INPUT DEVICES FOR 2D GRAPHICS

Document Number:	P0249
Date:	02/05/2016
Reply-To:	brett.searles@attobotics.net

Contributing Authors:	Michael McLaughlin < mikebmcl@gmail.com > Jason Zink < jzink_1@yahoo.com >

I. Table of Contents

II. Introduction

With 2D Graphics, this proposal extends that proposal in discussing input devices, in particular, the framework to support event and interrupt handlers. The proposal is composed of two (2) efforts. The first effort is to provide a framework for event handling. The second effort is to propose a framework for handling interrupts. Even though the two can be separated, there is the need to show the interconnection between the two. So this proposal takes the two concepts and shows how they work together in the process of event handling.

III. Motivation

The purpose of this work is three-fold.

1) To provide a process that C++ can be developed to handle Input-Output (IO) events similar to other UI based languages.
2) To provide developers ways to dynamically add, remove, invalidate and step through both synchronous and asynchronous events without the encumbrance of writing lengthy code.
3) To construct a framework for present and future development of IO devices, its data and its usage of that data.

Other languages strongly support Rapid Application Development of User Interfaces (UI) with canvas or surface objects connected to IO devices to handle events performed on the UI. Therefore, the effort of this standard was to eliminate macros or multiple inheritance supported in existing libraries asnd design a framework that supports a dynamic and customizable approach to describing events. It also outlines a framework to create a sequence of events that are stored in a container. This part of the framework would allow a single device interrupt to fire a sequence of events in real time.

This library is to be a standard for Multi-tasking and Real-Time Operating Systems. The standard could include the encapsulation of interrupts to read and set data that will be used by an event or series of events.

Some of the principles defined in the container that will hold the events have already been discussed in the QT Library, Allegro library and in Boost using the Signals library[Boost]. The main difference here is that these libraries mainly treat their containers as queues, whereas the proposed library uses the flexibility of a **vector** based container.

The framework is defined around a number of base classes. They are all abstract classes because the actual event, device or data retrieved from a device can be dependent upon the manufacturer or graphic developer's intent for the data or usage of the data. Also the use of abstract classes gives flexibility to the standard to provide a framework for current and future implementation of IO devices that will interact with UI.

A. Motivation Scope

The library that is being proposed covers three main problems to solve, which is the ability to

1) Provide a flexible framework to create Events and retrieve information from Input devices ,
2) Provide a collection that can sequence events based on a device interrupt, and
3) Provide a Portable Interrupt Framework

However, as mentioned earlier, the third item can be separated into its own Standard.

1. Scope: Provide a flexible framework to create Events, retrieve information from Input devices

a.. Current practice

Boost Signal Library allows users to create a collection of functions pointers and fires them in the order described by the developer. At first, the connection needs to be declared

```
boost::signals2::signal<void ()> sig;
```

Then the slots are filled

```
sig.connect(&print_args);
sig.connect(&print_sum);
sig.connect(&print_product);
sig.connect(&print_difference);
sig.connect(&print_quotient);
```

or they can be added in a sequence defined by the developer as

```
sig.connect(1, World());  // connect with group 1
sig.connect(0, Hello());  // connect with group 0
```

Then will be called when the signal object is initiated with the arguments that were described in the declaration

```
      sig(5., 3.);
```

In the article about signals, the authors also showed how it could be used in event handling

```
// a pretend GUI button
class Button
{
  typedef boost::signals2::signal<void (int x, int y)> OnClick;
public:
  typedef OnClick::slot_type OnClickSlotType;
  // forward slots through Button interface to its private signal
  boost::signals2::connection doOnClick(const OnClickSlotType & slot);

  // simulate user clicking on GUI button at coordinates 52, 38
  void simulateClick();
private:
  OnClick onClick;
};

boost::signals2::connection Button::doOnClick(const OnClickSlotType & slot)
{
  return onClick.connect(slot);
}

void Button::simulateClick()
{
  onClick(52, 38);
}

void printCoordinates(long x, long y)
{
  std::cout << "(" << x << ", " << y << ")\n";
} [Boost]
```

There are two other aspects that the Boost library discusses that are employed in this proposal. The features provide techniques to scope and block slots.

