Miscellaneous draft documentation

Andrew Hunter

September 6, 1999

1 impulse_event

The impulse event system is a generalised POSIX multithreaded IO system, which basically means select() with wings. It is a component of libImpUtil that must be ported between thread libraries, as the select() call must be interruptable somehow[1].

1.1 Background

The most important part of any user interface, be it graphical or not, is the means by which input is achieved. Almost all GUI systems provide feedback in the form of events - structures that specify an occurance, and suggest the action that the software should take. There are also two forms of event that are not always dealt with by a GUI library - the timeout and the socket event, but both these events can be vital to the smooth operation of a piece of software. impulse_event provides a framework for generic event types, such as GUI events, and also provides an implementation of socket and timeout events. It does all this in a manner which ensures a multi-threaded application will operate correctly.

1.2 Using impulse_event

The most important functions is are impulse_event_get_next_predicate, impulse_event_listen and impulse_event_alarm. These functions get the next event matching a pattern, tell impulse_event to listen to a filedescriptor and set a timeout respectively. Unless otherwise specified, all functions are completely thread-safe.

1.2.1 int32 impulse_event_listen(int fd)

This tells impulse_event to listen on the given file descriptor. An event will be returned when it is ready for reading. Once the event has occured, the handler will be deregistered - this avoids events being generated unnecessarily. The

[1] Under POSIX threads, by a signal (interrupting the system call), under GNU pth, also a signal, but this time generating a pth event.

return value is an ID, unique to this registration, which can be retrieved from an impulse_event structure when the event occurs.

1.2.2 int32 impulse_event_alarm(struct timeval *when)

This sets an alarm at an absolute time of day, as returned by gettimeofday(3) with a NULL timezone. As for impulse_event_listen, the event is one-shot, deregistering when it is complete. The return value is a unique ID, which can be used to check which alarm has gone off.

1.2.3 void impulse_event_get_next_predicate(imnpulse_event *result, impulse_event_predicate_func pred, void *data, int priority)

This function returns the next event matching the given predicate, storing it in result. Priority indicates the priority of this routine if predicates are matched in several threads - the higher the value, the lower the priority, 0 being the highest priority. The predicate function has two arguments - a pointer to an impulse_event structure, and the data argument passed to impulse_event_get_next_predicate. The function should return non-zero for a match, and zero for a mismatch.

1.3 The impulse_event structure

The impulse_event structure is designed to represent a generic event. There are currently two types of event with their structure predefined and 17 capable of passing any data. For those types, allocation and deallocation of the data is the responsibility of the routine that sends and finally receives the event (through a handler or by impulse_event_get_next_predicate). The event types are:

- IMPULSE_EVENT_SYSTEM
- IMPULSE_EVENT_FILEDESCRIPTOR
- IMPULSE_EVENT_ALARM
- IMPULSE_EVENT_USER+0 through to
- IMPULSE_EVENT_USER+15

These are stored in the event_type member of the structure. The other member of the structure is the event_data union. This contains three structures, for each possible event type (system_event, fd_event and alarm_event). These structures are as follows:

1.3.1 system_event

This is used to store the data for SYSTEM events and USER events. These event types are general purpose, although USER events are intended for user applications, and SYSTEM events for a utility library built around impulse_event, such as ROWS. Other general purpose events may be declared in the future, and will also use this structure. It has four members: system_data (a void pointer), event_type (int32), object (int32) and id (int32). These are all for use as the implementation sees fit, although note that the object ID is used to check if there is a handler available for this event.

1.3.2 fd_event

This represents an event occuring on a filedescriptor. The structure has three values: fd_event_type (the type of filedescriptor event; IMPULSE_EVENT_FD_READ, IMPULSE_EVENT_FD_WRITE or IMPULSE_EVENT_FD_EXCEPTION), fd (the filedescriptor) and id (the unique ID).

1.3.3 alarm_event

This represents a timeout event. It has three values - set_for (when the alarm was set for, a timeval), went_off (when the alarm actually went off, a timeval) and id (the unique ID of this timeout)

1.4 Registering handlers

Sometimes, you may find it necessary to register a handler - a function that is called whenever an event of a given type occurs. These handlers are called by impulse_event_handle() whenever they occur - most event driven programs will want to use impulse_event_handle() in their main loop and register relevant handlers with impulse_event.

