INScore
OSC Messages Reference
v.1.17

D. Fober
GRAME
Centre national de création musicale
<fober@grame.fr>

February 9, 2016

INScore makes use of the following technologies:

The GUIDOEngine	http://guidolib.sf.net
The IRCAM Gesture Follower	http://imtr.ircam.fr/imtr/Gesture_Follower
The GRAME Faust Compiler	http://faust.grame.fr
The Qt5 cross-platform application and UI framework	http://qt-project.org/qt5

INScore research and development has been funded by the French National Research Agency [ANR] Interlude project [ANR- 08-CORD-010] and INEDIT project [ANR-12-CORD-0009].

Contents

Warning

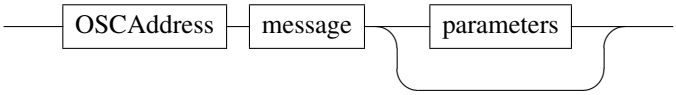
Throughout the documentation, all the sample code are given using scripting syntax i.e. that OSC messges are suffixed with a semi-colon ';'. This semi-colon is used as a message separator in INScore scripts and is not needed when sending messages over a network.

Chapter 1

General format

An OSC message is made of an OSC address, followed by a message string, followed by zero to n parameters. The message string could be viewed as the method name of the object identified by the OSC address. The OSC address could be string or a regular expression matching several objects.

OSCMessage



EXAMPLE

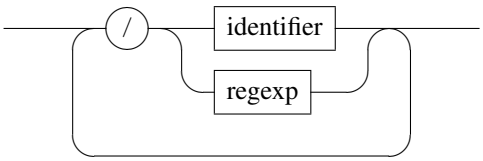
```
/ITL/scene/score x 0.5;
```

sends the message `x` to the object which address is `/ITL/scene/score` with `0.5` as parameter.

The address is similar to a Unix path and supports regular expressions as defined by the OSC specification (see at http://opensoundcontrol.org/). This address scheme is extended to address any host and applications (see section 15 p.61). Relative addresses have also been introduced for the scripting language (see section 16.2 p.69)

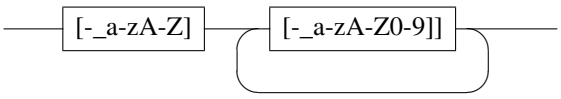
NOTE A valid legal OSC address always starts with `/ITL` that is the application address and that is also used as a discriminant for incoming messages.

OSCAddress



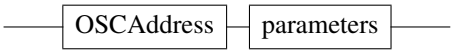
Identifiers may include letters, hyphen, underscore and numbers apart at first position (see lexical definition section 19.2 p.88).

identifier



Some specific nodes (like *signals* - see section 14.1.1) accept OSC messages without message string:

OSCMessage



1.1 Parameters

Message parameters types are the OSC types *int32*, *float32* and *OSC-string*. In the remainder of this document, they are used as terminal symbols, denoted by **`int32`**, **`float32`** and **`string`**.

When used in a script file (see section 16), **`string`** should be single or double quoted when they include characters not allowed in identifiers (space, punctuation marks, etc.). If an ambiguous double or single quote is part of the string, it must be escaped using a '\'.

Parameters types policy is relaxed: the system makes its best to convert a parameter to the expected type, which depend on the message string. With an incorrect type and when no conversion is applied, an incorrect parameter message is issued.

1.2 Address space

The OSC address space is made of static and dynamic nodes, hierarchically organized as in figure 1.1:

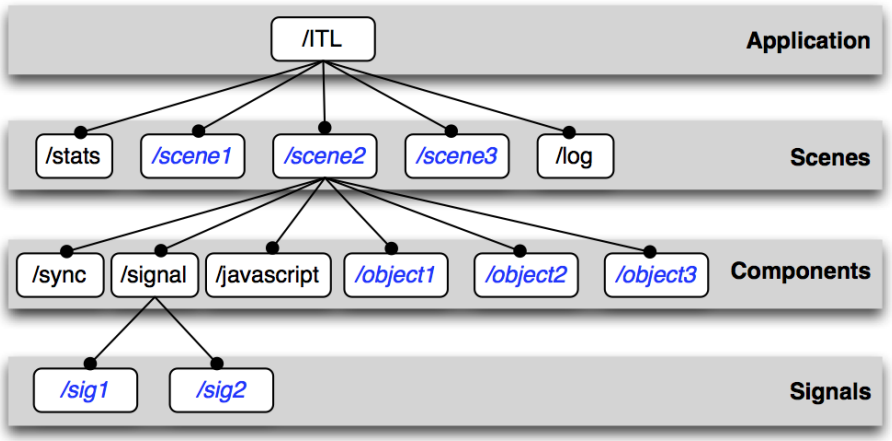


Figure 1.1: The OSC address space. Nodes in italic/blue are dynamic nodes.

OSC messages are accepted at any level of the hierarchy:

- **the application level** responds to messages for application management (udp ports management, loading files, query messages).

- **the scene level** contains *scores* that are associated to a window and respond to specific scene management messages. It includes a static node named `stats` that collects information about incoming messages, a static `log` node that control an embedded log window.
- **the component level** contains the score components and 3 static nodes:
  - a *signal* node that may be viewed as a folder containing signals
  - a *sync* node, in charge of the synchronization messages
  - a *javascript* node, that may be adressed to run javascript code dynamically.

  Each component includes a static node named `debug` that provides debugging information.

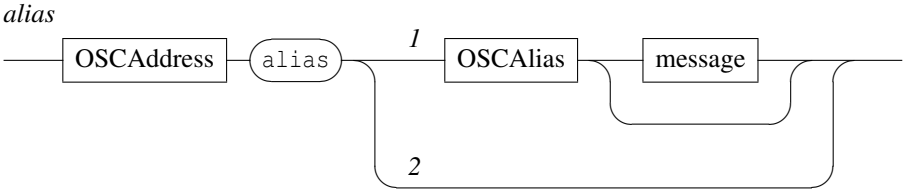
- **the signals level** contains signals i.e. objects that accept data streams and that may be graphically rendered as a scene component (see Signals and Graphic signals section 14 p.53).

NOTE Since version 1.05, each component of a score may also be a container and thus, the hierarchy described above has a potential infinite depth level. Note also that a `sync` node is present at each level.

1.3 Aliases

An alias mechanism allows an arbitrary OSC address to be used in place of a real address. An `alias` message is provided to describe aliases:

alias



- **[1]** sets `OSCAlias` as an alias of `OSCAddress`. The alias may be optionally followed by a message string which is then taken as an implied message i.e. the alias is translated to `OSCAddress message`.
- **[2]** removes `OSCAddress` aliases.

EXAMPLE

```
/ITL/scene/myobject alias '/1/fader1';
```

makes the object `myobject` addressable using the address `/1/fader1`.

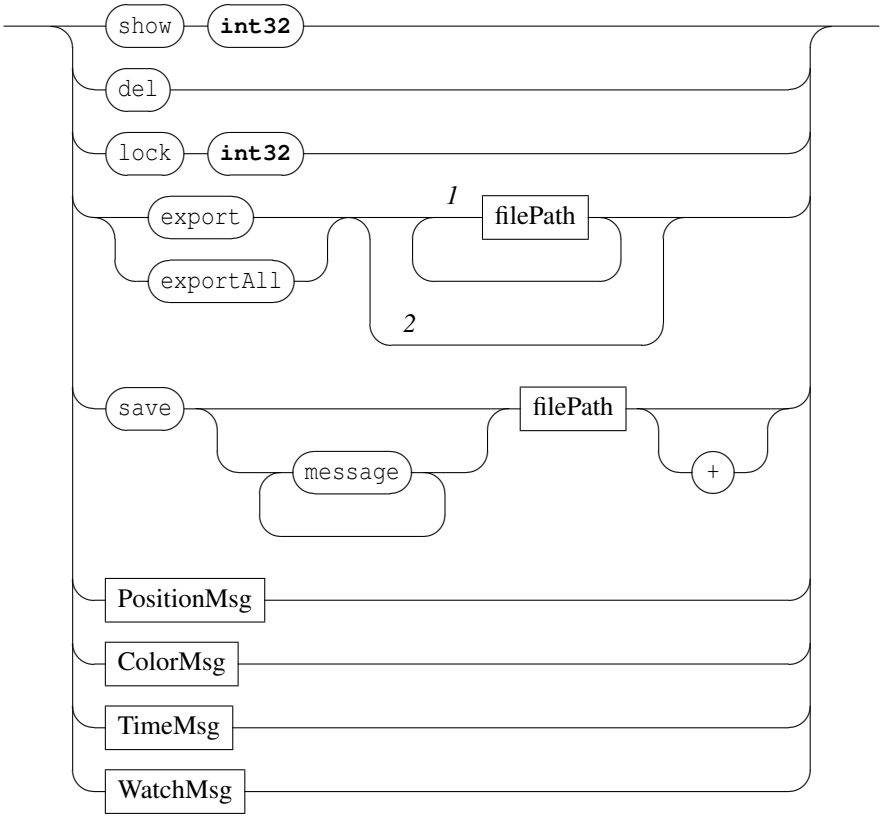
NOTE Regular expressions are not supported by the alias mechanism and could lead to unpredictable results.

Chapter 2

Common messages

Common messages are intended to control the graphic and the time space of the components of a scene. They could be sent to any address with the form `/ITL/`*`scene`* or `/ITL/`*`scene`*`/`*`identifier`* where *`identifier`* is the unique identifier of a scene component.

commonMsg



- `show`: shows or hides the destination object. The parameter is interpreted as a boolean value. Default value is `1`.
- `del`: deletes the destination object.
- `lock`: if not null, cancel any `del` message sent to this object. The object will still be deleted if its ancestors receive a `del` message. The parameter is interpreted as a boolean value. Default value is `0`.

- `export` and `exportAll`: exports an object to an image file respectively without or with its childrens. If the exported object is a scene, childrens are always exported.
  1) exports to the `filePath` name. The `filePath` can be relative or absolute. When the filename is not specified, exports to `path/`*`identifier`*`.pdf`. The file extension is used to infer the export format. Supported extensions and formats are: *pdf, bmp, gif, jpeg, png, pgm, ppm, tiff, xbm, xpm*.
  2) exports to `rootPath/`*`identifier`*`.pdf`.
  When the destination file is not completely specified (second form or missing extension), there is an automatic numbering of output names when the destination file already exists.

- `save`: recursively saves objects states to a file. When a `message` list is present, only the specified attributes are saved. The `filePath` can be relative or absolute. When relative, an absolute path is build using the current `rootPath` (see application or scene current paths p.34 and p.39). The optional `+` parameter indicates an append mode for the write operation. The message must be sent to the address `/ITL` to save the whole application state.
  Note: when a list of attributes is specified, unknown attributes are silently ignored.
  Note: the file extension for INScore files is `.inscore`. INScore files dropped on the application or on a window are interpreted as script files (see section 16 p.69).
- 'PositionMsg' are absolute and relative position messages.
- 'ColorMsg' are absolute and relative color control messages.
- 'TimeMsg' are time management messages. They are described in section 3 p.14.
- 'WatchMsg' are described in section 15 p.61.

EXAMPLE

Export of a scene to a given file as jpeg at the current root path:

```
/ITL/scene export 'myexport.jpg';
```

Saving a scene to `myScore.inscore` at the current root path, the second form saves only the `x`, `y` and `z` attributes, the third form uses the append mode:

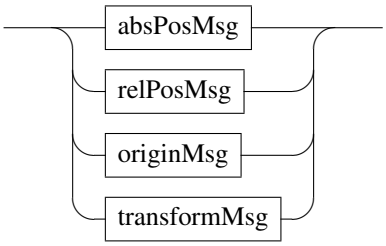
```
/ITL/scene save 'myScore.inscore';
/ITL/scene save x y z 'thePositions.inscore';
/ITL/scene save 'myScore.inscore' '+';
```

Hiding an object:

```
/ITL/scene/myObject show 0;
```

2.1 Positioning

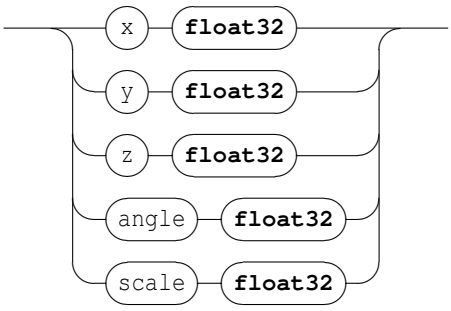
PositionMsg



Graphic position messages are absolute position or relative position messages. They can also control an object *origin* and transformations like rotation around an axis.

2.1.1 Absolute positioning

absPosMsg



- `x y`: moves the `x` or `y` coordinate of a component. By default, components are centered on their `x`, `y` coordinates. The coordinates space range is `[-1,1]`.
  For a `scene` component, -1 is the leftmost or topmost position, 1 is the rightmost or bottommost position. `[0,0]` represents the center of the `scene`.
  For the `scene` itself, it moves the window in the screen space and the coordinate space is orthonormal, based on the screen lowest dimension (*i.e.* with a 4:3 screen, `y=-1` and `y=1` are respectively the exact top and bottom of the screen, but neither `x=-1` nor `x=1` are the exact left and right of the screen).
  Default coordinates are `[0,0]`.
- `z`: sets the `z` order of a component. `z` order is actually relative to the `scene` components: objects of high `z` order will be drawn on top of components with a lower `z` order. Components sharing the same `z` order will be drawn in an undefined order, although the order will stay the same for as long as they live.
  Default `z` order is 0.
- `angle`: sets the `angle` value of a component, which is used to rotate it around its center. The angle is measured in clockwise degrees from the `x` axis.
  Default angle value is 0.
- `scale`: reduce/enlarge a component. Default scale is `1`.

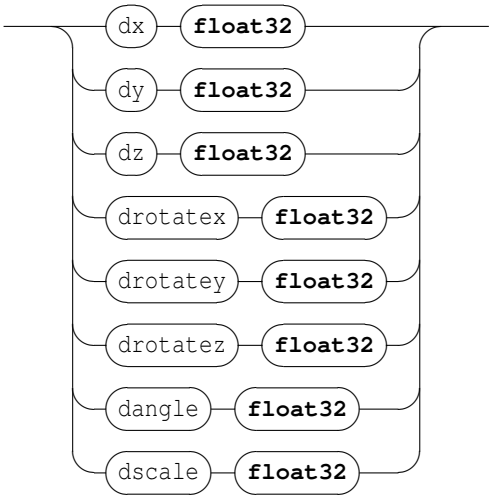
EXAMPLE

Moving and scaling an object:

```
/ITL/scene/myObject x -0.9;
/ITL/scene/myObject y 0.9;
/ITL/scene/myObject scale 2.0;
```

2.1.2 Relative positioning

relPosMsg



- `dx`, `dy`, `dz` messages are similar to `x`, `y`, `z` but the parameters represent a displacement relative to the current target value.
- `drotatex`, `drotatey`, `drotatez` are relative rotation messages. `dangle` is equivalent to `drotatez` and is maintained only for compatibility reasons.
- `dscale` is similar to `scale` but the parameters represents a scale multiplying factor.

EXAMPLE

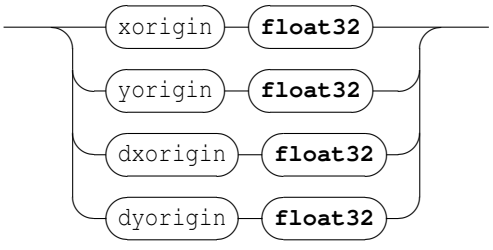
Relative displacement of an object:

```
/ITL/scene/myObject dx 0.1;
```

2.1.3 Components origin

The origin of a component is the point `(xo, yo)` such that the `(x, y)` coordinates and the `(xo, yo)` point coincide graphically. For example, when the origin is the top left corner, the component top left corner is drawn at the `(x, y)` coordinates.

originMsg



- `xorigin`, `yorigin` are relative to the component coordinates space i.e. `[-1,1]`, where -1 is the top or left border and 1 is the bottom or right border. The default origin is `[0,0]` i.e. the component is centered on its `(x,y)` coordinates.
- `dxorigin`, `dyorigin` represents displacement of the current `xorigin` or `yorigin`.

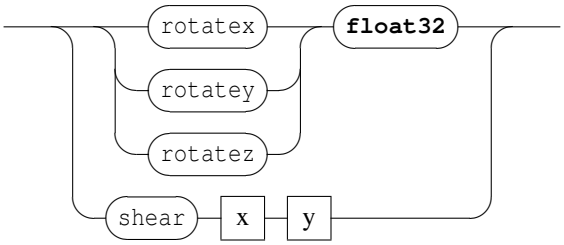
EXAMPLE
Setting an object graphic origin to the top left corner.

```
/ITL/scene/myObject xorigin -1.  ;
/ITL/scene/myObject yorigin -1.  ;
```

2.2 Components transformations

A component tranformation specifies 2D transformations of its coordinate system. It includes shear and object rotation.

transformMsg



- `rotatex rotatey rotatez`: rotates the component around the corresponding axis. Parameter value expresses the rotation in degrees.
- `shear` transforms the component in x and y dimensions. `x` and `y` are float values expressing the transformation value in the corresponding dimension.

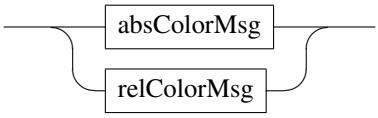
EXAMPLE
Rotating an object graphic on the `z` axis.

```
/ITL/scene/myObject rotatez 90.  ;
```

NOTE `angle` and `rotatez` are equivalent. `angle` has been introduced before the transformation messages and is maintained for compatibility reasons.

2.3 Color messages

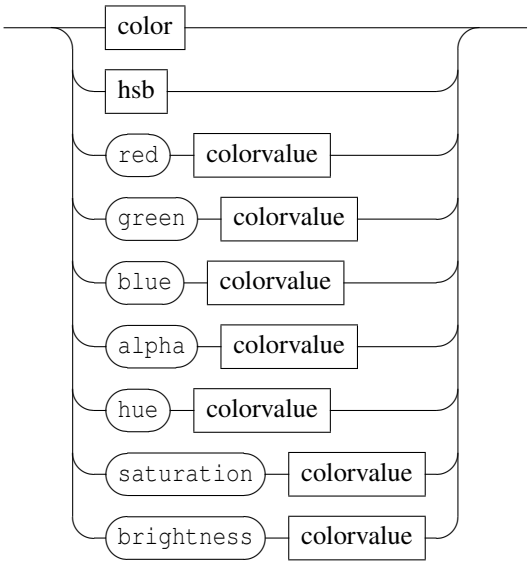
ColorMsg



Color messages are absolute or relative color control messages. Color may be expressed in RGBA or HSBA.