QT has a library that supports event handling with a base object, QEvent, and have incorporated the Boost Library how to handle actions around certain window features, like menus. [QT1]

In order to create a custom event in QT, the developer needs to create an object based on the parent event. QT has a virtual base class that registers event types defined either by QT or a developer creating custom events. QT uses the Observer Pattern where the QApplication object notifies any events in a collection that are registered in the base class. [QT2]

QT also offers an adaptation of the Boost Signal Library. The usage is limited to a few objects, yet it does have that feature.

jQuery, however, is the inspiration for this proposal because of the ease it is to create a sequence of events without using object inheritance.

```
$('#toggleButton').bind('click',function(){
  $dataBox[$dataBox.is(':visible')?'hide':'show'](100,'swing',function(){ alert('end of animation');});
  return false;
})
```

Also jQuery events support animation in its inclusion of an easing and speed parameters. [jQuery1]

b. What is being proposed

The library is designed to support inheritance, lambda functions and callback functions. The proposed classes are all pure virtual base classes to give developers and library writers the framework to build event handlers around a device interrupt. This library is designed to support both polling and real-time event handling. There are two distinct differences between this library and previous implementations of 2D graphics event handling. They are

1) Events can be overridden by changing the callback method
    1) Example of overriding a “Mouse Event”
        i. mouse_event((*newhandler)(e_args, 10, 100)); or
        ii. mouse_event([](e, 10, 100) { … })); or
        iii. (mouse_event, (*newhandler)(e_args, 10, 100)); or
        iv. (mouse_event, ([](e, 10, 100) { … }));
2) Add special effect animation to the events
    1) Can define the time it takes to show the complete control (speed)
    2) Can define the time it takes to render the control (easing)

The event_args_base, which is the data retrieved from a device, the speed and easing variables are all optional.

2. Scope: Provide a collection that can sequence events based on a device interrupt

a. Current Practice

QT supports a collection of events through its object QSignalMapper as demonstrated below:

```
signalMapper = new QSignalMapper(this);
signalMapper->setMapping(taxFileButton, QString("taxfile.txt"));
signalMapper->setMapping(accountFileButton, QString("accountsfile.txt"));
signalMapper->setMapping(reportFileButton, QString("reportfile.txt"));

connect(taxFileButton, SIGNAL(clicked()), signalMapper, SLOT (map()));
connect(accountFileButton, SIGNAL(clicked()), signalMapper, SLOT (map()));
connect(reportFileButton, SIGNAL(clicked()), signalMapper, SLOT (map())); [QT3]
```

However, each event is bound to a particular control so only one slot is fired even though it is in a collection.

Allegro allows for developers to queues events[Allegro]. Allegro supports creation and destruction of multiple event queues. The events will be executed via a polling technique as shown

```
while(1)
{
      ALLEGRO_EVENT ev;
      ALLEGRO_TIMEOUT timeout;
      al_init_timeout(&timeout, 0.06);

      bool get_event = al_wait_for_event_until(event_queue, &ev, &timeout);

      if(get_event && ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
         break;
      }

      al_clear_to_color(al_map_rgb(0,0,0));
      al_flip_display();
}
```

JQuery is a library that can better explain what the proposal is trying to accomplish. Basically, the developer can bind event handlers to a method call or an event handler. In order to execute each of the members of a

sequence, the developer uses the trigger method to execute a specific event when iterating through the list of events as shown below:

```
return this.each(function () {
   var obj = $(this),
      oldCallback = args[args.length-1],
      newCallback = function () {
         if ($.isFunction(oldCallback)){
            oldCallback.apply(obj);
         }
         obj.trigger('after'+m);
   };

   obj.trigger('before'+m);
   args[args.length-1]=newCallback;

   //alert(args);
   F.apply(obj,args);

});
```

[Rahen]

The only caveat is that the developer has to explicitly trigger each event defined in a sequence.

b. What is being proposed

The library is designed to support multiple container objects filled with sequencing events. The difference between this library and JQuery is that each event will be fired implicitly in the order described by the developer rather than explicitly triggering events.

The containers will also have features similar to functionality described in Boost Signals, where a slot can be blocked, scoped or inserted. However, the new library would expand available operations to allow for events to be removed, sequenced, and operated in reverse.