1.4.1 void impulse_event_register(int32 event_type, impulse_event_func func, int32 object, void **widget, void *data)

This function registers a handler for the given event type. object is a unique object identifier: in the case of timeouts this is the ID returned from impulse_event_alarm or impulse_event_alarm_offset; in the case of impulse_event_listen, this is the filedescriptor that is being listened on. For generic events, the object is defined by the user. widget and data are passed to the event function as arguments.

1.4.2 void impulse_event_deregister(int32 event_type, int32 object)

This function deregisters the handler for the given event type and object.

1.4.3 void impulse_event_handle(void)

This function calls impulse_event_get_next_predicate to get the next event that a handler is registered for, then calls the handler. It does not deregister the handler even though the two built-in event types are one-shot.

1.5 Generating events

An implementation might want to generate its own events, for example to add events from a window system into the queue. This is done via impulse_event_trigger, which will add an event in to the queue, and wake up any thread that may be waiting for it. This call takes one argument, a pointer to an impulse_event structure. The contents of the structure are copied away, so it may be deallocated after the call.

1.6 Convienience functions

There are a selection of convienience functions to make application writing simpler, doing extra work on behalf of the programmer. These functions include impulse_event_get_next, impulse_event_pending, impulse_event_check_next, impulse_event_loop and impulse_event_fd_listener.

1.6.1 impulse_event_get_next

This function is really designed for small applications only, as it has a major flaw in multithreaded situations. The function takes a pointer to an impulse_event structure, and returns the next available event in it. If other threads are listening for events, this routine has a low priority. The flaw is that the routine is never selective. If you have one thread listening for specific events, and one listening for general ones, most of the time the specific thread will receive its events and the general one will receive all others. However, if a specific event arrives while the specific thread is still dealing with a previous event, the general thread will receive it. This race condition can cause major problems and a lot of head-scratching...

1.6.2 impulse_event_pending

This routine takes a predicate function and its data as an argument, and returns non-zero if a matching event is found in the queue.

1.6.3 impulse_event_check_next

This routine takes a pointer to an impulse_event, a predicate function and its data as arguments, returns non-zero if an event is found, and copies it into the event structure.

1.6.4 impulse_event_loop

This routine calls impulse_event_handle until impulse_quit is called.

1.6.5 impulse_event_fd_listener

This routine provides a work-around for the one-shot nature of filedescriptor events. It takes a filedescriptor, an event function and its data as arguments. Whenever the filedescriptor becomes available to read, the event function is called as for a normal registered function. However, the event is reregistered as soon as the handler has completed, so that the routine is called whenever the file descriptor has data waiting. This routine solves the race conditions associated with leaving a file descriptor listener registered while its handler executes.

2 ROWS_X

ROWS itself consists of two parts - a platform-independent widget library and some platform-specific drivers. Those drivers that talk to X are in the ROWS_X* set of source files. I will provide a description of the functions implemented there to provide a aid to future porters - there is no reason to keep the current file structure with future ports apart from to maintain consistency.

2.1 ROWS_X.c

This library provides general purpose X functionality - the X drawable, error handling and initialisation code, basically all code that does not fit in any of the other libraries. The most important function provided here is ROWS_internal_initialise, which starts up the X window system.

2.2 ROWS_Xdrawable.c

This is the actual X drawable widget. The source file does not contain the X drawable code itself, which is in ROWS_X.c. All that is provided is a simple widget and initialisation of the drawable function pointers.

2.3 ROWS_Xevent.c

This file receives X events and passes them to impulse_event using impulse_event_trigger: using the impulse event system ensures correct operation in a multi-threaded environment, and makes implementing libraries such as ROWS_Xwindow, ROWS_Xcomm and ROWS_Xchild simpler. Also provided here is ROWS_main(), the function called to enter the event loop.

2.4 ROWS_Xwindow & ROWS_Xchild

ROWS_Xwindow provides top-level windows, those managed by the window manager. ROWS_Xchild provides support for containing child windows in lightweight windows, and provides various relevant events, ensuring that an 'actualised' widget receives all events. An actualised widget is one with implementation-defined special handling. ROWS specifies that an non-actualised widget need only recieve click and expose events, but an actuallised widget must receive all events, including the rollover events, keypress events when input focus is appropriately placed, etc. A non-actualised widget should have a smaller memory footprint and resource overhead than an actualised one.

2.5 ROWS_Xcomm.c

Although it is intended to support a range of communication protocols, the only guaranteed communication medium is that provided by the graphics system. ROWS_Xcomm.c provides facilities for finding other applications, establishing communications with them and sending messages.