2016

# P0434 Portable Interrupt Library

SG13 – HMI

Document Number: P0434

Date: 10/18/2016

Reply-To: brett.searles@attobotics.net

AUTHOR: BRETT SEARLES

## Table of Contents

## Introduction

As with event handling, this proposal extends the P0269R2 proposal by discussing how input devices interact with interrupt handlers and the data block(s) that can be consumed by events. The effort is to propose a library that defines a framework for handling interrupts for device driver and firmware developers. Because interrupts do can be independent from events, therefore this proposal is written separately. However, there is an interconnection between the events and interrupts and this proposal takes the two concepts and demonstrates how they work together when an interrupt is fired.

## Motivation and Scope

1) To construct a framework for present and future development of IO devices, its data and its usage of that data via a Portable Interrupt Framework

### 1. Scope: Provide a Portable Interrupt Framework

#### a. Current Practice

Currently, there is no specification in the C++ Standard for defining interrupts or interrupt handling. There was an article written how to write classes in C++ for interrupt handling, however, the functionality was based on a static implementation and focusing all interrupt handling to be within one class. [Dobbs]

#### 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_base object, defined is the **Trigger** function. The **Trigger** function is a pure virtual function that allows firmware developers the flexibility to implement their own code due to system constraints like OS for which the firmware is applied. The method will take one parameter which is a function pointer. The pointer will point to a function defined either in the event_basecontainer class or the event class itself depending upon if the event is a sequence or stand-alone. The base class allows for implementation of both a single static instance or by instantiating several objects. The proposal allows each interrupt to be defined independently and allows for dynamic definition of functionality.

# Design Decisions

## device_base

The object will be the abstraction of the actual hardware's implementation of device management. It will contain the location of the interrupt, a device_id, a timer interrupt, events attached to the device when interrupt is set, location of memory to access or store information and a pure abstract function that will allow firmware developers to override with code that will execute when an interrupt is set.

### int_handler

The concept of int_handler was designed around the fact that hardware contains an Interrupt Descriptor Table. It is an id is to point where the object maps to in that table.

### timer_handler

The timer_handler is similar to the int_handler, yet it points to Interrupt 0, the timer interrupt. It is used for cases for when the existing interrupt is set and another has fired. This allows for the second interrupt to be executed after the initial interrupt has been cleared. This is very important in Real-Time Operating Systems. If in a multi-tasking Operating System (OS), the Trigger function may be written to disable interrupts until the interrupt is finished execution.

### trigger_flag

The use of the trigger_flag that is a volatile variable with type int is that this can be overwritten by the hardware, OS, or external IO with multiple status levels so that if other interrupts are triggered, the new interrupt can handled the way the firmware developer so desire. It will describe five(5) states of the interrupt when it occurs. The fifth state is used for asychrounous file reading, etc.

### Collections

#### template<class U, class T = std::vector<*event_base>> T events<U>

This collection contains the events or sequence of events that are registered to the interrupt. When an event subscribes, that event will register the event or sequence with the interrupt handler. With subscription, the event or sequence will be placed in the defined container. When the device_base is triggered, it will loop through the subscribed events and fire them off.

#### template<class V = std::vector<*std::thread>>  V threads

This collection is for the threads that will be created when the events are fired. The threads will reset based on the condition of wait_all or wait_any or possibly some barrier(s). The purpose of the collection of threads was to execute the subscribed events more rapidly. However, the threads may have different criteria for execution. Some threads will operate under conditional requirements, while some work in a barrier related situation where several threads need to complete before other events can be executed. The barrier concept would be more in demand for the event_base_container object described in P0249R2 where events are fired in a sequence.

## Technical Specification

The base class that is defined in the proposal can be defined as the framework to write firmware for device drivers.

**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. This object is the base framework for inherited objects to 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.

## 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 trigger_flag;  // will be updated by Interrupt Service Routine (ISR) for a
                                    // particular
        // device like mouse, keyboard
        // 000001 means                        in-use
        // 000010 means                        ready
        // 000100                        “     busy
        // 001000                        “     suspend
        // 010000                        “     device error
        // 100000                        streaming data

        const int[32, 64, 128] device_id;


        template<class U, class T = std::vector<*event_base>> T events<U>;
        template<class V = std::vector<*std::thread>>  V threads;

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

        event_args_base& _args; // correlation with P0249r2

    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 Service Routine (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 executes the events the event_base_container’s Execute function or the Fire function in the event_base object (See proposal P0249r2). For each event that has subscribed to the ISR, a thread will be created. Depending upon the condition set for each event, the trigger value will be reset based on that condition statement. The function pointer that is executed is defined in the copy constructor. The default copy constructor is removed by the compiler because only one object should be created for each device. To have more than one object defined for one device could cause deleterious results. However, there is a need to have the copy constructor for events so that the device will know what object to call when an interrupt is set.

The event_args_base, which is defined in Proposal P0249r1, will store information that the device will either read or write to that object’s implementation. This information will then be used by an event handler.

## Future Work

1) Stronger support for asynchronous processes
2) Better development of the threading models to be incorporated for firing events.

## Acknowledgements

Would like to acknowledge Herb Sutter for his support for this library development. Also need to acknowledge Michal McLaughlin for his input as to separate the two concepts. One for event handling and the second got the Portable Interrupt Framework. Of course, the biggest motivation in my life, my son Christopher.

## References

[Dobbs] http://www.drdobbs.com/implementing-interrupt-service-routines/184401485

[Saks] http://www.embedded.com/electronics-blogs/programming-pointers/4025714/Modeling-interrupt-vectors

[OSDev] http://wiki.osdev.org/Interrupt_Descriptor_Table