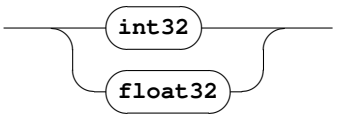
2.3.1 Absolute color messages

absColorMsg



`red`, `green`, `blue`, `hue`, `saturation`, `brightness`, `alpha` messages address a specific part of a color using the RGB or HSB scheme.

colorvalue



The value may be specified as integer or float. The data range is given in table 2.1. When the alpha component is not specified, the color is assumed to be opaque.

Component	integer range	float range
`red` [R]	[0,255]	[-1,1]
`green` [G]	[0,255]	[-1,1]
`blue` [B]	[0,255]	[-1,1]
`alpha` [A]	[0,255]	[-1,1]
`hue` [H]	[0,360]	[-1,1] mapped to [-180,180]
`saturation` [S]	[0,100]	[-1,1]
`brightness` [B]	[0,100]	[-1,1]

Table 2.1: Color components data ranges when expressed as integer or float.

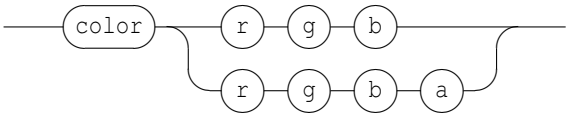
EXAMPLE

The same alpha channel specified as integer value or as floating point value:

```
/ITL/scene/myObject alpha 51 ;
/ITL/scene/myObject alpha 0.2 ;
```

2.3.2 The color messages

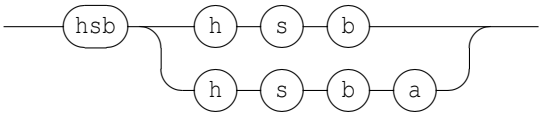
color



`color` sets an object color in the RGBA space. When A is not specified, the color is assumed to be opaque. Default color value is `[0,0,0,255]`.

2.3.3 The hsb messages

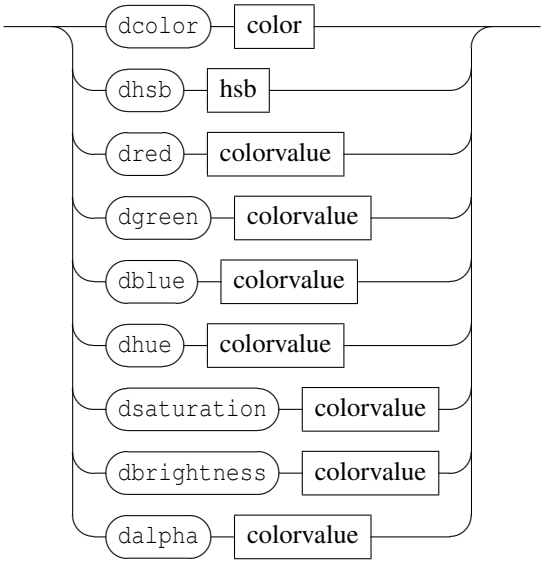
hsb



`hsb` sets an object color in the HSBA space. When A is not specified, the color is assumed to be opaque.

2.3.4 Relative color messages

relColorMsg



- `dred`, `dgreen`, etc. messages are similar to `red`, `green`, etc. messages but the parameters values represent a displacement of the current target value.
- `dcolor` and `dhsb` are similar and each color parameter represents a displacement of the corresponding target value.

EXAMPLE

Moving a color in the RGBA space:

```
TL/scene/myObject dcolor 10 5 0 -10 ,
```

will increase the red component by 10, the blue component by 5, and decrease the transparency by 10.

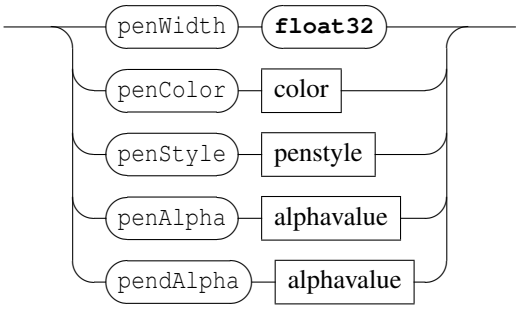
NOTE Objects that are carrying color information (images, SVG) don't respond to color change but are sensitive to transparency changes.

2.4 Pen control

Pen messages accepted by all the components and result in 2 different behaviors:

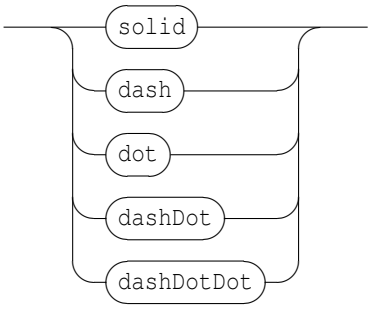
- for components types `rect | ellipse | polygon | curve | line | graph | fast graph | grid`, it makes the object border visible using the pen attributes;
- for the other components and when the pen width is greater than 0, it makes the object bounding box visible.

penMsg



- `penWidth` controls the pen width. The default value is 0 (excepted for `line` objects, where 1.0 is the default value). It is expressed in arbitrary units (1 is a reasonable value).
- `penColor` controls the pen color. The color should be given in the RGBA space. The default value is opaque black (0 0 0 255).
- `penStyle` controls the pen style.
- `penAlpha, pendAlpha` controls the pen transparency only. See section 2.3 p.8 for the expected

penstyle



The pen style default value is `solid`.

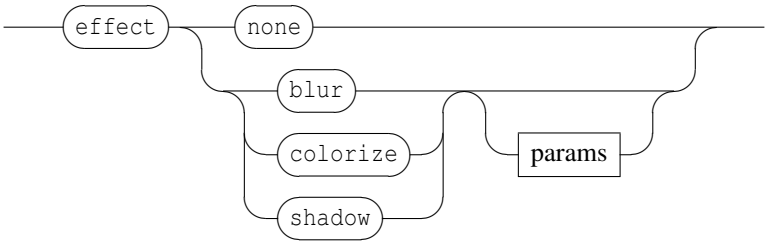
EXAMPLE
Setting a rectangle border width and color:

```
/ITL/scene/rect set rect 0.5 0.5 ;
/ITL/scene/rect penWidth 2.  ;
/ITL/scene/rect penColor 255 0 0 ;
```

2.5 The 'effect' messages

The `effect` message sets a graphic effect on the target object.

effectMsg

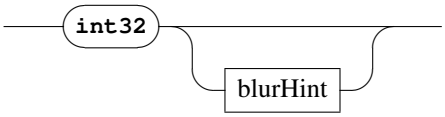


- `none`: removes any effect set on the target object.
- `blur, colorize, shadow`: sets the corresponding effect. An effect always replaces any previous effect. The effect name is followed by optional specific effects parameters.

NOTE An effect affects the target object but also all the target slaves.

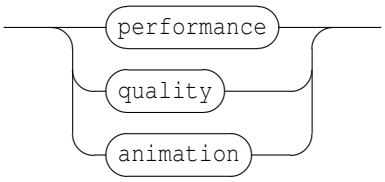
2.5.1 The blur effect

blurParams



Blur parameters are the blur radius and a rendering hint. The radius is an int32 value. By default, it is 5 pixels. The radius is given in device coordinates, meaning it is unaffected by scale.

blurHint



Use the `performance` hint to say that you want a faster blur, the `quality` hint to say that you prefer a higher quality blur, or the `animation` when you want to animate the blur radius. The default hint value is `performance`.

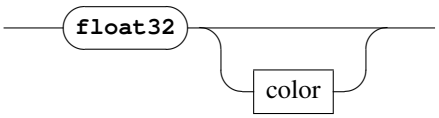
EXAMPLE

Setting a 8 pixels effect on `myObject`

```
/ITL/scene/myObject effect blur 8;
```

2.5.2 The colorize effect

colorizeParams



Colorize parameters are a strength and a tint color. The strength is a float value. By default, it is 1.0. A strength 0.0 equals to no effect, while 1.0 means full colorization.
The color is given as a RGB triplet (see section 2.3 p.8) by default, the color value is light blue (0, 0, 192).

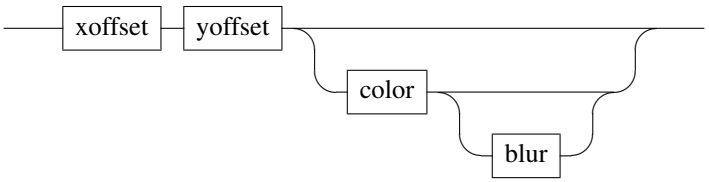
EXAMPLE

Setting a red colorize effect on `myObject` with a 0.5 strength.

```
/ITL/scene/myObject effect colorize 0.5 200 0 0;
```

2.5.3 The shadow effect

shadowParams



`xoffset` and `yoffset` are the shadow offset and should be given as int32 values. The default value is 8 pixels. The offset is given in device coordinates, which means it is unaffected by scale.
The color is given as a RGBA color (see section 2.3 p.8) by default, the color value is a semi-transparent dark gray (63, 63, 63, 180)
The blur radius should be given as an int32 value. By default, the blur radius is 1 pixel.

EXAMPLE

Setting a shadow effect on `myObject`.
The shadow offset is (10,10) pixels, the color is a transparent grey (100,100,100, 50) and the blur is 8 pixels.

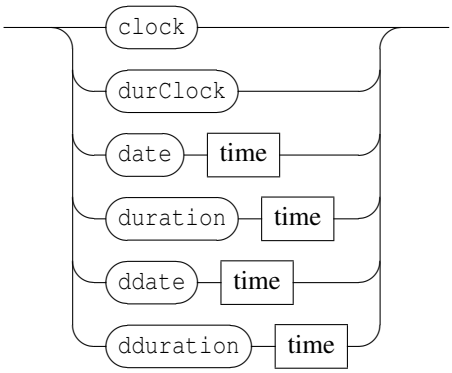
```
/ITL/scene/myObject effect shadow 10 10 100 100 100 50 8;
```

Chapter 3

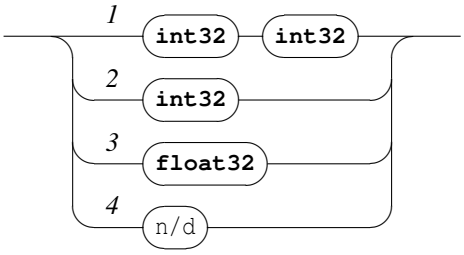
Time management messages

Time messages control the time dimension of the score components. They could be sent to any address with the form `/ITL/scene/identifier` where *identifier* is the unique identifier string of a scene component.

timeMsg



time



- 1) Time is specified as a rational value `d/n` where `1/1` represents a whole note.
- 2) Time may be specified with a single integer, then 1 is used as implicit denominator value.
- 3) Time may be specified as a single float value that is converted using the following approximation: let `f` be the floating point date, the corresponding rational date is computed as `f x 10000 / 10000`.
- 4) Time may also be specified as a string in the form `'n/d'`.

- `clock`: similar to MIDI clock message: advances the object date by 1/24 of quarter note.
- `durClock`: a clock message applied to duration: increases the object duration by 1/24 of quarter note.
- `date`: sets the time position of an object. Default value is `0/1`.

- `duration`: changes the object duration. Default value is `1/1`.
- `ddate`: relative time positioning message: adds the specified value to the object date.
- `dduration`: relative duration message: adds the specified value to the object duration.

EXAMPLE

Various ways to set an object date.

```
/ITL/scene/myObject date 2 1 ;
/ITL/scene/myObject date 2;        // the denominator is 1 (implied)
/ITL/scene/myObject date 0.5;      // equivalent to 1/2
/ITL/scene/myObject date '1/2';    // the string form
```

Similar ways to move an object date.

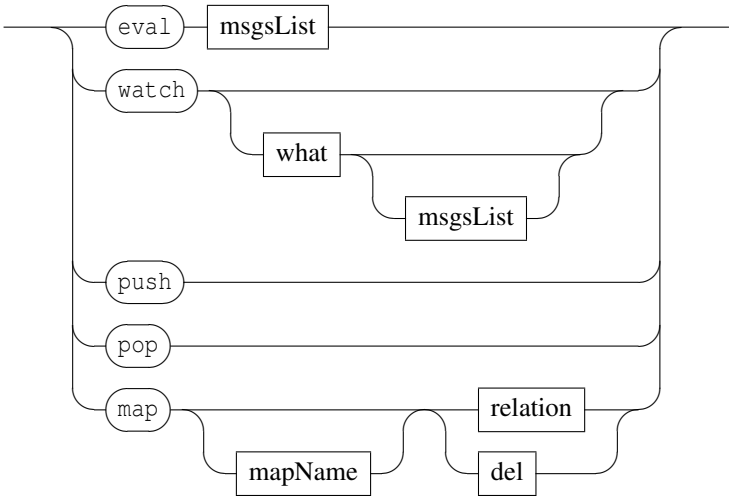
```
/ITL/scene/myObject clock;
/ITL/scene/myObject ddate '1/96';
```

Chapter 4

Miscellaneous messages

The following messages are supported by all the objects. They are detailled in specialized sections.

miscMsgs



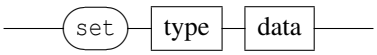
- `eval`: evaluates a list of messages in the context of the receiver object. See section 16.2 p.69 for more details.
- `watch`: used to manage the object interaction with various events. See section 15 p.61 for more details.
- `push`, `pop`: saves and restores the object interaction state. See section 15.3 p.67 for more details.
- `map`: used to describe the relations between graphic and time spaces. See section 12 p.45 for more details.

Chapter 5

The 'set' message

The `set` messages can be sent to any address with the form `/ITL/scene/`*`identifier`*. The global form of the message is:

setMsg

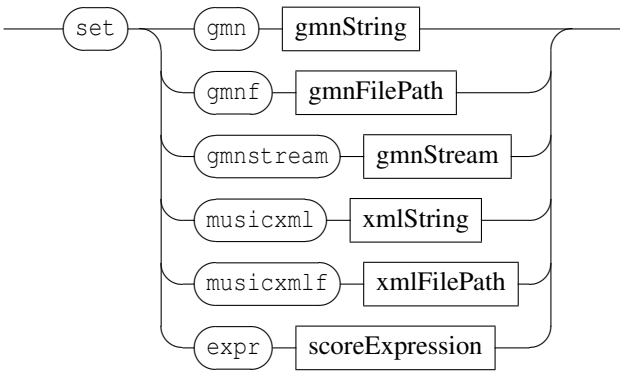


It sets a `scene` component data.
When there is no destination for the OSC address, the component is first created before being given the message.
When the target destination type doesn't correspond to the message `type`, the object is replaced by an adequate object.

5.1 Symbolic music notation

Symbolic music notation support is based on the Guido Music Notation format [GMN] or on the MusicXML format. MusicXMl is supported via conversion to the GMN format when the MusicXML library is present.

setMsg



- `gmn`: a Guido score defined by a GMN string.
- `gmnf`: a Guido score defined by a GMN file.
- `gmnstream`: a Guido score defined by a GMN stream (a GMN string that can be written in several times).

- `musicxml`: a score defined by a MusicXML string.
- `musicxmlf`: a score defined by a MusicXML file.
- `expr`: a score defined by a *score expression*.

EXAMPLE

Creating a music score using a Guido Music Notation language string.

```
/ITL/scene/myObject set gmn "[ a b g ]";
```

Creating the same music score as a stream.

```
/ITL/scene/myObject set gmnstream "[ a";
/ITL/scene/myObject write "b";
/ITL/scene/myObject write "g";
```

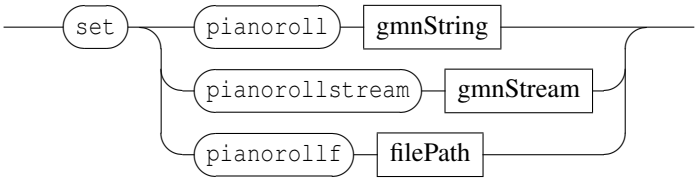
NOTE For compatibility with previous versions, passing a MusicXML string to a `gmn` object or a MusicXML file to a `gmnf` object may succed since the system tries to parse the content as GMN content or as MusicXML content when the former fails.

NOTE Conversion from MusicXML to GMN could be achieved manually using a command line tool that is distributed with the MusicXML library (see at https://github.com/dfober/libmusicxml). It allows to improve the output GMN code afterhand.

5.2 Piano roll music notation

Piano roll music notation is based on the Guido Music Notation format [GMN].

setMsg



- `pianoroll`: a piano roll defined by a GMN string.
- `pianorollstream`: a piano roll defined by a GMN stream (a GMN string that can be written in several times).
- `pianorollf`: a piano roll defined by a guido file (with ".gmn" extension) or by a midi file (with ".mid" extension). Warning: url forms are not supported for midi files.

EXAMPLE

Creating a pianoroll using a Guido Music Notation language string.

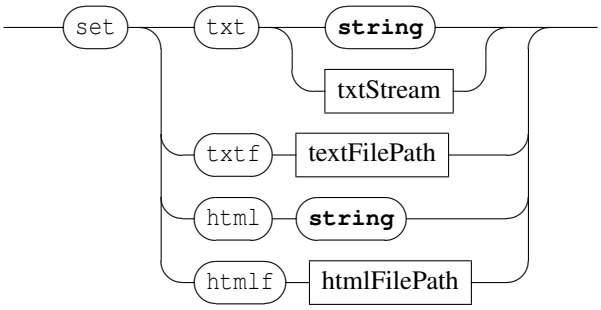
```
/ITL/scene/myObject set pianoroll "[ a b g ]";
```

Creating the same piano roll as a stream.

```
/ITL/scene/myObject set pianorollstream "[ a";
/ITL/scene/myObject write "b";
/ITL/scene/myObject write "g";
```

5.3 Textual components

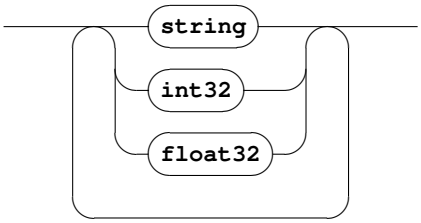
setMsg



- `txt`: a textual component.
- `txtf`: a textual component defined by a file.
- `html`: an html component defined by an HTML string.
- `htmlf`: an html component defined by an HTML file.

Text may be specified by a single quoted string or using an arbitrary count of parameters that are converted to a single string with a space used as separator.

txtStream



EXAMPLE

Creating a text object.

