INScore Web

Version 1.33

D. Fober

GRAME
Centre national de création musicale

Contents

1 Introduction

Since version 1.27, the INScore engine is available as Javascript libraries:

- a WebAssembly [WASM] library providing all the services of the abstract INScore model,
- a Javascript library providing an HTML view of the INScore model.

The web environment provides a very different runtime context than a native application: it is much more modular; due to the absence of a 'concrete machine' a number of INScore primitives do not make sense in a web environment; finally it provides new rendering capabilities with CSS.

This document is intended to present the differences between the native and web versions of INScore. A special section is also devoted to the implementation of INScore Web in standalone HTML pages.

2 Main differences

The OSC protocol is not supported in the Web version. As a result, the mode of communication with the INScore engine is different (see section 8) and may also depend on the application that uses this engine.

By default:

- the OSC output and error ports are redirected to the Javascript console.
- drag & drop works like in the native version: you can drop files or text to an INScore scene (an HTML div in the Web version).

The `log` window (address `/ITL/log`) is dependent on the host application. By default, input messages addressed to the `log` node are directed to the Javascript console.

2.1 Optional components

As the architecture of the web version is completely modular, the available objects depend on the host application: e.g. a page which wants to use objects in symbolic notation (type `gmn`) will have to include the Guido library. This architecture allows applications to be optimized to fit their needs. It also facilitates the extension of the INScore engine. The table 1 presents the current supported components.

Component	Name	Dependent types
Guido Engine	libGUIDOEngine.js	`gmn`, `gmnf`, `gmnstream`, `pianoroll`, `pianorollf`, `pianorollstream`
MusicXML library	libmusicxml.js	`musicxml`, `musicxmlf` use of MusicXML implies to have also the Guido Engine
Faust compiler	libfaust-wasm.js FaustLibrary.js	`faust`, `faustf`

Table 1: Components required by specific objects

2.2 Using files in a script

You can use file based objects in an INScore script but the file path is interpreted differently:

- when using an absolute path, it refers to the document root of the HTTP server,
- when using a relative path, it refers to the location of the HTML page

NOTE

Browsers infer a MIME type from the file extension and generally, download any file which extension is not recognized (this behavior depends on the browser you are using). It is therefore recommended to use a `.txt` extension for any textual resource with non-standard extension. For example, a `score.gmn` file could be renamed and used as `score.gmn.txt`.

3 Unsupported

3.1 Unsupported objects

The table 2 presents the objects that are not supported:

Type	Comment
`fileWatcher`	unsupported
`httpd`	unsupported server
`websocket`	unsupported server
Faust plugins	deprecated and redesigned (see section 7.2)
Gesture follower	unsupported

Table 2: Unsupported objects

The following components are not yet implemented:

- graphic signals objects: `graph`, `fastgraph`, `radialgraph`
- Pianoroll stream: `pianorollstream`
- Misc.: `grid`
- Memory image: `memimg`
- Sensors: `acceleromter`, `gyroscope`, `compass`, etc.

3.2 Unsupported messages

A number of messages are not supported in the web version, either because they do not make sense in the runtime context or because they cannot be implemented.

3.2.1 Common messages

- `export`, `exportAll`: not implemented
- `shear`, `dshear`: not implemented
- `effect`: `colorize`: not implemented
- `edit`: not implemented

3.2.2 Application messages

- `quit`: do not make sense in the web environment
- `rootPath`: not implemented
- `mouse`: not yet implemented
- `read`: not implemented
- `port`, `outport`, `errport`: do not make sense without OSC

3.2.3 Application log window

As mentioned above, the `log` window (address `/ITL/log`) is dependent on the host application. By default, input messages addressed to the `log` node are directed to the Javascript console.

