# A Brief Overview of the Re-Implementation of the Android Push Event Stream Model

Dustin McAfee and Brad Vander Zanden
E-mail: dmcafee2@utk.edu, bvanderz@utk.edu

## 1 Introduction

The work described, here, is an ongoing attempt to re-implement an existing model, the Android Event Push Stream Model (ESM), put forth in a number of papers by Drs. Marz, Vander Zanden, and Gao at the University of Tennessee, Knoxville, [1], [2], [3]. This re-implementation differs from Marz's original implementation in that it attempts to implement only the event push model in the Android kernel while Marz's implementation implemented the event push model in the Android kernel, moved the display system into the kernel, and implemented a GUI scheduler to better manage different GUI tasks. At this point we are only re-implementing the event push model so that we can examine its ability to reduce the power consumption of apps using sensor events.

The goal of this re-implementation is to minimize the amount of change that must be made to the Android kernel and Android middleware, so that it is more portable than the original implementation. It seeks to implement the functionality of the kernel-level functions "esm_wait", "esm_register", "esm_interpret", and "esm_dispatch" [1]. This paper assumes that the reader has read the original Marz paper in ACM Transactions on Embedded Computing Systems and is familiar with the four algorithms listed above.

The rest of this paper is organized as follows. Section 2 compares the basic functionality of the Push Event Stream Model (ESM) to the current poll model that is used in modern devices, and introduces the data structures used for the ESM. Section 3 describes our first attempt to re-implement this model, which involves using architecture-dependent assembly code to load the user-stack with a user-defined input handler, and context-switch from the bottom half of an interrupt handler to the user's input handler. Section 4 describes our second and simpler method of copying the events to user-space, along with a user-defined input handler identifier, so that the kernel is not forced to use architecture-dependent code to execute the user's input handler. Section 5 describes how the second implementation replaces the epoll API inside of the Android middleware with the ESM re-implementation, and section 6 describes the results and current status of this re-implementation.

• *D. McAfee and B. Vander Zanden are with the Department of Electrical Engineering and Computer Science, 1520 Middle Drive, Min H. Kao Building, University of Tennessee, Knoxville, TN 37996.*

## 2 Methods

The Android input stack is comprised of multiple layers for polling input and demultiplexing it to the user processes (applications). The polling interface to the Linux kernel that the Android operating system uses is epoll (see "frameworks/native/services/inputflinger/EventHub.cpp" in the Android source code, and "fs/eventpoll.c" in the Linux kernel source code). The Linux kernel running inside of Android uses the evdev input event interface (see "drivers/input/evdev.c" in the Linux kernel source code). After an input event fires an interrupt in the Linux kernel, evdev is responsible for directing the input event to the correct input device file (located in "/dev/input/").

The Android operating system is using epoll to poll all input devices that are of any interest to the system, which include input devices that have been registered by a user application or system application. Only one epoll instance is used to poll the input devices, so our goal is to be able to replace the one instance of epoll and its call to "epoll_wait" and calls to "epoll_register" to a new ESM model API, with calls to "esm_wait" and "esm_register". A big obstacle to note is that the epoll API uses file descriptors to not only identify the input devices, but also to identify itself. In the ESM, the input event is being streamed from the bottom half of an interrupt handler (deferred work) to the user, possibly even before it is being written to the input device file.

Marz describes in [1] how "esm_wait", "esm_register", "esm_interpret", and "esm_dispatch" are implemented, along with a mapping from the input events to the user-defined input handlers with which they are registered. The mapping differs between Sections 3 and 4, and so the mapping from Section 3 will be discussed here, and the differences will be explained in the later section.

In our implementation, a 'struct application_l' (application list item) structure is created for each input event type and code, and contains the input handler virtual address that is registered. The structures that share the same input type and code with different input handlers are shared in the same linked list, so as to organize the lists by the type/code of the input event. When "esm_register" is called, a new 'application_l' is allocated and added to the appropriate linked list, or if "esm_register" is called with the argument to de-register an input handler, then the appropriate 'application_l' is de-allocated and removed from the linked list.

The idea is that the user application can register an input event to a given input handler address, so that after it calls ”esm_wait”, it will receive these events without having to poll a device file. In order to record registered events that happen before the user calls ”esm_wait”, we added another linked list to the Process Control Block structure, ’struct task_struct’ (see ”linux/sched.h”). This linked list contains a list of input events that are registered to the user application and pending to the ESM model. This list should act as a queue in order to push the events to the user in order. Unfortunately, a FIFO queue implementation was never actually developed, and this structure remains a linked-list. Future development should replace this with a KFIFO implementation from the Android kernel. For the rest of this discussion, this list will be referred to as a queue, and the implications of using a linked-list rather than a FIFO queue is discussed further in Section 6.

