Document Number: NXXXX
Date: 2014-11-07
Project: SG13 - IO
Reply to: Cleiton Santoia Silva
<cleitonsantoia@gmail.com>
Daniel Auresco
<auresco@gmail.com>

IO device requirements for C++.

Contents

1 Intro [intro]

In this paper we present an I/O device requirement specification to C++, our basic approach is similar to container requirements, that does not present a "container" base class, in the same sense, instead of proposing some "device" class, we propose some basic interface that all devices must implement.

1.1 Simple one-page sample [intro.simple]

Let's get one of the most common type of devices, the Screen, in the most naive approach:

```
class Screen {
 int color(int x, int y);
 void set_color(int x, int y, int color);

 int cur_res_h();
 int cur_res_v();
 void set_res(int x, int y);

 vector<std::pair<int x, int y>> get_possible_res();   // all possible resolutions
 map<std::string, std::string> get_all_hw_features();  // all hardware features
};
```

After analyzing this example (and comparing to other possible devices), we propose to organize device capabilities into some groups, for instance, "IO", "features", "possible configurations".

```
class device {
 const container_feat<_Feature>& features() const; // get features of the device
 const container_cfg<_Config>&   configs() const;  // get possible configurations

 void set_config(const _Config&);                  // set current config
 const _Config& get_config() const;                // get current config

 const _Value& operator[](const _Coord& c) const;  // I/O coordinate device get
       _Value& operator[](const _Coord& c);        // I/O coordinate device set
};
```

Fitting "Screen" in the standard device:

```
typedef uint32_t RGBA;
typedef std::pair<int, int> Point2D;
typedef std::pair<int, int> Resolution;

class Screen {
 const map<std::string, std::string>& features() const;  // get features of the device
 const vector<Resolution>&            configs() const;   // get possible configurations

 void set_config(const Resolution&);                     // set current config
 const std::pair<int x, int y>& get_config() const;      // get current config

 const RGBA& operator[](const Point2D& c) const;         // I/O coordinate device get
       RGBA& operator[](const Point2D& c);               // I/O coordinate device set
};
```

2 Motivation and impact [mi]

2.1 Motivation [motivation]

Today, many different types of devices exists : Smartfones, Cameras, Sensors, Monitors, Keyboard, Etc..., and this number is growing fast, and, as expected, they differ in many ways. The standard I/O library is mostly (if not entirely) focused on streams, and many of this devices cannot be used this way, however, after few decades of evolution, C++ has and now a new set of tools and ideas that we can be used to solve reasonably the problem:

— Encapsulating devices in some standard interface.

— Organizing basic common capabilities in groups.

— Virtual high-level devices can encapsulate other (more low-level) devices.

— Allow the use of iterators and algorithms on devices.

— Allow vendors to present any extra and specific functionality they need, while maintain standard interface.

2.2 Impact on Standard [impact]

This proposal establishes the device requirements, in the same sort of section [ISO14, N3797] 23.3 “Container requirements” of the standard. Our intent is to present guidelines to be used when device implementation must be created.

3 Design decisions [design]

3.1 General concepts [design.general]

The device is some object that the program uses to communicate to the external world. It may vary from hardware abstraction like Screen, Keyboard, a Telescope, an Ocean Buoy, Racing telemetry to virtual "logical" device as a Canvas or a VideoWall with many monitors. Regarding this differences, its possible to define some characteristics that device interfaces must respect. This characteristics are presented here as "device requirements", in the same sense container library do on section 23.2 "Container Requirements" of the standard, this way, not all devices must present all concepts, but when they do, they must respect the predefined interface. We classify the requirements in some groups:

1. Device as an associative container.
2. Streaming devices.
3. Features and Configurations.
4. Synchronicity policy.
5. Acquire/Release policy.
6. Iteration.
7. Device traits.

3.2 Device as an associative container [design.map]

Some devices, the "Screen" for instance, may be used as an associative container from a coordinate type to a value type, in this "Screen" case, it's possible set or get the RGB color from a coordinate.

