc = k_msgq_get(&this->lem_queue,
40          &lem_state, K_NO_WAIT);
41      if (rc == 0) {
42          //handle lem event
43      }
44      rc = k_msgq_get(&this->relay_queue,
45          &relay_state, K_NO_WAIT);
46      //...
47  }
```

- more abstract, but still upcall
- priority inversion in parent
- parent implicit priorities
- type unsafe (can be fixed with C++)

- conclusion
  - more modular
  - explicit but static priorities per module
  - for proper seperation a lot of overhead
  - data passing expensive
  - software-engineering wise still not perfect

event-based

- shared workqueue threads
- per-module state
- “loop”
  - react to incomming event
  - switch state
  - generate event



event-based

```
2  void Lem::handle(GPIOEvent &event, GPIOEvent::GeneratorType &generator) {
3      switch (state) {
4          case LEM_OK:
5              this->try_state_change(LEM_FAULT);
6              break;
7      //...
8  }
9
10 void Lem::try_state_change(LemState state) {
11     // handle state change
12     //...
13
14     LemEvent event;
15     event.new_state = this->state;
16     NOTIFY(event);
17 }
18
```

event-based

```
20  class Charger : public EventListener<LemEvent> //...
21  private:
22      EventStreamQueue<LemEvent, 1> lem_event_q;
23      //EventStreamDirectSync<LemEvent> lem_event_q; //direct-call alternative
24      void handle(LemEvent &event, LemEvent::GeneratorType &lem) override;
25      Charger(WorkerQ &workerq)
26          : //...
27          , lem_event_q(*this, lem, RingBufferPolicy::KEEP_LAST, workerq)
28          //, lem_event_q(*this, lem) //direct-call alternative
29      {}
30  };
31
32
33  // runs in workerq or irq depending on line 22/23
34  void Charger::handle(LemEvent &event, LemEvent::GeneratorType &lem) {
35      if (event.new_state != LEM_OK) {
36          this->try_state_change(CHARGER_FAULT_RECOVERABLE_EXTERNALLY_LEM);
37          return;
38      }
39  }
```

event-based

- conclusion
  - data passing lowest overhead and easy
  - modular
  - explicit priorities and thread seperation
  - no tuning of buffers needed
  - no up or downcalls needed, just a link/relationship
  - type-safe

Demonstration

Sync

```
1  struct Radio1EventData { int fm1; };
2  using Radio1Event = GenEvent<BaseEvent, Radio1EventData>;
3
4  class Radio1 : public EventGeneratorFull<Radio1Event> {
5      void tune1(int fm1) { // ~30 instructions
6          Radio1Event e;
7          e.get<Radio1Event>().fm1 = fm1;
8          NOTIFY(e);
9      }
10 };
11
12 class BaseListener : EventListener<BaseEvent> {
13     EventStreamDirectSync<BaseEvent> event_q;
14     int event_cnt = 0;
15     // ~10 instructions (including event_q.handle)
16     void handle(BaseEvent &event, BaseEvent::GeneratorType &generator) {
17         event_cnt++;
18     }
19     BaseListener(BaseEvent::GeneratorType &target)
20         : event_q(*this, target) {}
21 };
```

Sync

```
25  int demo_main(const struct device *_) {
26      Radio1 radio;
27      BaseListener listener(radio);
28      PRINT_SIZE(radio); //16
29      PRINT_SIZE(listener); //36
30
31      LOG_DBG("Count: %d", listener.event_cnt); // 0
32      radio.tune1(50);
33      LOG_DBG("Count: %d", listener.event_cnt); // 1
34
35      return 0;
36  }
```

Async

```
1  struct Radio1EventData { int fm1; };
2  using Radio1Event = GenEvent<BaseEvent, Radio1EventData>;
3
4  class Radio1 : public EventGeneratorFull<Radio1Event> {
5      void tune1(int fm1) { // ~30 instructions
6          Radio1Event e;
7          e.get<Radio1Event>().fm1 = fm1;
8          NOTIFY(e);
9      }
10 };
11
12 class BaseListener : EventListener<BaseEvent> {
13     EventStreamQueue<BaseEvent, 1> event_q;
14     int event_cnt = 0;
15     // ~40 caller, ~30 callee thread
16     void handle(BaseEvent &event, BaseEvent::GeneratorType &generator) {
17         event_cnt++;
18     }
19     BaseListener(BaseEvent::GeneratorType &target)
20         : event_q(*this, target, RingBufferPolicy::KEEP_LAST, WorkerQSys) {}
21 };
```

Async

