# demo project

Select hardware device or on-screen keyboard. MIDI events received from either source displayed

A screenshot of a computer

Description automatically generated

**The application window**

# MIDI input

JUCE makes it easy to find connected hardware MIDI interfaces. Also provides MidiKeyboardComponent class that allows you to display on-screen keyboard. Look at member variables in MainContentComponent

juce::AudioDeviceManager deviceManager; // [1]

juce::ComboBox midiInputList; // [2]

juce::Label midiInputListLabel;

int lastInputIndex = 0; // [3]

bool isAddingFromMidiInput = false; // [4]

juce::MidiKeyboardState keyboardState; // [5]

juce::MidiKeyboardComponent keyboardComponent; // [6]

juce::TextEditor midiMessagesBox;

double startTime;

* [1]: We use the **[AudioDeviceManager](https://docs.juce.com/master/classAudioDeviceManager.html" \o "Manages the state of some audio and midi i/o devices.)** class to find which MIDI input devices are enabled.
* [2]: We display the names of the MIDI input devices in this combo-box for the user to select.
* [3]: Used to de-register a previously selected MIDI input when the user selects a different input.
* [4]: indicate if MIDI data arriving from external source, rather than mouse-click on-screen keyboard
* [5]: The **[MidiKeyboardState](https://docs.juce.com/master/classMidiKeyboardState.html" \o "Represents a piano keyboard, keeping track of which keys are currently pressed.)** class keeps track of which MIDI keys are currently held down.
* [6]: This is the on-screen keyboard component.

In the MainContentComponent constructor we initialise [3], [4], and [6]. We also take a note of the application start time so we can display the MIDI data timestamps relative to this.

MainContentComponent()

: keyboardComponent (keyboardState, [juce](https://docs.juce.com/master/namespacejuce.html)::[MidiKeyboardComponent](https://docs.juce.com/master/classMidiKeyboardComponent.html)::horizontalKeyboard),

startTime ([juce](https://docs.juce.com/master/namespacejuce.html)::[Time](https://docs.juce.com/master/classTime.html)::getMillisecondCounterHiRes() \* 0.001) {

We must pass a **[MidiKeyboardState](https://docs.juce.com/master/classMidiKeyboardState.html" \o "Represents a piano keyboard, keeping track of which keys are currently pressed.)** object to initialise the **[MidiKeyboardComponent](https://docs.juce.com/master/classMidiKeyboardComponent.html" \o "A component that displays a piano keyboard, whose notes can be clicked on.)** object. And, since these are statically allocated objects the **[MidiKeyboardState](https://docs.juce.com/master/classMidiKeyboardState.html" \o "Represents a piano keyboard, keeping track of which keys are currently pressed.)** must be listed first in our member variables.

## MIDI input list

Combo-box containing MIDI inputs list populated using MidiInput::getDevices() function:

addAndMakeVisible (midiInputList);

midiInputList.setTextWhenNoChoicesAvailable ("No MIDI Inputs Enabled");

auto midiInputs = juce::MidiInput::getAvailableDevices();

juce::StringArray midiInputNames;

for (auto input : midiInputs)

midiInputNames.add (input.name);

midiInputList.addItemList (midiInputNames, 1);

midiInputList.onChange = [this] { setMidiInput(midiInputList.getSelectedItemIndex());};

// find the first enabled device and use that by default

for (auto input : midiInputs) {

if (deviceManager.isMidiInputDeviceEnabled (input.identifier)) {

setMidiInput (midiInputs.indexOf (input));

break;

}

}

// if no enabled devices were found just use the first one in the list

if (midiInputList.getSelectedId() == 0) setMidiInput (0);

If user changes selected MIDI input, call lambda function assigned to [**ComboBox::onChange**](https://docs.juce.com/master/classComboBox.html#a9cf2e20990541b9fbb539cd4a8e0ac4e) helper object

midiInputList.onChange = [this] {setMidiInput (midiInputList.getSelectedItemIndex());};

setMidiInput() makes our application start listening to selected device. It also enables device if disabled

void setMidiInput (int [index](https://docs.juce.com/master/group__juce__gui__basics-detail.html#ga32b76ccceab9e1bba3e8c4ef2947d1d8)) {

auto [list](https://docs.juce.com/master/group__juce__gui__basics-accessibility.html#ggaf1349587ab6cac7e882cd63e1976682ca10ae9fc7d453b0dd525d0edf2ede7961) = juce::MidiInput::getAvailableDevices();

deviceManager.removeMidiInputDeviceCallback([list](https://docs.juce.com/master/group__juce__gui__basics-accessibility.html#ggaf1349587ab6cac7e882cd63e1976682ca10ae9fc7d453b0dd525d0edf2ede7961)[lastInputIndex].identifier, this);

auto newInput = [list](https://docs.juce.com/master/group__juce__gui__basics-accessibility.html#ggaf1349587ab6cac7e882cd63e1976682ca10ae9fc7d453b0dd525d0edf2ede7961)[[index](https://docs.juce.com/master/group__juce__gui__basics-detail.html#ga32b76ccceab9e1bba3e8c4ef2947d1d8)];

if (! deviceManager.isMidiInputDeviceEnabled (newInput.identifier))

deviceManager.setMidiInputDeviceEnabled (newInput.identifier, true);

deviceManager.addMidiInputDeviceCallback (newInput.identifier, this);

midiInputList.setSelectedId ([index](https://docs.juce.com/master/group__juce__gui__basics-detail.html#ga32b76ccceab9e1bba3e8c4ef2947d1d8) + 1, juce::dontSendNotification);

lastInputIndex = [index](https://docs.juce.com/master/group__juce__gui__basics-detail.html#ga32b76ccceab9e1bba3e8c4ef2947d1d8);

