Universiteit Leiden

Opleiding Informatica

Implementing an interface for virtual input devices into the MGSim simulator

Name:	Koen Putman
Date:	14/02/2017
Supervisor:	Raphael Poss (UvA)
2nd reader:	Todor Stefanov (LIACS)

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

MGSim is a simulator used for research and education. One of the features it lacks is a convenient method of accessing real-time external input in the simulated system. We want to design and implement an interface that would give the simulation access to external input from joysticks, gamepads, mice, and touch devices. After researching existing frameworks, we show our interface design and describe its functionality. By modifying existing parts of MGSim we connect it to external input devices. We use this connection to create a component that implements our interface design. Finally we measure its performance and provide a usage example.

Contents

Chapter 1

Introduction

MGSim is a simulator and full system emulator designed for multi-core processing [1]. It is generally used for research and education. MGSim allows students to get a better idea of the components that make up a computer and how they interact in a relatively simple environment. One of the current shortcomings of the simulations is a lack of interactivity when running. MGSim already supports a virtual graphical output interface (gfx), however an external input source alongside this graphics device would allow for creation of interactive software, like games.

1.1 MGSim

MGSim can be used to simulate a wide variety of system configurations by just changing a configuration file. Its main benefit is to reduce the cost of prototyping ideas for chips and benchmarking the performance of new designs. Next to being configurable it is also easily extensible to support new components and instruction set architectures. To aid research MGSim keeps track of performance metrics and allows inspection of components while it is running.

For education MGSim offers a relatively simple infrastructure that is possible for a student to understand. It aims to be easily deployable and offers them a view into a computer's inner workings. While a student might find MGSim interesting from a technical standpoint, it could be modified to provide a more engaging learning experience. As alluded to above, there are very few ways to interact directly with a running simulation and none of those are easily accessible for a student. As a result most of their simulations will just perform a task and exit, with any variation usually requiring them to recompile the binary they run.

Providing a method of direct interaction that is easily accessible, like a simple interface for various external input devices, could increase student engagement. It would also teach them how to work with memory mapped I/O. By combining an interface that connects to a joystick and the graphics device they can create interactive experiences, like games, while learning how computers function. Outside of educational use there is some potential merit to having an unpredictable source of I/O data to test how various system configurations with I/O bridges handle such a load.

MGSim already uses the keyboard for various control tasks, which leaves joysticks, gamepads, mice, and touch devices for potential input source candidates. The Simple DirectMedia Layer library [2], which MGSim already uses for the graphics device, can provide access to these devices across platforms.

1.2 Thesis overview

We set out to implement an interface into MGSim that allows access to external input sources like joysticks, gamepads, mice, and touch. In chapter 2 we list our requirements and research existing frameworks to compare their capabilities and gather inspiration. We describe and discuss our our design in chapter 3. Chapter 4 describes our modifications to MGSim. We start with modifying existing MGSim components for a prototype and move on to the modifications required to gather data from external input devices. We modify the prototype to support the new system and create the component that implements our interface design. In chapter 5 we measure the performance of our work and describe an example program that uses our interface. The conclusion in chapter 6 lists what works and what could be improved in the future.

Chapter 2

Analysis of requirements and prior work

The first stop on our journey from idea to implementation is specifying requirements. We move on to studying several existing frameworks and their internals if available. We discuss their capabilities, how they are used, and compare them to other frameworks. In the last section we summarise the results of our studies.

2.1 Design requirements

There are some initial requirements and expectations for the functionality of the interface and implementation.

Our implementation will be using SDL 2.0 to handle input devices because it provides platform-agnostic access to input devices and is already used in MGSim to render the graphics framebuffer. We should aim to provide access to all of the data that SDL provides, which includes information on the connected external device, access to the current state of that device, and an event system that tracks changes in device state.

While it does not have to emulate existing hardware, the interface should strive to resemble one that might occur in hardware. An interface design by itself is not going to be enough, so a component that implements this interface should be provided as well.

We will not be writing a driver and our interface serves an educational purpose, so it should be simple to utilise. To this end, there should also be clear documentation and example code to study.

The main purpose of this interface is adding interactivity to the simulation, so we would like to keep our input latency low.

Additional functionality like recording the actions of our component and being able to replay those on another execution can aid in testing and debugging. As such we would like this replay to be as deterministic as possible for our component regardless of what the rest of the system is like. This way a replay could be used in tests with different system configurations to compare their performance.

2.2 Examining existing frameworks

2.2.1 Simple DirectMedia Layer (SDL)

Simple DirectMedia Layer [2], usually called SDL, is a cross-platform library mainly used for games as it provides easy access to video, audio, and input devices. This is invaluable when developing for multiple platforms. From this point onwards any mention of SDL will refer to SDL 2.0.

SDL covers a wide range of input devices including joysticks, gamepads, mice, keyboards, and touch devices. It accesses these devices through the APIs provided by operating systems and presents them in a convenient way. Of all the options covered it is the most complete and appropriate for our case. It allows access to device state directly through some methods and all activated input devices will also send their events to the main SDL event loop [3].

It should be noted that directly accessing the state actually reads from the internal state that SDL keeps, which it keeps up to date using the events it gets from devices. When the devices do not provide events, like analogue joysticks, SDL will regularly poll them for their state instead. Mouse and keyboard state can be obtained directly as well.

Touch does not allow easy direct state access and is entirely based on the events it creates, so if we wanted to allow direct access to some touch data we would need to keep the state inside our interface. For touch input SDL also has support for single finger gesture recognition and can provide some data for multiple finger gestures like the amount of pinch and rotation [4], which we could potentially incorporate into our interface if needed.

One of the limitations of SDL is that it limits the amount of mice to one and unifies their events into a single device like a lot of operating systems do [5]. This means that we are unable to pass through individual mice and would require a separate source if we wanted this functionality.

2.2.2 DirectInput and XInput

DirectInput [6] is the old API for Windows that supports joysticks as well as keyboards and mice, but for the latter it is now advised to use the window events instead. Nevertheless they are available to us through the API and can be accessed either individually or as a system device unifying all the mice or keyboards.

Acquiring a joystick, which gives access to its data, requires enumerating through devices to find it, while the system keyboard an mouse are available for acquisition without enumeration. Before acquisition the client can define what kind of data it wants to gather, usually this is either full device state or a buffer of events since the last check. After setting up and acquiring the device it is recommended to query the state or event buffer during a main loop to receive the updated state of our device. Analogue joysticks do not provide any kind of events so they are polled when their state is requested.

XInput [7] is the newer API that only supports up to four Xbox-like controllers. It provides a simpler interface and access to more features compared to DirectInput. The same API is used on the Xbox console for cross compatibility.

Determining if a controller is connected with XInput is done by simply querying its state using an index of 0-3 and checking if it returns an error. The lights on connected controllers give a visual indication of their index. This API only supports getting full state packets from the controller and does not have an event based system. The data packets are numbered so it is easy to determine if anything changed since the last check.

The XInput and DirectInput APIs can be used simultaneously allowing the user to choose the most appropriate one for each device, which is what SDL does on Microsoft Windows platforms. These APIs are limited to a single operating system family and thus not very appropriate for our case, but the way they function was useful to research.

They are quite similar to SDL, though DirectInput requires significantly more effort to set up. The main difference being that XInput and DirectInput do not use querying state of individual parts of the joystick but instead opt for returning the entire state in one packet, like is usually the case in USB packets. Another difference is that, instead of getting all events in a single queue like SDL, event buffers are per-device and not unified. These per-device queues need to be tracked and checked individually, which is more complex than using a unified queue, but this method does make the event source clear without requiring an event field to indicate the source device like SDL. It also allows the user to check certain devices more often without having to process events from all other sources.

There is no explicit support for touch input in either of these APIs, so supporting touch requires window events like is recommended for mouse and keyboard events.

2.2.3 Linux input devices

Linux input drivers [8] can provide data through file descriptors typically found in `/dev/input/`. The main way to access the devices we aim to support is through evdev, which lists individual devices as `/dev/input/event[0-31]`.

These event file descriptors can be read to obtain a standard event structure that contains all relevant data for that device. This is essentially the same as a per-device event queue like DirectInput uses. The file descriptor can be checked for new events using `select()` and information like the current state of the device and extra information on its capabilities can be gathered using `ioctl()`. Accessing mice and keyboards directly through this system requires superuser permissions, so it is not useful for our purpose.

SDL uses this unified evdev system to gather data from joysticks, but a lot of drivers also provide more specialised access to this data. Mice and keyboards can be read individually or through a unified descriptor, though these still require superuser permissions.

Joysticks are also accessible through the joystick API [9], which lists all connected joysticks as `/dev/input/js[0-31]`. These joystick file descriptors only support buttons and axes and work a lot like the evdev joysticks, but their events are smaller and they can not be queried directly for their state. Instead they queue their entire initial state in events when their file descriptor is opened. This joystick API is attractive for implementing a proof of concept as it only requires reading from a file descriptor.

Actual access to mice, keyboards, and touch devices as a non-superuser usually utilises the X Window System. This is elaborated upon in the next section on the Xinput extension for X11.