— These two functions must be used for coordinate-based devices:

```
const _Value& operator[](const _Coord& coord) const;
      _Value& operator[](const _Coord& coord);
```

— More extensive example:

```
struct point2D {
 int x;
 int y;
};
typedef uint32_t RGBA;
class Screen {
public:
 const RGBA& operator[](const point2D& coord) const;
       RGBA& operator[](const point2D& coord);
};
```

— Another example: a traffic light.

```
enum TrafficLightStatus { on, off, flashing };
enum TrafficLightIndex { red, yellow, green };
class TrafficLight {
public:
 const TrafficLightStatus& operator[](const TrafficLightIndex& coord) const;
       TrafficLightStatus& operator[](const TrafficLightIndex& coord);
};
```

3.3 Device as a stream [design.stream]

First, we want to explain that we are not endorsing or avoiding the use of I/O streams, we are aware that exists a discussion if this schema should continue or be deprecated, anyway, if it continues we present a simple standard for it. The common way of using streams are overloading `operator «()` and `operator»()` to many typed value I/O;

Some requirements for stream devices:

```
device& operator<<(const value_type&);
device& operator>>(value_type&);
```

It's possible to define a surface as a stream device.

```
class Square { /*...*/ };
class Elipse { /*...*/ };
class Line { /*...*/ };

class surface { // device
public:
  Canvas& canvas();
  surface& operator<<(const Square&);
  surface& operator<<(const Elipse&);
  surface& operator<<(const Line&);
};
```

Another example using only one type of DrawPrimitive:

```
class DrawPrimitive { /* some pimpl idiom */ };

class surface { // device
public:
  Canvas& canvas();
  surface& operator<<(const DrawPrimitive&);
};
```

This way, you just `std::copy()` from a container (`some std::vector<DrawPrimitive>`) to surface and get the result in `surface::canvas()`.

3.4 Device configuration [design.config]

A configuration is a possible state the device may assume, that changes the behavior of subsequent I/O, changes in screen resolution, or changes in mouse sensibility, or page configuration of a printer. We propose three member functions on each device, get and set the current configuration, and one to retrieve a container of possible configurations, this returned container is implementation defined, however it is presented here in the way that can be replaced by many common containers from std, like `std::vector<_Config>`.

— Requirements of the return object from `configs()` function:

```
template<typename _Config>
class device_config { // the name is not defined
public:
 typedef unsigned    size_type;
 typedef _Config     value_type;
 typedef std::iterator<std::input_iterator_tag, const _Config> const_iterator;

 size_type size() const;
 const_iterator begin() const;
 const_iterator end() const;
};
```

— More complete example:

```
typedef std::tuple<int, int> Point2D;
typedef unsigned RGBA;
struct ScreenConfig {
 int res_x;
 int res_y;
 int frequency;
 enum Orientation { vertical, horizontal } orientation;
};

class Screen{
public:
 std::vector<ScreenConfig>& configs(); // the function to get configurations
 void set_config(const ScreenConfig&);
 const ScreenConfig& get_config() const;
 .
 .
 .
};
```

— Another example

```
typedef map<std::string, boost::any> config_bag_type;
class device {
public:
 config_bag_type& config();
 void set_config(const config_bag_type&);
 const config_bag_type& get_config() const;
};

void init(device& d) {
 auto c = d.config();
 c["parm1"] = 1;
 c["parm2"] = "some_important_value";
 d.set_config(c);
}
```

This way you could set as many parameters you need at once, and `get_config()` and `configs()` may return the same map;

— A ranged configuration:

```
struct range {
```

```
 int min;
 int max;
};

typedef map<std::string, int>  configuration;
typedf map<std::string, range> possible_config;

class device {
public:
 possible_config& config();
 void set_config(const configuration&);
 const configuration& get_config() const;
};
```

3.5 Device features [design.feat]

The features are the immutable capabilities of the device. A graphic card may have a `Texture::bltfast()` function hardware acceleration or not, or a maximum of 70Hz of video frequency. The list of features may be implemented as an associative iterable container. The device feature type must implement the following interface (may be implemented as `std::map<>` if needed).