3. Scope: Provide a Portable Interrupt Framework

a. Current Practice

Currently, there is no specification in the C++ Standard for defining interrupts or interrupt handling.

b. What is being proposed

The library exposes a pure virtual base class that will define the framework for firmware developers to write device drivers and for how the information from the device will be stored in an event_args_base object. In the device object is where the Trigger functionality takes place. The method will take one parameter which is a function pointer. The pointer will point to a method defined either in the container class or the event class itself depending upon if the event is a sequence or stand-alone.

IV. Impact On C++

The proposal is only a framework to standardize event handling and does not add any new functionality, operators or keywords beyond what C++ 11 supports.

V. Technical Specification

For this standard, there will be two **base** classes defined that will provide the framework for developers to utilize data received from low-level device handlers to affect the User Interface(UI) Surface object and a collection object to sequence events around a particular device trigger. These objects are defined as follows:

1) event_base
2) event_args_base
3) event_baseContainer

event_base will define how the UI will be manipulated once an interrupt is fired by the device. event_base can be inherited in order to customize the object in order to fit the software requirements. There is a listing of potential events that can be handled at the end of the document. It also has the feature that the callback function can be overwritten using the **( )** operator. The callback function may either be a specific function that has a signature or a lambda method.

event_args_base is the data that is sent to or received from a device. This will be one of the arguments used in the event_base implementation of the Fire method. Since this is a framework class, it will need to be implemented by the device manufacturer to provide the data context and memory location that is to be used by the events when fired.

event_baseContainer is the container that may contain a sequence of events that will execute due to an external device's input. It is designed to give developers the framework and flexibility to add, insert, remove, or invalidate events within an existing collection around a specific input device firing an interrupt.

The framework will also include one class that will define the interface between the events that will manipulate the UI and the 2D Graphics TS called **io_surface**.

The other **base** class that is defined in the proposal has multiple purposes for this TS. It can be

1) a placeholder for demonstration of how to connect a device input to the UI via the event handler
2) a description of the placeholder and a framework for development of a portable interrupt library
3) the actual framework for standardizing interrupt based device inputs to manipulate UI and other systems

device_base is the framework to create an interrupt service handler, give the device a deviceId and to interface with the timer to query the state of the device. Again this object defines the framework that inherited objects use to meet the device specifications. It also demonstrates how event handlers will interact with a device when an interrupt is fired and then the interrupt triggers the firing of events.

However, the discussion about the framework for standarizing interrupt handling can be put in a separate TS. Yet it is part of this proposal to show the implementation and interconnection between input devices and the effects on the UI when events are triggered by interrupts.

A. event_base

event_base is a pure virtual base class that can be inherited for each type of event that the developer would like to create. Since a control object, event_base is designed so that it cannot be copied or moved.

event_base_args is the data received from a device, which in the event allows for the manipulation of the UI.

The basic structure of the class event_base will be as follows:

```
class event_base
{
        bool inset = true; // used by the container to turn off events for certain sequences

        void (*event)(event_base_args& eventargs = NULL, int speed = 0, int effect = 0));
        std::function<T> fevent_base;

        virtual void Fire(void  (*event)(event_base_args& eventargs = NULL, int speed = 0, int effect = 0)) ;
        virtual void Fire(void) = 0; // maybe the class that inherits event_base will have its own
                                        implementation

        const std::string _name;
        const device_base * _device;

    public:
        event_base(std::string name) : _name(name);
        event_base(std::string name, device_base *device) : _name(name), _device(device);
        virtual ~event_base();

        event_base& operator()(void (*event)( event_base_args& eventargs = NULL, int speed = 0, int effect
                                                                                                = 0) );
        event_base& operator()(std::function<void>(( event_base_args& eventargs = NULL, int speed = 0,
                                                                                    int effect = 0));
        event_base& operator()(const event_base* evt, void (*event)( event_base_args& eventargs = NULL,
                                                                    int speed = 0, int effect = 0) );
        event_base& operator()(const event_base* evt, std::function<void>((event_base_args& eventargs =
                                                                    NULL, int speed = 0, int effect = 0));

        event_base(const event_base& obj) = delete;
        event_base(event_base&& obj) = delete;
        event_base& operator=(const event_base& obj) = delete;
        event_base& operator=(event_base&& obj) = delete;

        void setInSet(bool);

};
```

Therefore events can use callbacks to describe what the event will do when fired. The callbacks can be external functions or lambda statements.