2.2.4 X Input Device Extension Library

The X Input Device Extension Library [10], or Xinput for short, is the primary way of getting keyboard, mouse, and touch device information when using the X Window System. Though it shares a name with a Microsoft API their similarities end there. Proper support for multi-touch devices was not implemented until Xinput 2.2.

Its method of distributing events can be compared to SDL. There is a single unified event queue that the user reads from and the kinds of events it provides can be selected. There is a mechanism to enumerate other devices besides keyboards and mice and open them to receive their events. This does not seem to be listing joysticks though, so supporting those is still going to require the joystick API. When Xinput 2.2 or higher is present on a system it is possible to turn on multi-touch support, which will also send touch events into the queue.

Overall Xinput works a lot like SDL with a main event queue that receives all window, keyboard, and mouse events with the option to also receive touch events. It also supports querying the state of devices directly.

2.2.5 Kivy

Kivy [11] is a cross-platform GUI toolkit aimed at touch interfaces and thus works a bit differently from the previous APIs we looked at. It dispatches touch events to all visible widgets that are listening for them, so those widgets get to do something with them.

Kivy can access joysticks through the PyGame API, which works much like a joystick in SDL. This system is separated entirely from the touch input event system. In addition to motion events from touch devices it also supports motion events for accelerometers as this framework is also aimed at mobile devices. It does not dispatch any keyboard events like other frameworks, as it was built with touch devices in mind where keyboard input is usually through text fields using an on screen keyboard. The mouse can be used to emulate multitouch, but normal mouse input data is not directly accessible. There is no gesture detection built in at this point in time.

Overall this approach is very appropriate for a GUI toolkit, but not very logical for our input interface as our implementation would not be GUI focused.

2.3 Summary

We give a more concise overview of the capabilities of the frameworks and how they compare in table 2.1.

Overall SDL provides the best abstraction for all devices and it does so in a single unified event loop. The cross platform aspect and the fact that it is already present in MGSim make it the obvious choice for gathering input data from the sources we want to support. The lack of access to individual keyboards or mice is certainly a limitation, but we are not planning to support keyboards in this interface and the mouse data can be based on the system pointer.

Framework	Joystick	Keyboard	Mouse	Touch	Events	State access
SDL	Many	Unified	Unified	Multi	Unified	Yes
DirectInput	Many	Individual	Individual	No	Per device	Yes
XInput (Microsoft)	4 gamepads	No	No	No	No	Yes
Linux kernel API	Many	Individual	Individual	Multi	Per device	Yes
XInput (X11)	No	Unified	Unified	Multi	Unified	Yes
Kivy	Many	No	No	Multi	Per widget	No

Table 2.1: Comparison of frameworks for handling input devices

Chapter 3

Interface design

With our list or requirements we can start designing an interface that satisfies them. This chapter only deals with the design of the interface and how it functions, while chapter 4 deals with the implementation of a component that emulates it. We start with a general overview of the design and move on to a more detailed description. In the last section we discuss some of our design decisions. A simplified diagram of the architecture model is available in figure 3.1 at the end of this chapter.

3.1 Design overview

We designed our interface for memory mapped I/O, which means that we have an address space and all interactions will be read or write requests. It supports sending interrupts, but this feature is optional. While MGSims packet based I/O network was what we designed for, it should be adaptable to other MMIO based models.

Our design is a mock-up and not suited for any production components. While the address space division and protocol could be implemented, supporting the actual functionality would require a separate microcontroller to handle and process all USB communications with external devices.

The interface provides information on the layout of the connected device, read access to the front element of a FIFO event queue, and direct access to the device state. While the next sections of this chapter may sometimes refer to specifics of our implementation, the interface is designed to be extensible and to support many different implementations.

3.2 Address space division and protocol

The address space is divided into several sections and subsections, each with their own purpose. These sections are spaced out in a way that would allow a (de)multiplexer to easily select sections based on certain bits of the address.

When we refer to a bit by number in the following sections we use a system that starts numbering from 1 at the least significant bit. Any addresses we mention are offsets from the base address of the device.

Major sections are selected by bits 11 and up giving us kibibyte (1024 bytes) size blocks. We currently use five of these blocks but support further expansions to the interface. The first of these sections has an extra subdivision on bit 10 resulting in two blocks, the first of which also divides itself based on bit 9.

The resulting sections and their functionality is summarised in table 3.1, but we will cover these things in more detail in the following sections. Some of the details in this

Address bits value				Hex	Width	R/W	Description
11+	10	9	1-8				
0	0	0	0	0	1	R	0 if disabled, device type otherwise
						W	0 disables, non-zero enables device
			1	1	1	R	1 if events are enabled
						W	0 disables, non-zero enables events
			2	2	1	R	1 if interrupts are enabled
						W	0 disables, non-zero activates interrupts
			3	3	1	R	the interrupt channel
						W	set the interrupt channel
			4	4	1	R	The amount of queued events
						W	Pop the front of the event queue
		1	0	100	4	R	Information on the axes section
			4	104	4	R	Information on the buttons section
			8	108	4	R	Information on the hats section
			12	10c	4	R	Information on the balls section
			16	110	4	R	Reads 0 to indicate no further sections
	1	0,4,8,12,16		200-210	4	R	Read 4 byte chunks of the current event
1	0,2,4,...,510			400-5fe	2	R	Direct access to axis states
2	0,1,2,...,31			800-81f	1	R	Direct access to bitsets with button states
3	0,1,2,...,255			c00-cff	1	R	Direct access to hat states
4	0,2,4,...,1022			1000-13fe	2	R	Direct access to ball states

Table 3.1: Overview of the interface address space in our implementation

table are based on our implementation, but we will clearly explain what parts of the interface are implementation-defined.

3.2.1 Interface control and status

This section at the start of device memory is the only section that can be written to. All access to this section is done using I/O operations of 8-bit width. This is the only section that is about the interface itself, every section after this is dedicated to information about the external input device it is connected to.

The first byte can enable and disable the interface. Writing 0 disables the interface while any other value activates it. When read it returns either 0 when the interface is disabled or an implementation-defined device type when it is enabled.

The following two bytes are used to activate events and interrupts respectively. Writing 0 disables these features while any other value activates them. When read they return 1 if that feature is enabled and 0 otherwise.

The fourth byte can change the interrupt channel by writing to it and when read it returns the current channel.

The fifth byte contains the length of the current event queue. While there is no hard limit on the event queue size, this value is capped at 255. Writing to this address can be used to pop the front event off the queue.

Interrupts only function when events are enabled and they indicate that there are events in the queue, so when an event is placed in the empty queue it sends the interrupt. The interrupt is cleared once the last event is popped of the queue.

3.2.2 Device information

The second section describes the layout of the device that is connected to the interface and how its state is described. It can be accessed using 32-bit aligned read operations. The addresses for this section start at `0x100`.

Each value read from this section corresponds to one of the direct state access sections of the device. It describes how to access that section and how the state is conveyed. The first value describes the second kibibyte block of device address space, the second describes the third kibibyte and so on. When a value of 0 is read it indicates that there are no more sections.

This section is used to signal some of the implementation-defined details. Our implementation has four sections for direct state access, so we have four values in this section. These four values correspond to the different types of joystick parts that produce input data: axes, buttons, hats, and balls. We will discuss these types in more detail in section 4.3.1.

The 32-bit values read from this section actually represent four unsigned 8-bit values. The most significant byte, bits 25-32, represents the amount of items in this section. Bits 17-24 represent the required access width of this section in bytes. Bits 9-16 describe the amounts of bits used to represent a value in this section. Bits 1-8 describe amount of values that are stored per item.

To make this easier to understand we will give an example. Consider a connected device with six axes which produce one 16-bit value each. These values are served individually so the access width is two bytes. In this case 32-bit value would be composed of the following bytes `0x06 0x02 0x10 0x01`.

All this data is exposed to make interface extensions easier and to allow data representation to differ between implementations. If some implementation chooses to represent whole other kinds of data through this interface, all information required to access this data is available during runtime.

3.2.3 Event access

The third section can be used to access up to 512 bytes of event data. This section can be accessed using aligned 32-bit read operations. Addresses for this section start at `0x200`.

This section can be used to access the event that is currently at the front of the event FIFO. Accessing this section is only allowed if events are enabled and there is at least one event in the queue.

The structure of the events that are presented here is implementation-defined. We provide a header with our implementation that describes the event structures for each of our device types. Knowing the event structure allows the user to determine which 32-bit chunks of the event are useful to copy.

After copying the relevant chunks of event data to local memory the event can be popped by writing to the queue status byte in the control section.

3.2.4 Direct state access

The remaining device address space is divided into kibibyte sized blocks used for direct device state access. The access width of these individual blocks can be determined using the information in section 3.2.2. All accesses to this section have to be aligned. Each block can hold up to 256 32-bit values, which should be enough to represent any conventional device.

Our implementation defines four of these sections, which correspond to the four types of data produced by joysticks that we will describe in section 4.3.1.