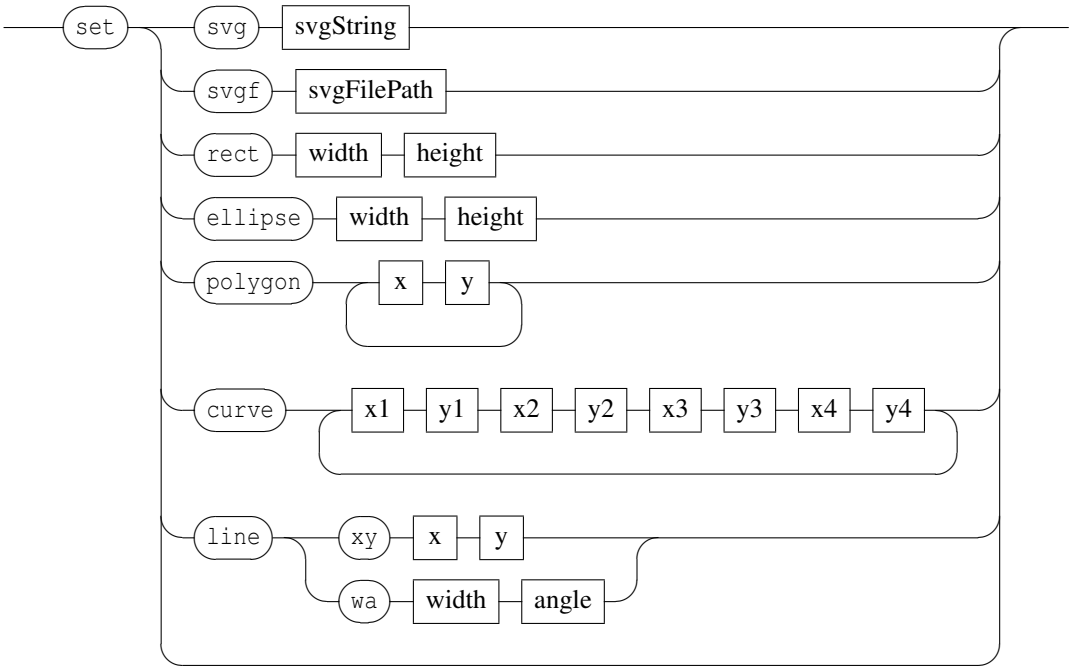
```
/ITL/scene/myObject set txt "Hello ...  world!";
```

Setting the content of a text object using a values stream.

```
/ITL/scene/myObject set txt Hello 1 world and 0.5;
```

5.4 Vectorial graphics

setMsg



- `svg`: SVG graphics defined by a SVG string.
- `svgf`: vectorial graphics defined by a SVG file.
- `rect`: a rectangle specified by a width and height. Width and height are expressed in `scene` coordinates space, thus a width or a height of 2 corresponds to the width or a height of the `scene`.
- `ellipse`: an ellipse specified by a width and height.
- `polygon`: a polygon specified by a sequence of points, each point being defined by its (x,y) coordinates. The coordinates are expressed in the `scene` coordinate space, but only the relative position of the points is taken into account (*i.e* a polygon A = { (0,0) ; (1,1) ; (0,1) } is equivalent to a polygon B = { (1,1) ; (2,2) ; (1,2) }).
- `curve`: a sequence of 4-points bezier cubic curve. If the end-point of a curve doesn't match the start-point of the following one, the curves are linked by a straight line. The first curve follows the last curve. The inner space defined by the sequence of curves is filled, using the object color. The points coordinates are handled like in a `polygon`.
- `line`: a simple line specified by a point (x,y) expressed in `scene` coordinate space or by a width and angle. The point form is used to compute a line from (0,0) to (x,y), which is next drawn centered on the scene.

EXAMPLE

Creating a rectangle with a 0.5 width and a 1.5 height.

```
/ITL/scene/myObject set rect 0.5 1.5;
```

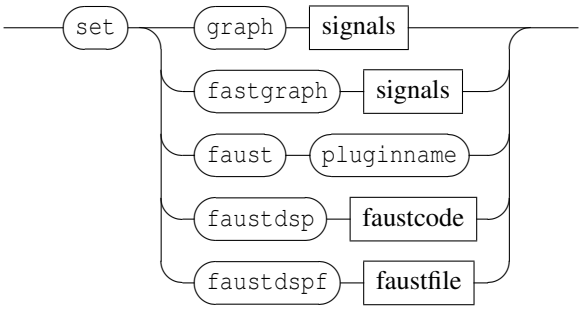
Creating a line specified using width and angle.

```
/ITL/scene/myObject set line wa 1.  45.;
```

5.5 Signals and graphic signals

Signals are special objects that are stored in a special `signal` node and that may be composed in parallel to produce graphic signals. Signals and graphic signals are decribed in section 14 p.53.
Signals and computation on signals may be based on FAUST objects that are actually signals processors. FAUST objects are decribed in section 18.1 p.79.
For more information about the FAUST language, see at `http://faust.grame.fr`.

setMsg

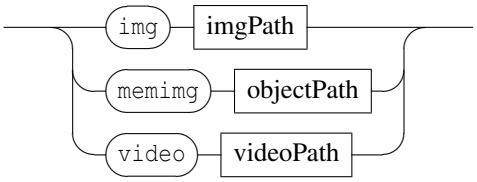


- `graph`: graphic of a signal. See section 14 p.53 for details about the `graph` objects data.
- `fastgraph`: fast rendering graphic signal. See also section 14 p.53.
- `faust`: a FAUST object as a plugin (see section 18.1)
- `faustdsp`: a FAUST object defined by a string (see section 18.1 p.79)
- `faustdspf`: a FAUST object defined by a file (see section 18.1 p.79)

5.6 Images and video

Images and video are supported using various formats. See section 5.9 p.23 for more details on the supported formats.

setMsg



- `img`: an image file. The image format is infered from the file extension.
- `memimg`: a memory capture of the object given as argument. `objectPath` indicates the target object that is captured with all its childrens. It may be an object name or a path to an object. Simple object names and relative path are looked for in the receiver layer.
- `video`: a video file. The video format is infered from the file extension. Note that navigation through the video is made using its `date`.

EXAMPLE

Creating an image.

```
/ITL/scene/myObject set img "myImage.png";
```

Creating a memory image of a scene.

```
/ITL/scene/myObject set memimg "/ITL/scene";
```

NOTE

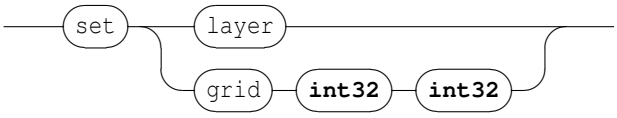
It is necessary to have an object or scene graphically rendered before a capture can be made. Since the actual graphic rendering is made asynchronously to the model update, a sequence of messages like the following:

```
/ITL/scene/myObject set gmn "[a f g]";
/ITL/scene/capture set memimg myObject;
```

won't work if the messages are handled by the same time task. A delay is necessary between the two messages. To make sure all the objects have been rendered, you can use the scene `endPaint` event.

5.7 Miscellaneous

setMsg



- `layer`: a graphic layer, may be viewed as a container (see section 11 p.44).
- `grid`: a white transparent object that provides a predefined time to graphic mapping (see section 7.5 p.31 for more details and section 12 p.45 for time to graphic relations). The parameters are int32 values representing the number of columns and rows.

5.8 File based resources

Most of the types can be either expressed with the corresponding data, or by a path to a file containing the data. For the latter form, the object type is generally suffixed with an 'f' (e.g. `txtf`, `htmlf`, `gmnf`, `musicxmlf`, `svgf`, `faustf`). The `img` and `video` types have only a file form (and no 'f' suffix).
A file path can be expressed as a Unix path (absolute or relative - see the scene or application `rootPath` message for relative paths handling), but also as an URL. Only the `http:` protocol is currently supported.
When the system encounters an URL, it creates an intermediate object that is in charge of retrieving the corresponding data. This object has a specific `url` type that takes the target type and an url as arguments. It has a graphic appearance (actually a light gray box containing the object name and the target url) that can be controled like for any regular object.

urlType

The `url` intermediate object acts as a proxy for the target object and will transfer all its properties once the data are ready. A client can thus interact transparently with the target adress, whatever the status of the download request.

EXAMPLE

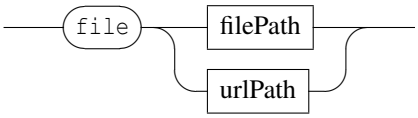
Creating a score using an URL:

```
/ITL/scene/score set gmnf "http://anyhost.adomain.org/score.gmn";
   is equivalent to
/ITL/scene/score set url gmnf "http://anyhost.adomain.org/score.gmn";
```

NOTE The `url` object handles specific events : `success`, `error` and `cancel` (see the section 15.1.4 p.64).

5.9 The file type

setFile



- `file`: a generic type to handle file based objects. Actually, the `file` type is translated into a one of the `txtf`, `gmnf`, `img` or `video` types, according to the file extension (see table 5.1).

See also: the application `rootPath` message (section 8 p.34) for file based objects.

Table 5.1: File extensions supported by the `file` translation scheme.

file extension	translated type
.txt .text	txtf
.htm .html	htmlf
.gmn	gmnf
.xml	musicxmlf
.svg	svgf
.jpg .jpeg .png .gif .bmp .tiff	img
.avi .wmv .mpg .mpeg .mp4 .mov .vob	video
.dsp	faustdspf

EXAMPLE

Creating an image using the `file` type.

```
/ITL/scene/myObject set file "myImage.png";
   is equivalent to
/ITL/scene/myObject set img "myImage.png";
```

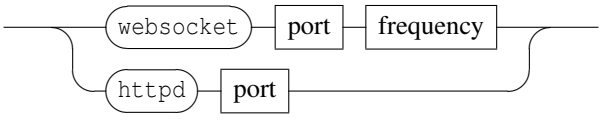
5.10 Web objects

A score can make its content available to the Internet using specific components that provide an image of the scene over http or websocket protocols.

The `httpd` server depends on the Httpd server plugin and is described in section 18.3 p.85.

The `websocket` server provides a two-ways communication between INScore and distant clients. The server sends notifications to client using a `Screen updated` text message when the scene is updated. Clients can request an image by sending a `getImage` text message to the server. The server responds with a image of the scene in png format, using a Blob type javascript object.

webobject



- `port`: a port number for the socket communication.
- `frequency`: a minimum time in millisecond between two `Screen updated` notifications.

NOTE

A busy port prevents the server to start. The server status can be checked with the `get status` message.

EXAMPLE

Creating an websocket server using the port `1234` and limiting the notifications rate to one per 500 milliseconds.

```
/ITL/scene/myObject set websocket 1234 500;
```

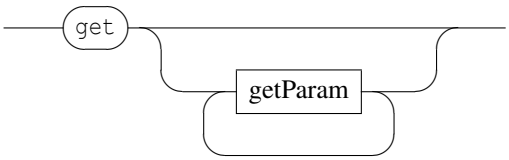
See also: the http web server plugin (section 18.3 p.85).

Chapter 6

The 'get' messages

The `get` messages can be sent to any valid OSC address. It is intended to query the system state. It is the counterpart of all the messages modifying this state. The result of the query is sent to the OSC output port with the exact syntax of the counterpart message. The global form of the message is:

getMsg



The `get` message without parameter is the counterpart of the `set` message. When addressed to a container (the application `/ITL`, a scene `/ITL/scene`, the signal node `/ITL/scene/signal`) is also distributed to all the container components.

Specific `get` forms may be available, depending on the component type (see sections 7.3, 8.3, 8.4.2, 13, 14.1.2, 18.1.2).

The `get frame` message is supported by all the components. An object frame is available for read only. It represents the polygon that encloses the object, taking account of scaling, rotations, and shear. The polygon is returned as a set of 4 points (x, y) expressed in the parent object coordinates space.

EXAMPLE

Sending the following request to an object which position is 0.3 0.5

```
/ITL/scene/myobject get x y;
```

will give the following messages on outpout port:

```
/ITL/scene/myobject x 0.3;
/ITL/scene/myobject y 0.5;
```

Querying an object content

```
/ITL/scene/myobject get;
```

will give the corresponding `set` message:

```
/ITL/scene/myobject set txt "Hello world!";
```

Querying an object frame

```
/ITL/scene/myobject get frame;
```

will give the corresponding `frame` message:

```
/ITL/scene/myobject frame -0.5 -0.25 0.50 -0.25 0.50 0.25 -0.5 0.25;
```

NOTE

The `get width` and `get height` messages addressed to components that have no explicit width and height (text, images, etc.) returns 0 as long as the target component has not been graphically rendered.

Chapter 7

Type specific messages

Some of the messages are accepted only by specific components.

7.1 Brush control

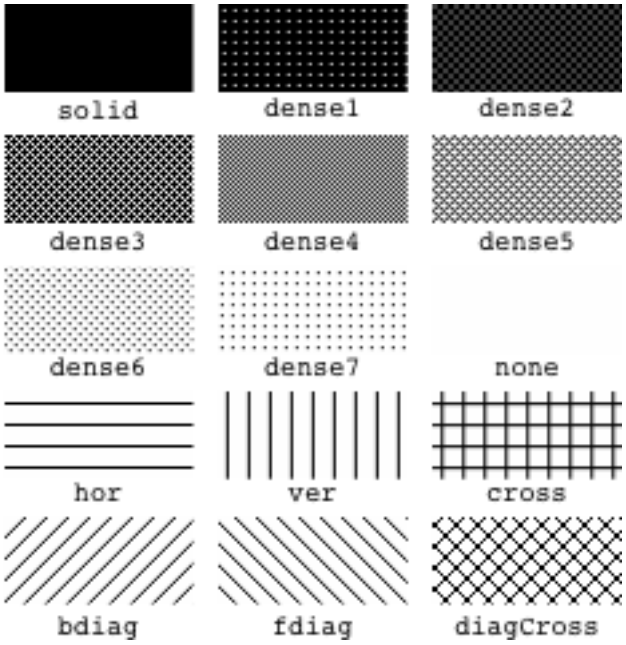
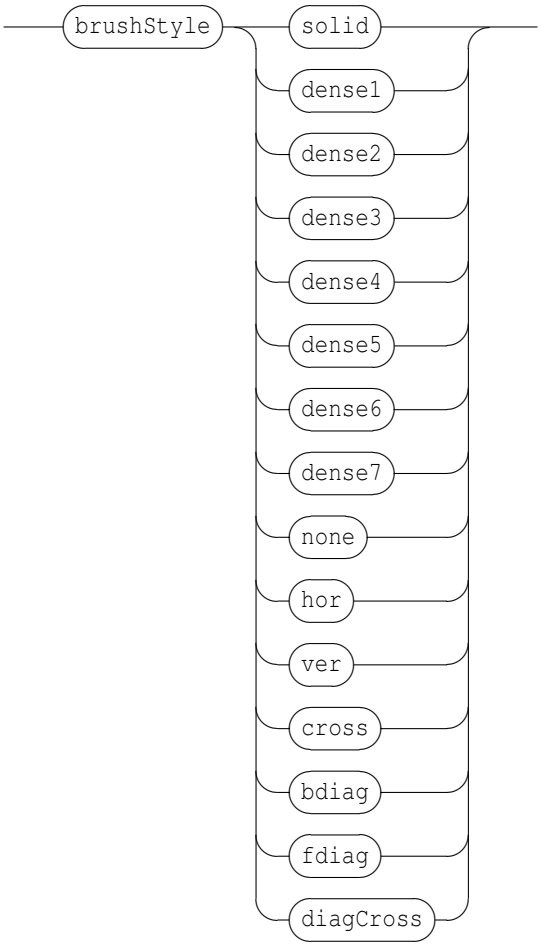


Figure 7.1: Brush styles

Specific brush messages accepted by the following components: `rect | ellipse | polygon | curve | layer`.

brushMsg



- `brushStyle` controls the brush style (see figure 7.1).

The brush style default value is `solid`.
For the `layer` object, the brush style default value is `none`.

EXAMPLE

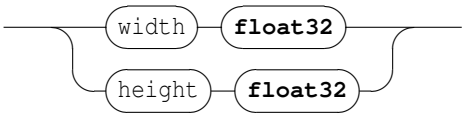
Setting a rectangle style :

```
/ITL/scene/rect set rect 0.5 0.5 ;
/ITL/scene/rect brushStyle dense4;
```

7.2 Width and height control

`width` and `height` messages are accepted by the following components: `rect | ellipse | graph | fastgraph | grid | pianoroll | pianorollf`.

widthMsg

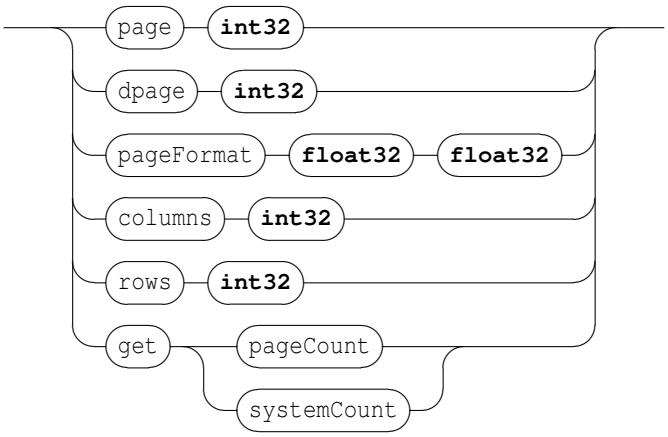


NOTE Querying the `width` and `height` of any object is always supported, provided that the object has been graphically rendered.

7.3 Symbolic score management

The following messages are accepted by the components types `gmn | gmnstream | gmnf`.

scoreMsg



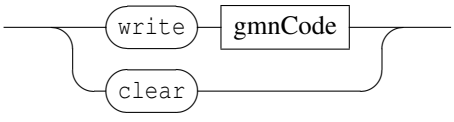
- `page`: set the score current page
- `dpage`: moves the score current page
- `pageFormat`: set the page format. The parameters are the page width and height. Note that the message has no effect when the score already includes a \pageformat tag.
- `columns`: for multi pages display: set the number of columns.
- `rows`: for multi pages display: set the number of rows.
- `pageCount`: a read only attribute, gives the score pages count.
- `systemCount`: a read only attribute, gives the number of systems on each of the score pages. The result is given as a list systems count ordered by page number (index 0 is page 1, etc.).