One more major addition our implementation makes to the Android kernel is a new task state, ’TASK_EV_WAIT’, as described in [1]. This is to ensure that the system knows which applications are waiting for events, so that any call to ’wake_up_state’ must be given the argument ’TASK_EV_WAIT’ in order to wake up the process, notifying it that there are input events in its queue. In Marz’s implementation of the GUI Scheduler, dynamic clock ticks are set, such that, when the system is doing nothing but waiting on input events (no wake-locks in use), the clock timer will not interrupt and wake up any processes that otherwise should be asleep waiting for events [3]. This is not yet a part of the re-implementation.

After an interrupt occurs in the Android kernel for an input event, it is passed to evdev. A call to ”esm_interpret” is used to intercept this event just before it is written to the character device. In this re-implementation, ”esm_interpret” iterates through each ’application_l’ type to find the corresponding linked list that contains the registered input event and event handlers, and creates deferred work (work queue) that calls ”esm_dispatch” for each input handler interested in the input event.

A call to ”esm_dispatch” will enqueue the input event if the task is not in ’TASK_EV_WAIT’; otherwise, it is responsible for pushing the event into userspace. The original Marz implementation uses a combination of the Kernel Display Server and the Android Push Event Stream Model to not only push the event into userspace, but also context switch from the kernel to the user process’s registered input handler, while allowing the process to return back into the call to ”esm_wait” after the event is handled. Unfortunately, this code is dependent on the Nvidia TK-1 architecture and is not portable. The implementation we are currently describing also attempts to make use of our hardware architecture, which is the x86/x86_64. In Section 4 we describe a simpler model that deviates from this architectural decision in order to simplify the way that the input handler is called and to make it more architecture-independent.

## 3 ESM Load User Stack

The first attempt in re-implementing ESM in the Android kernel involved registering single events to the kernel and loading the user process’s stack with its own input handler and the type of the triggered input event. For each input event of interest (left mouse click, right mouse click, specific keyboard buttons, etc.), the user process calls ”esm_register” with the virtual address of the input handler, which belongs to the user process, that should handle the event. esm_register creates an item in a linked list that contains the virtual address of the input handler, the type of input event, and a pointer to the Process Control Block that is interested in the input event (and owns the input handler).

After registering for events, the user process calls ”esm_wait”, which puts the process into the new task state ’TASK_EV_WAIT’ and schedules it. When an event of interest occurs via interrupt, ”esm_interpret” is called, which in turn calls ”esm_dispatch” for each event. ”esm_dispatch” is responsible for dispatching the input event to the input handler (taking the process out of ’TASK_EV_WAIT’), and in this case, calls a special architecture dependent context-switch function that loads the user process’s stack with its own input handler and input event and switches the context to that user process. This architecture dependent code was only written for x86/x86_64, and never properly worked; there are protections set in place by either the hardware or the kernel from switching out of kernel mode to user mode that halt the kernel when attempting this method and we were not able to overcome this problem.

However, the main reason this approach was abandoned is because it required re-writing assembly code for x86 and arm devices (32-bit and 64-bit) and we realized that our re-implementation would be no more portable than the original Marz implementation. While it is appropriate for commercial hardware developers with large groups to create proprietary implementations, we cannot afford to keep writing proprietary implementations every few years and need something more portable.

Another reason for trying a different approach is that the epoll API does not call the input handler from kernel space. Instead, Android relies on its middleware to read input events from the character devices after epoll returns and figure out what to do with the input events. This is a big reason why Marz moved much of the Android middleware to the kernel when developing the Kernel Display Server [2]; it is because the Android middleware has to re-figure out what to do with input events after receiving them from the kernel, and because of this, much of the kernel input driver code is re-written inside of the Android HAL (Hardware Abstraction Layer).

## 4 ESM Copy to User

Since we decided not to re-implement Dr. Marz’s KDS, it does not make much sense to call the input event handler from kernel space–it is easier to let the Android middleware make the call to the appropriate input event handler. This makes our goal much simpler: Remove the call to epoll from the Android inputflinger and replace it with a similar API that does not poll. This means registering groups of input events based on input device file descriptors, and somehow delivering the interested events to the user after a call to ”esm_wait”. This goal of removing the polling loop in inputflinger eliminates the point of waiting on character device files, and so this means there will be no writing

to, or reading from character devices inside of the ESM. However eliminating the character input devices causes a problem since epoll uses open input device file descriptors to register input events to user processes. To solve this problem, "esm_register" takes as a parameter a user buffer of a 'struct input_id', which is a unique identifier of an input device. This input ID structure consolidates the 'struct application_l' type (Section 2) to a single linked-list of input IDs and Process Control Block pointers. The Android kernel knows upon an interrupt which device generated the input event and passes this information to "esm_interpret" (from evdev). In this case, "esm_dispatch" just wakes up the process (if it is in 'TASK_EV_WAIT', scheduled in "esm_wait") and "esm_wait" copies the events to userspace (to a 'void __user*' supplied buffer argument) before returning.

As a test case, we developed a single threaded user application that uses the ESM API to wait on, collect, and deliver input events to corresponding handlers. The ESM successfully delivers to this application multiple events without having read from any open file descriptors (however, it does have to make calls to ioctl to read the input id's). Successful tests were performed on this single threaded program for collecting events before and after the call to "esm_wait".