- `clear`: dependent on the host application (do nothing by default)
- `save`: not supported
- `foreground`: dependent on the host application (do nothing by default)
- `wrap`: dependent on the host application (do nothing by default)
- `scale`: dependent on the host application (do nothing by default)
- `write`: dependent on the host application (write to the Javascript console by default)

3.2.4 Scene messages

- `foreground`: dependent on the host application (do nothing by default)
- `frameless` `windowOpacity`: not supported

3.2.5 Type specific messages

- `brushstyle`: not yet implemented
- Piano roll messages: not yet implemented
- Video `get` messages: not yet implemented
- SVG `animate`: not yet implemented
- Arc `close`: not yet implemented
- Line `arrows`: not yet implemented
- `debug` node: not yet implemented

Symbolic score:

- `columns` `rows`: not supported
- `pageFormat`: not yet implemented
- `get`: `pageCount` `systemCount`: not yet implemented

3.2.6 Synchronization

The following synchronisation modes are not yet supported:

- Stretch modes: `h`, `hv`

3.2.7 Events

The following events are not yet supported:

- Touch events: `touchBegin`, `touchEnd`, `touchUpdate`
- Url events: `success`, `error`, `cancel`
- Score event: `pageCount`
- `export`: not supported since the `export` message is not supported
- `endPaint`: not supported

4 Behavioral changes

Some messages behave differently with the Web version.

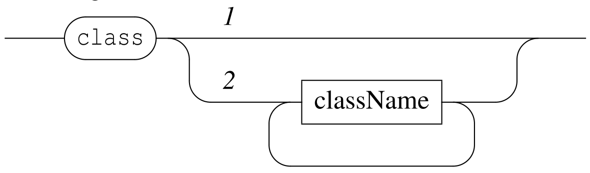
- `save`: download the INScore content to the file given as argument. You should find this file in your download folder.

5 Specific new messages

5.1 Leveraging CSS

Cascading Style Sheets [CSS] are a powerful way to control the appearance of elements on a web page. The Web version of INScore provides a specific `class` message to use CSS in parallel to the standard mechanisms. This message is supported by all the INScore objects.

classMsg



- 1: without argument, remove all class settings
- 2: set the CSS classes of an object

NOTE

If a `class` message has not the expected effect, it's likely because the CSS properties of the target object are set by the standard INScore mechanisms (like color, border, font-size, etc.). As the own attributes of an object have precedence over the class of the object, the properties of the class are then ignored. You can force the properties of the class by adding the CSS rule `!important` which will override all previous styling rules for that specific property.

6 MIDI support

A `midi` node is automatically created at application level (`/ITL/midi`). Before using the MIDI features, the `midi` node must be explicitly initialized using the `init` message. On success, the `ready` event is triggered, otherwise an `error` event is triggered.

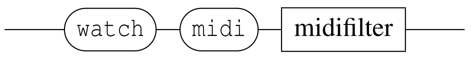
EXAMPLE

```
/ITL/midi watch ready ( do something to use the MIDI interface );
/ITL/midi watch error ( do something else );
/ITL/midi init;
```

6.1 MIDI events

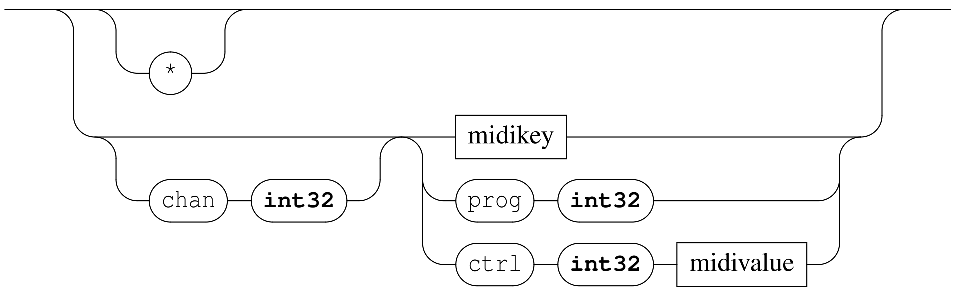
MIDI is supported using a specific event that you can configure using the `watch` message. Before using MIDI, you should check that your browser is supporting the Web MIDI API.