3.3 Design decisions

This design went through several iterations before arriving at this version and we will use this section to elaborate on some of the decisions we made along the way.

3.3.1 Access width

One major decision has been access widths, in the current iteration these are fixed per section but can vary between sections. We could have chosen a fixed 32-bit access width as well, given that the only section that is currently forced to 8-bit width is the control section which could just as well be 32-bit. The latter sections could easily provide direct access in 32-bit values by presenting multiple items in a single response.

However, using 32-bit operations for the data we want to present in these sections is not always convenient, because extracting individual values could require shifting or masking. The control section at the start simply does not require more precision than 8 bits provide, so we decided to keep it this way.

Another way of handling access width is to allow the user free access to any valid address with any width. This obviously would not work for the control section, but the other sections could technically be handled this way. This requires handling cases like requests that start at the end of valid address space and read into undefined space. Unaligned reading and reading parts of larger values would need to take the endianness of host and simulated system into account. To avoid making the interface more complex we decided against doing this.

3.3.2 Event queue popping

The way we pop events on the queue is by writing to an address, but this was not the only method we considered.

Another way to handle this is popping the queue once the entire event has been read, but with our variable event structures that could end up wasting cycles on copying useless or empty chunks.

An alternative is to pop the queue once the first chunk is read so the event could be copied back to front, but this first chunk might not contain useful information and requiring to read it could slow everything down.

So we arrived at our current solution of writing to the interface, because the asynchronous I/O interface of MGSim does not have to wait for a response after sending a write request.

3.3.3 Extensibility

The design is undeniably based on joystick data, but we did manage to make it easily extensible if new types of data need to be presented. The sections we defined also have a lot of unused address space to allow for extensions of functionality. We made sure that it is always possible to get direct access to state data, even if there is no external information available on how an implementation presents it.

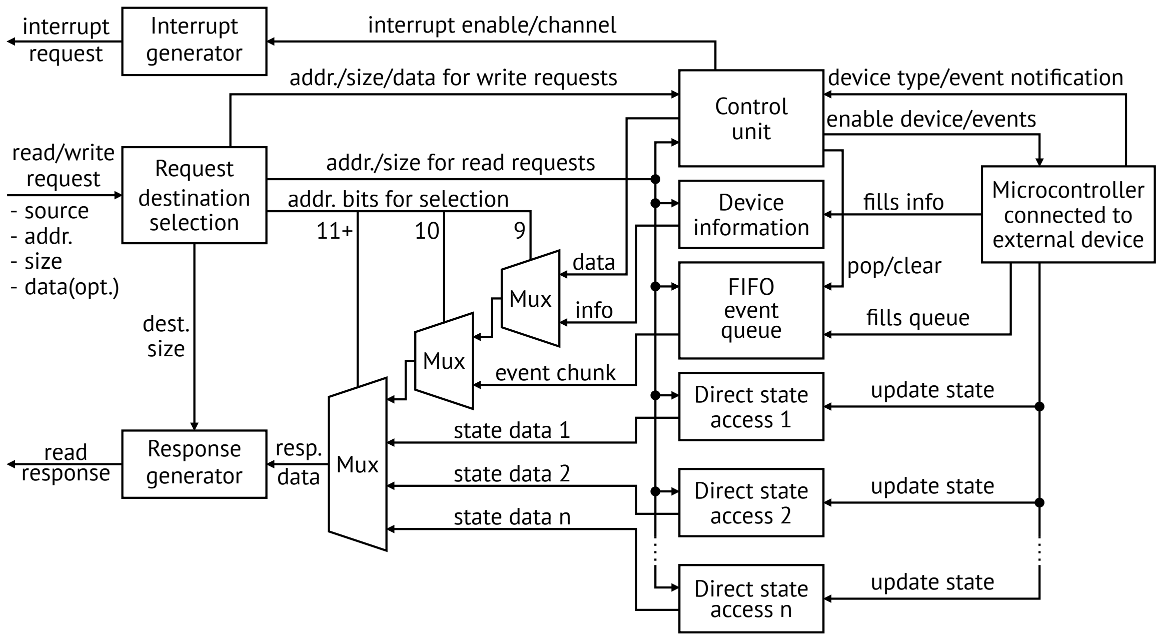

Figure 3.1: Diagram of architecture model

Chapter 4

Implementation in MGSim

Implementing our interface design in an MGSim component that connects it to an external device took several steps. This chapter discusses these steps and how they worked towards achieving our final result.

4.1 Implementation process overview

We started by familiarising ourselves with how MGSim functions. To learn how to use memory mapped I/O we wrote a simple program that uses the graphics device. Our understanding of how components function improved greatly after we studied the source of the graphics component to learn how it was interpreting our commands. This armed us with the required knowledge to start working towards our goal.

Our contributions to MGSim started with creating a simple proof of concept that uses the existing UART to pass events in section 4.2. This proof of concept is very limited, so in section 4.3 we describe the modifications we made to properly connect MGSim to external input devices through SDL. We created a system to gather and distribute events and device information to components. We extended our proof of concept to support receiving events through our new system in section 4.4. And finally, to properly support all the data our new system provides, we created the component that implements our interface design in section 4.5.

Before we move on to specifics it is useful to know how I/O components for MGSim function. An I/O component registers itself with the I/O interface when it is initialised and has to implement a few methods. Two of the important methods handle incoming read and write requests respectively. Handling read requests requires sending a response with requested data back through the I/O interface.

All of our modifications to MGSim, including code in `arch/dev/` and documentation in `doc/`, are available in a branch on GitHub [12]. An additional repository with example code and configurations can also be found on GitHub [13]. We will explain one of these examples in more detail in section 5.2. A class diagram showing a simplified view of all of our modifications and how they interact can be found at the end of this chapter in figure 4.1.

4.2 Proof of concept using the UART

Being a proof of concept we wanted to keep things simple, so we used the Linux joystick API [9] that we mentioned in section 2.2.3. It only requires reading from a file descriptor and has a constant packet size, making it easy to work with. Our other option was using evdev, but those events are bigger in size and it does not provide an initial device state.

```c
volatile uint8_t   *uart; //pointer to the base address of the UART
struct js_event       e; //event as defined in joystick API documentation
uint8_t *buff = (void*)e; //Use the event as buffer

//This next part would preferably be in the main loop
if (uart[5] & 1){ //Check the data ready bit of the LSR for a new event
    for (int i = 0; i < sizeof(e); i++)
        buff[i] = uart[0]; //copy individual bytes
}
//Process the event that was copied
```

Listing 4.1: Using the proof of concept

We decided to use the existing MGSim component that implements a universal asynchronous receiver/transmitter, usually referred to as UART. A UART facilitates serial communication between two systems, usually by transmitting and receiving single bytes as packets of individual bits without requiring elaborate synchronisation protocols. Using a UART to communicate is simple and straightforward, writing data to the UART transmits it, while reading returns the data that it received. Internally the data is sent from and received in shift registers, which transform data into individual bits and vice versa. Many UARTs have FIFO buffers to combat data loss when the CPU writes too quickly or takes too long to read received data. UARTs are common on microcontrollers, as their simple but versatile abilities can be used implement many features like debug shells.

We decided to modify the UART component because it was very simple to use and its implementation already allowed reading from file descriptors, including ones that do not constantly provide data like our joystick. The UART component does not emulate to the level of transmitting individual bits, but keeps everything byte sized instead. It features two FIFO buffers for both receiving and transmitting.

The code for the UART component can be found in the files `UART.cpp` and `UART.h`. Documentation about configuration and usage can be found in `mgsimdev-uart.rst`.

We added a new mode to the component for reading from joysticks and allowed the user to define what file descriptor it used in the configuration file.

The original way the component reads from file descriptors is byte-by-byte, but single byte reading from the joystick API file descriptors is invalid. The joystick API requires read requests of at least eight bytes, which is the size of one event. If a request asks for more than eight bytes it fits as many full events as it can in the requested amount of bytes. Only allowing complete events to be read is convenient for the joystick API, because it can just move the head position in its internal ring buffer to pop the events that were read.

Changing the component to read eight bytes from joysticks lead to the problem that the UART input latch only holds one byte at a time, so we only move the first byte to the latch and store the rest in a queue. When a byte from the input latch is moved to the FIFO buffer, the next byte from the queue is put in the latch.

Listing 4.1 contains a simple example of interacting with the proof of concept from inside the simulation. This example only shows copying the bytes of an event from the FIFO inside the UART and does not include locating the UART base address, the joystick API event structure [9], and processing the event after copying. This example also assumes that the UART either has the entire event ready in the FIFO or is clocked fast enough to copy a new byte from the input latch for each subsequent read request.

A more elaborate example called `uartjsvizold.c` is available in the examples repository [13].

While functional, this method is slow, can only be used on Linux, and only provides very simple joystick events. It lacks support for all other device types we want, direct access to device state and info, and the joystick events only distinguish between axes and buttons with no proper support for hats and balls.

