OpenAV Ctlra:
A Library for Tight Integration of Controllers

Harry VAN HAAREN
OpenAV
Bohatch,
Mountshannon,
Co Clare, Ireland.
harryhaaren@gmail.com

Abstract

Ctlra is a library designed to encourage integration of hardware and software. The library abstracts events from the hardware controller, emitting generic events which can be mapped to functionality exposed by the software.

The generic events provide a powerful method to allow developers and users integrate hardware and software, however a good development workflow is vital to users while tailoring mappings to their unique needs.

This paper proposes an implementation to enable a fast scripting-like development workflow utilizing on-the-fly recompilation of C code for integrating hardware and software in the Ctlra environment.

Keywords

Controllers, Hardware, Software, Integration.

1 Introduction

Ctlra aims to enable easy integration between DAWs and controllers. At OpenAV we believe that enabling hardware controllers to be 1st class citizens in controlling music software will provide the best on-stage workflow possible.

Ctlra has been developed due to lack of a simple C library that affords interacting with a range of controllers in a generic but direct way, that enables tight integration.

1.1 Existing Projects

Although many projects exist to enable hardware access, very few aim to provide a generic interface for applications to use.

Projects such as maschine.rs[Light, 2016], HDJD[Pickett, 2017], OpenKinect[OpenKinect-Community, 2017] and CWiid[Smith, 2007] all enable hardware access, however they each expose a a unique API to the application, resulting in the need to explicitly support each controller.

The o.io[Freed, 2014] project aims to unify communications for various types of interaction using an OSC API, which is similar to the generic events concept. Discoverability and familiarity with the implementation presented possible issues, so Ctlra is designed as a simple C API that will be instantly familiar to seasoned developers.

Hence, Ctlra is implemented as a C library that provides generic events to the application, regardless of the hardware in use.

1.2 Modern Controllers

Each year there are new, more powerful and complex hardware controllers, often with large numbers of input controls, and lots of feedback using LEDs etc. The latest generations have seen an uptake in high-resolution screens built into the hardware.

The capabilities of these devices require an equally powerful method to control the hardware, or risk not utilizing them to the full potential. As such, any library to interface with these controllers should afford handling these complex and powerful controller devices easily.

1.3 Why a Controller library?

Although every application could implement its own device-handling mechanism, there are significant downsides to this approach.

Firstly, a developer will not have access to all controllers that are available, so only a subset of the controllers will have tight integration with their software. As an end result, the users controller may not be directly supported by the application.

Secondly, duplication of effort is significant, both in the development and testing of the controller support. This is particularly true if a device supports multiple layers of controls.

Thirdly, advanced controller support features like hotplug and supporting multiple devices of the same type must also be tested - requiring both access to multiple hardware units and time.

The Ctlra library shares the effort required to develop support for these powerful devices, pro-

viding users and developers with an easy API to communicate with the hardware.

1.4 Tight integration

The terms "tight integration" or "deep integration" are often used to describe hardware and software that collaborate closely together, perhaps they are even specifically designed to suite one other.

Tight integration leads to better workflows for on-stage usage of software, as it allows operations from inside the software to be controlled by the hardware device and appropriate feedback returned to the user.

The advantage of tight integration is providing a more powerful way of integrating the physical device and the software. As an example, many DAWs support MIDI Control Change (CC) messages, and allow changing a parameter with it. Although such a 1:1 mapping is useful, most workflows require more flexibility. For example, each physical control could effect a number of parameters with weighting applied to provide a more dynamic performance.

1.5 Controller Mapping

The Ctlra library allows mappings to be created between physical controls and the target software. DAWs could expose this functionality for technical users - giving them full control over the software.

Given the variation in live-performances and on-stage workflows, there is no ideal mapping from a device to the application - it depends on the user. As a result, OpenAV is of the opinion that enabling users to create custom mappings from controllers to software using a generic event as a medium to do so is the best approach.

1.6 Scripting APIs

Various audio applications provide APIs to allow users script functionality for their controller. Enabling users to script themselves requires technical skill from the user, however it seems like there is no viable alternative.

The solution proposed in section 4 also proposes "crowd-sourcing" the effort in writing controller mappings to the users themselves, as they have access to the physical device and have knowledge of their ideal workflow.

Examples of audio applications that provide scripting APIs are Ardour[Davis, 2017], Mixxx[Mixxx, 2017] and Bitwig Studio[Bitwig, 2017]. Although Ableton Live[Ableton, 2017] doesn't officially expose a scripting API, the are members of the community that have investigated and successfully written scripts to control it[Petrov, 2017].