Requirements to device feature container:

```
template<typename _Key, typename _Value>
class device_features_type<_Key, _Value> {
public:
 typedef _Key     key_type;
 typedef _Value   mapped_type;
 typedef std::iterator<std::input_iterator_tag, const pair<key_type, mapped_type>> const_iterator;

 const_iterator begin() const;
 const_iterator end() const;
 size_type size() const;
 const_iterator find(const key_type&) const;
};
```

And the device type must present a public const member function `features()` that return the container of features:

```
template<typename _Value, typename _Coordinate>
struct device_tag {
public:
 typedef _Coordinate coord_type;
 typedef _Value      value_type;
};
typedef std::tuple<int, int> Point2D;
typedef unsigned RGBA;
class screen : device_tag<RGBA, Point2D> {
public:
 const std::map<std::string, std::string>& features(); // map matches the requirements
 const value_type& operator[](const coord_type& coord) const;
       value_type& operator[](const coord_type& coord);
};
```

3.6 Device goods [design.goods]

Those devices that need to be checked before I/O.

```
class device {
 explicit operator bool() const { return good(); };
 bool operator !() const { return !good(); }
 boolean good() const;
};
```

Not much new here, just the similar from stream library, if you ask a device and it respond "I'm good" means that is is ready for I/O, but it does not imply that the subsequent I/O will not throw some exception, some thing may happens between those calls.

3.7 Device synchronicity [design.sync]

If some device provides functions for "event driven architecture", the case that the device will send a signal to the program when something happens, this signal might be implemented as a `std::condition_variable` wrapper, hold and locked by the device itself, and used via common wait methods (wait, wait_for, wait_until) in similar way. However it's possible to use return of the wait methods, also to bring information about the event that happened.

— Requirements for wait methods:

```
template<typename _EventType>
struct device_waiter_type {
 EventType wait();
 template <class Predicate> EventType wait (Predicate pred);

 template <class Rep, class Period>
 EventType wait_for(const chrono::duration<Rep,Period>& rel_time);

 template <class Rep, class Period, class Predicate>
 EventType wait_for(const chrono::duration<Rep,Period>& rel_time, Predicate pred);

 template <class Clock, class Duration>
 EventType wait_until (const chrono::time_point<Clock,Duration>& abs_time);

 template <class Clock, class Duration, class Predicate>
 EventType wait_until (const chrono::time_point<Clock,Duration>& abs_time, Predicate pred);
};
```

— Sample:

```
struct EventType {
 std::cv_status status;
 enum status_t {KEY_UP, KEY_DOWN} status;
 char key;
 int scan_row_h;
 int scan_row_v;
};

class keyboard { // the keyboard device
public:
        device_waiter_type<EventType>& waiter();
};

void foo(keyboard& k) { // how to use
 while (k.good()) {
  auto ev = k.waiter().wait();// wait until something happens
```

```
  if ( ev.status == EventType::status_t::KEY_UP ) {
   process_key_up(ev);
  }
 }
 throw std::exception("keyboard_problem");
}
```

— PnP sample:

```
class USBPenDrive {
public:
 filesystem::path root;
};

struct USBPenDriveEvent { // Event
 std::cv_status;
 enum status_t {Connect, Disconnect} status;
 USBPenDrive drive; // when you connect or disconnect, this is the root of the pen-drive.
};

class USBDriveController { // Device
public:
        device_waiter_type<USBPenDriveEvent>& waiter();
};

void foo(USBDriveController& controller) { // how to use
 while (controller.good()) {
  auto ev = controller.waiter().wait();// wait until something happens
  if ( ev.status == USBPenDriveEvent::status_t::Connect ) {
   open_filesystem_window(ev.drive.root);
  }
 }
 throw std::exception("controller problem");
}
```

One important issue in this wait approach is the use of thread resource, instead of some common `register_callback(event_type callback)` technique, we make the caller calls for `wait()` to avoid two problems:

1. If we use callback, your program will work inside the device thread resource, it's not good ! You should provide your own thread for this;
2. If some exception occurs before the callback, it more difficult to notify the program what happened;

Another issue is that, if you put all methods of wait directly inside the device class, it's interface may become messy and big, and in next topic we will talk about waited acquires, that will make it more messy.

3.8 Device acquire policy [design.acquire]

Our advice is: "you should prefer RAII and don't acquire directly nor release it". But, for those device design that need to acquire and release:

1. The basic example should be like this:

```
class Device {
public:
 void acquire();
 void release();
};
```

2. But in reality, acquire may be some method that need more control, since it can take a time and you may want a timeout for everything that take time, we propose the acquire method similar to the six `wait()` variants from `std::condition_variable`.
   These is the standard requirements for acquire methods:

```
template<typename _AcquireEventType> // possibly _AcquireEventType == void
struct device_acquire_waiter_type {
 _AcquireEventType wait();
 template <class Predicate> _AcquireEventType wait (Predicate pred);

 template <class Rep, class Period>
 _AcquireEventType wait_for(const chrono::duration<Rep,Period>& rel_time);

 template <class Rep, class Period, class Predicate>
 _AcquireEventType wait_for(const chrono::duration<Rep,Period>& rel_time, Predicate pred);

 template <class Clock, class Duration>
 _AcquireEventType wait_until (const chrono::time_point<Clock,Duration>& abs_time);

 template <class Clock, class Duration, class Predicate>
 _AcquireEventType
 wait_until (const chrono::time_point<Clock,Duration>& abs_time, Predicate pred);

};

class Release_Result_Type {...};

template<typename _Event>
class device_acquire_waiter_type {...};

class device {
public:
 class Event {...};
 device_acquire_waiter_type<Event>& acquirer();
 Release_Result_Type release();
};
```

   So you can acquire and release as you need. The the `release()` implementation should end the wait iterators pointed to the device, free wait locks and turn `device::good()` to false, also `release()` may also return some information and after you call it.

3.9 Iteration [design.iteration]

We propose few new features to help the usage of standard algorithms with devices.

1. Coordinate input/output iterators

   The device may implement common iterators `begin()`, `end()`, `rbegin()`, `rend()` (and their const versions), iterating over a pair of coordinate(key) and value(mapped), same as a `std::map<coord,`

value>.
This is an example of a motion capture suit with many motion "capture points devices" indexed by a integer number;

```
typedef std::tuple<double, double, double> Point3D;
class motion_capture_array_device {
public:
 const Point3D& operator[](int index) const;
 const_iterator begin() const; // implement iterator to const std::pair<int, Point3D>
 const_iterator end() const;   // implement iterator to const std::pair<int, Point3D>
};

void foo(const motion_capture_array_device& mc_array) {
        for (auto& mc_item : mc_array) {
                // do something important with mc_item, like store or something
        }
}
```

2. Stream input/output iterators

   A camera device may be implemented as stream of frames, where you pass each frame on a filter and then save to a file.

```
class Frame {
 ...
 friend std::ostream& operator<<(std::ostream&, const Frame& f);
};

class Camera {
 const_iterator current() const; // implement iterator to a Frame
 const_iterator end() const;     // implement iterator to a Frame
};

const Frame& apply_filter(const Frame& frame) {
 ...
};

void foo(const Camera& c, const std::string& nm) {
 ofstream out(nm);
 // c.begin() == c.end() whenever the device is released/closed (probably for other thread)
 std::transform(c.current(), c.end(), std::ostream_iterator<Frame> (out), apply_filter );
};
```

3. Wait iterators

   This is a complex topic that was not fully explored in this proposal, probably a topic for SG1 group, many todo's here, please see [GS08, ParallelIterator]. The idea here is to bring another problem over the shoulders of the iterators: iterate when events/signals occurs. This bring us only two problems, when "wait" and when "not wait";

   Some guidelines to wait iterators:

   a) The iterators must become == end() when the device is released/close
   b) The release/close may occur during a wait operation, in this case the iterator must become == end()

So, we should put wait before each check, what leads us to:

a) When wait: on compare, on move++, on constructor and output `operator*()`;

b) When not wait: input `operator*()`;

The main motivation for this two rules happens when we look to some algorithms in C++ (in my compiler library):

a) In `std::for_each()`, if we wait on ==, ++ and not wait on *, we are fine, if we wait in *, then some wait may incur in a ==end() operation and the `_Func()` will be called over an invalid iterator, this behavior is ok;

```
for (; _First != _Last; ++_First)
 _Func(*_First);
```

b) In `std::transform()` Same case as `std::for_each()`

```
for (; _First != _Last; ++_First, ++_Dest)
 *_Dest = _Func(*_First);
```

c) In `std::count_if()` the last statement *_Dest++ will wait the output in `operator* ()`, this is fine.

```
for (; _First != _Last; ++_First)
 if (_Pred(*_First))
        *_Dest++ = *_First;
```

d) In the `std::merge()` there is a little more code, but the `*_First2++` is the important part, this case, the input will come from `*First2`, without wait and `postfix operator++()` will wait (even if the result is ready), this behavior is not ok ! But... it's the best we got so far.

```
*_Dest++ = *_First2++;
```

Back to the sample, the same case again, a camera as a stream of frames, but we want also a timeout for the read operation. We could implement "wait_iterators" when every increment/decrement operation wait. One should expect that wait operators may return a different type from common begin/end iterators, it's pointed type is whatever the `device::wait()` method returns.