4.3 Supporting SDL-based input devices in MGSim

To get around the platform-binding limitations and support the wide variety of features we were aiming for, we have to modify MGSim to use SDL for input gathering. This will give us cross-platform access to all the devices we aim to support with our interface.

SDL was already used by MGSim to render displays for all graphics devices, which required an event processing system that handled window and keyboard events for these displays. We moved the existing event handling code contained in the `DisplayManager` subclass of the graphics device component, into a new `SDLInputManager`. This new manager retains all display related functionality while also providing ways to access data from input devices.

The code for the `SDLInputManager` can be found in the files `SDLInputManager.cpp` and `SDLInputManager.h`.

We will start by describing how to interact with the manager and and what data it provides before moving on to some implementation details.

4.3.1 Interacting with the SDLInputManager

The main functionality of manager is dispatching events to registered clients. To become a client a class needs to implement a simple interface called `ISDLInputClient`, which contains a single method called `void OnInputEvent(MGInputEvent event)`. This method will be called by the manager to dispatch events to its clients.

There can only a single instance of the manager at any time, so before trying to interact with it for the first time we recommend calling `CreateManagerIfNotExists`. In order for SDL to process events properly, there needs to be at least one active window, so an active gfx device is required.

While we will cover general information about events, the specifics about their structure and contents are explained in appendix A and available in header form as `MGInputEvents.h` on GitHub [12]. Specifics about method calls of the manager can be found in the header `SDLInputManager.h`.

Client registration

A client can register with the manager to receive events for either a joystick, the mouse pointer, or touch devices. For the latter two of these, there can only be one client registered at a time. Mouse events generated by a touch interface are distributed to the mouse client unless there is also a registered touch device client.

Registration for a joystick requires the client to provide a joystick index, which is used to select one of the connected devices. Every joystick can only have one registered client, so we provide a method that checks if a certain joystick index is valid

```
struct JoystickInfo
{
    int                 id; //SDL instance ID (not joyindex)
    unsigned char    naxes; //Number of axes
    unsigned char nbuttons; //Number of buttons
    unsigned char    nhats; //Number of hats
    unsigned char   nballs; //Number of balls
    JoystickInfo()
        : id(-1), naxes(0), nbuttons(0), nhats(0), nballs(0) {}
};
```

Listing 4.2: JoystickInfo structure

```
struct JoystickState
{
    int              instance_id; //SDL instance ID
    std::vector<int16_t>    axes; //Absolute positions
    std::vector<uint8_t> buttons; //Bitstring with 1-bit states
    std::vector<uint8_t>    hats; //encodes state of a d-pad
    std::vector<int16_t>   balls; //Relative values in two dimensions
    JoystickState()
        : instance_id(-1), axes(), buttons(), hats(), balls() {}
};
```

Listing 4.3: JoystickState structure

and available for registration. This index is also required to access the additional information our interface provides.

Any registered client can also undo its registration to stop receiving events and free a certain device for another client.

Device information and state

Next to the events it distributes, the manager also allows clients to get additional information about the device they are connected to. This is mainly useful to get information about connected joysticks, so the data structures we use are based on the way SDL represents joysticks. While data from the mouse pointer can be adapted to fit this format, this additional information is not available for a touch client.

There are two types of additional information available, `JoystickInfo` which describes device layout and `JoystickState` which describes its state. These structures are visible in listings 4.2 and 4.3 respectively.

The joystick layout is described by the amount of inputs of a certain type. We support four types of input sources: axes, buttons, hats, and balls. The kind of joystick parts these types represent and how we store their state is explained below. The joystick state structure contains a vector for each input type, which contains the states of all inputs of that type.

An axis produces a signed 16-bit integer representing an absolute position and is usually used to represent clearly bounded input sources. Controls that present themselves as axes include triggers, sliders, joy- and analogue sticks which present their state as two axes, and we use them to indicate the absolute position of the mouse cursor on the SDL window.

The state of a button can be encoded as a single bit of data, so button states are presented in one or more bytes, each representing eight buttons. The state of a single button can be extracted using a mask. In addition to the buttons of joysticks this also encodes the mouse button state. The header we provide shows how the mouse buttons are numbered and provides a macro to generate a mask for a given index.

Hats encode their direction in the lower 4-bits of the 8-bit value we store, Each bit indicates one of the four main directions and they can be combined to get a direction. We provide masks to simplify extracting the state in the header. This data type is used to represent directional pads that can point in eight directions.

A ball represents relative movement on two axes in the form of two 16-bit signed integers, so the size of our vector in the state is actually twice as large as the amount of balls. This data type is generally used to represent trackball movement, but it could also be used to represent relative mouse movement of the pointer and the last movement directions of the mouse wheel.

Event data

The events that a client receives are based on SDL events [3], but while the numerous structures SDL uses are no problem on modern systems they are less suitable for use in a simulator like MGSim. SDL provides a different structure for nearly every event type, which is unnecessary as most of the fields are reusable. There is a lot of padding and every event has extra values to help identify things like the source device, which is useless information for a simulated system. Several fields are also very wide for the values they will contain. To make transmitting events to the simulation as fast as possible we created a new set of three optimised event structures to represent joystick, mouse, and touch events. The input manager translates SDL events into our own event structures before dispatching them to clients. More information about the events and their fields can be found in appendix A and the header we provide.

We chose to separate event types for device types because the data their events produce is so different. While it would be possible to translate every mouse event into several joystick events, it would not be very efficient. Touch events are a separate category entirely and they would not translate well at all.

The touch events provided by SDL are filled with many floating point values, which takes a lot of space and is problematic for systems that have no floating point unit. For our own events they are translated into signed 16-bit fixed point representations. The values in the SDL event are normalised in the interval $(-1.0, 1.0)$, so by multiplying with 32768 we can obtain a fixed point value in 16-bit signed range with the radix right after the sign bit. To handle the case where the float value is either -1.0 or 1.0 we change it to the closest floating point value that does fit in our interval. The conversion function is available in appendix A.4.

A client is recommended to keep an internal joystick state up to date instead of constantly requesting it from the manager. Keeping the state up to date based on events is significantly less performance intensive than updating the entire state in the manager.

```
struct SDLInputClientContext
{
    SDL_Joystick      *joystick; //SDL joystick representation
    ISDLInputClient     *client; //Pointer to the client
    int                joyindex; //joyindex used for registration
    struct JoystickInfo joyinfo; //Information on the joystick
    SDLInputClientContext()
        : joystick(0), client(0), joyindex(-1), joyinfo() {}
};
```

Listing 4.4: Input client context structure

4.3.2 Implementation details

The display manager that we started with already had an event loop that handled keyboard and window events for connected displays. Event checking was only done right before refreshing the display. This was on a large delay because it is expensive in terms of performance and refreshing often is not really useful in the case of a slow simulation.

To achieve our goal of low input latency we added a way of checking for events without refreshing the display and made the amount of cycles between event checks configurable separately from the display refresh timing.

Client management

Information about registered clients is kept in a client context shown in listing 4.4. This contains a pointer to the SDL joystick, a pointer to the client itself used to pass events, the joystick index they registered with and a JoystickInfo describing the connected device.

We only allow one mouse client and one touch client, so the client contexts for these devices are allocated statically. When a client registers for mouse or touch events it updates the client pointer and sets the `joyindex` to 1 to indicate that there is a client. The `JoystickInfo` for the mouse is also updated to the standard mouse description. When these devices undo their registration the client context is reset.

Keeping track of joystick clients is slightly more complicated. When a client wants to register for a joystick, we start by checking if the requested joystick exists and is not yet in use by another client. If the joystick is available, we save the client pointer and the `joyindex` in the client context. The manager then fills the `joystick` field by asking SDL to connect to the joystick which returns the joystick pointer. If the connection was successful the `joyinfo` is filled by calling SDL functions. This data includes the joystick instance ID, which should not be confused with the joystick index. The instance ID is used in joystick related SDL events to indicate the source joystick, which allows us to distinguish between multiple connected joysticks. The event processing loop is the most executed part of the manager code: to optimise locating the correct client for an event, we store our clients in a map using the instance ID as the key.

When a client undoes their registration, the corresponding context is located in the map, the SDL joystick connection is closed, and the context is removed from the map.

Event processing

The event checking loop polls SDL for events until the event queue runs out. We updated it to add cases to catch events for the devices we support.

For mouse and touch events, these cases are fairly straightforward, since we only support a single client for these devices at any time. In the case of joystick events the instance ID from the event is used to find the relevant client in the map.

Once a client is located the event is converted into one of our own event structures and passed to the `OnInputEvent` method of the client.

Event conversion is a fairly straightforward process that starts around line 235 in `SDLInputManager.cpp`. Some of the mouse fields are cast from 32-bit to 16-bit integers to save space, which is not a problem as getting these values out of 16-bit range would require an SDL window with a horizontal width over 32767. This window would need to span more than 17 horizontally stacked 1080p displays. Touch events have their floating point members converted to fixed point, which results in enough precision to determine the touched pixel on a 32K (30720x17280) display.