}

## Handling external MIDI input

We implement **[MidiInputCallback::handleIncomingMidiMessage()](https://docs.juce.com/master/classMidiInputCallback.html" \l "acdf6d7e27b96a50fe06922737f0d082e" \o "Receives an incoming message.)** [pure virtual](http://www.learncpp.com/cpp-tutorial/126-pure-virtual-functions-abstract-base-classes-and-interface-classes/) function. This updates the keyboard state (which in turn will update the **[MidiKeyboardComponent](https://docs.juce.com/master/classMidiKeyboardComponent.html" \o "A component that displays a piano keyboard, whose notes can be clicked on.)** object):

void handleIncomingMidiMessage (juce::MidiInput\* source, const juce::MidiMessage& message) override {

const juce::ScopedValueSetter<bool> scopedInputFlag (isAddingFromMidiInput, true);

keyboardState.processNextMidiEvent (message);

postMessageToList (message, source->getName());

}

Notice scopedInputFlag variable makes use of the **[ScopedValueSetter](https://docs.juce.com/master/classScopedValueSetter.html" \o "Helper class providing an RAII-based mechanism for temporarily setting and then re-setting a value.)** class. This does the following:

* It stores current state of isAddingFromMidiInput member.
* It sets isAddingFromMidiInput member to true.
* When function exits it reset value of isAddingFromMidiInput  to state it was in at function start

## The MIDI keyboard state and component

In MainContentComponent constructor [**MidiKeyboardComponent**](https://docs.juce.com/master/classMidiKeyboardComponent.html) object added to parent component and made visible. Also listen to [**MidiKeyboardState**](https://docs.juce.com/master/classMidiKeyboardState.html) object (not the component):

addAndMakeVisible (keyboardComponent);

keyboardState.addListener (this);

MidiKeyboardStateListener class has two pure virtual functions that we must implement, MidiKeyboardStateListener::handleNoteOn() and MidiKeyboardStateListener::handleNoteOff()

void handleNoteOn(juce::MidiKeyboardState\*,int midiChannel, int midiNoteNumber, float velocity) override {

if (! isAddingFromMidiInput) {

auto m = juce::MidiMessage::noteOn (midiChannel, midiNoteNumber, velocity);

m.setTimeStamp (juce::Time::getMillisecondCounterHiRes() \* 0.001);

postMessageToList (m, "On-Screen Keyboard");

}

}

void handleNoteOff(juce::MidiKeyboardState\*, int midiChannel, int midiNoteNumber, float /\*velocity\*/) override {

if (! isAddingFromMidiInput) {

auto [m](https://docs.juce.com/master/namespacegl.html#a0927a7b4232e61481fa501d12209467e) = juce::MidiMessage::noteOff (midiChannel, midiNoteNumber);

[m](https://docs.juce.com/master/namespacegl.html#a0927a7b4232e61481fa501d12209467e).setTimeStamp (juce::Time::getMillisecondCounterHiRes() \* 0.001);

postMessageToList (m, "On-Screen Keyboard");

}

}

See how isAddingFromMidiInput member is used. This prevents events that arrived from hardware input from being posted to our list more than once.

## Posting messages to the list

The postMessageToList() function may look a little unusual at first:

void postMessageToList (const juce::MidiMessage& message, const juce::String& source) {

(new IncomingMessageCallback (this, message, source))->post();

}

 IncomingMessageCallback class is subclass of **[CallbackMessage](https://docs.juce.com/master/classCallbackMessage.html" \o "A message that invokes a callback method when it gets delivered.)** class. We need to use this since we can't be sure from which thread postMessageToList() function will be called. It will be called from message thread if user clicks on **[MidiKeyboardComponent](https://docs.juce.com/master/classMidiKeyboardComponent.html" \o "A component that displays a piano keyboard, whose notes can be clicked on.)** object. But, if data arrives from external MIDI source then it will be called from the background MIDI thread (possibly an operating system thread).

CallbackMessage class provides means of calling function on message thread. Its a kind of ReferenceCountedObject class. We don't need to store IncomingMessageCallback object anywhere. IncomingMessageCallback::post() function (which is MessageManager::MessageBase::post() ) adds object to queue handled by MessageManager class. MessageManager class finds this object in queue & calls IncomingMessageCallback::messageCallback() on message thread. Once this has been called, the IncomingMessageCallback object will be deleted. So lifetime of this object is handled automatically.

Only needed since we send data to message thread. likely that some inter-thread communication is necessary in a MIDI application but the exact implementation depends on the circumstances.

## Displaying the messages

addMessageToList() and getMidiMessageDescription() are similar to these functions from Tutorial: Create MIDI data, but we make note of MIDI message source [7] (which hardware input, or on-screen keyboard)

void addMessageToList (const juce::MidiMessage& message, const juce::String& source){

auto time = message.getTimeStamp() - startTime;

auto hours = ((int) (time / 3600.0)) % 24;

auto minutes = ((int) (time / 60.0)) % 60;

auto seconds = ((int) time) % 60;

auto millis = ((int) (time \* 1000.0)) % 1000;

auto timecode = juce::String::formatted ("%02d:%02d:%02d.%03d",

hours,minutes,seconds,millis);

auto description = getMidiMessageDescription (message);

juce::String midiMessageString (timecode + " - " + description + " (" + source + ")"); // [7]

logMessage (midiMessageString);

}

Add sliders to UI that transmit and respond to messages such as modulation wheel (CC1) and pitch wheel