A brief review shows high-level scripting languages are favoured over compiled languages. Mixxx and Bitwig are both using JavaScript, while Ableton Live uses Python, and Ardour uses the Lua language.

These solutions are all valid and workable, however they do require that the application developer to exposes a binding API to glue the scripting API to the core of the application.

With the exception of Lua, none of the above scripting languages provide real-time safety unless very carefully programmed - which should not be expected of user's scripts.

OpenAV feels that providing controller support in the native language of the application ensures that all operations that the application is capable of are also mappable to a controller. Other advantages of having the controller mappings in the native language of the application is that they can be compiled into the application itself.

2 Ctlra Implementation

This section details the design decisions made during the implementation of the Ctlra library. The core concepts like the context, device and events are introduced.

2.1 Ctlra Context

The main part of the Ctlra library is the context, it contains all the state of that particular instance of the Ctlra library. This state is represented by a `ctlra_t` in the code. Using a state structure ensures that Ctlra is usable from inside a plugin, for example an LV2 plugin.

Devices and metadata used by Ctlra are stored internally in the `ctlra_t`. The end goal is to enable multiple `ctlra_t` instances to exist in the same process without interfering with one-another. This is more difficult than it sounds as not all backends provide support for context style usage.

2.2 Generic Events

Ctlra is built around the concept of a generic event. The generic event is a C structure `ctlra_event_t` which may contain any of the available event types. The available event types include all common hardware controller interaction types, such as `BUTTON`, `ENCODER`, `SLIDER` and `GRID`. The events are

prefixed by CTLRA_EVENT_, so BUTTON becomes CTLRA_EVENT_BUTTON.

Once the type of the event is established, the contents of the event can be decoded. The generic event has a union around all events, so an event must represent one and only one type of event. It is expected that the application will use a switch() statement to decode the event types, and process them further.

The power of generic events is shown by the examples/daemon sample application, which translates *any* Ctlra supported device into an ALSA MIDI transmitting device.

2.2.1 Button

The button event represents physical buttons on a hardware device. It contains two variables, id and pressed. The button id is guaranteed to be a unique identifier for this device, based from 0, and counting to the maximum number of buttons. The pressed variable is a boolean value set high when the button is pressed by the user.

2.2.2 Slider

The slider event represents physical controls that have a range of values, but the interaction is of limited range, eg: faders on a mixing desk. The slider has and id as a unique identifier for the slider, and floating-point value that represents the position of the control. The value variable range is normalized as a linear value from 0.f to 1.f to allow generic usage of the event.

2.2.3 Encoder

The encoder represents an endless rotary control on a hardware device. There are two types of encoders, which we will refer to as "stepped" and "continuous". Stepped controls have notches providing distinct steps of movement, while the continuous type is smooth and provides no physical feedback during rotation.

The stepped controls notify the application for each notch moved by setting the ENCODER_FLAG_INT, and the delta change is available from delta. Similarly the ENCODER_FLAG_FLOAT tells the application to read the delta_float value, and interpret the value as a continuous control.

2.2.4 Grid

The grid represents a set of controls that are logically grouped together, eg: the squares of the Push2 controller. The grid event type contains multiple variables: id, flags, pos, pressure and pressed.

The id identifies the grid number, allowing controllers with more than one grid to distinguish between them. The flags allows the event to identify which values are valid in this event. Currently two flags are defined, BUTTON and PRESSURE, there are 14 bits remaining for future expansion.

If the flag GRID_FLAG_BUTTON is set, the pressed variable is valid to read, and represents if the button is currently pressed or not. The BUTTON flag should only be set in the device backend if the state of the grid-square has changed, this eases handling events in the application. When GRID_FLAG_PRESSURE is set, the floating-point pressure variable may be read, The pressure value is normalized to the range 0.f to 1.0f.

2.3 Devices

In Ctlra, any physical controller is represented internally in by a ctlra_dev_t. Devices do not appear available to the application directly, but instead operations on the device are performed through the ctlra_t context. There is an abstracted representation of a device at the API level, which the application has access to in the event_handle() callback.

The reason that the device is not exposed to the application directly is that ownership and cleanup of resources becomes blurred when hotplug functionality is introduced. Using the ctlra_t context as a proxy for multiple devices not only simplifies the application handling of controllers, but actually helps define stronger memory ownership rules too. See section 2.4 for hotplug implementation details.

2.3.1 Device Backends

A device backend is how the software driver connects to the physical device.