EXAMPLE

Displaying a multi-pages score on two pages starting at page 3:

```
/ITL/scene/myScore columns 2 ;
/ITL/scene/myScore page 3 ;
```

gmnstreamMsg



- `write`: add the gmn code to the current gmn stream
- `clear`: reinitialize the stream

EXAMPLE

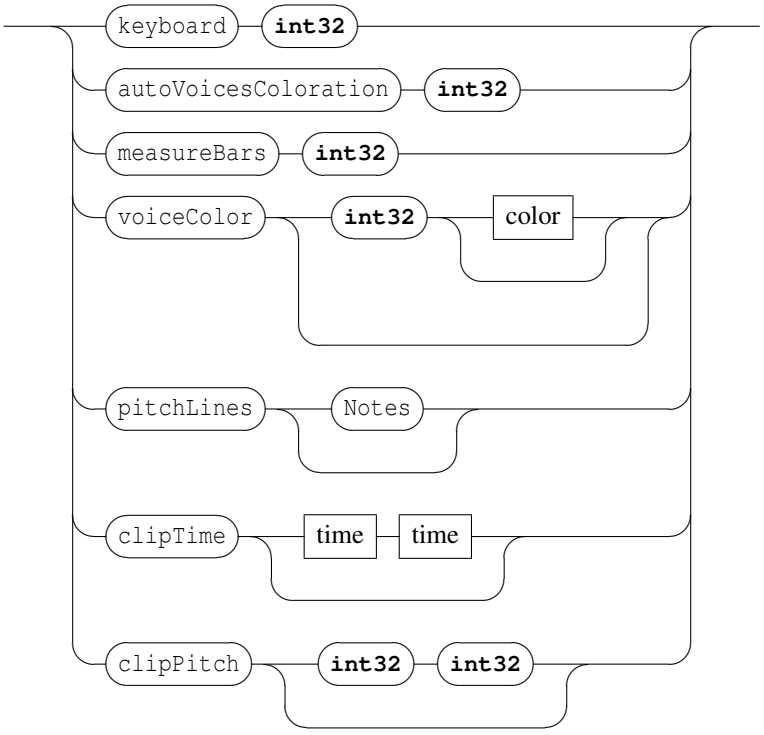
Writing a score in 3 steps:

```
/ITL/scene/myScore set gmnstream "[ c";
/ITL/scene/myScore write " d e";
/ITL/scene/myScore write " f]";
```

7.4 Piano roll management

The following messages are accepted by the components types `pianoroll | pianorollstream | pianorollf`.

pianorollMsg



- `keyboard`: display the keyboard on left of piano roll. Default value to 0.
- `autoVoicesColoration`: enable voices automatic coloration. If voiceColor is used for a voice, automatic voices coloration do nothing for it. Default value to 0.
- `measureBars`: Display measure bars on piano roll. Default value to 0.
- `voiceColor`: set a color to a voice. The parameters are voice number (start to 1), and RGBA color (See section 2.3 p.8). If not color is present, voice color is reset to default color. If voice number and color are not present, reset all voices to default color.
- `pitchLines`: Display pitch lines on pianoroll. Parameters are a note list in english notation (A A# B ...) with case insensitive. Default to all lines. An '`empty`' note (i.e. the litteral '`empty`' string) can be used to not display any line.
- `clipTime`: set time limits for piano roll (See section 3 p.14 to set a time). The two times have to be wrote in the same format. If no time is present, time limits are reset to default.
- `clipPitch`: Set pitch limits to piano roll. The pitch is in midi format. If no value is present, pitch limits are reset to default.

EXAMPLE

Set a color on voice 2 with transparency and display C and F pitch lines:

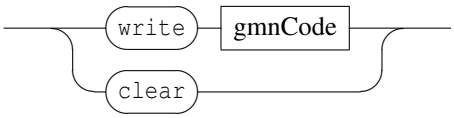
```
/ITL/scene/myPianoroll voiceColor 2 154 234 45 100;
/ITL/scene/myPianoroll pitchLines 'C' 'F';
```

Removes the pitch lines:

```
/ITL/scene/myPianoroll pitchLines empty;
```

Piano roll streams support the same messages than Guido streams:

pianorollstreamMsg



- `write`: add the gmn code to the current gmn stream
- `clear`: reinitialize the stream

EXAMPLE

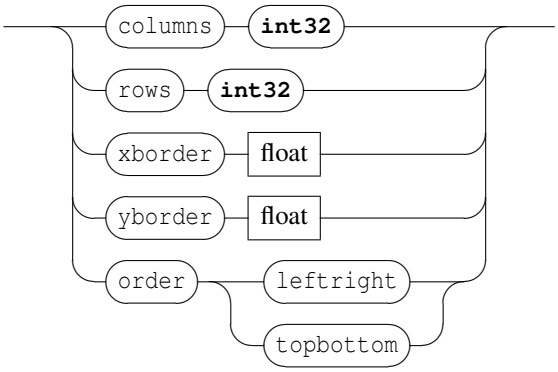
Writing a pianoroll in 3 steps:

```
/ITL/scene/myPianoroll set pianorollstream "[ c";
/ITL/scene/myPianoroll write " d e";
/ITL/scene/myPianoroll write " f]";
```

7.5 The 'grid' object

The `grid` object provides a pre-defined time to graphic mapping organized in columns and row. By default, it is not visible (white, transparent) but supports all the attributes of rectangles (color, pen, effects, etc.). Each element of a grid has a duration that is computed as the grid duration divided by the total number of elements (columns x rows) and is placed in the time space from the date 0 to the end of the grid duration.

gridMsg



- `columns` set the number of columns of the grid,
- `rows` set the number of rows of the grid,
- `xborder` set the horizontal spacing between the elements of the grid (default is 0.),
- `yborder` set the vertical spacing between the elements of the grid (default is 0.),
- `order` defines the time order of the elements. By default, elements are organized from left to right first and from top to bottom next (`leftright`). The `topbottom` parameter changes this order from top to bottom first and from left to right next.

EXAMPLE

Creating a 10 x 10 grid organized from top to bottom with a border:

```
/ITL/scene/grid set grid 10 10 ;
/ITL/scene/grid xborder 3.  ;
/ITL/scene/grid yborder 3.  ;
/ITL/scene/grid order topbottom ;
```

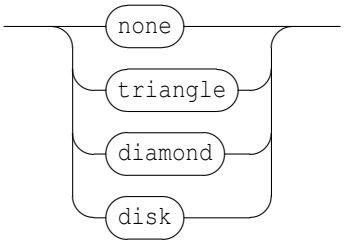
7.6 Arrows

Specific arrows message is accepted by the component type `line`. It add capability to draw arrow heads to the begining and the end of a line object.

arrowsheadMsg

- `arrowStyleBegin` Set the arrow head of the begining of the line.
- `arrowStyleEnd` Set the arrow head of the end of the line.

arrowStyle

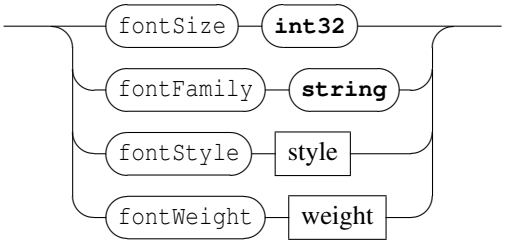


The arrow style default value is `none`.

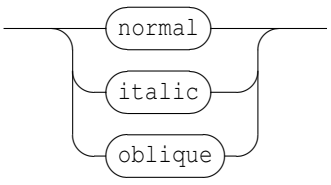
7.7 Font control

Specific font messages are accepted by `txt` components.

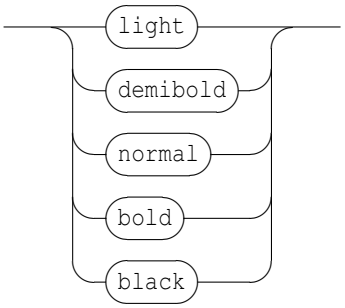
fontMsg



fontStyle



fontWeight



- `fontSize` controls the font size in pixel. The default value is 13px.
- `fontFamily` controls the font family. The default value is 'Arial'. If a non existing value is used, system default font is used.
- `fontStyle` controls the pen style. The font style default value is `normal`.

- `weightValue` controls the font weight. The font weight default value is `normal`.

EXAMPLE

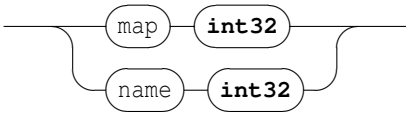
Setting a text object with a font family Times and bold weight:

```
/ITL/scene/text set txt "text sample";
/ITL/scene/text fontFamily Times;
/ITL/scene/text fontWeight bold;
```

7.8 The 'debug' nodes

Each component includes a static `debug` nodes provided to give information about components.

debugMsg



- `map` is used to display the time to graphic mapping. The parameter is a int value: 0 prevents mapping display, 1 displays only the bounding boxes and 2 displays also the dates along with the boxes. Default is 0 (no map).
- `name` is used to display both the object name and bounding box. The parameter is a boolean value. Default is 0.

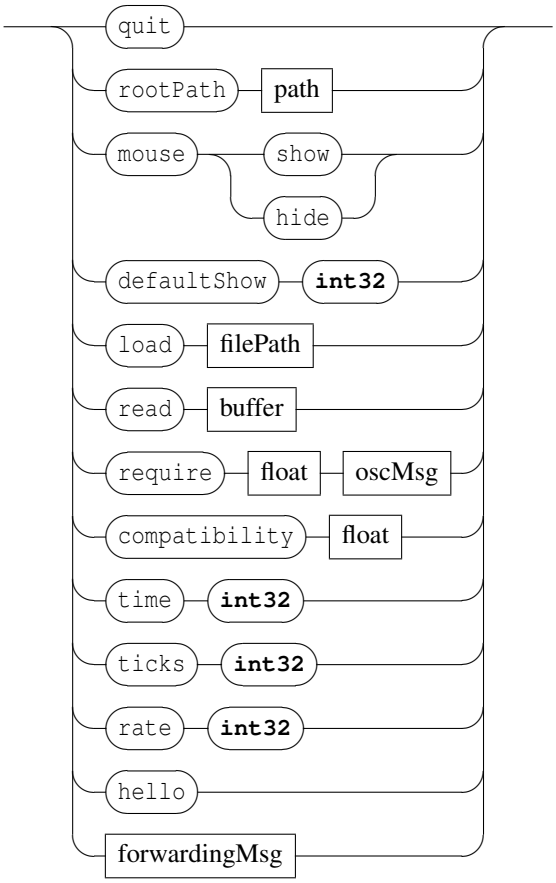
Chapter 8

Application messages

Application messages are accepted by the static OSC address `/ITL`.

8.1 Application management

ITLMsg



- `quit`: requests the client application to quit.

- `rootPath`: *rootPath* of an INScore application is the default path where the application reads or writes a file when a relative path is used for this file. The default value is the user home directory. Sending the `rootPath` message without parameter resets the application path to its default value.
- `mouse`: hide or show the mouse pointer.
- `defaultShow`: changes the default `show` status for new objects.
  The default `defaultShow` value is 1.
- `load`: loads a file previously saved using the `save` message (see section 2 p.4). Note that the load operation appends the new objects to the existing scene. When necessary, it is the sender responsibility to clear the scene before loading a file. URL are supported for the file path (see section 5.8 p.22);
- `read`: read a buffer that is expected to contain a valid inscore script.
- `require`: check that the current INScore version number is equal or greater to the number given as argument. The version number is given as a float value. A message is associated to the `require` message, which is triggered when the check fails. See section 15 p.61 for more details.
- `compatibility`: preserve INScore previous behavior. The argument corresponds to a version number, INScore will preserve the corresponding behavior (objects scaling, default size, etc.).
- `rate`: changes the time task rate. Note that null values are ignored.
  The default `rate` value is 10.
- `time`: sets the application current time. The time is expressed in milliseconds.
- `ticks`: sets the application current ticks count. The ticks count indicates the number of time tasks performed by the application.
- `hello`: query the host IP number. The message is intended for ITL applications discovery. Answer to the query has the following format:
  `IP inPort outPort errPort` where `IP` is sent as a string and port numbers as integer values.
- `forwardingMsg`: application support message forwading and filtering. See section 10 p.42.

EXAMPLE

when sending the message:

```
/ITL hello;
```

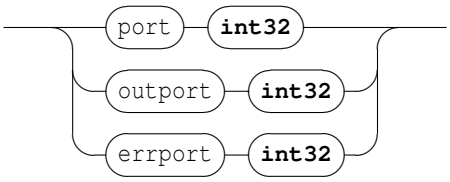
the application will answer with the following message:

```
/ITL 192.168.0.5 7000 7001 7003
```

when it runs on a host which IP number is `192.168.0.5` using the default port numbers.

8.2 Ports management

ITLPortsMsg



Changes the UDP port numbers:

- `port` defines the listening port number,
- `outport` defines the port used to send replies to queries,

- `errport` defines the port used to send error messages.

The `int32` parameter should be a positive value in the range `[1024-49150]`.
The default `port`, `outport` and `errport` values are 7000, 7001 and 7002.

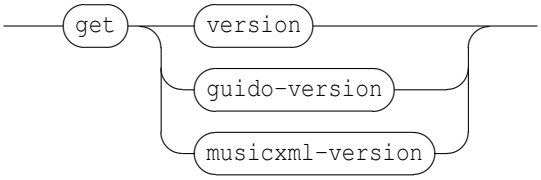
NOTE

Error messages are sent as a single string.

8.3 Application level queries

The application supports the `get` messages for its parameters (see section 6 p.25). In addition, it provides the following messages to query version numbers.

ITLRequest



- `version`: version number request.
- `guido-version`: Guido engine version number request.
- `musicxml-version`: MusicXML and Guido converter version numbers request. Returns "not available" when the library is not found.

EXAMPLE

Querying INScore version:

```
/ITL get version;
```

will give the following as output:

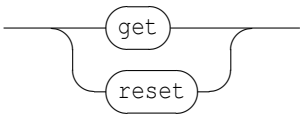
```
/ITL version 1.00
```

8.4 Application static nodes

The application level provides the static nodes - `stats`, `debug` and `log`, available at `/ITL/stats` `/ITL/debug` and `/ITL/log` to help debugging communication and INScore scripts design.

8.4.1 The 'stats' nodes

ITLStats



- `get` gives the count of handled messages at OSC and UDP levels: the UDP count indicates the count of messages received from the network, the OSC count includes the UDP count and the messages received internally.

- `reset` resets the counters to zero. Note that querying the `stats` node increments at least the OSC the counter.

EXAMPLE

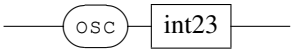
Answer to a `get` message addressed to `/ITL/stats`

```
/ITL/stats osc 15 udp 10
```

8.4.2 The 'debug' nodes

The `debug` node is used to activate debugging information.

ITLdebug



- switch the debug mode ON or OFF. The parameter is interpreted as a boolean value. When in debug mode, INScore sends verbose messages to the OSC error port for every message that can't be correctly handled. Debugging is ON by default.

EXAMPLE

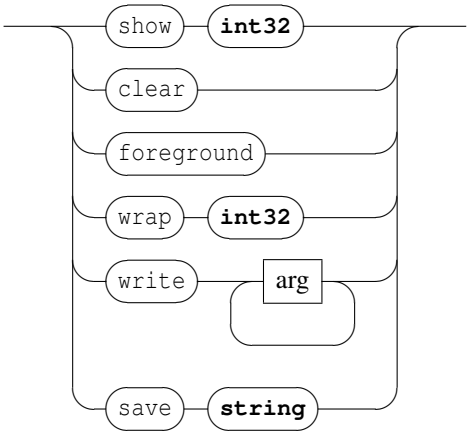
Error messages generated on error port in debug mode:

```
error:  incorrect OSC address:  /ITL/stat
error:  incorrect parameters:  /ITL/scene/foo unknown 0.1
error:  incorrect parameters:  /ITL/scene/foo x "incorrectType"
```

8.4.3 The 'log' nodes

The `log` node controls a console window that display all the messages sent to the OSC error port. Typical content is given by the example above.

ITLLog

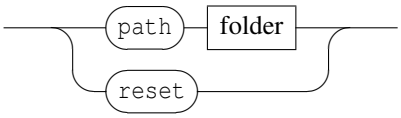


- `show` show or hides the console. The parameter is a boolean value.
- `clear` clear the console window.
- `foreground` put the console window to front.
- `wrap` control line wrapping of the console. The parameter is a boolean value.
- `write` write the `arg` list formatted as a string to the log window.
- `save` save the current log content to a file. The parameter is a file name. When expressed as a relative path, the file is saved under the current application root path.

8.4.4 The 'plugins' nodes

The `plugins` node controls the search path for plugins. See section 18 p.79 for more information on plugins and search strategies.

ITLPlugin



- `path` add `folder` as a user path. The system will look for plugins in this folder first.
- `reset` clear the current user path.

Chapter 9

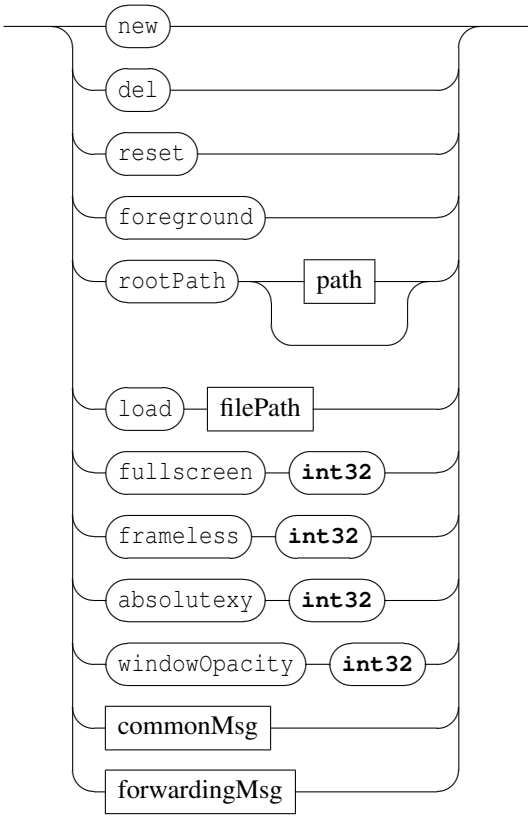
Scene messages

A scene may be viewed as a window on the score elements. Its address is `/ITL/`*`sceneIdentifier`* where *`sceneIdentifier`* is the scene name.

9.1 Scene control

The following messages are available at scene level, to control the scene appearance and behaviour:

sceneMsg



- `new`: creates a new scene and opens it in a new window.

- `del`: deletes a scene and closes the corresponding window.
- `reset`: clears the scene (i.e. delete all components) and resets the scene to its default state (position, size and color).
- `foreground`: display scene window in foreground of all other windows in the system windows manager.
- `rootPath`: *rootPath* of a scene is the default path where the scene reads or writes a file when a relative path is used for this file. When no value has been specified, the application *rootPath* is used. Calling `rootPath` without argument clears the scene *rootPath*.
- `load`: loads an INScore file to the scene. Note that the OSC addresses are translated to the scene OSC address.
- `fullscreen`: requests the scene to switch to full screen or normal screen. The parameter is interpreted as a boolean value. Default value is `0`.
- `frameless`: requests the scene to switch to frameless or normal window. The parameter is interpreted as a boolean value. Default value is `0`.
- `absolutexy`: requests the scene to absolute or relative coordinates. Absolute coordinates are in pixels relative to the top left corner of the screen. Relative coordinates are in the range [-1, 1] where [0,0] is the center of the screen. The message parameter is interpreted as a boolean value. Default value is `0`.
- `windowOpacity`: switch the scene window to opaque or transparent mode. When in transparent mode, the scene alpha channel controls the window opacity (from completely opaque to completely transparent). In opaque mode, the scene alpha channel controls the background brush only. Default value is `0` (transparent).
- `commonMsg`: a scene support the common graphic attributes. See section 2 p.4.
- `forwardingMsg`: a scene support message forwading and filtering. See section 10 p.42.