```
25  int demo_main(const struct device *_) {
26      Radio1 radio;
27      BaseListener listener(radio);
28      PRINT_SIZE(radio); //16
29      PRINT_SIZE(listener); //112
30
31      LOG_DBG("Count: %d", listener.event_cnt); // 0
32      radio.tune1(50);
33      LOG_DBG("Count: %d", listener.event_cnt); // 0
34      k_msleep(1000);
35      LOG_DBG("Count: %d", listener.event_cnt); // 1
36
37      return 0;
38  }
```

Implementation

Requirements

- no heap allocation
- infinite listeners per generator
- infinite generators per listener
- asynchronous event handling - generation
  - dynamic queue sizes
  - different dropping strategies
- every listener receives every event
- listeners can access generator
- listeners know if and how many events are dropped
- as much as possible done/checked during compile time
- events are not necessarily specific to generators
- hierarchical events
- events can pass data

Overview

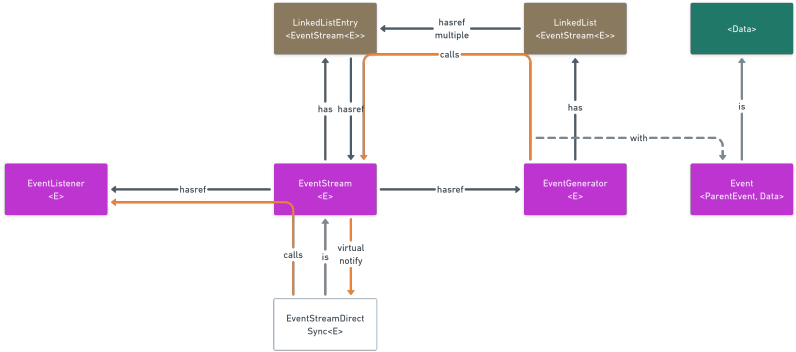


Figure: event-system with direct-call

Overview

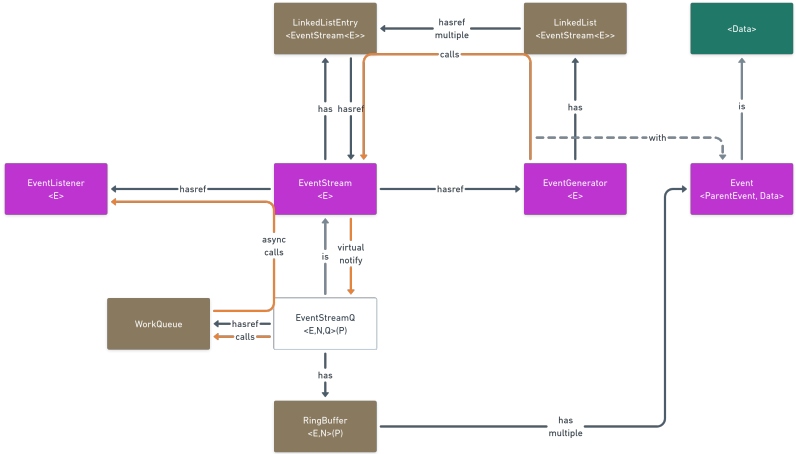


Figure: event-system with async-call

Instantiation

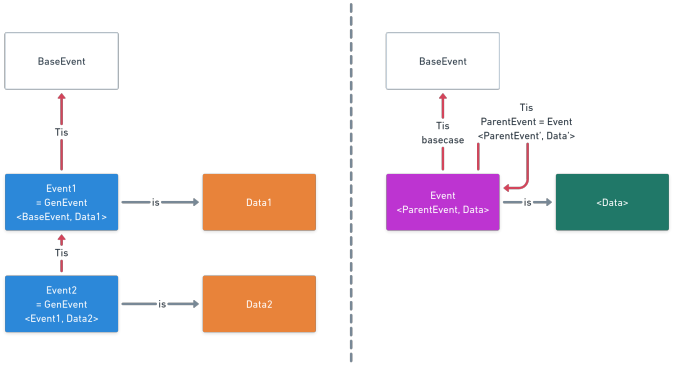


Figure: event instantiation

Instantiation

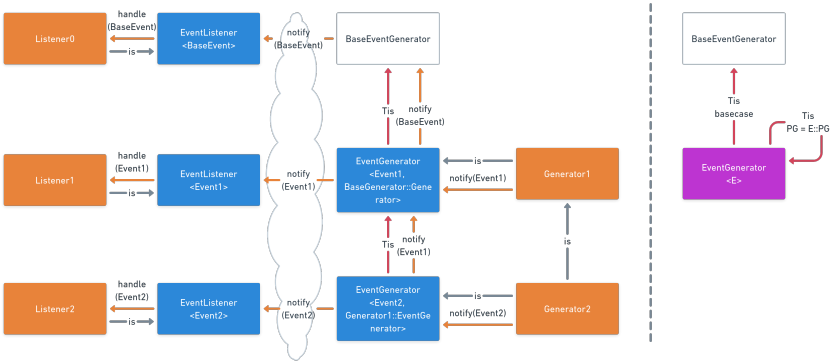


Figure: generator/listener instantiation

Conclusions

Conclusions

- event-based architectures can be useful to create safe, low-footprint embedded applications

Conclusions

- event-based architectures can be useful to create safe, low-footprint embedded applications
- events cooperate well with RTOS and statemachines

Conclusions

- event-based architectures can be useful to create safe, low-footprint embedded applications
- events cooperate well with RTOS and statemachines
- C++20 and Zephyr make it reasonably easy to implement event-based architectures

THANK YOU!

Questions?

https://sched.co/10CEa

Appendix

Event inheritance