watchMIDI



A filter is used to select the MIDI messages that will trigger the event.

midifilter



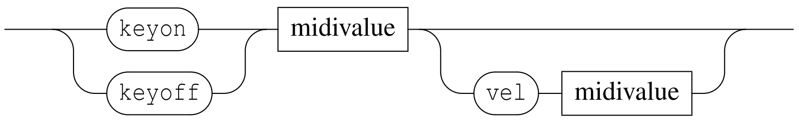
- an empty filter can be used to stop watching MIDI input
- '*' denotes any MIDI message (no filter at all)
- `chan`: an optional channel number may be used to select only the MIDI messages on a given MIDI channel
- midikey: is used to select key on/off messages according to their values and to an optional velocity.
- `prog`: is used to select program change messages. A program change number is expected as next argument.

- `ctrl`: is used to select control change messages. A control change number is expected as next argument.

EXAMPLE
Accept all MIDI messages that are on channel 0

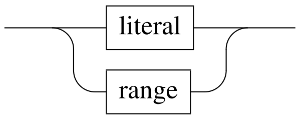
```
/ITL/scene/obj watch midi chan 0 (inscore messages list);
```

midikey

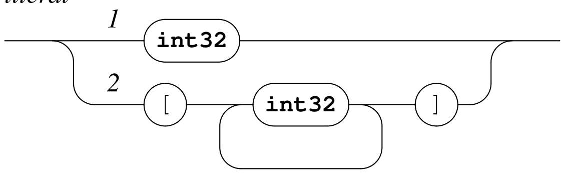


A MIDI key is either `keyon` or `keyoff`. It may be followed by an optional velocity selector.
A MIDI value is either literal values or a range of values.

midivalue



literal



Literal values are:

- 1: a single value,
- 2: a list of space separated values enclosed in brackets,

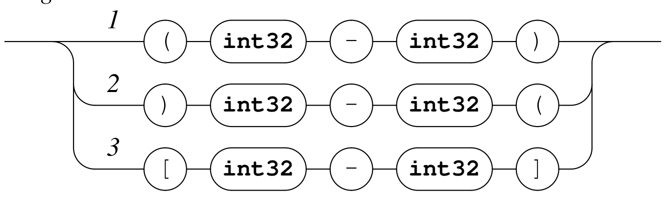
All the values must be in the MIDI data range (0-127).

EXAMPLE
Accepting MIDI key on messages for 3 specific pitches.

```
/ITL/scene/obj watch midi keyon [60 62 64] (inscore messages list);
```

Range may be used when a value is within the specified range, or enters or leaves the range.

range



- 1: trigger the event when the value enters the range.
- 2: trigger the event when the value leaves the range.
- 3: trigger the event when the value is within the range.

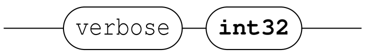
EXAMPLE
Accepting `keyon` MIDI messages only when entering and leaving the range 60 - 67;

```
/ITL/scene/obj watch midi keyon '[60-67]' (inscore messages list);
/ITL/scene/obj watch midi keyon ']60-67[' (inscore messages list);
```

6.2 MIDI verbose mode

In order to facilitate the detection of messages sent by a MIDI interface, a ’verbose’ mode allows incoming messages to be displayed on the browser console. To enable or disable verbose mode, a specific message must be sent to the `/ITL/midi` object:

verbMIDI



The parameter is a numerical value that controls which MIDI events are displayed:

- 0: disable the verbose mode
- 1: displays the channel MIDI events (Real-time and system exclusive events are filtered out).
- 2: displays all the MIDI events

EXAMPLE

```
/ITL/midi verbose 1;
```

Activates the MIDI verbose mode.