As shown in the Scope statements, there is no need to inherit new events if one already exists with the framework requested. Yet the developer may want to change the orignal functionality to a manner that complies with different specifications. Therefore, the developer only need change the callback function that the

event is pointing. This function will be executed when the event is fired. To demonstrate, let us assume that a developer wants to write an event handler for a mouse click. If the library has a MouseClick object written, the developer only needs to overwrite the pointer to the function that will be executed when the MouseClick event is activated. The developer can also create a lambda function to overwrite as well. However, for events that do not have an object associated with it, then the developer will need to create a new class object.

The class can be instantiated in two ways. If for example the event is an individual event based on an input device, the developer needs to supply the device that the event is attached. Then when the device executes the Trigger function, it will execute the events Fire method. However, if the event is part of a sequence of events, the class is instantiated with only the name parameter supplied. It would be nice to have that parameter be an enum for reliability, yet it would restrict the flexibility of the developer to provide custom events.

The purpose of the isSet variable is that it will tell the container object whether to fire the event or not. It is default equal to true. Once set to false and a sequence is fired, the value will be set to true again.

B. event_args_base

The basic structure of the class event_args_base will be as follows:

```
class event_args_base
{
        virtual void SetData(void) = 0; // still need to determine the exact method

    public:

        event_args_base ();
        virtual ~ event_args_base ();

        virtual event_args_base GetData() = 0;

        event_args_base (const event_args_base & obj) = delete;
        event_args_base (event_args_base && obj) ) = delete;
        event_args_base & operator=(const event_args_base &obj) ) = delete;
        event_args_base && operator=( event_args_base &&obj) ) = delete;

}
```

event_args_base is a class that is designed to hold a memory location where the device can communicate its data. It provides a context for storage of device data to be used by the event.

Therefore an event can use this location to send data to the device or receive data from the device. Since the data is specific to a device, the object's data is not transferrable to other event_base_args derived objects. However, there may be more than one event_base object that uses only one set of data from a device. Examples are that the mouse device firmware would provide certain data, yet there may be events like,

MouseClick,
MouseDown,
MouseUp,
MouseEnter

that would use the same data. Therefore, there is only need for one event_base_args to contain data that the mouse transmits, yet can be used by multiple events that use that data.

This class is the base class for all device data and needs to be configured specifically to the data that the developer wants to pull from or push to a device. So the developer would write derived classes for each device.

C. event_baseContainer

event_baseContainer is used to contain a sequence of events in a **vector** container. The class is shown below as follows:

```
class event_baseContainer
{
                // ************************
                //  operators listed below
                // ************************

                const device_base *_device;

                std::vector<event_base> events;
        public:

                event_baseContainer(const device_base *device) : _device(device) { }

                event_baseContainer() = delete;
                event_baseContainer (const event_args_base & obj) = delete;
                event_baseContainer (event_args_base && obj) ) = delete;
                event_baseContainer & operator=(const event_args_base &obj) ) = delete;
                event_baseContainer && operator=( event_args_base &&obj) ) = delete;

                void (*Execute)();

}
```

The reason why the event_baseContainer contains a contructor with one parameter is that when the interrupt is handled by the **device_base** object, the device will trigger the Execute method. The Execute function then fires the events in the container, which for the sake of this proposal, called events.

event_baseContainer operators as follows:

1. To add a single event

a. control.event (void *event(event_args_base eventargs = NULL, int speed = 0, int effect = 0))
b. control.event(std::function<T>(event_args_base eventargs = NULL, int speed = 0, int effect = 0))

The .event is just a placeholder for a specific event that is being bound to a control, like keydownevent.