The implementation of the driver calls a read() function, which indicates the driver wishes to receive data. The backend library will send an async read to the physical device, and return immediately. Upon completion of the transaction a callback in the driver is called which decodes the newly received data, and can emit events to the application if required. To write data to the device, a write() is provided.

Note that a single device driver may open multiple backends, or utilize multiple connections of the same backend in order to fully support the capabilities of the hardware. An example could be a USB controller that exposes both a USB interrupt endpoint for buttons *and*

a USB bulk endpoint for sending data to a high-resolution screen.

Note that more backends can be added to support more devices if it is required in future.

2.4 Hotplug Implementation

Implementing a hotplug feature is difficult; it requires handling device additions and removals in the library itself, as well as a method to communicate any changes of environment with the application.

As Ctlra is a new library built from the ground up, hotplug was a consideration from the start as a required feature. As such, the API has been influenced by and designed for hotplug capabilities. The concept of a `ctlra_t` context that contains devices was introduced to allow transparent adding of devices without blurring memory ownership rules.

Hotplug of USB devices is enabled by LibUSB, which provides a hotplug callback, when a hotplug callback is registered and hotplug is supported on the platform. The USB hotplug callback is utilized to call the `accept_device()` callback in the application, providing details of the controller. The info provided allows the application to present the user with a choice of accepting or rejecting the controller, and if accepted, it will be added to the `ctlra_t` context.

3 Application Usage of Ctlra

This section will introduce the reader to the steps required to integrate Ctlra into an application. Refer to the `examples/simple/simple.c` sample to see a minimal program in action.

The following steps summarize Ctlra usage:

1. `ctlra_create()`
2. `ctlra_probe()`
   - Accept controller in callback
3. `ctlra_iter()`
   - Handle events in callback
4. `ctlra_exit()`

This creates a single `ctlra_t` context, probes and accepts any supported controller. The accepted controllers are connected to the particular context that it was probed from.

Calling `ctlra_iter()` causes the event to be polled and the application is given a chance to send feedback to the device. Finally, `ctlra_exit()` releases any resources and gracefully closes the context.

3.1 Interaction

The main interaction between Ctlra and the application happens in two functions. Events from the device are handled in the application provided `event_handle()` function, while feedback can be sent to a device from the `feedback_func()`.

These functions are callback functions, and they are invoked for each device when the application calls `ctlra_iter()`.

To understand the events passed between the device and the application, please review the generic events (Section 2.2), and browse the `examples/` directory.

3.2 Controller's View of State

Each application has its own way of representing its state. Similarly, each controller has its own capabilities in terms of controls and feedback to the user. Given the specific application state and capabilities of the hardware, it is useful to create a struct specifically for storing the view that the controller has of the application.

Note that the controller view should be tracked *per instance* of the controller, as users may have multiple identical controllers. This controller's instance of the struct is very useful for remapping the controls to provide an alternate map when a "shift" key is held down. As the struct depends on the application and device, this problem can not be solved elegantly at the library layer.

Ctlra provides a userdata pointer for each instance which can be purposed for to point to the state struct. If the application's state must be accessed from the state-struct, a "back-pointer" to the application elegantly provides that.

The memory for the state struct can be allocated in the `accept_device()` callback from Ctlra, and the memory can be released in when the device is disconnected using the `remove_device()` callback.

4 Device Scripting in C

This section describes a solution to providing a fast and interactive development workflow for scripting mappings between software and device using the C language.

C is typically a compiled and static language, not one that comes to mind when discussing dynamic and scripting type workflows. Although generally accurate, C can be used as a dynamic language with certain compromises. The following section details how applications can imple-

ment a C scripting workflow for users to quickly develop "Ctlra scripts".

4.1 Dynamic Compilation

Dynamically compiling C at runtime can be achieved by bundling a small, lightweight C compiler with your application. This may sound a little crazy, but there are very small and lightweight C compilers available designed for this type of usage. The "Tiny C Compiler", or TCC[Bellard, 2017] project is used to enable compiling C code at runtime of the application.

Please note that the security of dynamically compiling code is not being considered here as the goal is to enable user-scripted controller mappings for musical performance. If security is a concern, the reader is encouraged to find a different solution.

4.2 TCC and Function Pointers

The TCC API has various functions to create a compilation context, set includes, and add files for compilation. Once initialized, TCC takes an ordinary `.c` source file, and compiles it.

When compilations completes successfully, TCC allows requesting functions from the script by name, returning a function pointer.

The returned function pointer may be called by the host application, forming the method of communicating with the compiled script.