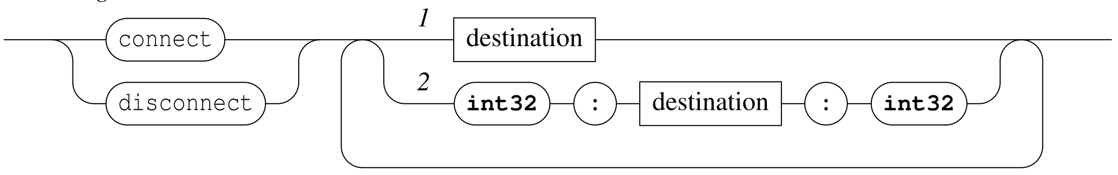
7 Audio

7.1 Audio objects

Audio objects are *connectable* objects i.e. objects which output can be connected to the input of another audio object. In the INScore model, the following objects are audio objects: `audio`, `video`, `faust` (see section 7.2) and `audioio` (see section 7.3).

Audio objects support the following messages:

audioMsgs



- `connect`: connect the outputs of the object to the destination inputs. `destination` must be another audio object. Without numeric prefix and suffix [1], the connexion may be subject of up or down mixing as specified by the Web Audio API. The destination supports regular expressions. The second form [2] connects an output channel to the a destination input channel. The first number refers to the target object channel, the second one to the destination channel. Channels are indexed from 1, using 0 is equivalent to all channels (that are then mixed together). See the examples below.
- `disconnect`: disconnect the outputs of the object from the destination inputs. Note that errors (e.g. no existing connection with the destination) are silently ignored.

EXAMPLE

Connect the first channel of an audio object to the second channel of the audio output :

```
/ITL/scene/obj connect '1:audioOutput:2';
```

Connect all channels of an audio object to the first channel of the audio output :

```
/ITL/scene/obj connect '0:audioOutput:1';
```

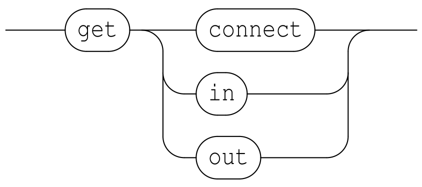
NOTE

The messages `/ITL/scene/obj connect '0:dest:0';` is equivalent to `/ITL/scene/obj connect dest;`

NOTE

The `connect` message assumes that the source and destination are located in the same hierarchy (i.e. they have the same parent).

audiogetMsgs



- `in`: gives the number of inputs of the audio object
- `out`: gives the number of outputs of the audio object

7.2 Faust objects

Faust is a functional programming language for sound synthesis and audio processing. Faust objects are available, providing that the Faust library has been loaded.

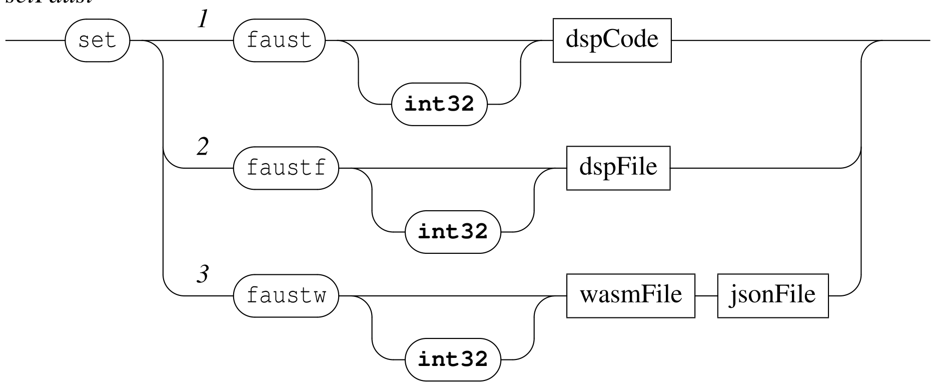
WARNING
A page containing Faust objects require an https connection, unless it runs on localhost.