```
1  struct Radio1EventData { int fm1; };
2  struct Radio2EventData { int fm2; };
3  struct Pillow1EventData { int sft1; };
4
5  using Radio1Event = GenEvent<BaseEvent, Radio1EventData>;
6  using Radio2Event = GenEvent<Radio1Event, Radio2EventData>;
7  using Pillow1Event = GenEvent<BaseEvent, Pillow1EventData>;
8
9  class Radio1 : public EventGeneratorFull<Radio1Event> //...
10 class Radio2 : public EventGeneratorExtendFull<Radio1, Radio2Event> //...
11 class Pillow1 : public EventGeneratorFull<Pillow1Event> //....
12
13 class Pillow1Radio2withBar
14     : public EventDiamond<
15         Pillow1,
16         Radio2,
17         EventGeneratorFull<BarEvent>
18     > //...
```

Implementation - Events - template inheritance

```
1  template<isEvent T_BASE, typename T_EXT>
2  struct GenEvent : public T_EXT {
3      using ParentType = T_BASE;
4      using GeneratorType = Generator<GenEvent<T_BASE, T_EXT>>;
5      T_BASE m_base_event;
6
7      template<isEvent T_TARGET>
8      T_TARGET& get() {
9          if constexpr (std::is_same_v<GenEvent<T_BASE, T_EXT>, T_TARGET>) {
10             return *this;
11         }
12         else if constexpr (std::is_same_v<T_BASE, T_TARGET>) {
13             return this->m_base_event;
14         }
15         else if constexpr (std::is_same_v<BaseEvent, T_BASE>) {
16             static_assert(!std::is_same_v<BaseEvent, T_BASE>, "not a base of this event");
17             return *((T_TARGET*)nullptr);
18         }
19         else {
20             return this->m_base_event.template get<T_TARGET>();
21         }
```

Implementation - Generators

```
25  template <isEvent E>
26  class Generator {
27  private:
28      using EventStreamType = EventStream<E>;
29      LinkedList<EventStreamType> event_streams;
30  public:
31      using ParentGenerator = E::ParentType::GeneratorType;
32      using ParentType = ParentGenerator;
33      using GeneratorType = Generator<E>;
34
35      bool subscribe(LinkedListEntry<EventStreamType>& event_stream_node) {
36          this->event_streams.add_prev(event_stream_node);
37          return true;
38      }
39
40      template<typename GeneratorObjectPtr>
41      void notify(E& e) {
42          //...
43      }
44  };
```

Implementation - Generators - recursive notify

```
46  template<typename GeneratorObjectPtr>
47  void notify(E& e) {
48      // check that we are passed something that might actually be a child of this
49      static_assert(std::is_base_of_v<
50          GeneratorType,
51          typename std::remove_pointer<GeneratorObjectPtr>::type
52      >);
53
54      // notify all listeners indirectly by feeding the event streams
55      for (EventStreamType& event_stream : this->event_streams) {
56          event_stream.handle(e);
57      }
58
59      // pass further to top
60      if constexpr (!std::is_same_v<ParentGenerator, void>) {
61          static_cast<GeneratorObjectPtr>(this)->ParentGenerator
62              ::template notify<GeneratorObjectPtr>(e.get_base_event());
63      }
64  }
```

Implementation - Streams - async

```
66  template<isEvent E, size_t N>
67  class EventStreamQueue : public EventStream<E> {
68      RingBuffer<E,N> events;
69      WorkQ<&EventStreamQueue<E,N>::handle_async> async;
70      EventStreamQueue(EventListener<E> &listener, EventStream<E>::G &generator,
71          RingBufferPolicy replacement_policy, WorkerQ &q)
72          : EventStream<E>(listener, generator)
73          , events(replacement_policy)
74          , async(q, *this) {}
75
76      void handle_async(void *obj) {
77          std::optional<E> eventopt = events.pop();
78          this->listener.handle(*eventopt, this->generator);
79      }
80
81      void handle(E const& event) override {
82          if(!events.try_push(event)) {
83              return;
84          }
85          this->async.schedule(K_NO_WAIT, false);
86      }
```