4.3 The Illusion of Scripting

To provide the illusion that the code is a script, the application can check the modified time of a script file, and recompile the file if needed. By swapping in the new function pointers, the update code runs. The old program can then be freed, cleaning up the resources that were consumed by the now outdated script.

The `examples/tcc_scripting/` directory contains a minimal example showing how the event handling for any Ctlra supported device can be dynamically scripted.

Providing this workflow requires some extra integration from the application, however the time pays off easily in developer time saved when time save in scripting support for each controller is considered.

4.4 C and C++ APIs

Note that TCC is a C compiler only - explicitly not a C++ compiler. This has some impact on how scripts can interact with applications, as many large open-source audio projects are written in C++. The solution is to provide wrapper functions to C, if the hosts language is C++.

Often real-time software uses message-passing in plain C structs through ringbuffers. This is a good way to communicate between dynamically compiled scripts and the host, as it provides a native C API, as well as a method to achieve thread-safe message passing.

5 Case Study: Ctlra and Mixxx

This section briefly describes the work performed to integrate Ctlra with the open-source Mixxx DJ software. It is presented here to showcase how to integrate the Ctlra library in an existing project.

5.1 Implementation

This section details the steps taken to integrate the Ctlra library in Mixxx to test Ctlra in the real-world.

5.1.1 Class Structure

Mixxx has a very object oriented design, utilizing C++ classes to abstract behaviour of control devices and managers of those control devices. The `ControllerManager` class aggregates the different types of `ControllerEnumerator` classes, which in turn add `Controller` class instances to the list of active controllers. Ctlra has been integrated as a `ControllerEnumerator` sub-class for this proof-of-concept implementation, really it should be integrated at the `ControllerManager` level.

5.1.2 Threading in the Mixxx Engine

The Mixxx engine currently creates many threads. This design is supported by the use of an "atomic database" of values (see next Section 5.1.3). Given this design, the Ctlra integration is done by spawning a Ctlra handling thread, which performs any polling and interacting with Ctlra supported devices.

5.1.3 Communicating with the Engine

The Mixxx engine is composed of values, which can be controlled from any thread anywhere in the code. These values are represented in the code by `ControlObject` and `ControlProxy` classes. A `ControlObject` is the equivalent to owning a value, while the `ControlProxy` allows atomic access to update the value. Lookup of these values is performed using "group" and "key" strings. The strings are constant allowing Ctlra and the Mixxx engine to understand the meaning of each value represented by a particular `ControlProxy`.

5.1.4 Mixxx's C++ API

An issue arises due to Mixxx having a ControlProxy being a C++ class which is not possible to access from a TCC compiled script (refer to C and C++ APIs, Section 4.4).

The solution is to create a C wrapper function, which simply provides a C API to the desired C++ function to be called on a `ControlProxy` instance. This provides the power of the Mixxx engine to the dynamically compiled script code:

```
void mixxx_config_key_set(
        consl char *group,
        const char *key,
        float value);
```

5.2 Mixxx and Hotplug

Since Ctlra hides the hotplug functionality from the application due to the design of the `accept_device()` callback, Mixxx supports on-the-fly plug-in and plug-out transparently.

This is achieved by the Ctlra library having its own thread to poll events (see Section 5.1.2), and handling the connect or disconnect events. The Mixxx application code did not have to be modified to support hotplugging of controllers in any way (beyond adding basic Cltra support).

5.3 Scripting Controller Support

With the Ctlra library integrated in Mixxx, users are now able to script the tight integration of the Ctlra supported hardware and Mixxx. The next sections demonstrate simple mappings from a device to Mixxx and vice-versa.

5.3.1 Event Input to Mixxx

When a user presses a physical control on a device, the action is presented to the application as an event. The user can map these events to the application in a variety of ways, in order to suit their own requirements on how they wish to control the software application.

For example, the following snippet shows how we can bind slider ID 10 to channel 1 volume in Mixxx (note the usage of the C function from Section 5.1.4):

```
case CTLRA_EVENT_SLIDER:
  switch(e->slider.id) {
  case 10:
    mixxx_config_key_set(
        ''[Channel1]'',
        ''volume'',
        e->slider.value);
    break;
```

5.3.2 Mixxx Feedback to Device

The reverse of the previous paragraph is to send Mixxx state to the physical device, providing feedback to the user. Each parameter that Mixxx exposes via the `ControlProxy` is available for reading as well as writing. The allows the script to query the state of a particular variable from Mixxx, and update the state of an LED on the device, using the Ctlra encoding for colour and brightness:

```
int play;
play = mixxx_config_key_get(
          ''[Channel1]'',
          ''play_indicator'');

led = play > 0 ? 0xffffffff : 0;

ctlra_dev_light_set(dev,
          DEVICE_LED_PLAY,
          led);
```

6 Future Work

To make Ctlra a ubiquitous library for event I/O is a huge task, however the benefit to all applications if such a library did exist would be huge too.