EXAMPLE

Setting a scene current path:

```
/ITL/scene rootPath "/path/to/my/folder";
```

Loading an INScore file:

```
/ITL/scene load "myscript.inscore";
```

will load `/path/to/my/folder/myscript.inscore` into the scene.

Setting a scene to fullscreen:

```
/ITL/scene fullscreen 1;
```

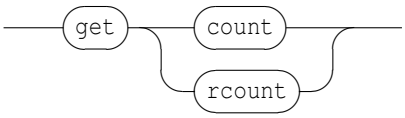
Creating a new score named `myScore`:

```
/ITL/myScore new;
```

9.2 Scene queries

A scene may respond to queries regarding its elements:

sceneQuery



- `count`: count the number of elements in the scene.
- `rcount`: recursively count the number of elements in the scene.

EXAMPLE

Counting the elements in a scene:

```
/ITL/scene get count;
   will give a message like the following as output:
/ITL/scene count 200;
```

Chapter 10

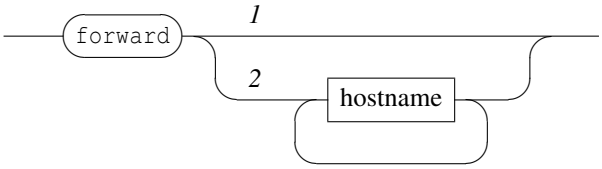
Messages forwarding

The messages handled by the application or by a scene can be forwarded to arbitrary remote hosts. A filtering mecanism can be used to have a fine control of forwarded messages.

10.1 Remote hosts list

Remote hosts lists can be set using the `forward` message at scene or application level. Hosts lists of the application and of each scene are independent. At scene level, only messages handle by the scene are forwarded (ie message for the scene itself or for one of his children object). The `forward` message itself can't be forwarded. A message from a host cannot be forwarded to him to avoid direct loop.

ITLMsgForward



- 1) removes the set of forwarded destinations,
- 2) set a list of remote hosts for forwarding. Note that `hostname` can be any legal host name or IP number, optionally extended with a port number separated by a semi-colon. By default, when no port number is specified, the default application listening port number is used (7000).

EXAMPLE

Forwarding messages handle by application to `host1.adomain.org` using the default application listening port number (7000) and to `host2.adomain.org` on port number 5100.

```
/ITL forward host1.adomain.org host2.adomain.org:5100;
```

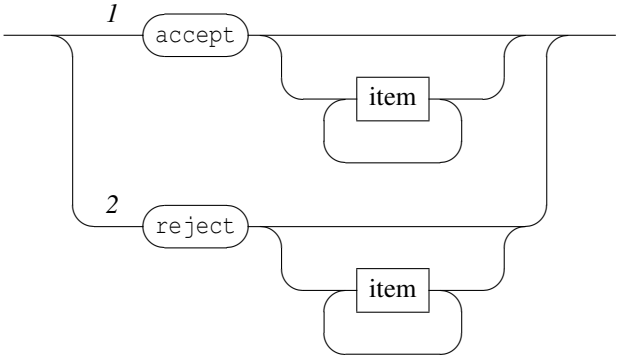
Forwarding messages handle by the scene `scene1` to `host3.adomain.org` using the default application listening port number (7000) and to `host4.adomain.org` on port number 5100.

```
/ITL/scene1 forward host1.adomain.org host2.adomain.org:5100;
```

10.2 Filtering messages forwarding

The messages forwarded to arbitrary remote hosts using the `forward` message can be filtered to send only wanted messages. The static `filter` node is use manage the filter. A static `filter` node is created for each scene and one at application level. The filter can be construct with OSC address and messages.

ITLFilteringForward



- 1) Remplace the actual accepted list by the new list or by an empty list. Item in accepted list are not filtered by the reject item list.
- 2) Remplace the actual accepted list by the new list or by an empty list. Item in reject list are filtered if they not match the accept list.

When a new message is incoming, if they match to an accepted item, filter is not apply.

EXAMPLE
Filter at application level :

```
/ITL/filter reject /ITL/scene/line* /ITL/scene/rect;
/ITL/filter accept /ITL/scene/line2 scale arrows;
```

Message with OSC message with address starting with `/ITL/scene/line` or with address `/ITL/scene/rect` are filtered only if message address is not `/ITL/scene/line2` or if the content is not `scale` or `arrows`.

Filter at scene level :

```
/ITL/scene1/filter reject fontWeight /ITL/scene1/rect;
```

The `fontWeight` message and message for `/ITL/scene1/rect` are rejected.

Chapter 11

Layers

Layers may be viewed as containers or as groups. They represent a way to structure both the address space and the graphic space.

From graphic viewpoint, a layer is a scene inside a scene. All the properties of 'rect' components are available to layers: position, scale, color, transparency, etc.). By default, a layer is not visible: it has no brush and no pen, but changing the brush style (see section 7.1 p.27) - e.g. to `solid` - makes it visible.

From time viewpoint, a layer has the common time attributes i.e. a date, a duration.

A layer may be synchronized to other objects, including other layers. It includes a `sync` node and supports synhcronization of the enclosed objects. However, synchronization is restricted to objects from the same layer and cannot cross the border of a layer.

EXAMPLE

Creating a layer and its content:

```
/ITL/scene/layer1 set layer;
/ITL/scene/layer1/score set gmnf 'myscore.gmn';
/ITL/scene/layer1/cursor set rect 0.01 0.1;
```

Synchronizing 2 components of a layer :

```
!'score' and 'cursor' must be enclosed in layer1
/ITL/scene/layer1/sync cursor score;
```

Making a layer visible :

```
/ITL/scene/layer1 brushStyle solid;
/ITL/scene/layer1 color 120 120 120;
```

11.1 Layers generalization

The idea of layer is generalized to all the type of objects: any INScore object can be a container without depth limitation.

Layers but also any object respond to the `count` and `rcount` queries described in section 9.2 p.40.

Chapter 12

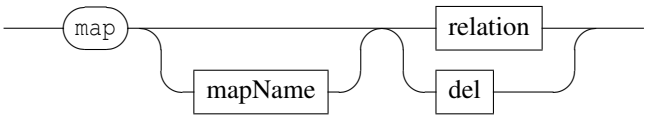
Mapping graphic space to time space

Time to space mapping refers to the description of relationship between an object local graphic space and its time space. A mapping consists in a set of relations between the two spaces. INScore provides specific messages to describes mappings and to synchronize arbitrary objects i.e. to display their time relationships in the graphic space.

12.1 The 'map' message

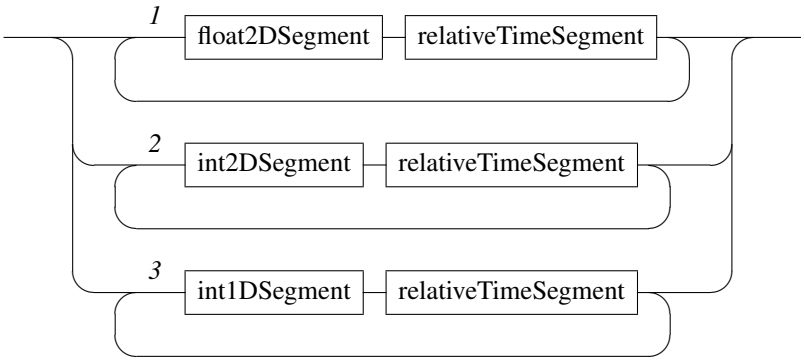
The `map` messages can be sent to any address with the form `/ITL/`*`scene`*`/`*`identifier`*. It is intended to describe the target object relation to time and sets a relation between an object segmentation and a time segmentation. The global form of the message is:

mapMsg



The `relation` parameter must be sent as a single string which format is described below. It consists in a list of associations between the object local space and its time space expressed as segments.

relation



Segments are expressed as a list of intervals. For a 1 dimension resource, a segment is a made of a single interval. For a 2 dimensions resource, a segment is a made of 2 intervals: an interval on the `x`-axis and one

on the y-axis for graphic based resource, or an interval on columns and one on lines for text based resources. Intervals are right-opened.

The different kind of relations corresponds to:

- **[1]** a relation between a 2 dimensions segmentation expressed in float values and a relative time segmentation. These segmentations are used by `rect, ellipse, polygon, curve, line` components.
- **[2]** a relation between a 2 dimensions segmentation expressed in integer values and a relative time segmentation. These segmentations are used by `txt, txtf, img` components.
- **[3]** a relation between a 1 dimension segmentation expressed in integer values and a relative time segmentation. These segmentations are used by the `graph` component and express a relation between a signal space and time.

Table 12.1 summarizes the specific local segmentation used by each component type.

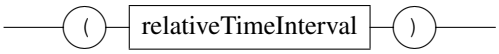
The specified `map` can be named with an optional `mapName` string; this name can be further reused, during object synchronization, to specify the mapping to use. When `mapName` is not specified, the mapping has a default *empty name*.

The `del` command deletes the mapping specified with `mapName`, or the *'empty name'* mapping if no map name is specified.

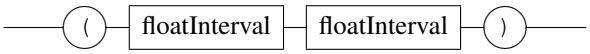
Table 12.1: Local segmentation type for each component

component type	segmentation type
`txt, txtf`	int2DSegments
`img`	int2DSegments
`rect, ellipse, polygon, curve`	float2DSegments
`graph`	int1DSegments

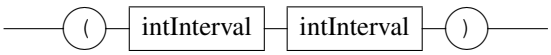
relativeTimeSegment



float2DSegment

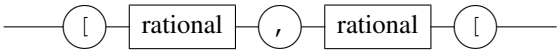


int2DSegment

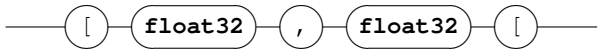


int1DSegment

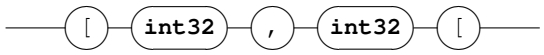
relativeTimeInterval



floatInterval

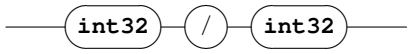


intInterval



Relative time is expressed as `rational` values where `1` represents a whole note.

rational



EXAMPLE

Mapping an image graphic space to time:

```
/ITL/scene/myImage map
      "( [0, 67[ [0, 86[ ) ( [0/2, 1/2[ )
       ( [67, 113[ [0, 86[ ) ( [1/2, 1/1[ )
       ( [113, 153[ [0, 86[ ) ( [1/1, 3/2[ )
       ( [153, 190[ [0, 86[ ) ( [3/2, 2/1[ )
       ( [190, 235[ [0, 86[ ) ( [2/1, 5/2[ )" ;
```

the image is horizontally segmented into 5 different graphic segments that express pixel positions. The vertical dimension of the segments remains the same and corresponds to the interval `[0, 86[`. Each graphic segment is associated to a time interval which duration is 1/2 (a half note).

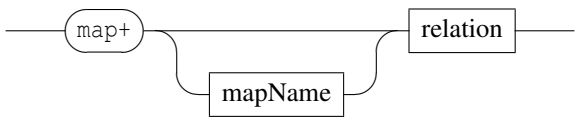
NOTE ABOUT LOCAL SPACES

- Text objects (`txt txtf`) local space is expressed by intervals on columns and rows.
- Html object (`html, htmlf`) do not support mapping because there is not correspondence between the text and the graphic space.
- Vectorial objects (`rect, ellipse, polygon, curve, svg,...`) express their local graphic space in internal coordinates system i.e. on the [-1.,1.] interval.
- Bitmap objects (`img`) express their local graphic space in pixels.

12.2 The 'map+' message

The `map+` messages is similar to the `map` message but doesn't replace the existing mapping data: the specified relations are added to the existing one.

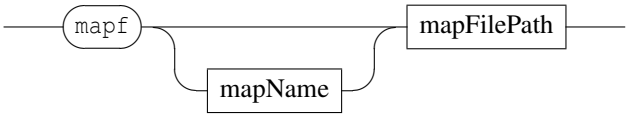
mapAddMsg



12.3 Mapping files

The `mapf` messages is similar to the `map` message but gives the path name of a file containing the mapping data, along with the optional map name.

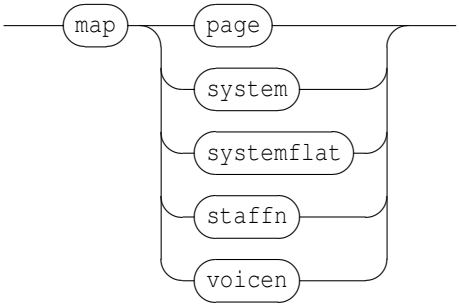
mapfMsg



12.4 Symbolic score mappings

Mapping between the graphic and time space is automatically computed for symbolic score `gmn, gmnstream, gmnf`. However and depending on the application, the graphic space may be segmented in different ways, for instance: different graphic segments for different staves, a single graphic segment traversing all a system, etc. Thus for a symbolic score, the `map` message different and is only intended to select one king mapping supported by the system.

scoreMap



- `page`: a page level mapping
- `system`: a system level mapping
- `systemflat`: a system level mapping without system subdivision (one graphic segment per system)
- `staff`*n*: a staff level mapping: the staff number is indicated by `n`, a number between 1 and the score staves count.
- `voice`*n*: a voice level mapping: the voice number is indicated by `n`, a number between 1 and the score voices count.

The default mapping for a symbolic score is unnamed but equivalent to `staff1`.

EXAMPLE

Requesting the mapping of the third staff of a score:

```
/ITL/scene/myScore map staff3;
```

Requesting the system mapping :

```
/ITL/scene/myScore map system;
```

NOTE

A voice may be distributed on several staves and thus a staff may contain several voices.

Chapter 13

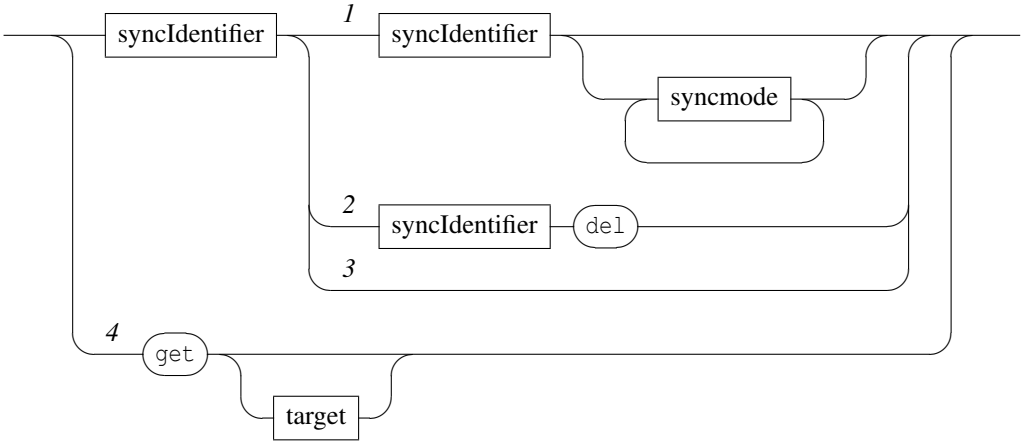
Synchronization

Synchronization between components is in charge of the static `sync` node, automatically embedded in each object. Its address is `/ITL/.../object/sync` and it supports messages to add or remove a master / slave relation between components or to query the synchronizations state.

NOTE

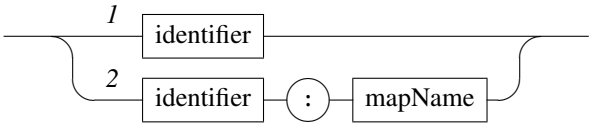
A master can naturally have several slaves, but a slave can have several masters as well. In this case, it will be drawn several times, corresponding to each master's space.

sync



- **[1]** the `slave master` form is followed by an optional synchronization mode (see below). It adds a slave / master relation between the first and the second component.
- **[2]** the `slave master del` form removes the specified slave/master relation.
- **[3]** the `slave` form without `master` removes all synchronizations with the slave.
- **[4]** the `get` message is intended to query the synchronization state. The optional parameter is the identifier of a component. The `get` message without parameter is equivalent to a `get` message addressed to each object declared in the `sync` node.

syncIdentifier



Synchronization identifiers indicates 1) the name of an object or 2) the name of an object associated to a mapping name. Using the first form (i.e. without explicit mapping name), the system uses the default unnamed mapping (see section 12.1 p.45 mappings and named mappings).

Synchronization between components has no effect if any of the required mapping is missing (see table 12.1).

EXAMPLE

Synchronizing an object on several masters:

```
/ITL/scene/myParent/sync mySlave myMaster1;
/ITL/scene/myParent/sync mySlave myMaster2;
```

Synchronizing two objects using a specific mapping (the second object is assumed to be a symbolic score (`gmn`, `gmnstream` or `gmnf`) which `system` mapping has been previously requested):

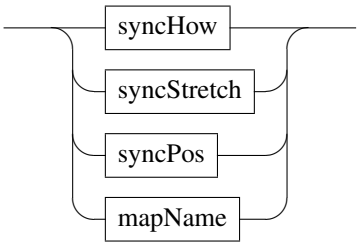
```
/ITL/scene/myParent/sync mySlave myMaster:system;
```

13.1 Synchronization modes

Synchronizing a slave component `A` to a master component `B` has the following effect:

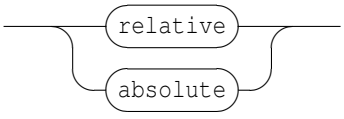
- `A` position (`x`) is modified to match the `B` time position corresponding to `A` date.
- depending on the optional `syncStretch` option, `A` width and/or height is modified to match the corresponding `B` dimension (see below).
- depending on the optional `syncPos` option, `A` vertical position (`y`) is modified. Note that the `y` position remains free and could always be modified using a `dy` message.
- if `A` date has no graphic correspondence in `B` mapping (the date is not mapped, or out of `B` mapping bounds), `A` won't be visible.

syncmode



13.1.1 Using the master date

syncHow



The synchronization mode makes use of the master time to graphic mapping to compute the slave position. It may also use the master current date, depending on the following options:

- `relative`: the time position where the slave appears is relative to the mapping and to the master current date (actually, it shifts the mapping from the master current date). The relative mode is used by default.

- `absolute`: the time position where the slave appears corresponds to the mapping date only.

NOTE

Use of the `absolute` mode may take sense with nested synchronizations: let's consider an object `A`, slave of `B`, which is slave of `C`. In `relative` mode and if `A` and `B` receive the same `clock` messages, `A` will remain at a fixed position on `B` although it is moving in time.