2. To add multiple chained events need to use **bind** method

a. control.bind(void* event (event_args_base eventargs = NULL, int speed = 0, int effect = 0))[.bind(void *event(event_args_base eventargs = NULL, int speed = 0, int effect = 0))]**
b. control.bind(std::function<T>(event_args_base eventargs = NULL, int speed = 0, int effect = 0))[.bind(std::function<T>(event_args_base eventargs = NULL, int speed = 0, int effect = 0))] **

3. To add a single event to the end of the list, use the **add** method

a. control.add(void* event (event_args_base eventargs = NULL, int speed = 0, int effect = 0));
b. control.add(std::function<T>(event_args_base eventargs = NULL, int speed = 0, int effect = 0));

4. To insert an event chained event, need to use the **insert** method

Definition of newevent_base
a. newevent_base = (event, fptr*(event_args_base eventargs = NULL, int speed = 0, int effect = 0))
b. newevent_base = (event, std::function<T>(event_args_base eventargs = NULL, int speed = 0, int effect = 0))

Can be added to the chain via the event before the new event to be inserted
a. control.insert(beforeevent_base, newevent_base)[.insert (beforeevent_base, newevent_base)] **
b. control.insert(beforeevent_base, newevent_base) [.insert(beforeevent_base, newevent_base)] **

Or the index of the event before the new event
a. control.insert(index, newevent_base) [.insert(beforeevent_base, newevent_base)] **
b. control.insert(index, newevent_base) [.insert(beforeevent_base, newevent_base)] **

5. To remove an event use **unbind** method

a. control.unbind(event);
b. control.unbind(index);

6. To remove multiple events

a. To remove from an event to the end of the list use the **wipe** method
a. control.wipe(event)
b. control.wipe (index)
b. To remove certain events in a list use **remove**
a. control.remove(event)[.remove(event)] **
b. control.remove(index)[.remove(index)] **

7. To add asynchronous event use **async** method

a. This call would execute each async event in order
async(void *fptr())
async(std::function<T>())

More suited for Sending or Posting Messages. This can be chained to create a number of threads that will be started at approximately one time, yet will only complete after all threads are completed.

b. To chain events, it will be similar to synchronous
async(void* fptr1())[.async (void* fptr2())]**
async(std::function<T>())[.async (std::function<T>())]**

c. For use in Post Messaging can also add events using
random(void *fptr())[.(void *fptr())]**
random(std::function<T>())[.(std::function<T>())]**

The above would execute each call randomly because there is no interconnection of data between events. The use case for this call can only be done via a container that contains BidirectionalIterator.

8. To remove asynchronous event, use the **stop** method

```
stop(event)
```

The next four operators can be applied to both synchronous and asynchronous event handling used for execution.

9. To prevent an event from firing, use **overrule**

```
control.overrule(event);
```

10. To filter events that you want to allow for certain processes to fire without removing elements from container.

```
control | event[x0] | event[x1] … | event[xn];
```

11. To remove the restriction of event filtering, use the **continue** method.

```
control.continue();
```

12. To indicate that the events start at a location within the container, need to use the **startAt** method

```
control.startAt(eventn);
control.startAt(index);
```

This will fire the events starting at the event named in the parameter

The following will work if the container has a backward iterator:

13. To start the event firing sequence from the last item in the container and iterate backwards, need to use the **reverse** method.

```
control.reverse();
```

14. To start the event firing sequence at a certain index within the container and iterate backwards, need to use **reverseAt** method.

```
control.reverseAt(eventn);
control.reverseAt(index);
```

The concept of “control” is a placeholder for the **surface** object that is described in the 2D graphics standard. The control would have the interface io_surface which contains two containers to hold devices and events that the surface may employ to alter itself.

D. io_surface

io_surface will be the class that interfaces between the 2D Graphics and the event handling process. The io_surface class will be constructed as follows:

```
class io_surface
{
		std::vector<event_base> detached_events;
		std::vector<event_baseContainer> sequenced_events;
		std::vector<device_base> devices;

		template <class T> std:multimap<device_base*, T> map;

	public:
/*
	Possible methods that could be used for the surface (2D) object to use to add devices or events:
*/

		void AddDevice(const device_base* device);
		void DeleteDevice(device_base *device);
		void AddEventContainer(const event_baseContainer* baseContainer);
		void DeleteEventContainer(event_baseContainer* baseContainer);
		void AddDetachedEvent(const event_base* event);
		void DeleteDetachedEvent(event_base* event);
}
```

io_surface will be the class that will be exposed by the device_surface class defined in N4073-successor paper. It will have a container of devices that it will expose to the surface object.

The interface will contain containers of devices, sequenced events and detached events. There will also be methods to add and remove as shown above that will be exposed to the surface object so it could decide what will happen when an input device triggers a change.