First we show some implementation of a wait_iterator and wait_for_iterator.

```
template<typename _Device, typename _T>
class device_wait_iterator {
public:
 // when create, you must wait to get some item
 device_wait_iterator(_Device& device) : device_(&device) {}
 device_wait_iterator() : device_(nullptr) {}

 device_wait_iterator& operator++();
 device_wait_iterator& operator++(int);
 const _T& operator*() const;
 bool operator==( const device_wait_iterator& w );
};

template<typename _Device, typename _T, typename _Duration>
class wait_for_iterator {
  _Duration duration;
```

```
public:
 wait_for_iterator(_Device& device, _Duration& rel_time );

 wait_for_iterator& operator++() {
  result = object_.wait(rel_time); // <- here is the difference in the call
        ...
 }
 ...
}
//The wait_until_iterator is analogue to wait_for_iterator.
```

Example of use the wait_iterator.

```
class Frame {
 ...
 friend std::ostream& operator<<(std::ostream&, const Frame& f);
};

class Event {
 Frame frame;
 std::cv_status status;
 friend std::ostream& operator<<(std::ostream&, const Frame& f);
};

class Camera {
 const_wait_iterator wcurrent() const; // implement iterator to Event
 const_wait_iterator wend() const;     // implement iterator to Event

 Event wait();
};

const Frame& apply_filter(const Event& event) {
 if (event.status == no_timeout) {
   ...
 } else {
   static Frame dummy_frame;
   return dummy_frame;
 }
};

ostream& operator<<(ostream&, const Frame&) {...}

void foo(const Camera& c, const std::string& nm, Surface& s) {
 ofstream out(nm);
 // c.wcurrent() == c.wend() whenever the device is released/closed (probably for other thread)
 std::transform(c.wcurrent(), c.wend(), std::ostream_iterator<Frame> (out), apply_filter );
};
```

3.10 Device traits [design.traits]

Devices may vary a lot, to organize them a little we propose the following traits:

— `static constexp bool is_acquirable()` for those who implement `acquire()` methods;

— `static constexp bool is_associative()` for devices that uses have `operator[]` defined;

— `static constexp bool is_input()` for devices that produces some input;

— `static constexp bool is_iterable()` for devices that can be iterated and implements `begin()` and `end()` functions;

— `static constexp bool is_output()` for devices that can be used as output;

— `static constexp bool is_stream()` for streams that have `operator«()` and `operator»()` implemented;

— `static constexp bool has_waiter()` for those who implement `waiter()` methods;

— `static constexp bool has_acquirer()` for those who implement `acquirer()` methods, this is important because the device may have void return type for acquire and release functions, but still got the functions;

— `static constexp bool has_features()` for those who implement `features()`;

— `static constexp bool has_config()` for those who implement `configs()`;

3.11 Device composition [design.composed]

Here we present some other device composition, none of them are part of standard, these are just some samples.

1. Composite devices:

```
class Desktop {
public:
 class Video {...};     // video device definition
 class Keyboard {...}; // keyboard device definition
 class Mouse {...};     // mouse device definition

 Video& video();
 Keyboard& keyboard();
 Mouse& mouse();
};

class Tablet {
public:
 TouchVideo& video();
 SD& sd();
 Battery& battery();
 NetworkInterface& net_if();
};
```

2. Device array, the Star Trek Voyager Astrometrics:

```
struct SensorStatus {
 double sensor_data;
 enum {
  ok,
  problem
 } sensor_status;
};

struct Polar3D {
 double angle_x;
 double angle_y;
 double distance;
```

```
};

class VoyagerAstrometricSensor { // Device
public:
 const SensorStatus& operator[](const Polar3D&) const;
};

class VoyagerAstrometricSensorArray { // Array of devices as a Device
public:
 const VoyagerAstrometricSensor& operator[](const int& index) const;
};
```