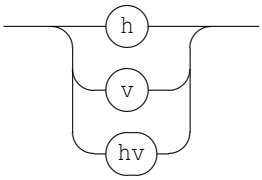
EXAMPLE

Describing nested synchronizations, the first one using the `absolute` mode:

```
/ITL/scene/sync slave masterSlave absolute ;
/ITL/scene/sync masterSlave master ;
```

13.1.2 Synchronizing an object duration

syncStretch



The synchronization stretch mode has the following effect on the slave dimensions:

- `h`: the slave is horizontally stretched to align its begin and end dates to the corresponding master locations.
- `v`: the slave is vertically stretched to the master map vertical dimension.
- `hv`: combines the above parameters.

By default, no stretching is applied.

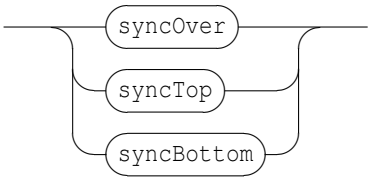
EXAMPLE

Synchronizing two objects, aligning the slave duration to the corresponding master space and stretching the slave to the master map vertical dimension:

```
/ITL/scene/sync mySlave myMaster hv ;
```

13.1.3 Controlling the slave y position

syncPos



The synchronization position mode has the following effects on the slave `y` position:

- `syncOver`: the center of the slave is aligned to the master center.
- `syncTop`: the bottom of the slave is aligned to the top of the master.
- `syncBottom`: the top of the slave is aligned to the bottom of the master.

The default position mode is `syncOver`. The `y` attribute of the slave remains available to displacement (`dy`).

NOTE

The `y` position of a synchronized object remains a free attribute. To control this position, you should send `dy` messages.

EXAMPLE

Synchronizing two objects, aligning the slave duration to the corresponding master space, the slave being below the master map:

```
/ITL/scene/sync mySlave myMaster h syncBottom;
```

Chapter 14

Signals and graphic signals

The graphic representation of a signal is approached with *graphic signals*. As illustrated in figure 14.1, the graphic representation of a signal could be viewed as a stream of a limited set of parameters : the `y` coordinate at a time `t`, a thickness `h` and a color `c`. A *graphic signal* is a composite signal including a set of 3 parallel signals that control these parameters. Thus the INScore library provides messages to create signals and to combine them into *graphic signals*.

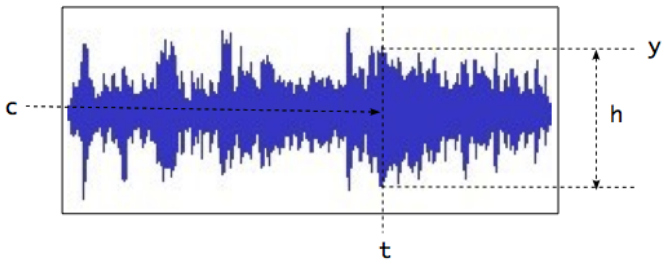


Figure 14.1: A simple graphic signal, defined at time t by a coordinate y, a thickness h and a color c

14.1 The 'signal' static node.

A `scene` includes a static signal node, which OSC address is `/ITL/`*`scene`*`/signal` which may be viewed as a container for signals. It is also used for *composing signals in parallel*.

The `signal` node supports the `get` message that gives the list of the defined signals and also the `get connect` message that gives a list of all connections, but doesn't take any argument.

EXAMPLE

Querying the signal node:

```
/ITL/scene/signal get;
```

will give the enclosed signals definitions:

```
/ITL/scene/signal/y size 200 ;
/ITL/scene/signal/h size 200 ;
```

And :

```
/ITL/scene/signal get connect;
```

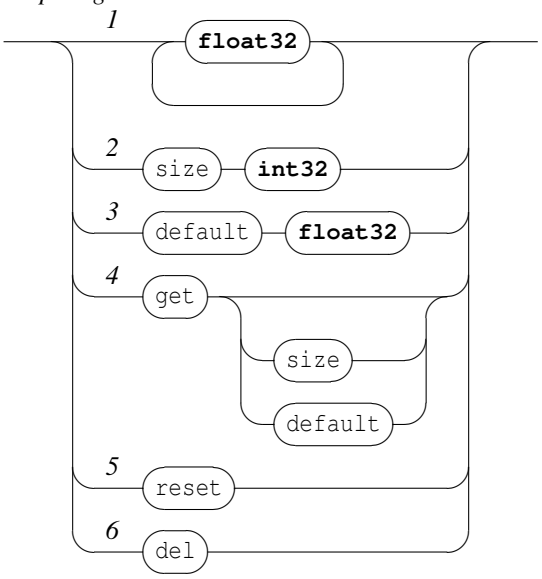
will give the signal connections :

```
/ITL/scene/signal connect cos object1:method1 ;
/ITL/scene/signal connect sin object2:method2 ;
```

14.1.1 Signal messages.

Signal messages can be sent to any address with the form `/ITL/`*`scene`*`/signal/`*`identifier`*, where *`identifier`* is a unique signal identifier. The set of messages supported by a signal is the following:

simpleSignal



- **[1]** push an arbitrary data count into the signal buffer. The expected data range is `[-1,1]`. Note that the internal data buffer is a ring buffer, thus data are wrapped when the data count if greater than the buffer size.
- **[2]** the `size` message sets the signal buffer size. When not specified, the buffer size value is the size of the first data message.
- **[3]** the `default` message sets the *default signal value*. A signal *default value* is the value returned when a query asks for data past the available values.
- **[4]** the `get` message without parameter gives the signal current values. The `size` and `default` parameters are used to query the signal size and default values.
- **[5]** the `reset` message clears the signal data.
- **[6]** the `del` message deletes the signal from the `signal` space. Note that it is safe to delete a signal even when used by a graphic signal.

EXAMPLE

Creating a signal with a given buffer size:

```
/ITL/scene/signal/mySig size 200;
```

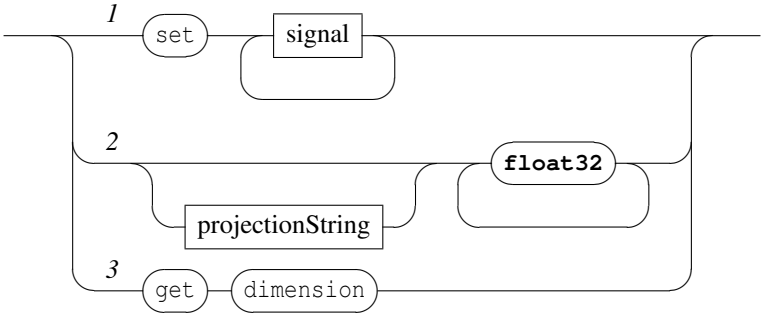
Creating a signal with a given set of data (the buffer size will be the data size):

```
/ITL/scene/signal/mySig 0. 0.1 0.2 0.3 0.4 0.5 0.4 0.3 0.2 0.1 0. -0.1 -0.2 ;
```

14.1.2 Composing signals in parallel.

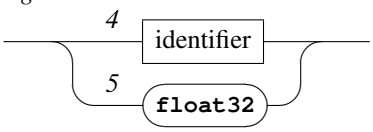
Composing signals in parallel produces a signal which value at a time `t` is a vector of the composed signals values. Thus an additional read-only attribute is defined on *parallel signals* : the signal *dimension* which is size of the signals vector. Note that the dimension property holds also for simple signals.
The format of the messages for parallel signals is the following:

parallelSignal



where

signal



- **[1]** defines a new signal composed of the signals given as parameters. A signal parameter is defined as:
  - **[4]** an `identifier` i.e. a signal name referring to an existing signal in the `signal` node.
  - **[5]** or as a float value. This form is equivalent to an anonymous constant signal holding the given value.
- **[2]** sets the values of the signals using a projection string. See section 14.1.3 p.56.
- **[3]** in addition to the `get` format defined for signals, a parallel signal supports the `get dimension` message, that gives the number of simple signals in parallel. The dimension of a simple signal is 1.

EXAMPLE

Putting a signal `y` and constant signals 0.01 0. 1. 1. 1. in parallel:

```
/ITL/scene/signal/mySig set y 0.01 0.  1.  1.  1.  ;
```

Querying the previously defined parallel signal:

```
/ITL/scene/signal/mySig get ;
will give the following output:
/ITL/scene/signal/mySig set y 0.01 0.  1.  1.  1.
```

NOTE

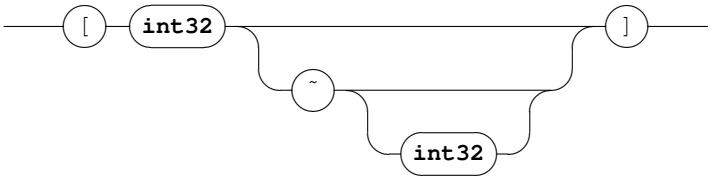
For a parallel signal:

- the `get size` message gives the maximum of the components size.
- the `get default` message gives the default value of the first signal.

14.1.3 Distributing data to signals in parallel

When signals are in parallel, a *projection string* may be used to distribute data over each signal. Individual components of a parallel signal may be addressed using a *projection string* that is defined as follows:

projectionString



The projection string is made of a *index value*, followed by an optional *parallel marker* (`~`), followed by an optional *step value*, all enclosed in brackets.

The *index value* `n` is the index of a target signal. When the *parallel marker* option is not present, the values are directed to the target signal. Indexes start at 0.

EXAMPLE

Sending data to the second component of a parallel signal:

```
/ITL/scene/signal/sig '[1]' 0. 0.1 0.2 0.3 0.4 0.5 0.4 0.3 0.2 0.1 0.  ;
```

is equivalent to the following message (assuming that the second signal name is 's2'):

```
/ITL/scene/signal/s2 0. 0.1 0.2 0.3 0.4 0.5 0.4 0.3 0.2 0.1 0.  ;
```

Note that:

- the message is ignored when `n` is greater than the number of signals in parallel. Default `n` value is `0`.
- setting directly the values of a simple signal or as the projection of a parallel signal are equivalent.

The *parallel marker* (`~`) and the *step value* `w` options affect the target signals. Let's consider `s[n]` as the signal at index `n`. The values are distributed in sequence and in loop to the signals `s[n], s[n+w]...s[m]` where `m` is the greatest value of the index `n+(w.i)` that is less than the signal dimension. The default *step value* is `1`.

EXAMPLE

Sending data to the second and third components of a set of 3 parallel signals:

```
/ITL/scene/signal/sig [1~] 0.1 0.2 ;
```

is equivalent to the following messages (assuming that the signal dimension is 3):

```
/ITL/scene/signal/sig [1] 0.1 ;
/ITL/scene/signal/sig [2] 0.2 ;
```

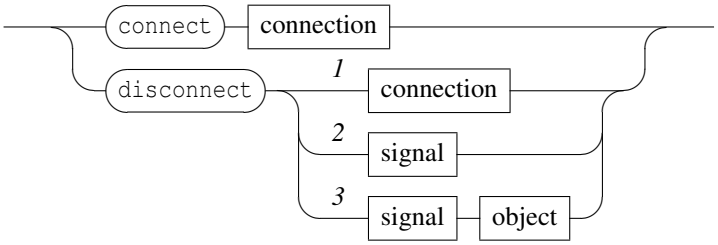
or to the following (assuming that the target signal names are 's2' and 's3'):

```
/ITL/scene/signal/s2 0.1;
/ITL/scene/signal/s3 0.2;
```

14.2 Connecting signals to graphic attributes.

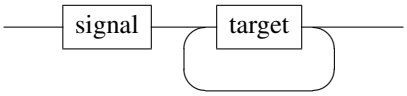
A signal may be connected to one or several graphic attributes of an object. Only the common attributes (see section 2 p.4) support this mechanism. When a connection between a signal and an object attribute is set, sending values to the signal is equivalent to send the values to the connected object attribute. A similar behavior could be achieved by sending the equivalent messages, however the connection mechanism is provided for efficiency reasons and in addition, it supports values scaling.

signalcnx



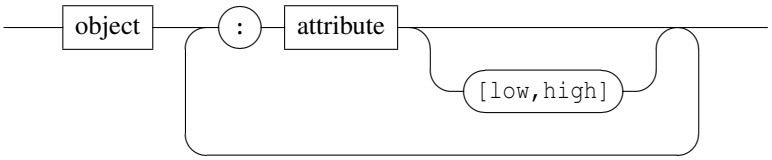
- the `connect` message makes a connection between a signal and one or several attributes of one or several objects.
- the `disconnect` message breaks a specific connection **[1]** or all the connections of a given signal **[2]**, or all connections between a given signal and a given object **[3]**.

connection



- `signal` is a name referring to an existing component of the `signal` node.

target



- `object` is the name of an object (must be on the same hierarchy level than the `signal` node).
- `attribute` is the name of the object target attribute (same name as the method used to set the attribute, e.g. `x`, `angle`, etc.).
- an optional scaling feature is provided with the `[low,high]` suffix: signal values are expected to be between -1 and 1, the scaling suffix re-scale the input values between `low` and `high`.

NOTE

Connections are restricted to one-dimensional signals as source and to one-dimensional attribute as target. This is not a real limitation since any component of a multi dimensional attribute (e.g. `color`) is always available as a single attribute (e.g. `red` or `blue`).

NOTE

A connection can't cross the borders of a component i.e. the target object and the signal node should have the same parent.

EXAMPLE

Connecting signals to attributes:

```
!  connects the values of sig1 to the red attribute of the 'rect' object
/ITL/scene/signal connect sig1 "rect:red";
!  connects the values of sig2 to several objects and attributes
/ITL/scene/signal connect sig2 "rect:blue:x:rotatey[0,360]" "cursor:date[0,15]";
```

Disconnecting some of the previous connections :

```
/ITL/scene/signal disconnect sig2 "cursor:date" "rect:rotatey:blue";
```

14.3 Graphic signals.

A graphic signal is the graphic representation of a set of parallel signals. It is created in the standard scene address space. A simple graphic signal is defined by a parallel signal controling the `y` deviation value, the thickness and the color at each time position. The color is encoded as HSBA colors (Hue, Saturation, Brightness, Transparency). The mapping of a signal value (`[-1,1]`) to the HSBA color space is given by the table 14.1.

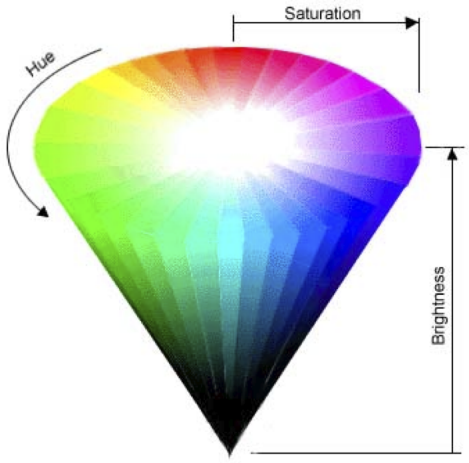


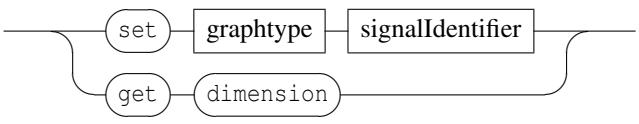
Figure 14.2: The HSB color space

Table 14.1: HSBA color values.

parameter	mapping	
`hue`	`[-1,1]`	corresponds to `[-180,180]` angular degree where `0` is red.
`saturation`	`[-1,1]`	corresponds `0%` to `100%` saturation.
`brigthness`	`[-1,1]`	corresponds `0%` (black) to `100%` (white) brithgness.
`transparency`	`[-1,1]`	corresponds `0%` to `100%` tranparency.

A graphic signal responds to common component messages (section 2 p.4). Its specific messages are the following:

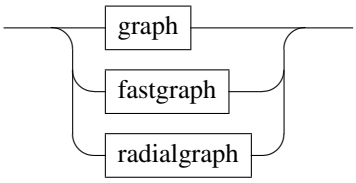
graphicSignal



- the `set` message is followed by the graph type and a *`signalIdentifier`*, where `signalIdentifier` must correspond to an existing signal from the `signal` address space. In case `signalIdentifier` doesn't exist, then a new signal is created at the `signalIdentifier` address with default values.

- the `get dimension` message gives the number of graphic signals in parallel (see section 14.3.2 p.59).

graphtype



The signal representation type is among:

- `graph`: a classical signal representation as illustrated in figure 14.1, where time is mapped to the x coordinate.
- `fastgraph`: a representation similar to the `graph` type, using a more efficient drawing strategy, but at the expense of a degraded graphic rendering.
- `radialgraph`: a signal representation where time is mapped to the polar coordinates. The rendering takes place in the ellipse enclosed in the object dimensions.

EXAMPLE

Creating a signal and its graphic representation:

```
/ITL/scene/signal/y size 200 ;
!  use of constant anonymous signals for thickness and color
/ITL/scene/signal/sig set y 0.1 0.  1.  1.  1.  ;
/ITL/scene/siggraph set graph sig ;
```

14.3.1 Graphic signal default values.

As mentionned above, a graphic signal expects to be connected to parallel signals having at least an `y` component, a graphic thickness component and HSBA components. Thus, from graphic signal viewpoint, the expected dimension of a signal should be equal or greater than 6. In case the `signalIdentifier` dimension is less than 6, the graphic signal will use the default values defined in table 14.2.

Table 14.2: Graphic signal default values.

parameter	default value	
`y`	`0`	the center line of the graphic
`thickness`	`0`	
`hue`	`0`	meaningless due to brigthness value
`saturation`	`0`	meaningless due to brigthness value
`brigthness`	`-1`	black
`transparency`	`1`	opaque

14.3.2 Parallel graphic signals.

When the dimension d of a signal connected to a graphic signal is greater than 6, then the input signal is interpreted like parallel graphic signals. More generally, the dimension n of a graphic signal is:

$$n \mid n \in \mathbb{N} \wedge 6.(n-1) < d \leqslant 6.n$$

where *d* is the dimension of the input signal.

When *d* is not a mutiple of 6, then the last graphic signal makes use of the default values mentionned above.