7.2.1 Creating a Faust object

The `faust` or `faustf` types must be used to create a Faust object.

NOTE
The `faust` type exists with the native version but to load a pre-compiled DSP. `faustdsp` and `faustdspf` types are not supported.

setFaust



The expected arguments of the `set` message are:

- an optional integer that indicates a number of voices used to create a polyphonic DSP (see section 7.2.4). Note that when present, a polyphonic DSP is created even if equal to 1.
- 1: Faust DSP code (see the Faust language for more information).
- 2: a Faust DSP file.
- 3: a Faust pre-compiled wasm file and the associated json file. Note that theses files can be generated using the `get wasm` message (see section 7.2.2 p.7).

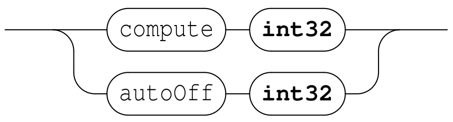
By default, a Faust DSP appears as a browsable block diagram.

NOTE
The Faust language uses characters that have a special meaning in HTML: for example, with the split operator `<:`, the '<' character will be interpreted as an opening HTML tag and you should use HTML escapes (e.g. `&lt;` instead of <). This is necessary for inline DSP code only, DSP files are not concerned.

7.2.2 Faust messages

Faust objects are active by default, i.e. the output signals is computed whatever the audio connection state. However, it is possible to disable the signals computation when it is not needed (e.g. for objects that are only temporarily active), which allows a large number of faust objects to be embedded in a score, while saving a significant CPU. The `compute` message is provided in this intend.

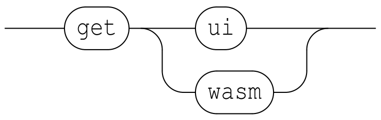
faustMsgs



- `compute`: start or stop computing the audio signals. The parameter is a boolean value. Default value is 1.
- `autoOff`: automatically switch buttons Faust UI elements back to off. The parameter is a boolean value. Default value is 0.

Faust objects support the audio objects messages plus additional query messages:

faustgetMsgs



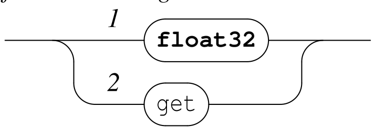
- `ui`: gives the Faust processor parameters (see section 7.2.3) i.e. for each parameter: its OSC address followed by the parameter UI type, a label, the minimum and maximum values and the parameter step.
- `wasm`: generates the Faust dsp as a precompiled wasm file with the associated json file. The Faust object name is used for the files name. On output, the files `xxx.wasm` and `xxx.json` should be present in the local download folder (where `xxx` is the Faust object name).

7.2.3 Faust objects parameters

A Faust DSP code can declare UI elements that are used by *architecture files* to build controllers providing users with dynamic control of the DSP parameters. In INScore, DSP UI elements are used to extend the Faust object address space. For example, when a DSP code declares a UI element named 'Volume', a Faust object which address is `/ITL/scene/dsp` is extended as `/ITL/scene/dsp/Volume`.

Faust parameters support two types of messages:

faustParamMsgs

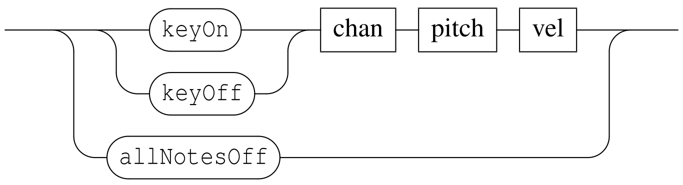


- 1: set the parameter value
- 2: gives the parameter value

7.2.4 Polyphonic objects

Polyphonic objects (i.e. Faust objects created using the optional voice number) support additional messages:

faustPolyMsgs

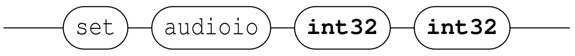


- `keyOn` and `keyOff` messages take 3 integer parameters: a MIDI channel, the note pitch and the velocity.
- `allNotesOff`: similar to MIDI all notes off message