3. Oceanic buoys:

```
struct Buoy_data {
 float gps_height;
 float water_temperature;
 float air_temperature;
 float air_pressue;
 float wind_speed;
 struct {
  int gps_height_status : 1;
  int water_temperature_status : 1;
  int air_temperature_status : 1;
  int air_pressue_status : 1;
  int wind_speed_status : 1;
 } bouy_status;
};

struct LatLongRad {
 float latitude;
 float longitude;
 float radius;
};

class OceanBuoyArray {
public:
 const Bouy_data& operator[](const int& n) const; // pick n-esim buoy from array
};
class OceanBuoyMap {
public:
 const OceanBuoyArray& operator[](const LatLongRad& x) const; // pick nearest buoys from x
};
```

4. Another device array, the multi-monitor (video wall). Imagine a set of 8 monitors as an array of 2x4. One could create a virtual device “VideoWall” as an array of monitors.

```
typedef std::tuple<int, int> Point2D;
typedef unsigned RGBA;
struct ScreenConfig {
 int res_x;
 int res_y;
 int frequency;
 enum Orientation { vertical, horizontal } orientation;
};
```

```
class Screen { // screen device
 std::vector<ScreenConfig> configs() const;
 const RGBA& operator[](const Point2D& c) const;
       RGBA& operator[](const Point2D& c);
};

typedef std::pair<int, int> cell_pos;
struct VideoWallCell {
 ScreenConfig screen_config;
 cell_pos pos;
};

struct VideoWallConfig {
 int cells_x; // total of horizontal screens
 int cells_y; // total of vertical screens
 std::vector<VideoWallCell> cells;
};

class VideoWall { // device May act as a 2D array of Screens
 std::vector<VideoWallConfig> configs();
 const Screen& operator[](const cell_pos& c) const; // get the Screen at pos x,y
};
```

4 Technical Specification [spec]

4.1 Full specification [spec.full]

The full specification of feature container, config container and device itself.

1. Device features structure:

```
template<typename _Key, typename _Mapped>
class device_feature { // The name of the class is not part of standard
public:
 typedef _Key    key_type;
 typedef _Mapped mapped_type;
 typedef std::iterator<std::input_iterator_tag,
                       std::pair<key_type, mapped_type>> const_iterator;
 typedef [some_int_type] size_type;

 const_iterator begin() const;
 const_iterator end() const;
 size_type size() const;
 const_iterator find(const key_type&) const;
};
```

2. The device configuration structure:

```
template<typename _Config>
class device_config { // The name of the class is not part of standard
public:
 typedef _Config     config_type;
 typedef std::iterator<std::input_iterator_tag, const _Config> const_iterator;
 typedef [some_int_type] size_type;

 const_iterator begin() const;
 const_iterator end() const;
 size_type size() const;
};
```

3. The waiter structure

```
template<typename _Event>
class device_wait_type { // The name of the class is not part of standard
public:
 typedef _Event            event_type;          // void if not applicable

 // iterators
 typedef std::iterator<std::xxx_iterator_tag, event_type> wait_iterator;
 typedef std::iterator<std::xxx_iterator_tag, event_type> wait_for_iterator;
 typedef std::iterator<std::xxx_iterator_tag, event_type> wait_until_iterator;

 // const iterators
 typedef std::iterator<std::xxx_iterator_tag, event_type> const_wait_iterator;
 typedef std::iterator<std::xxx_iterator_tag, event_type> const_wait_for_iterator;
 typedef std::iterator<std::xxx_iterator_tag, event_type> const_wait_until_iterator;
```

```
                            event_type wait();
 template <class Predicate> event_type wait(Predicate pred);
 template <class Rep, class Period>
 event_type wait_for(const chrono::duration<Rep, Period>& rel_time);
 template <class Rep, class Period, class Predicate>
 event_type wait_for(const chrono::duration<Rep, Period>& rel_time, Predicate pred);
 template <class Clock, class Duration>
 event_type wait_until(const chrono::time_point<Clock, Duration>& abs_time);
 template <class Clock, class Duration, class Predicate>
 event_type wait_until(const chrono::time_point<Clock, Duration>& abs_time, Predicate pred);

 // wait_iterators
 wait_iterator       wcurrent();
 wait_iterator       wend();
 wait_for_iterator   wfor_current();
 wait_for_iterator   wfor_end();
 wait_until_iterator wuntil_current();
 wait_until_iterator wuntil_end();

 // const analogues
 const_wait_iterator       wcurrent() const;
 const_wait_iterator       wend() const;
 const_wait_for_iterator   wfor_current() const;
 const_wait_for_iterator   wfor_end() const;
 const_wait_until_iterator wuntil_current() const;
 const_wait_until_iterator wuntil_end() const;

 // const analogues
 const_wait_iterator       cwcurrent() const;
 const_wait_iterator       cwend() const;
 const_wait_for_iterator   cwfor_current() const;
 const_wait_for_iterator   cwfor_end() const;
 const_wait_until_iterator cwuntil_current() const;
 const_wait_until_iterator cwuntil_end() const;
};
```