EXAMPLE

Creating parallel graphic signals:

```
/ITL/scene/signal/y1 size 200 ;
/ITL/scene/signal/y2 size 200 ;
!  use of constant anonymous signals for thickness and color
/ITL/scene/signal/sig1 set y1 0.1 0.  1.  1.  1.  ;
!  use a different color for 'sig2'
/ITL/scene/signal/sig2 set y2 0.1 0.6 1.  1.  1.  ;
!  put 'sig1' and 'sig2' in parallel
/ITL/scene/signal/sig set sig1 sig2;       !  'sig' dimension is 12
/ITL/scene/siggraph set graph sig;
```

NOTE

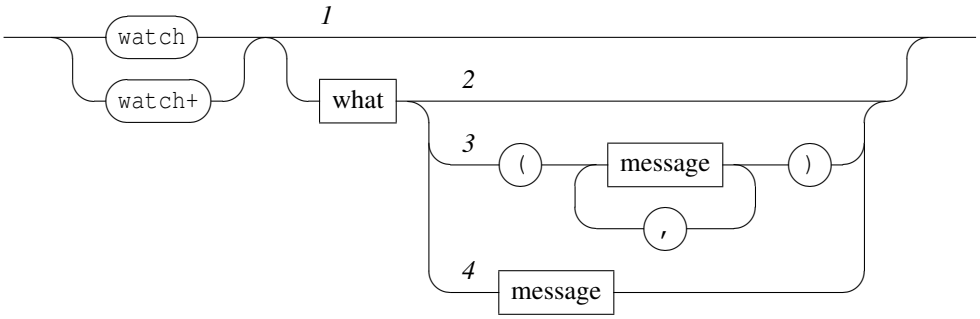
Using data projection may be convenient when the input signal represents interleaved data. For example, the projection string `[n~6]` distribute data over similar components of a set of graphic signals, where `n` represents the index of the graphic signal target component.

Chapter 15

Events and Interaction

Interaction messages are user defined messages associated to events and triggered when these events occur. These messages accept variables as message arguments.
The general form of the message is the following:

interactMsg



`what` represents the event to watch and `message` is a list of associated messages, separated by a comma.

1 : clear all the messages for all the events.

2 : clear the messages associated to the `what` event.

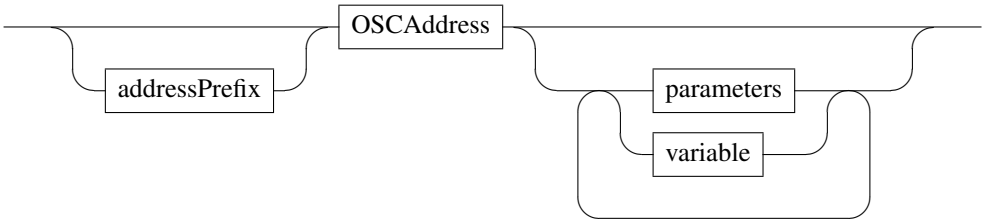
3 : associate a list of messages to the `what` event. With `watch`, the messages replace previously associated messages. Using `watch+`, the messages are appended to the messages currently associated to the event.

4 : associate or add a single message to the `what` event. This form is provided for compatibility with previous versions.

NOTE

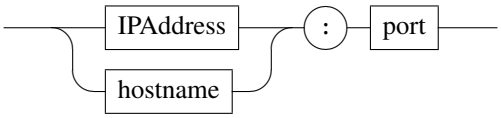
The [1] and [2] form has no effect with the `watch+` message.
In some environments, the comma has a special meaning, making tricky to use it as a message separator. This is why ':' is also accepted as separator in OSC messages.
The `get watch` message gives all the watch messages of a node, but doesn't take any argument.

message



The associated messages are any valid OSC message (not restricted to the INScore message set), with an extended address scheme, supporting IP addresses or host names and udp port number to be specified as OSC addresses prefix. The message parameters are any valid OSC type or variable (see section 15.2).

addressPrefix



EXAMPLE

An extended address to send messages to `localhost` on port `12000`:

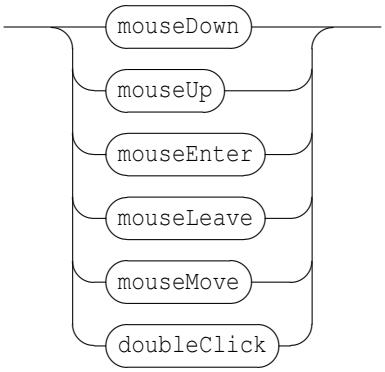
```
localhost:12000/your/osc/address;
```

15.1 Events

15.1.1 Mouse events

User interface events are typical mouse events:

what



EXAMPLE

Triggering a message on mouse down:

```
/ITL/scene/myObject watch mouseDown (/ITL/scene/myObject show 0);
```

the object hides itself on mouse click.

Triggering a message on mouse down but addressed to another host on udp port 12100:

```
/ITL/scene/myObject watch mouseDown (host.domain.org:12100/an/address start);
```

NOTE

UI events are not supported by objects that are synchronized as slave.

Mouse events can be simulated using a `event` message:

uievt

where `mouseEvent` is one of the events described above, `x` and `y` are integer values giving the click position, expressed in pixels and relative to the target object.

EXAMPLE

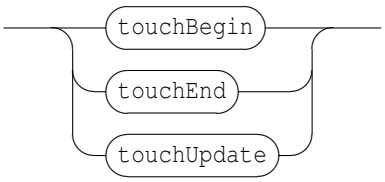
Simulating a mouse down at position 10, 10 :

```
/ITL/scene/myObject event mouseDown 10 10;
```

15.1.2 Touch events

Depending on the display device, multi-touch events are supported by INScore :

touch

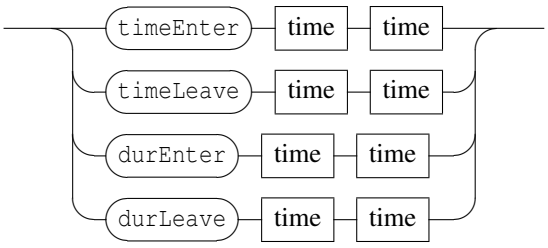


A typical sequence of generated events consists in a `touchBegin` event, followed by `touchUpdate` events and closed by a `touchEnd`.

15.1.3 Time events

Events are also defined on the time domain:

what



Each event takes a time interval as parameter, defined by two `time` specifications (see section 3 p.14 for the time format)

- `timeEnter`, `timeLeave` are triggered when an object date is moved to or out of a watched time interval,
- `durEnter`, `durLeave` are triggered when an object duration is moved to or out of a watched time interval.

EXAMPLE

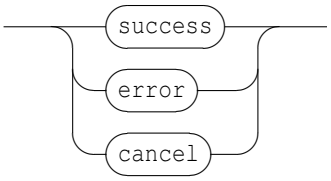
An object that moves a score to a given page number when it enters its time zone.

```
/ITL/scene/myObject watch timeEnter 10/1 18/1 (/ITL/scene/score page 2);
```

15.1.4 URL events

`url` objects (i.e. intermediate objects for URL based objects (see section 5.8 p.22) support specific events, intended to reflect the transaction state:

what



- `success` is triggered when the data have been downloaded,
- `error` is triggered when an error has occurred during the download,
- `cancel` is triggered when the target url or the object type is changed while downloading.

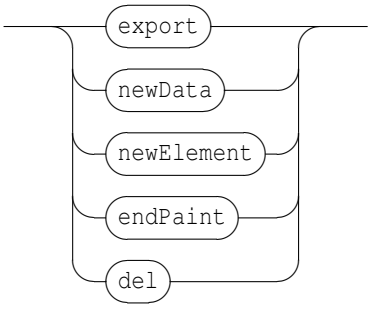
EXAMPLE

Triggering an error message in case of failure :

```
/ITL/scene/score set gmnf "http://ahost.adomain.org/score.gmn";
/ITL/scene/score watch error(
   /ITL/scene/status set txt "Failed to download file"
);
```

15.1.5 Miscellaneous events

what



- the `export` event is supported by all the components. It is triggered after an export message has been handled and could be used to simulate synchronous exports.
- the `newData` event is supported by all the components. It is triggered when the object specific data are modified (typically using the `set` message).
- the `newElement` event is supported at scene level only and triggered when a new element is added to the scene.
- the `endPaint` event is supported at scene level only and triggered after a scene has been painted.
- the `del` event is triggered when an object is being deleted.

EXAMPLE

Displaying a welcome message to new elements:

```
/ITL/scene watch newElement (/ITL/scene/msg set txt "Welcome");
```

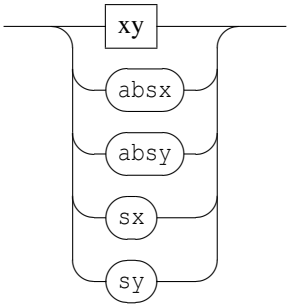
15.2 Variables

Variables are values computed when an event is triggered. These values are send in place of the variable. A variable name starts with a '$' sign.

15.2.1 Position variables

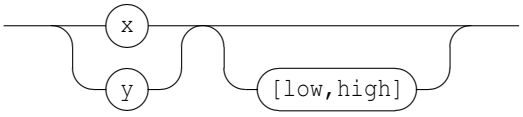
Position variables reflects the current mouse position for mouse events or the current touch position for touch events. They are set to `0` for the other events.

posVar



where

xy



- `$x` `$y`: denotes the mouse pointer position at the time of the event. The values are in the range `[0,1]` where 1 is the object size in the x or y dimension. The value is computed according to the object origin: it represents the mouse pointer distance from the object x or y origin (see 2.1.3 p.7). `$x` and `$y` variables support an optional range in the form `[low, high]` that transforms the `[0,1]` values range into the `[low, high]` range.
- `$absx` `$absy`: denotes the mouse pointer absolute position at the time of the event. The values represent a pixel position relative to the top-left point of the target object. Note that this position is unaffected by scale. Note also that the values are not clipped to the object dimensions and could exceed its width or height or become negative in case of mouse move events.
- `$sx` `$sy`: denotes the mouse pointer position in the scene coordinates space.

EXAMPLE

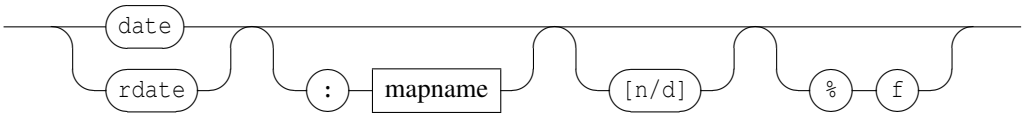
An object that follows mouse move.

```
/ITL/scene/myObject watch mouseDown (
                    /ITL/scene/myObject x '$sx',
                    /ITL/scene/myObject y '$sy' );
```

15.2.2 Time variables

Time variables reflects the date corresponding to the current mouse position for mouse events. They are set to `0` for the other events.

timeVar



- `$date`: denotes the object date corresponding to the mouse pointer position at the time of the event. It is optionnaly followed by a colon and the name of the mapping to be used to compute the date. The `$date` variable is replaced by its rational value (i.e. two integers values). The optional rational enclosed in brackets may be used to indicate a quantification: the date value is rounded to an integer count of the specified rational value. The optional `%f` may be used to get the date delivered as a float value.
- `$rdate`: is similar to `$date` but ignores the target current date: the date is relative to the object mapping only.

NOTE

A variable can be used several times in a message, but several `$date` variables must always refer to the same mapping.

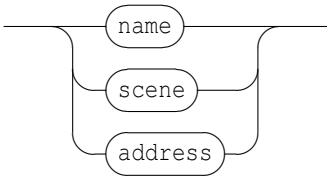
EXAMPLE

Sending the current date as a float value to an external application:

```
/ITL/scene/myObject watch mouseDown ( targetHost:12000/date '$date%f' );
```

15.2.3 Miscellaneous variables

variable



- `$name` is replaced by the target object name.
- `$scene` is replaced by the target object scene name.
- `$address` is replaced by the target object OSC address.

NOTE

For the `newElement` event, the target object is the new element.

EXAMPLE

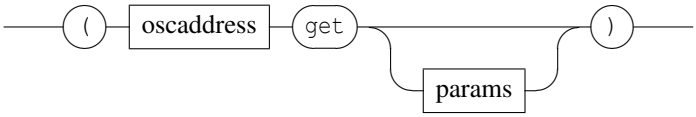
Using an object name:

```
/ITL/scene watch newElement (/ITL/scene/welcome set txt "Welcome" '$name');
```

15.2.4 Message based variables

A message based variable is a variable containing an OSC message which will be evaluated at the time of the event. They are supported by all kind of events. Like the variables above, a message based variable starts with a '$' sign followed by a valid 'get' message enclosed in parenthesis:

msgVar



The evaluation of a 'get' message produces a message or a list of messages. The message based variable will be replaced by the parameters of the messages resulting from the evaluation of the 'get' message. Note that all the 'get' messages attached to an event are evaluated at the same time.

EXAMPLE

An object that takes the `x` position of another object on mouse down:

```
/ITL/scene/myObject watch mouseDown
                   (/ITL/scene/myObject x '$(/ITL/scene/obj get x)');
```

15.2.5 OSC address variables

The OSC address of a message associated to an event supports the following variables:

- `$self`: replaced by the object name.
- `$scene`: replaced by the scene name.

EXAMPLE

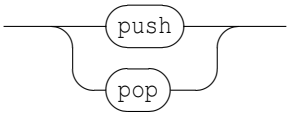
Requesting a set of objects to send a message to themselves on a mouse event:

```
/ITL/scene/* watch mouseDown               !  request all the objects of the scene
                (/ITL/scene/$self x '$sx'); !  to send a message to themselves
```

15.3 Interaction state management

For a given object, its *interaction state* (i.e. the watched events and the associated messages) can be saved and restored.

stateMsg

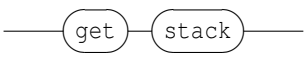


Interaction states are managed using a stack where the states are pushed to or popped from.

- `push`: push the current interaction state on top of the stack.
- `pop`: replace the current interaction state with the one popped from the top of the stack.

The different states stored in this stack can be obtain with the message :

stackMsg



NOTE

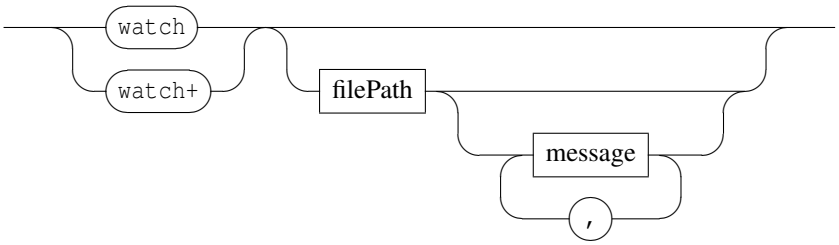
The effect of a pop message addressed to an object with an empty stack is to clear the object current interaction state.

15.4 File watcher

The `fileWatcher` is a static node of a scene that is intended to watch file modifications.
It receives messages at the address `/ITL/scene/fileWatcher`.

The `fileWatcher` support the `watch` and `watch+` messages as described in section 15 p.61 with a file name used in place of the `what` parameter.

fileWatcher



EXAMPLE

Reload aa INScore script on file modification:

```
/ITL/scene/fileWatcher wach 'myScript.inscore'
                  ( /ITL/scene load 'myScript.inscore' );
```

Chapter 16

Scripting

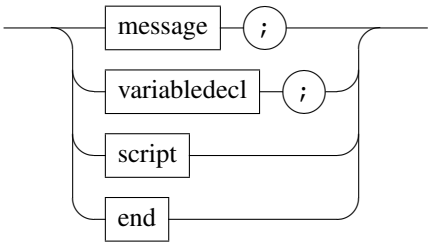
INScore saves its state to files containing textual OSC messages. These files can be edited or created from scratch using any text editor. In order to provide users with a scripting language, the OSC syntax has been extended at textual level.

16.1 Statements

An INScore file is a list of textual expressions. A script expression is:

- a message: basically a textual OSC message extended to support URL like addresses and variables as parameters.
- a variable declaration.
- a foreign language script that may generate messages as output.
- an end marker '__END__' to declare a script end. After the marker, the remaining part of the script will be ignored.

expression



Messages and variables declarations must be followed by a semicolon, used as statements separator.

16.2 Messages

Messages are basically OSC messages that support the address extension scheme described in section 15 p.61 and relative addresses that are described below. Messages parameters can be replaced by variables that are evaluated at parsing level. Variables are described in section 16.4.

Using the address extension scheme, a script may be designed to initialize an INScore scene and external applications as well, including on remote hosts.

EXAMPLE

Initializing a score and an external application listening on port 12000 and running on a remote host named `host.adomain.net`.

```
/ITL/scene/score set gmnf 'myscore.gmn';
host.adomain.net:12000/run 1;
```

Relative addresses have been introduced to provide more flexibility in the score design process. A relative address starts with '`./`'. It is evaluated in the context of the message receiver: a legal OSC address is dynamically constructed using the receiver address that is used to prefix the relative address.

EXAMPLE

```
the relative address   ./score
addressed to           /ITL/scene/layer
will be evaluted as    /ITL/scene/layer/score
```

The receiver context may be:

- the INScore application address (i.e. `/ITL`) for messages enclosed in a file loaded at application level (using the `load` message addressed to the application) or for files dropped to the application or given as arguments of the INScoreViewer application.
- a scene address for messages enclosed in a file loaded at scene level (using the `load` message addressed to a scene) or for files or messages dropped to a scene window.
- any object address when the messages are passed as arguments of an `eval` message (see section 4 p.16).

EXAMPLE

Using a set of messages in different contexts:

```
score = (
   ./score set gmn '[a f g]',
   ./score scale 2.
);
/ITL/scene/l1 eval $score;
/ITL/scene/l2 eval $score;
```

NOTE

Legal OSC addresses that are given as argument of an `eval` message are not affected by the evaluation.

16.3 Types

Using OSC, the message parameters are typed by the OSC protocol. With their textual version, any parameter is converted to an OSC type (i.e. int32, float or string) at parsing level. A special attention must be given to strings in order to discriminate addresses and parameters. Strings intended as parameters must:

- be quoted, using single or double quotes. Note that an ambiguous quote included in a string can be escaped using a '`\`'.
- or make use of the following characters set: `[-a-zA-Z0-9]+` or `[_a-zA-Z][_a-zA-Z0-9]*`.

EXAMPLE

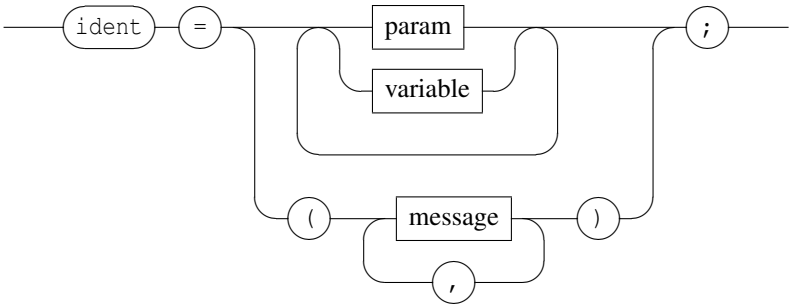
Different string parameter

```
/ITL/scene/text set txt "Hello world"; !  string including a space must be quoted
/ITL/scene/img set file 'anImage.png'; !  dots must be quoted too
/ITL/scene/foo set txt no_quotes_needed;
```

16.4 Variables

A variable declaration associates a name with a list of parameters or a list of messages. Parameters must follow the rules given in section 16.3. They may include previously declared variables. A message list must be enclosed in parenthesis and a comma must be used as messages separator.

variabledecl



EXAMPLE

Variables declarations

```
color = 200 200 200;
colorwithalpha = $color 100; !  using another variable
msgsvar= (                   !  a variable refering to a message list
       localhost:7001/world "Hello world",
       localhost:7001/world "how are you ?" );
```

A variable may be used in place of any message parameter. A reference to a variable must have the form `$ident` where `ident` is a previously declared variable. A variable is evaluated at parsing level and replaced by its content.