7.3 Audio Input / Output

Audio input / output objects are provided as audio object (see section 7.1) to represent the physical audio inputs and outputs, in order to homogenize the connection process. An audio input / output type is `audioio`. It is created with a number of inputs and outputs as arguments.

audioioMsgs



NOTE

The current Javascript implementation automatically creates an audio input objet named `audioInput` and an audio output object named `audioOutput`. Thus the names `audioInput` and `audioOutput` are reserved and you must not use them. `audioOutput` and `audioInput` are created on audio request only, i.e. when an audio object is created (`audio`, `video` or `faust`).

NOTE ABOUT I/O OSC ADDRESSES

Since audio connections can only be made between objects that are in the same hierarchy, `audioOutput` and `audioInput` objects are created in all hierarchies that contain audio objects. For example, by creating an `faust` object in the `/ITL/scene/folder` hierarchy, audio I/O will be created at `/ITL/scene/folder/audioInput` and `/ITL/scene/folder/audioOutput`. Thus audio I/O objects may appear at different levels of a scene.

7.3.1 Audio Input

By default, the `audioInput` object is created with 0 inputs. It must be explicitly initialized using the `init` message. On success, the `ready` event is triggered and the number of input channels is set, otherwise an `error` event is triggered. This initialization is intended to avoid capturing the audio input when not used.

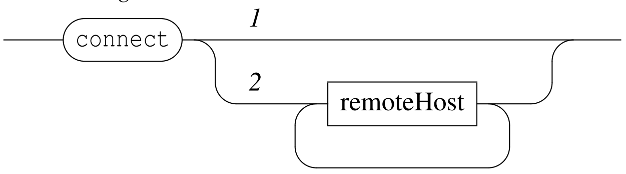
EXAMPLE

```
/ITL/scene/audioInput watch ready ( do something to use the audio input );
/ITL/scene/audioInput watch error ( do something else );
/ITL/scene/audioInput init;
```

8 Communication scheme

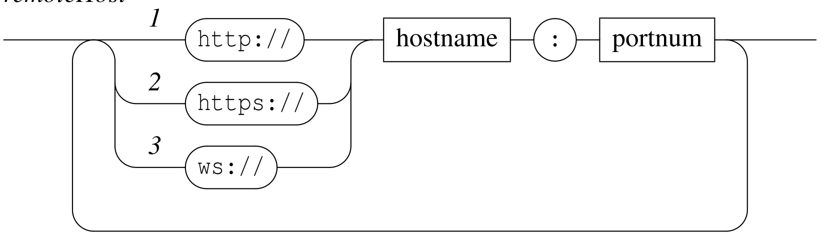
INScore forwarding mechanism supports `http` and `ws` protocols since version 1.27. Since these protocols are connection based, a counterpart `connect` message is provided with the Web version.

connectMsg



- 1) removes the existing set of connections,
- 2) connect to a list of remote hosts. Once the connection is established, the local INScore engine may receive messages from the remote hosts. To establish the connection, the remote hosts must have set a forwarding mechanism using the same protocol with the same port number.

remoteHost



- 1) uses `http` as communication protocol,
- 2) uses `https`,
- 3) uses `websockets`.

8.1 Web messages format

Messages emitted by the `http` and `ws` forwarding mechanism and received by connected clients are encoded in JSON and transmitted as base64 encoded packets. The JSON format is the following:

```
{
   'id' :  number ,
   'method' :  'post' ,
   'data' :  textual inscore messages
}
```

- `id`: is a packet unique identifier (currently unused)
- `method`: value must be 'post'
- `data`: must contain a valid inscore script

Although the native version of INScore supports this format for the `http`, `https` and `ws` protocols, nothing prohibits another application to control INScore web pages, provided that the format described above is respected.