Info and state requests

Any requests for `JoystickInfo` just return the one stored in the client context. Requests to update the joystick state use the passed joystick index to find the related client context. The vectors are resized to fit the state information and SDL functions are used to fill it using the internal state of SDL. Mouse state requests do something similar, but there are no SDL functions to query the last relative mouse and wheel movement so those ball values are set to 0. These state update requests are useful for getting the initial state, but more expensive than a client keeping the state up to date based on the events it receives, so we recommend using that method instead.

4.4 Updating the proof of concept

With our input manager ready for use, we decided to update the UART once more. The new mode we added requires the user to define a joystick ID in the configuration file. The UART uses that ID to connect to the input manager and receive events for the corresponding joystick. The callback handler puts the events on the same queue of bytes we used for the other UART joystick mode.

The only difference for a program running in the simulation is that we use the different event structure now which is two bytes larger. To adapt the example we gave in listing 4.1, the only change would be to change the event type to `MGJoyInputEvent`. A more elaborate example called `uartjsviznew.c` is available in the examples repository [13].

This method is slightly slower than the previous one because we have to transfer two additional bytes, but it does add support for hats and balls and is now platform independent.

One of the potential issues with this implementation would be additional latency due to the extra layer that SDL adds compared to the joystick API, but as we show later on this is not the case.

4.5 The JoyInput component

As the title of this thesis suggest the goal was to implement an interface, so armed with our design from chapter 3 we can finally create the component we desired. Our implementation takes the form of a component called `JoyInput`, which might be deceptive as it also supports mouse and touch input, but as far as names go this one does bring joy.

The code for the component can be found in the files `JoyInput.cpp` and `JoyInput.h`. Documentation on how to configure and use it can be found in `mgsimdev-joyinput.rst`. In this section we will describe its configuration options before moving on to describing several aspects of the implementation.

4.5.1 Configuration

The component can be added to any system configuration by defining a `JoyInput` component and choosing a device type. The required option `:JoyInputDeviceType` can be set to one of the following values: `JOYSTICK, MOUSE, TOUCH, REPLAY`.

`MOUSE` and `TOUCH` require no further configuration, while `JOYSTICK` requires setting the additional option `:InputJoystickIndex` to the desired joystick index.

The `REPLAY` mode will read data from a log file created on a previous run of MGSim and reply to requests the exact same way as long as they arrive in the same order.

The `:JoyInputReplayFile` option defines the file used for either saving or reading a replay. If a replay file is defined and the device type is anything but `REPLAY` a log is written.

The last setting we discuss does not impact the `JoyInput` device directly and needs to be put in a global block. The optional `SDLInputPollDelay` defines the delay between polling for events in cycles. This defaults to 1000 cycles when it is not defined.

4.5.2 Communications with SDLInputManager

In anything but replay mode the device relies heavily on the `SDLInputManager`, starting during component initialisation where it makes sure there is a manager to work with. If the component is configured to connect to a joystick a quick check is done if the defined joystick index is valid.

When the component is turned on through the interface we register for the device we want and request the `JoystickInfo` to update our local copy. If the device is either a joystick or a mouse the local `JoystickState` is passed to the manager to update it for accurate state data. If either the registration or state update fails the component is not enabled.

When the component is turned off through the interface it undoes the registration with the manager.

In the `OnInputEvent` method, which is required to be a client, we receive events and use them to update the local `JoystickState`. If events are enabled the event is also added to the queue and if interrupts are enabled the flag to send them is set.

4.5.3 Request handling

Given that our implementation outsources event collection, most of the code is dedicated to handling read and write requests. The address space division for our implementation is the one we showed in table 3.1 back in section 3.2.

Write requests are handled in a fairly straightforward manner given that only the first few addresses allow writing and they are all 8-bit width. After catching invalid writes there is a simple switch statement on the address that handles enabling the device and its various features. When the event queue popped its last event the interrupt flag is cleared.

Read requests are slightly more complex given the variable access widths and the amount of different sections, so any request is considered invalid until proven otherwise. The bulk of the code is nested switch statements on various parts of the address up to three levels deep to mimic multiplexers. Any responses bigger than a byte are appropriately converted to the correct endianness for the simulation. All sections personally verify if access width and alignment are correct and if the address is in bounds.

The first section is fairly straightforward as it just returns information about the component state.

The second section describing device layout and direct state access is static apart from the number of items in the upper byte.

The third section providing event access uses a pointer into the current event to extract the appropriate chunk.

The latter sections that provide direct state access are all just translating addresses into indexes to use for the vectors in the state.

4.5.4 Replay functionality

One of the challenges posed at the start of this project was implementing a deterministic replay capability. This would make it easier to compare performance between configurations and can reproduce exact situations to make tracking down bugs easier.

The way we decided to implement this replay was at the level of requests and responses in the component itself, with every request and response logged with all necessary data in a plain text format. This log can be read back on a subsequent execution of the simulator to send the exact same responses and check if the same exact writes happen in sequence.

This approach does not allow activating interrupts as that would require more extensive changes to MGSim to track them outside of our component. Currently this does not yet stall the processor if a request is early and only checks if the requests are exactly the same.

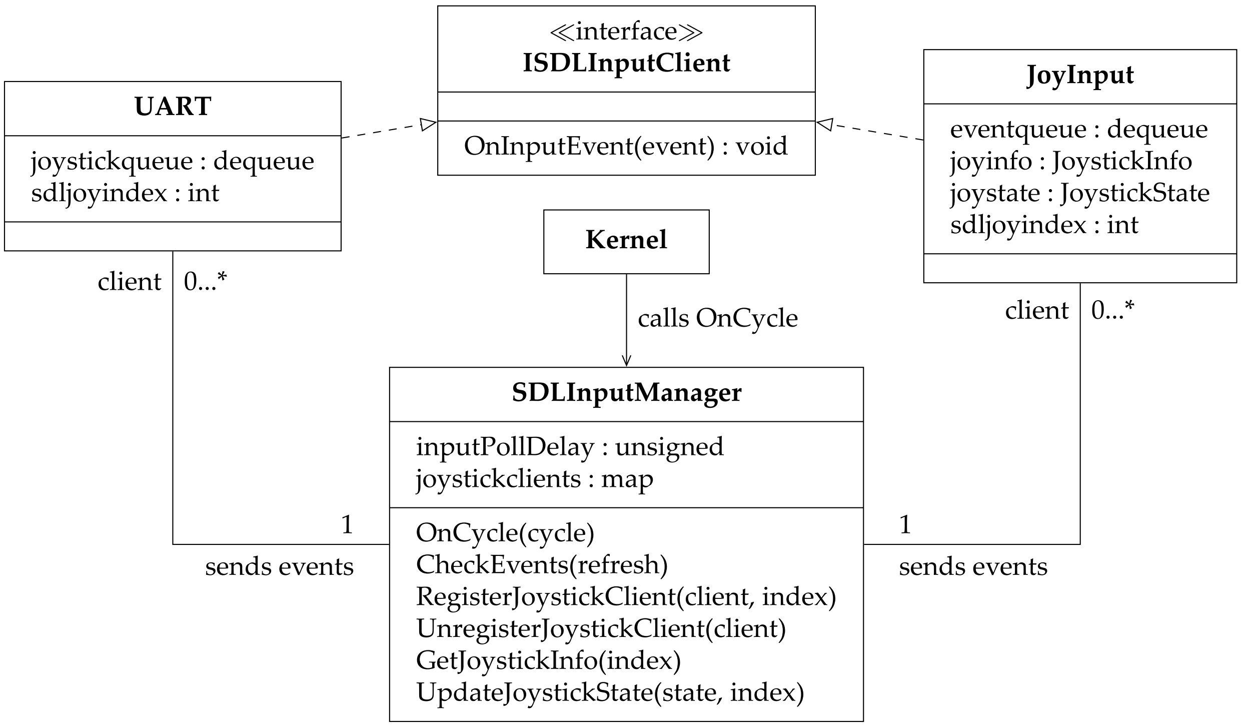

Figure 4.1: Simplified class diagram showing joystick related interaction

Chapter 5

Results

5.1 Performance measurements

While there is no other implementation to compare against, there are some aspects of our implementation that we can assign numerical values to. We will compare our final component implementation to the UART prototypes.

The main measurement here is efficiency in accessing event data, which is the only aspect that is available across all interfaces. The results will, of course, depend heavily on the environment, but with the same configuration they can still provide useful insight.

The comparison was done on the MT-Alpha architecture with a configuration very close to the example that comes with MGSim. We are timing the amount of cycles it takes to copy the entire event into a local buffer. Timing starts when the CPU receives the result from reading the status register or queue size and it ends when the last chunk of the event is received in a register. To get a stable cycle count for executions we had every test copy 10 events before exiting, this ensured that our results were not affected by the cache.

For our component we tested both copying the entire event and just the chunk with the fields we most commonly use. The UART was tested in both of the modes we implemented. After the copying completes we verify that the event was saved correctly to avoid compiler optimisation from manipulating the results by discarding data. Excerpts of the test programs which show the code for the section we timed can be found in appendix B. The results for our test are visible in table 5.1.