The purpose of the multimap container is to correlate a device with either an event_base object or an event_baseContainer object. Since a device may trigger multiple detached events, the multimap allows a better expression of this correlation.

Because of the efforts of Michael Mclaughlin and I to construct the basic frameworks for our separate proposals, the discussion on the full implementation is still progressing. Therefore, this item will be added to the Future Work section to complete the interface.

E. device_base

The basic structure of the class device_base will be as follows:

```
class device_base
{
        const int[32, 64, 128]* int_handler;
        const int[32, 64, 128]* timer_handler;

        /// triggerFlag is to give the state of the device. If the device is busy or suspended, the timer
        /// interrupt is enabled to monitor the trigger until the ready state is set
        /// once set, the timer interrupt will call the Trigger method, which in turns fires the events
        /// listed in the container, event_baseContainer.
        /// Otherwise, the interrupt handler directly will call the Trigger method.

        volatile int triggerFlag;   // will be updated by Interrupt Service Routine (ISR) for a particular
        // device like mouse, keyboard
        // 00001 means                  in-use
        // 00010 means                  ready
        // 00100                  "     busy
        // 01000                  "     suspend
        // 10000                  "     device error

        const int[32, 64, 128] deviceId;

        virtual void Trigger(void* fptr()) = 0; // trigger would execute a callback in the container of events

        event_args_base& _args;

    public:

        device_base();
        virtual ~device_base();

        event_args_base getArgs(void);

        device_base (const device_base & obj) = delete;
        device_base (device_base && obj) ) = delete;
        device_base & operator=(const device_base& obj) ) = delete;
        device_base & operator=(device_base&& obj) ) = delete;

}
```

The device_base object is to map directly to the ISR for a particular device and the Timer interrupt ISR. The Timer interrupt is designed to handle cases where the interrupt has been set, yet the UI was busy. Then when the tigger value is set to "ready", it will execute the function pointer defined as the parameter in the Trigger method.

When the interrupt is set, the Trigger will either call the event_base's function, Fire or the event_baseContainer's function, Execute.

F. Examples of events to be handled by event_base

- Timer

- Keyboard
  - OnKeyUp
  - OnKeyDown
  - OnKeyPress

- Mouse
  - OnClick
  - OnMouseOver
  - OnMouseDown
  - OnMouseUp
  - OnMouseMove
  - OnContent

- Touch
  - OnTouch
  - OnSwipe
    - SwipeUp
    - SwipeDown
    - SwipeLeft
    - SwipeRight
  - OnTouchPosition
    - TouchStart
    - TouchEnd
    - TouchCancel
    - TouchMove
    - TouchHold

- Show
  - Show
    - BeforeShow
    - AfterShow
  - Hide
    - BeforeHide
    - AfterHide
  - OnDraw
    - Before
    - After

VI. Future Work to Consider

1. Completion of Interface between the 2D Surface object and the event handling objects
2. Messaging
    a. Synchronous
    b. Asynchronous
3. Multithreading

    Need to consider synchronization of events for messaging

4. Security
5. Support more input devices
6. Parallel Hardware handling of events
7. Simpler coding of the event sequencing defined in the event_baseContainer class.

8. Acknowledgements

Would like to aknowledge Michael McLaughlin for inspiring me to work on the standard. Without his efforts on developing the standard for 2D Graphics, this would not have been possible. Would also like to thank Jason Zink for his input about other work that is currently being done which allowed me to show that this standard emphasizes current work done already. Of course, without Herb Sutter's input, none of this work would have taken place. Would like to thank my son, Christopher, because he is the biggest motivation in my life.

9. References

[Boost] http://www.boost.org/doc/libs/1_59_0/doc/html/signals2/tutorial.html

[QT1] http://doc.qt.io/qt-4.8/eventsandfilters.html

[QT2] http://doc.qt.io/qt-4.8/qevent.html

[QT3] http://doc.qt.io/qt-4.8/signalsandslots.html

[Allegro] https://wiki.allegro.cc/index.php?title=Allegro_5_Tutorial/Events

[jQuery1] https://api.jquery.com/category/events/

[jQuery2] https://api.jquery.com/bind/

[Rahen] http://jsfiddle.net/rahen/eUaAw/5/

** [] means optional.