The single threaded case was a success, but the Android inputflinger service is multi-threaded, with one thread that registers/de-registers input devices, and another dedicated thread that runs "EventHub::GetEvents", which is the function that calls "epoll_wait".

## 5 Implementation in the Android Middleware

When we installed the ESM interface into EventHub, the events get registered to a thread separate from the thread that calls "esm_wait" (separate threads do not share Process Control Blocks in the Android kernel); the thread that waits for events in "EventHub::GetEvents" goes to sleep and never has any events registered to its Process Control Block. Therefore, it never wakes up when any events happen.

To solve this issue, we created a new system call named "esm_ctl". This function is given two process IDs, and is called right before "esm_wait". It transfers the thread's registered input devices and queued input events to another thread, allowing the callee of "esm_wait" to have the correct registered and queued events.

However, to completely replace "EventHub"'s dependency on the epoll interface, another argument must be supplied to "esm_register" and "esm_wait". The epoll interface uses a 'struct epoll_event', which is primarily used by epoll to mask events from file-descriptors, along with error reporting, and for the user to store device specific data. "EventHub" sets the 'events' member to "EPOLLIN", which sets the input event mask to take all input events from the input device. When there is an error such as a hang-up event in epoll, then epoll is liable to change this value to indicate the error; however, we have not yet implemented this error handling in our push model.

One other member of the 'epoll_event' structure is used by the Android middleware, which is the 'data' field (of type 'epoll_data_t'). This field holds a value that is used to index an array of input device structures defined by the middleware. So, it is important to add an association of the supplied 'struct epoll_event' to the supplied 'struct input_id' (Section 4) inside of the ESM API, specifically, inside of 'struct application_l' (See Section 2).

One minor discrepancy between the current ESM API and the epoll API used by the Android middleware is that the ESM API uses 'struct input_value' structures, and the epoll API uses 'struct input_event' structures. The difference is that the 'struct input_event' structures contain a timestamp member. This difference can be easily remedied by replacing all of the 'struct input_value' instances in the ESM API with 'struct input_event' instances, and including the timestamp in the 'struct input_event' when invoking "esm_interpret" from the evdev interface. Because of this, the Android middleware (specifically inputflinger) currently emits warnings about the wrong clock being used for the input devices when input arrives.

## 6 Results

Using this new ESM API, we replaced all calls to the epoll interface with calls to the ESM interface in an AOSP 7.1 build, and tested it in an x86_64 emulator. There are no inputflinger process crashes that are seen in the logs, and the input is delivered in a timely manner. However, the input is a little jittery, with the USB keyboard having the most issues as single key presses are periodically reported as two or three key presses. This multiple reporting could be because the event queue in the Process Control Blocks is not implemented as a FIFO queue, causing key-up/key-down events to come in out of order. The underlying structure is a simple linked-list, and a KFIFO structure would be much more likely to work (See "include/linux/kfifo.h").

Another issue we have not yet addressed in our modification of the Android middleware is that during streaming of input events in the Android kernel, a special kind of event with type, code, and value equal to 0 is generated to mark the end of an event. This event is called an "EV_SYN" event, which is used as a marker to separate events in time or space (e.g. multi-touch protocol). These events are streamed to the Android middleware, which expects the Android kernel to have already handled these events. This issue could likely be addressed by moving the invocation of "esm_interpret" to a later function in the Android input stack, where the synchronization events have already been handled, but may not be as trivial to solve as the previous issues that have been discussed. A different solution would be to handle the "EV_SYN" events in the ESM API, which is still not trivial. More experimentation on the handling of these "EV_SYN" events is required to finish the implementation of the ESM model in the Android middleware.

One final issue with the new ESM implementation is that the source code for the ESM API contains artifacts from the first architecture-dependent re-implementation of the ESM model that were never re-written or removed. A fine example of such an artifact is that it is likely that the input event queue which is stored in the Process Control Block could be removed from the Process Control Block and put into a global event queue, since the Android middleware is only using a single instance of the ESM. This would make the "esm_ctl" call obsolete, as well. However, keeping the

input event queues stored inside of the PCB allows for multiple instances of the ESM for different processes (or threads) in the system.

## ACKNOWLEDGMENTS

The research reported in this paper is being supported by NSF grant CNS-1617198.

## REFERENCES

[1] S. Marz and B. V. Zanden, "Reducing power consumption and latency in mobile devices using an event stream model," *ACM Trans. Embed. Comput. Syst.*, vol. 16, pp. 11:1–11:24, Oct. 2016.
[2] S. Marz, B. T. Vander Zanden, and W. Gao, "Reducing event latency and power consumption in mobile devices by using a kernel-level display server," *IEEE Transactions on Mobile Computing*, pp. 1–1, 2018.
[3] S. Marz, "Reducing power consumption and latency in mobile devices by using a gui scheduler," 2018.