Configuration	Original cycles	Corrected cycles
JoyInput (partial event)	22	36
JoyInput (full event)	24	39
UART using joystick API	49	51
UART using SDL	44	64

Table 5.1: Interface performance

Partially reading the event is the fastest method as one would expect, followed rather closely by reading the entire event. The difference between these is so small because all load instructions went to a different address making it possible to send them without waiting for the previous result. Our first results for the UART seemed odd with the configuration that has to copy ten bytes instead of eight being five cycles faster.

After disassembling the code generated for the tests it became clear why this occurs. The compiler was reusing registers and storing some of the bytes it copied already to the stack. In the case of copying eight bytes more of these stores are planned before loading the last byte into a register. Our timing only cares for when the last byte arrives in a register, so the implementation that copies more bytes ends up with a lower cycle count.

The initial measurement ended up being inaccurate, so we looked at the disassembly of all of the test binaries and noticed that all of them store at least some of the results to the stack after reading them. Armed with the disassembly and detailed logs we calculated corrected cycle counts to indicate when the final store is finished and added them to the table. The corrected results match up with our initial expectations and show that our interface definitely has lower latency.

One other question we wanted to answer was if the added layer of SDL added extra latency compared to the Linux joystick API. To test this we attached the same joystick to the UART through the joystick API and through our interface using SDL. We then set both to check for events every cycle to make sure we would not introduce extra delays. In our testing both received the events the same cycle most of the time, but sometimes our interface actually received events a cycle earlier through SDL. After checking the SDL source code it becomes obvious that polling the SDL event queue makes it check the joystick file descriptor directly. The cycle difference is probably due to SDL using evdev instead of the joystick API on Linux.

The last measurement is subjective, but in all our testing we never noticed significant input latency. As long as applications process input regularly it should not be a problem.

5.2 Example program

While the code snippets and descriptions throughout this thesis give an idea of how to use the interface, we would like to show a complete example that actually uses the data. There are several pieces of example code available in the repository [13], but we decided to explain our joystick state visualisation. The commented code is available in appendix C and in the repository as `joyinputviz.c`. An example of how this looks when running is shown in figure 5.1.

The example is showing the state of an Xbox 360 controller. The bar on top indicates the state of individual buttons, the bars below it show the values of all the axes, and the block on the bottom moves to visualise the direction the d-pad is pointing.

We will now give a more detailed explanation of how this is achieved by dividing the code into four sections.

We start by including `MGInputEvents.h` which describes the event structures from appendix A. We also define some other data structures that we use later on.

The first section with executable code sets up the pointers into JoyInput memory and activates the device. The device is configured to produce events and to notify us of new events on channel 0 which we activate. We read information about the layout of the connected joystick and exit cleanly if it exceeds the limits of what we can visualise. The initial states for buttons, axes, and hats are read into the local data structure from the interface.

The next section sets up the graphics. We do not use the gfx library to control the graphics so to understand this code we recommend looking at the documentation, `doc/mgsimdev-gfx.rst`, in the MGSim repository [12].

Drawing starts with the background, which is a single stretched black pixel. A 1-bit palette is created for drawing the bar that shows the button state.

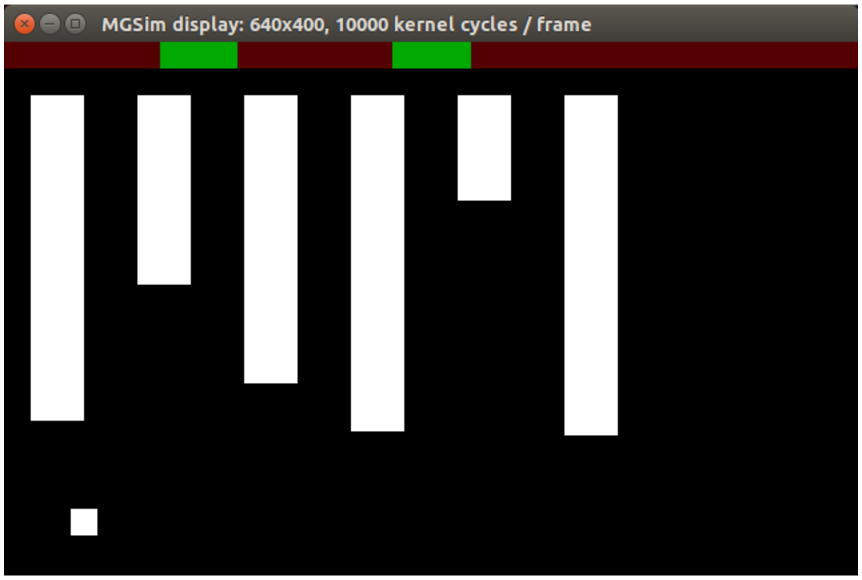

Figure 5.1: joyinputviz connected to an Xbox 360 controller

The button bar is rendered by a line with as many pixels as there are buttons, which we stretch to screen width. This way we can render the correct state by directly copying the state we keep into graphics memory.

The bars for axis states are rendered next, which is done using stretched white pixels scaled based on the value in the state. Hats are also single white pixels, but turned into a cube and positioned to reflect the direction in their state.

With all the graphics prepared we move to the final section. The main loop that processes any events that our interface presents. The loop starts by waiting for a notification so we can avoid constantly checking the interface for new events. When there are events in the queue on the interface we enter a loop that runs until the queue is empty. It starts by copying the event data to the local event and popping the event queue. It first checks if the event matches the stopping condition, which is releasing button 0, before moving on to processing it further. Processing is a simple switch statement that updates the local joystick state and makes the required changes to the graphics framebuffer to reflect this change in state.

Axis values are mapped from the signed 16-bit value to unsigned 8-bit for display, hat state is decoded using masking and the square is positioned accordingly, and the button state is updated using masking and bitwise operations and copied to graphics memory.

This example demonstrates almost all of the functionality our interface provides for joysticks so it was suitable for a more detailed explanation, but we have listed other examples in the repository in table 5.2 with a small description.

File	Description
`joyinput.c`	Prints text about events of up to four connected JoyInput devices.
`joyinputvizmouse.c`	Visualises mouse position and movement.
`joyinputviznoint.c`	The same visualisation explained in section 5.2 without interrupts.
`multijoytest.c`	Draws movable dots for up to 4 connected devices.
`pong.c`	Simple 2 player pong implementation that uses the graphics library.
`uartjsvizold.c`	Joystick visualisation using the UART and Linux joystick API.
`uartjsviznew.c`	Joystick visualisation using the UART and SDLInputManager.

Table 5.2: Description of examples in the repository

Chapter 6

Conclusion

We set out to design and implement an interface for accessing external input devices inside the simulation and in that regard we succeeded. There are still unresolved issues and problems with our approach, but it provides a lot of the required functionality.

6.1 What works

The component can be configured to connect to joysticks, mice and touch devices through SDL and provides multiple ways to access the data produced by these devices. In the end the design is mostly inspired by joysticks with all the different kinds of data being based on features found on these devices, but mouse data can be mapped to this quite easily. For joysticks and mice we currently provide access to both events and device state while touch input is limited to events. Accessing the event and device state is fairly robust and will work consistently during normal operation. Although we did not test it with touch devices, the limited functionality we provide for them should function as well. Recording and replaying sessions also functions and while it does not stall the system, it does verify received requests, reproduces accurate responses, and terminates in case the simulation deviates from the recording.

6.2 Future work

Although the design and its implementation are functional there is always room for improvement.

The handling of touch input is very limited and could be improved quite a bit. We had no environment to test it in and as such could only provide a theoretically functional event passing solution. The main improvements would include storing per-finger data in the state access regions of the address space as well as supporting the gesture capabilities built into SDL, although that would not necessarily be the most realistic capability to implement.

There are features like force feedback on controllers that can be controlled through SDL, which would be nice expose through our interface to enable the simulated system to provide feedback to the outside world.

One rather unfortunate limitation of our current implementation is that in order for SDL to start processing events there needs to be an active window. It will not work properly and consistently when there is no window. This requires more research into the internal workings of SDL which we did not have time for.

To make the interface easier to use inside the simulation one of the future developments could be a driver to simplify usage of an input device. We could even have it run on a separate core with I/O access. That way the main program does not have to concern itself with checking for and copying the events.

The deterministic replay functionality could be modified to also stall the processor in the future. There is a potential other way of implementing replay that does so at the event supplier level. This involves logging the events that the component receives when recording and sending those same events to the component during replay without needing to connect a device.

While we did list its disadvantages, we would still like to improve direct state access in the design by allowing variable access width. This would be limited to only allow bigger widths than the original and would require them to be aligned. That way we can deal with endianness effectively and we can handle requests that go out of defined space by zeroing undefined parts.

Some miscellaneous possible improvements are handling device dis- and reconnection properly and allowing relative mouse mode [5].

6.3 What I learned

This project has taught me a lot about computer architecture and how the CPU can interact with separate memory mapped I/O devices. I learned about designing protocols that could work with this model. In my research I learned about frameworks for input gathering as well as how part of the USB stack on Linux functions to facilitate the interfaces used to present them. In extending the MGSim simulator I learned more about emulating systems and how it was approached in this case. Writing examples and testing taught me to write simple C programs while having no access to a standard library and no backing operating system.