Imagine easily scripting your DIY controller to easily control any aspect of any software - huge potential for customized powerful user-experience. OpenAV intends to use the Ctlra library and integrate it with any projects that would benefit from a powerful customizable workflow.

6.1 Device Support

At time of writing, the Ctlra library supports 6 advanced USB HID devices, one USB DMX device, a generic MIDI backend, and plans are in place to support a common bluetooth console controller - but more must be added to make the Ctlra library really useful!

An interesting angle may be so that DIY platforms like Arduino can be used to build controllers that use a generic Ctlra backend, allowing controllers to be auto-supported.

The previously mentioned hardware enabling projects that provide access to specific hardware devices could be integrated with Ctlra, transparently benefiting applications that use Ctlra.

The number of supported hardware devices is paramount to the success of the Ctlra library, so OpenAV welcomes patches or pull-requests that add support for a device.

6.2 Software Environments

From the software point-of-view there is huge potential for integrating into existing software.

For example mapping Ctlra events to LV2 Atoms would expose the Ctlra backends to any LV2 Atom capable host.

Integration with DSP languages like FAUST or PD may prove interesting and allow for faster prototyping and more powerful control over performance using those tools.

Hardware platforms like the MOD Duo[MOD, 2017] could use the Ctlra library to enable musicians to use a wider variety of controllers in thier on-stage setups in conjunction with the DSP on the DUO.

7 Conclusion

This paper presents Ctlra, a library that allows an application to interface with a range of controllers in a powerful and customizable way.

It shows how applications and devices can interact by using generic events. A case study showcases integrating Ctlra with the open-source Mixxx project as a proof of concept.

To enable a fast development workflow for creating mappings between applications and devices, a method to dynamically compile C code is introduced. This enables developers and users to write mappings between devices and applications as if C was a scripting language, but provides native access to the applications data structures.

Ctlra is available from github here[OpenAV, 2017], please run the sample programs in the `examples/` directory of the source to experience the power of Ctlra yourself.

8 Acknowledgements

OpenAV would like to acknowledge the linux-audio community and open-source ecosystem as a whole for providing novel solutions to various problems and being a great place to collaborate and innovate. For the work on Ctlra certain people and projects provided lots of inspiration and support, thanks!

Thanks to the TCC project, which allows dynamically compiling Ctlra scipts, it is awesome to script in C!

Thanks to William Light for writing `maschine.rs`, David Robillard for the creation of PUGL[Robillard, 2017], the Mixxx project devs (particular shout outs to `be_`, `Pegasus_RPG` and `rryan` on `#mixxx` on irc.freenode.net.

References

Ableton. 2017. Music production with live and push — ableton. https://www.ableton.com.

Fabrice Bellard. 2017. Tcc : Tiny c compiler. https://http://www.bellard.org/tcc/.

Bitwig. 2017. Bitwig music productiona nd performance system for windows, macos and linux. https://www.bitwig.com.

Paul Davis. 2017. Ardour: Record, edit, and mix on linux, os x and windows. http://ardour.org/.

Adrian Freed. 2014. o.io: a unified communications framework for music, intermedia and cloud interaction. *International Computer Music Conference (ICMC) 2014*.

William Light. 2016. Maschine.rs, open-source ni maschine device handling. https://github.com/wrl/maschine.rs.

Mixxx. 2017. Mixxx dj software, dj your way. for free. https://mixxx.org/.

MOD. 2017. Mod duo, the definitive stompbox. https://moddevices.com/pages/mod-duo.

OpenAV. 2017. Ctlra is a library providing support for controllers, designed to integrate hardware and software. https://github.com/openAVproductions/openAV-Ctlra.

OpenKinect-Community. 2017. Open source libraries that will enable the kinect to be used with windows, linux, and mac. https://openkinect.org.

Hanz Petrov. 2017. Introduction to the ableton framework classes. http://remotescripts.blogspot.com/2010/03/introduction-to-framework-classes.html.

Neale Pickett. 2017. Hercules dj controller driver for linux. https://github.com/nealey/hdjd.

David Robillard. 2017. Pugl is a minimal portable api for opengl guis. https://drobilla.net/software/pugl.

Donnie Smith. 2007. A collection of linux tools written in c for interfacing to the nintendo wiimote. http://abstrakraft.org/cwiid/.