4. The acquirer structure, in this we took out iterators.

```
template<typename _Event>
class device_acquire_type { // The name of the class is not part of standard
public:
 typedef _Event            event_type;          // void if not applicable

                            event_type wait();
 template <class Predicate> event_type wait(Predicate pred);
 template <class Rep, class Period>
 event_type wait_for(const chrono::duration<Rep, Period>& rel_time);
 template <class Rep, class Period, class Predicate>
 event_type wait_for(const chrono::duration<Rep, Period>& rel_time, Predicate pred);
 template <class Clock, class Duration>
 event_type wait_until(const chrono::time_point<Clock, Duration>& abs_time);
 template <class Clock, class Duration, class Predicate>
 event_type wait_until(const chrono::time_point<Clock, Duration>& abs_time, Predicate pred);
};
```

5. Finally the full device structure with all methods, all of them are optional excluding type traits and typedefs, both obligatory.

```
// Minimalist device interface
template<typename _Coordinate,
         typename _Value,
         typename _Config  = void,
         typename _Feature = void,
         typename _Event   = void,
         typename _AcquireEventType = void,
         typename release_type = void>
class device { // the name is not standard
public:
 // traits (obligatory for all devices)
 static constexp bool is_acquirable();
 static constexp bool is_associative();
 static constexp bool is_input();
 static constexp bool is_iterable();
 static constexp bool is_output();
 static constexp bool is_stream();
 static constexp bool has_waiter();
 static constexp bool has_acquirer();
 static constexp bool has_features();
 static constexp bool has_config();

 // typedefs ( obligatory )
 typedef _Coordinate       coord_type;          // void if not applicable
 typedef _Value            value_type;          // void if not applicable
 typedef _Config           config_type;         // void if not applicable
 typedef _Feature          feature_type;        // void if not applicable
 typedef _Event            event_type;          // void if not applicable
 typedef _AcquireEventType acquire_event_type;  // void if not applicable
 typedef _ReleaseEventType release_result_type; // void if not applicable

 typedef device_wait_type<_Event>               waiter_type;   // void if not applicable
 typedef device_acquire_type<_AcquireEventType> acquirer_type; // void if not applicable

 typedef device_feature<_Feature>    feature_container_type;
 typedef device_config<config_type>  config_container_type;

 // iterator types
 typedef std::iterator<std::xxx_iterator_tag, std::pair<coord_type, value_type>> iterator;
 typedef std::iterator<std::xxx_iterator_tag, std::pair<coord_type, value_type>> const_iterator;

 // features
 const feature_container_type&  features() const;

 // config
 const device_conf_type&        configs() const;
 const _Config& get_config() const;
 void           set_config(const _Config&);

 // io ccordinate devices
 const value_type& operator[](const coord_type& c) const;  // get the point in coordinate c
       value_type& operator[](const coord_type& c);        // set the point value in coordinate c
```

```
 // io stream devices, possibly different parameter types than value_type
 device& operator>>(value_type& c);
 device& operator<<(const value_type& c);

 // iterators
 iterator       begin();
 iterator       end();

 // const analogues
 const_iterator begin()  const;
 const_iterator end()    const;
 const_iterator cbegin() const;
 const_iterator cend()   const;

 waiter_type         waiter();
 acquirer_type       acquirer(); // even if acquirer type is void, this can exists
 release_result_type release();  // even if release type is void, this can exists

 // rule of 5 ( if applicable )
 device(const device&);
 device(device&&);
 device& operator=(const device&);
 device& operator=(device&&);
 ~device();
};
```

5 Acknowledgments [ack]

5.1 Few acks [ack.ack]

Bibliography

[GS08] Nasser Giacaman and Oliver Sinnen. Parallel iterator for parallelising object oriented applications. 2008.

[ISO14] C++ ISO. N3797 programming languages c++. Technical report, Programming Language C++, 2014.