Bibliography

[1] Mike Lankamp, Raphael 'kena' Poss, Qiang Yang, Jian Fu, M. Irfan Uddin, and Chris R. Jesshope. Mgsim - simulation tools for multi-core processor architectures. *CoRR*, abs/1302.1390, 2013.

[2] Simple DirectMedia Layer homepage. https://www.libsdl.org/.

[3] SDL Event Handling documentation. https://wiki.libsdl.org/CategoryEvents.

[4] SDL gesture recognition documentation. https://hg.libsdl.org/SDL/file/default/docs/README-gesture.md.

[5] SDL mouse API documentation. https://wiki.libsdl.org/CategoryMouse.

[6] Microsoft Corporation. DirectInput API documentation. https://msdn.microsoft.com/en-us/library/windows/desktop/ee416842(v=vs.85).aspx.

[7] Microsoft Corporation. XInput API documentation. https://msdn.microsoft.com/en-us/library/windows/desktop/ee417003(v=vs.85).aspx.

[8] Linux input documentation - input/input.txt. https://github.com/torvalds/linux/blob/master/Documentation/input/input.txt.

[9] Linux joystick API documentation - input/joystick-api.txt. https://github.com/torvalds/linux/blob/master/Documentation/input/joystick-api.txt.

[10] Mark Patrick, George Sachs. X11 Input Extension Library Specification. http://refspecs.linux-foundation.org/X11/Xinput.pdf.

[11] Kivy homepage. https://kivy.org/.

[12] MGSim branch with our modifications on Github. https://github.com/Fleppensteyn/mgsim.

[13] GitHub repository with examples. https://github.com/Fleppensteyn/joyinput-examples.

Appendix A

Event structure overview

A.1 Common event structure

These event structures, enumerations, constants, and masks are contained in the header `MGInputEvents.h` available on GitHub [12].

```
struct MGCommonInputEvent
{
    uint32_t timestamp;
    uint8_t type;
};
```

All events share these fields. The timestamp is copied from the SDL event and based on SDLs cycle count. The type is the main indicator for the event structure and can be used to determine what parts of the event need to be copied. Type can take the the values outlined in the enumeration below.

```
enum eventtypes
{
    MG_JOYAXISMOTION = 0x01,
    MG_JOYBUTTON,
    MG_JOYHATMOTION,
    MG_JOYBALLMOTION,
    MG_DEVICEATTACHED,
    MG_DEVICEREMOVED,
    MG_MOUSEMOTION = 0x11,
    MG_MOUSEBUTTON,
    MG_MOUSEWHEEL,
    MG_TOUCHDOWN = 0x21,
    MG_TOUCHMOTION,
    MG_TOUCHUP
};
```

The actual event type can be determined by looking at the value in the most significant 4 bits of the type. As their names suggest, `0x0X` means `MGJoyInputEvent`, `0x1X` means `MGMouseInputEvent`, and `0x2X` means `MGTouchInputEvent`. These events will be explained in more detail on the following pages.

A.2 Joystick events

```
typedef struct MGJoyInputEvent
{
    uint32_t timestamp;
    uint8_t type;
    uint8_t num;
    int16_t value;
    int16_t value2;
    int16_t padding;
} MGJoyInputEvent;

#define MG_HAT_UP        0x01
#define MG_HAT_RIGHT     0x02
#define MG_HAT_DOWN      0x04
#define MG_HAT_LEFT      0x08
```

Joystick events usually only require reading the second 4-byte block to get all the relevant data, the only event type that uses `value2` is ball motion. These fields and their function should be clear, with only three extra fields next to the common ones.

The `num` field relates to the index of the part that triggered the event, this index starts at zero and is a direct copy of the SDL assigned index.

The `value` field contains the absolute axis value, a binary button state, a 4-bit hat state, or the relative x movement for a ball, with the `value2` field containing the relative y movement for a ball. The direction represented in the hat state can be deciphered using binary and operations and the masks we provided.

If, for example, the fifth button of a device is pressed the resulting event would have:
`type = 1, num = 4, value = 1`

A.3 Mouse events

```
typedef struct MGMouseInputEvent
{
    uint32_t timestamp;
    uint8_t type;
    uint8_t num;
    uint8_t state;
    uint8_t clicks;
    int16_t xpos;
    int16_t ypos;
    int16_t xrel;
    int16_t yrel;
} MGMouseInputEvent;

#define MG_BUTTON_LEFT      0
#define MG_BUTTON_MIDDLE    1
#define MG_BUTTON_RIGHT     2
#define MG_BUTTON_X1        3
#define MG_BUTTON_X2        4
#define MG_BUTTONMASK(X)    (1 << (X))
```

Mouse events contain a lot more information and are more expensive to transmit, but if we were to convert every mouse event to multiple joystick events that effectively convey the same information we would have to transmit much more data.

These events are very straight translations of the SDL events they are based on.

The `num` field indicates what button is being pressed in the case of a button event. To determine which button is which we provide some helpful constants. We started numbering from zero instead of one unlike SDL.

In the `state` field we store the button state. In the case of button event this is simply 0 or 1. In the case of mouse motion events it contains an unsigned integer which encodes the state of all mouse buttons. This is similar to the way we normally represent button states so there is a bit for every button. To make masking easier there is a macro included to generate one for a given button.

The `clicks` field is only relevant in mouse button events where it indicates how many times the user has clicked consecutively, like double and triple clicks and beyond. This information is only available through events and is not mapped to any part of the state.

The `xpos` and `ypos` fields contain the absolute position of the pointer. This is the click position for button events and the stopping position for motion events. This value is relative to the window.

The `xrel` and `yrel` fields contain relative movement. In the case of mouse motion this is the difference between start and end positions and for scroll events these indicate the scroll direction.

If, for example, the mouse was moved from $100, 100$ to $120, 80$ while holding both left and right mouse buttons it would result in the following event data:

```
type = 0x11, state = 5, xpos = 120, ypos = 80, xrel = 20, yrel = -20
```

A.4 Touch events

```
typedef struct MGTouchInputEvent
{
    uint32_t timestamp;
    uint8_t type;
    uint8_t num;
    uint8_t device;
    uint8_t pad;
    int16_t xpos;
    int16_t ypos;
    int16_t xrel;
    int16_t yrel;
    int16_t pressure;
    int16_t pad2;
} MGTouchInputEvent;
```

Touch events are unique in the sense that all fields are filled for all the types of touch event. Their fields are basically copied from the SDL touch events, with the main difference being that all the floats have been converted to signed fixed point with all bits apart from the sign dedicated to the fractional part.

When using multitouch every finger is assigned an index as long as it is on the screen, this is contained in the `num` field.

The `device` field refers to which touch input device it originated from. Due to our lack of touch screens to test, it has been left in as we were not certain about its precise use and it was not taking up any additional space.

The `xpos`, `ypos`, `xrel`, and `yrel` fields are all fixed point numbers representing the same data as the mouse fields of the same name. The main difference is that the position is mapped on a number between 0.0 and 1.0, and the relative movement is mapped between -1.0 and 1.0.

The last field for `pressure` is fixed point mapped between 0.0 and 1.0, but many platforms do not actually provide this information to SDL.

The code we use for float to fixed conversion is shown in the listing below.

```
static short convertToFixed(float f)
{
    if (f > 0.9999999) f = 0.9999999;
    else if (f < -0.9999999) f = -0.9999999;
    return (short)(f * 32768.0);
}
```

Appendix B

Code for performance measurement

This is an excerpt of code for our tests in 5.1. The code we show is just the part we timed for our results. We unrolled copying to improve efficiency over loops. This code is slightly slower compared to manually optimised assembly, but this is an illustration of what normal users can expect when using a compiler.

We start by describing common variables before moving to copying code.

```
MGJoyInputEvent mgev; //Data storage for copied events
struct js_event jsev; //Linux joystick API event for storage
```

The code for UART copying with comments showing variations between the two tests.

```
uint8_t *bb = (void*)&jsev; //Buffer for testing Linux API events
uint8_t *bb = (void*)&mgev; //Buffer for testing MGJoyInputEvents
volatile uint8_t     *uart; //UART base address

if (uart[5] & 1){           //Timing starts when we receive uart[5]
  bb[0] = uart[0];          //Copy a byte from the UART
  bb[1] = uart[0];          //Copy the next byte
  ...
  bb[7] = uart[0];          //Last byte for Linux API events
  bb[9] = uart[0];          //Last byte for MGJoyInputEvents
}
```

The code for JoyInput copying with a comment indicating the chunk for partial events.

```
uint32_t *qb = (void*)&mgev; //buffer pointer for JoyInput
volatile uint8_t    *joydev; //JoyInput base address
volatile uint32_t    *joyev; //JoyInput event section (at 0x200)

if (joydev[4]){              //Timing starts when we receive joydev[4]
  qb[0] = joydevev[0];
  qb[1] = joydevev[1];       //The only chunk we copy for partial events
  qb[2] = joydevev[2];
  joydev[4] = 1;             //Pop the event queue
}
```

Appendix C

Example program source code

Also available as `joyinputviz.c` on our example repository on GitHub [13].

```c
#include <svp/testoutput.h>
#include <svp/mgsim.h>
#include <stdint.h>
#include <stddef.h>

#include "mtconf.h"
#include "MGInputEvents.h"

typedef struct joystickdata //Local joystick info and state
{
  uint8_t naxes;
  uint8_t nbuttons;
  uint8_t nhats;
  uint8_t nballs;
  uint32_t buttons;//Bitset for button state
  int16_t axes[8];//Should cover all reasonable joysticks
  uint8_t hats[8];//Never seen more than one on a device
  int16_t balls[16];//Enough for 8 balls
} joystickdata;

typedef struct joydevctl //simplifies access to control registers
{
  uint8_t enabled;
  uint8_t events;
  uint8_t notifications;
  uint8_t channel;
  uint8_t queuesize;
} joydevctl;

typedef struct joydevinfo //simplifies access to device info
{
  uint32_t axes;
  uint32_t buttons;
  uint32_t hats;
  uint32_t balls;
} joydevinfo;

typedef struct drawcmd //Helps with draw commands on the GPU
{
  uint32_t cmd;
  uint32_t mode;
  uint32_t offset;
  uint32_t scanlen;
  uint32_t size;
  uint32_t pos;
  uint32_t dsize;
} drawcmd;

int main(void){
  sys_detect_devs();
  sys_conf_init();

  //Set up helpful pointers
  volatile joydevctl *joydev = (void*)mg_devinfo.base_addrs[mg_joyinput_devids[0]];
  volatile uint8_t *joydevbase = (void*)mg_devinfo.base_addrs[mg_joyinput_devids[0]];
  volatile joydevinfo *joyinfo = (void*)&joydevbase[0x100];
  volatile uint32_t *evdata = (void*)&joydevbase[0x200];
  volatile long *notif = (void*)mg_devinfo.channels;
  joydev->enabled = 1;
  joydev->events = 1;
  notif[0] = 1; //enable channel
  joydev->channel = 0;
  joydev->notifications = 1;
  int i, cmdi = 0; //two counters

  joystickdata js; //Joystick info and state
  MGJoyInputEvent ev;//To store an event
  //Gather information about the joystick
  js.naxes = joyinfo->axes >> 24; //shifting to get the axes count
  js.nbuttons = joyinfo->buttons >> 24;
  js.nhats = joyinfo->hats >> 24;
  js.nballs = joyinfo->balls >> 24;

  //Make sure we can actually draw this joysticks state
  //These limits should not be a problem for most existing joysticks
  if (js.naxes > 8 || js.nbuttons > 32 || js.nhats > 8){
    output_string("Joystick layout breaks visualisation limits\n",1);
    return 0; //Exit the program instead of littering the code with if statements
  }

  //gather data on initial state of axes
  volatile int16_t *axesdata = (void*)&joydevbase[0x400];
  for (i = 0; i < js.naxes; i++)
    js.axes[i] = axesdata[i];

  //gather data on initial state of buttons (always 0 on Linux)
  volatile uint8_t *buttondata = (void*)&joydevbase[0x800];
  for (i = 0; i <= (js.nbuttons - 1) >> 3; i++)
    js.buttons += buttondata[i] << (i * 8);

  //gather data on initial state of hats (always 0 on Linux)
  volatile uint8_t *hatsdata = (void*)&joydevbase[0xc00];
  for (i = 0; i < js.nhats; i++)
    js.hats[i] = hatsdata[i];

  //initialise the graphics device
  mg_gfx_ctl[1] = 640;
  mg_gfx_ctl[2] = 400;
  mg_gfx_ctl[0] = 1;
  mg_gfx_ctl[3] = 5;//Start the command buffer after the palette and button texture
  volatile uint32_t *gfxcmd = (uint32_t*)mg_gfx_fb + 5; //command buffer

  //We put all the colours we use at the start
  volatile uint32_t *palette = (uint32_t*)mg_gfx_fb;
  palette[0] = 0x00000000U;//black
  palette[1] = 0x00ffffffU;//white
  palette[2] = 0x00550000U;//red
  palette[3] = 0x0000aa00U;//green

  //First command clears the screen to black
  gfxcmd[cmdi++] = 0x206;
  gfxcmd[cmdi++] = 0x20;//32 bit colour
  gfxcmd[cmdi++] = 0; //texture starts at 0
  gfxcmd[cmdi++] = 1;
  gfxcmd[cmdi++] = 1 << 16 | 1; //texture size is 1x1
  gfxcmd[cmdi++] = 0; //position is 0,0
  gfxcmd[cmdi++] = 640 << 16 | 400;//Stretch 1 pixel to screen size

  gfxcmd[cmdi++] = 0x102;//set a palette of red/green
  gfxcmd[cmdi++] = 2;//of size 2
  gfxcmd[cmdi++] = 2;//starting at offset 2

  //Location to store button data as a texture
  volatile uint32_t *gfx_buttondata = (uint32_t*)mg_gfx_fb + 4;
  *gfx_buttondata = js.buttons; //store the initial button state

  volatile drawcmd *buttoncmd = (drawcmd*)&gfxcmd[cmdi];
  buttoncmd->cmd = 0x206;
  buttoncmd->mode = 0x10001;//use 1 bit red/green palette
  buttoncmd->offset = 4; //Location of button state
  buttoncmd->scanlen = 32; //32 bits per line
  buttoncmd->size = js.nbuttons << 16 | 1; //only show existing buttons
  buttoncmd->pos = 0; //start at the origin
  buttoncmd->dsize = 640 << 16 | 20;//Stretch the bar to screen width


  //Set up locations of commands for axes/hats
  volatile drawcmd *axescmd = (drawcmd*)&gfxcmd[cmdi+7];
  volatile drawcmd *hatscmd = &axescmd[js.naxes];

  //Axes stretch a single pixel into a white bar
  for (i = 0; i < js.naxes; i++){
    axescmd[i].cmd = 0x206;
    axescmd[i].mode = 0x20;//32-bit colour
    axescmd[i].offset = 1;//Location of white value
    axescmd[i].scanlen = 1;
    axescmd[i].size = 1 << 16 | 1;
    axescmd[i].pos = ((i * 80) + 20) << 16 | 40;
    axescmd[i].dsize = 40 << 16 | (js.axes[i] + 32768) >> 8;
  }

  //Hats show their state with a white block
  //We initialise them centred for now
  for (i = 0; i < js.nhats; i++){
    hatscmd[i].cmd = 0x206;
    hatscmd[i].mode = 0x20;
    hatscmd[i].offset = 1;
    hatscmd[i].scanlen = 1;
    hatscmd[i].size = 1 << 16 | 1;
    hatscmd[i].pos = ((i * 80) + 30) << 16 | 330;
    hatscmd[i].dsize = 20 << 16 | 20;
  }
  mg_gfx_ctl[0] = 1;

  int loop = 1;
  uint32_t *evbuff = (void *)&ev; //To use as a buffer for event data
  while (loop){
    notif[0]; //wait for events
    while (joydev->queuesize){
      evbuff[0] = evdata[0];
      evbuff[1] = evdata[1];//Technically this is the only part we need
      evbuff[2] = evdata[2];
      joydev->queuesize = 1;//pop the queue
      if (ev.type == MG_JOYBUTTON && ev.num == 0 && ev.value == 0){
        loop = 0;
        break;//exit the loop if we release button 0
      }
      switch (ev.type){
        case MG_JOYAXISMOTION://Axis value is mapped to [0,255] for display
          js.axes[ev.num] = ev.value;
          axescmd[ev.num].dsize = 40 << 16 | (ev.value + 32768) >> 8;
          break;
        case MG_JOYHATMOTION:
          js.hats[ev.num] = ev.value;
          if (ev.value == 0){//hat in origin
            hatscmd[ev.num].pos = (ev.num * 80) + 30 << 16 | 330;
          } else {//Hat state is decoded to position the square
            int x = 1, y = 1;
            if (ev.value & MG_HAT_UP)
              y = 0;
            else if (ev.value & MG_HAT_DOWN)
              y = 2;
            if (ev.value & MG_HAT_LEFT)
              x = 0;
            else if (ev.value & MG_HAT_RIGHT)
              x = 2;
            hatscmd[ev.num].pos = (ev.num * 80) + 10 + (x * 20) << 16 | 310 + (y * 20);
          }
          break;
        case MG_JOYBUTTON://Button state is updated using masking
          if (ev.value == 0)
```

```
            js.buttons &= ~(1L << ev.num);
          else
            js.buttons |= (1L << ev.num);
          *gfx_buttondata = js.buttons; //Copy new data to framebuffer
          break;
        case MG_JOYBALLMOTION://No ball visualisation
        default:
          break;
      }
    }
  }
  joydev->notifications = 0;
  joydev->enabled = 0;
  return 0;
}
```