EXAMPLE

Using a variable to share a common position:

```
x = 0.5;
/ITL/scene/a x $x;
/ITL/scene/b x $x;
```

Variables can be used in interaction messages as well, which may also use the variables available in the interaction context (see section 15.2 p.65). To differentiate between a *script* and an *interaction* variable, the latter must be quoted to be passed as strings and to prevent their evaluation by the parser.

EXAMPLE

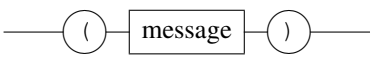
Using variables in interaction messages: $sx is evaluated at event occurrence and $y is evaluated at parsing level.

```
y = 0.5;
/ITL/scene/foo watch mouseDown (/ITL/scene/foo "$sx" $y);
```

16.5 Message based parameters

Similarly to message based variables (see section 15.2.4 p.66), a message parameter may also use the result of a `get` message as parameters specified like a message based variable. The message must be enclosed in parameters with a leading $ sign.

msgparam



EXAMPLE

Displaying INScore version using a message parameter:

```
/ITL/scene/version set txt "INScore version is" $(/ITL get version);
```

NOTE

Message based parameters are evaluated by the parser. Thus when the system state is modified by a script before a message parameter, these modifications won't be visible at the time of the parameter evaluation because all the messages will be processed by the next time task. For example:

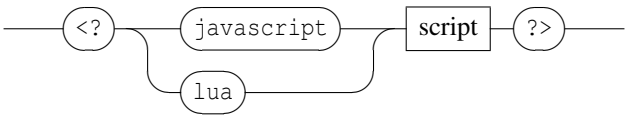
```
/ITL/scene/obj x 0.1;
/ITL/scene/foo x $(/ITL/scene/foo get x);
```

x position of `/ITL/scene/foo` will be set to x position of `/ITL/scene/obj` at the time of the script evaluation (that may be different to 0.1).

16.6 Languages

INScore supports Javascript and Lua as scripting languages. Javascript is embedded by default (using the Qt Javascript engine). INScore needs to be recompiled to embed the Lua engine[1]. A script section is indicated similarly to a Javascript section in html i.e. enclosed in an opening `<?` and a closing `?>`.

script



The principle of using an embedded programming language in script files is the following: *javascript* or *lua* sections are given to the corresponding engine and are expected to produce INScore messages on output. These messages are then parsed as if replacing the corresponding script section.

Note that INScore variables are exported to the current language environment.

EXAMPLE

```
<?javascript
  "/ITL/scene/version set 'txt' 'Javascript v." + version() + "';";
 ?>
```

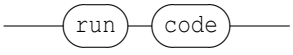
A single persistent context is created at application level and shared to each scene.

NOTE Lua support is going to be deprecated and should be removed in a future release.

16.6.1 The Javascript object

The Javascript engine is available at runtime at the address `/ITL/`*`scene`*`/javascript`. It has a `run` method that takes a javascript string as parameter.

javascript



[1] http://www.lua.org/

The `run` method evaluates the code. Similarly to javascript sections in scripts, the output of the evaluation is expected to be a string containing valid INScore messages that are next executed. Actually, including a javascript section in a script is equivalent to send the `run` message with the same code as parameter to the javascript object.

The Javascript engine is based on the Qt5 Javascrip engine, extended with additional functions:

- **`version()`** : gives the javascript engine version number as a string.
- **`print(val1 [, val2 [, ...]])`** : print the arguments to the OSC standard output. The arguments list is prefixed by 'javascript:'. The function is provided for debug purpose.
- **`readfile(file)`** : read a file and returns its content as a string. The file name could be specified as an absolute or relative path. When relative, the file is searched in the application current `rootPath` (see section 8.1 p.34).
- **`post(address [,...])`** : build an OSC message and post it for delayed processing i.e. to be processed by the next time task. `address` is an OSC or an extended OSC address. Optional arguments are the message parameters.
- **`osname()`** : gives the current operating system name. Returned value is among `"MacOS"`, `"Windows"`, `"Linux"`, `"Android"` and `"iOS"`.
- **`osid()`** : gives the current operating system as a numéric identifiant. Returned value is (in alphabetic order):
  - 1 for Android
  - 2 for iOS.
  - 3 for Linux,
  - 4 for MacOS,
  - 5 for Windows,

EXAMPLE 1

```
<?javascript
  post ("/ITL/scene/obj", "dalpha", -1);";
  // The message /ITL/scene/obj dalpha -1
  // will be evaluated by the next time task.
?>
```

EXAMPLE 2

```
<?javascript
  // declare a function foo()
  function foo(arg) {
    return "/ITL/scene/obj set txt foo called with " + arg + ";";
  }
?>

// call the foo function
<?javascript foo(1)?>

// or call the foo function using the run message
/ITL/scene/javascript run "foo(1)";
```

Chapter 17

Score expressions

Score expressions allows to defines score objects (`gmn` or `pianoroll`) by dynamically combine various resources using a formal expression. To define such object one should use the basic `set` messages using a score expressions as arguments:

EXAMPLE

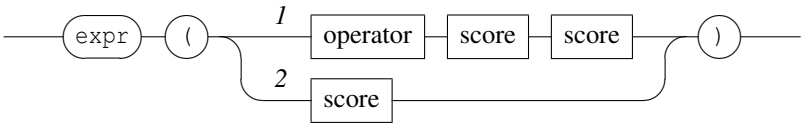
The following example defines a `gmn` and a `pianoroll` object using score expressions, the meaning of the expression is explained further.

```
/ITL/scene/score set gmn expr(seq [a] [b]);
/ITL/scene/pianoroll set pianoroll expr(score);
```

17.1 General Syntax

A score expression always starts with `expr(` and ends with `)`, then 2 syntaxes are handled:

EvaluableExpression



- **1**: Define an expression as an operation combining two scores. `operator` is the name of the operation used to combine them (see Section 17.2 for operators list), and `score` are the arguments passed to the operator (see Section 17.3 for arguments specification).
- **2**: Define on expression using a single score. This syntax is useful when defining an object as a dynamic copy of an other existing object or file.

Each of these tokens can, of course, be separated by spaces, tabulations or carriage returns (allowing multiline expression definition).

When defining an object using a score expressions, INScore will parse it, construct an internal representation and finally evaluate it, reducing the formal expressions to a valid GMN string.

EXAMPLE

Creating a guido object by sequencing two guido string

```
/ITL/scene/score set gmn expr( seq "[c d e]" "[f g h]");
```

is equivalent to

```
/ITL/scene/score set gmn "[c d e f g h]";
```

17.2 Score Operators

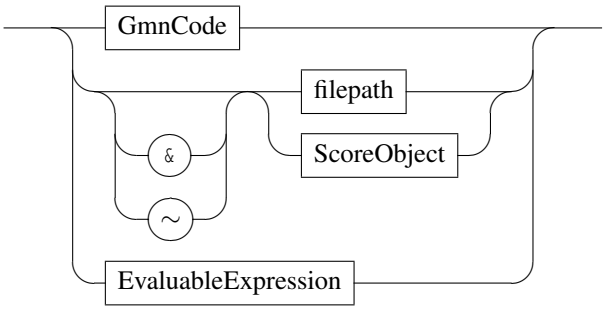
All the score operators of INScore make use of guido operators implemented in the GuidoAR library.

operation	arguments	description
seq	*s*1 *s*2	puts the scores *s*1 and *s*2 in sequence
par	*s*1 *s*2	puts the scores *s*1 and *s*2 in parallel
rpar	*s*1 *s*2	puts the scores *s*1 and *s*2 in parallel, right aligned
top	*s*1 *s*2	takes the *n* first voices of *s*1 where *n* is *s*2 voices count
bottom	*s*1 *s*2	cut the *n* first voices of *s*1 where *n* is *s*2 voices count
head	*s*1 *s*2	takes the head of *s*1 up to *s*2 duration
evhead	*s*1 *s*2	takes the *n* first events of *s*1 where *n* is the event's count of *s*2
tail	*s*1 *s*2	cut the beginning of *s*1 up to the duration of *s*2
evtail	*s*1 *s*2	cut the *n* first events of *s*1 where *n* is the event's count of *s*2
transpose	*s*1 *s*2	transposes *s*1 so its first note of its first voice match *s*2 one
duration	*s*1 *s*2	stretches *s*1 to the duration of *s*2 if not used carefully, this operator can output impossible to display rhythm
pitch	*s*1 *s*2	applies the pitches of *s*1 to *s*2 in a loop
rhythm	*s*1 *s*2	applies the rhythm of *s*1 to *s*2 in a loop

17.3 Score Arguments

The syntax for arguments is quite permissive and various resources can be used as arguments for score expressions. In any case, when evaluating the expression, all the arguments will be reduce to GMN string so they can then be processed by the operators.

Argument



Arguments specification

- `GmnCode` are not evaluated, passed as they are to operators. Both GMN and MusicXML string are supported.
- `filepath`: on evaluation INScore read all the content of the file. Again, both GMN and MusicXML are supported. `filepath` handle absolute or relative path (from the scene rootPath) as well as url.
- `ScoreObject`: Gmn code can be retreive from existing score objects (`gmn` or `pianoroll`) simply refering to them using their identifier (using absolute or relative path).
- `EvaluableExpression`: an expression can also be used as an argument, thus simple operator can be combined together to create more complex ones. In that case the `expr` token can be omitted: parenthesis are sufficient.

Arguments prefix

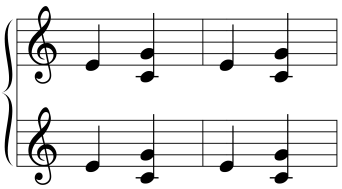
- `&`: When triggering the reevaluation of an expression (see Section 17.4) only the arguments prefixed with `&` are updated.
- `~`: before the first evaluation of a score expression, any `ScoreObjects` prefixed with a `~` shall be replaced by their own expression. In other words, score expressions containing `~` arguments will be expended with existing score expressions. This mechanism allows to compose not only scores and score expressions together.

EXAMPLE

Defining `/ITL/scene/score` as a copy of `/ITL/scene/simpleScore` duplicated 4 times.

```
/ITL/scene/simpleScore set gmn "[e {c,g} |]";

/ITL/scene/score set gmn expr( &simpleScore );
/ITL/scene/score set gmn expr( seq ~score ~score);
/ITL/scene/score set gmn expr( par ~score ~score);
```

`/ITL/scene/score` should look like:



Querying for the expanded expression of `/ITL/scene/score` (see Section 17.4) should return:

```
/ITL/scene/score expr
  expr( par
        ( seq
           &simpleScore
           &simpleScore
        )
        ( seq
           &simpleScore
           &simpleScore
        )
  )
```

NOTE ON ARGUMENTS QUOTING

Arguments using special characters (space, tabulation, parenthesis, braces...), should be simple or double quoted, otherwise quotes can be omitted.

17.4 `expr` commands

ITLObject defined using an evaluable expression gain access to these specific commands:

- `get expr`: return the expression used to define the object (before the expansion of `~` arguments).
- `get exprTree`: return the expanded expression

- `expr reeval`: re-evaluate the expression, updating only the value of arguments prefixed with `&`.
- `expr reset`: re-evaluate the expression, updating the value of all arguments.
- `expr renew`: reapply the definition of the object (similar to send its `set` message again)

Applied to an object which wasn't defined by an evaluable expression, all this commands will cause a bad argument error.

The `renew` command reset the internal state of the evaluated variable, forcing the re-evaluation and update of every arguments in the expression. Be aware that the track of copy evaluated arguments is lost after the first evaluation, thus renewing an expression defined using copy evaluated arguments won't update these arguments to their targeted ITLObject expression. Though, static arguments added by the copy shall be renewed.

17.5 newData event

`newData` is triggered by any object when its value change (generally because of a `set` message). Neither trying to set an object to its actual value without changing its type, nor re-evaluating an object to its actual value will trigger newData.

Of course, the `newData` event can be used together with `reeval` to automatically update an object when the value of an other changes.

EXAMPLE

Creating a copy of `score`, and automatise its update when `score` is changed

```
/ITL/scene/score set gmn "[c e]";
/ITL/scene/copy set gmn expr(&score);
/ITL/scene/score watch newData (/ITL/scene/copy expr reeval);
```

To avoid infinite loop when using recursion, `newData` event is delayed of one event loop, meaning that, in the previous example, during the event loop that follow `score`'s modification, `score` and `copy` are different (`copy` has not been updated yet...).

NOTE

Because newData event is delayed, if `score` experiences multiple modifications during the same event loop (because multiple `set` messages have been sent together), only his final value will be accessible when newData will be actually triggered, however the event will be sent as many times as `score` have been modified.

NOTE WHEN AUTOMATISING UPDATE

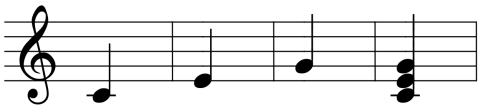
For the reasons raised in the previous note, one should be very careful to delayed update when automatise `reeval` with `newData`. Indeed, in some extreme case, executing a script one line after an other won't have the same result as executing the all script at once!!

EXAMPLE

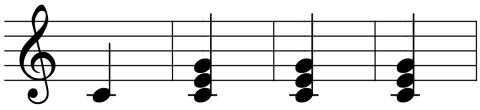
Creating a "score buffer", storing every state adopted by `score`

```
/ITL/scene/score set gmn "[c]";

/ITL/scene/buffer set gmn "[]";
/ITL/scene/buffer set gmn expr(seq &buffer (seq "[|]" &score));
/ITL/scene/score watch newData (/ITL/scene/buffer expr reeval);

/ITL/scene/score set gmn "[e]";
/ITL/scene/score set gmn "[g]";
/ITL/scene/score set gmn "[{c,e,g}]";
```

Won't have the same result if run line by line, or the all script as once:
Line by line:



All script at once:



To avoid such undeterministic behaviour, one should, in this case, manually trigger `reeval` after each modification of `score`.

Chapter 18

Plugins

A plugin is an external library that is dynamically loaded when an object that need it is created. The system looks for plugins in the following locations:

- in the current folder first
- in the PlugIns folder, located in the application bundle on macos, in the application folder on other systems
- in the system default locations for shared libraries

Additionaly, a user path can be set, where the system will look for plugins in first position. See section 8.4.4 p.38.
The plugins are shared libraries which extension is platform dependent. The plugin name should not include the extension. The expected extensions are the following: .dylib on MacOS and Linux, .dll on Windows.

18.1 FAUST plugins

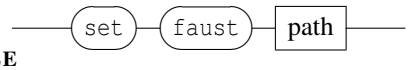
FAUST [Functional Audio Stream][1] is a functional programming language specifically designed for real-time signal processing and synthesis. A FAUST/INScore architecture allows to embed FAUST processors in INScore, for the purpose of signals computation. A FAUST plugin is viewed as a parallel signal and thus it is created in the `signal` address space. Similarly to signals, it is associated to an OSC address in the form `/ITL/`*`scene`*`/signal/`*`name`* where *`name`* is a user defined name.

18.1.1 Set Message

There are two ways to create a FAUST Processor :

1 - By charging a DSP as a plugin already compiled

faustprocessor



EXAMPLE

```
/ITL/scene/signal/myFaust set faust aFaustPlugin;
```

NOTE

The plugin name should not include the extension. The expected extensions are the following: .dylib

[1] http://faust.grame.fr

on MacOS and Linux, .dll on Windows.

2 - By charging libfaust as a plugin to compile a DSP on-the-fly (as a string or a file).

faustdsp

faustdspfile

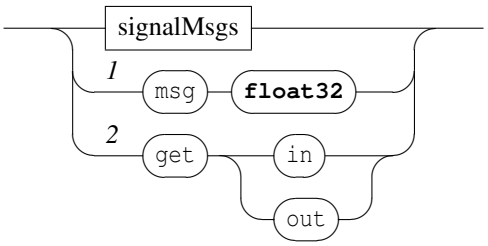
EXAMPLE

```
/ITL/scene/signal/plus set faustdsp "process=+;";
/ITL/scene/signal/mydsp set faustdspf "mydsp.dsp";
```

18.1.2 Specific messages

A FAUST processor is characterized by the numbers of input and output channels and by a set of parameters. Each parameter carries a name defined by the FAUST processor. The set of messages supported by a FAUST processor is the set of signals messages extended with the parameters names and with specific query messages.

faustmessage



1 *msg* is any of the FAUST processor parameters, which are defined by the FAUST processor.

2 the `get` message is extended to query the FAUST processor: `in` and `out` give the number of input and output channels.

EXAMPLE

Querying a FAUST processor input and output count:

```
/ITL/scene/signal/myFaust get in out;
```

gives as output:

```
/ITL/scene/signal/myFaust in 2;
/ITL/scene/signal/myFaust out 4;
```

Modifying the value of a FAUST processor parameter named `volume`:

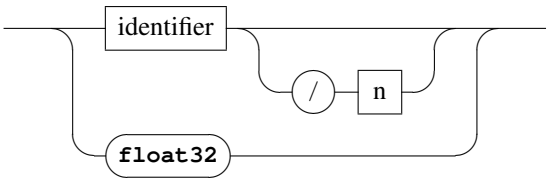
```
/ITL/scene/signal/myFaust volume 0.8
```

18.1.3 Feeding and composing FAUST processors

A FAUST processor accepts float values as input, which are taken as interleaved data and distributed to the input channels.

From composition viewpoint, a FAUST processor is a parallel signal which dimension is the number of output channels. Thus, a FAUST processor can be used like any parallel signal. However, the signal identifier defined in 14.1.2 is extended to support adressing single components of parallel signal as follows:

signal



where `n` selects the signal #n of a parallel signal. Note that indexes start at 0.

EXAMPLE

Creating 3 parallel signals using the 3 output channels of a FAUST processor named `myFaust`:

```
/ITL/scene/signal/y1 set 'myFaust/0' 0.01 0.  1.  1.  1.  ;
/ITL/scene/signal/y2 set 'myFaust/1' 0.01 0.5 1.  1.  1.  ;
/ITL/scene/signal/y3 set 'myFaust/2' 0.01 -0.5 1.  1.  1.  ;
```

18.2 Gesture Follower

INScore supports gesture following using the technology developed by the IRCAM IMTR team. These features are available as a plugin that is included in the INScore distribution (version 1.03 or greater) or available from the IRCAM.

18.2.1 Basic principle

Gesture following is provided as a mean to interact with a score. From input viewpoint, the gesture follower is similar to signals (see section 14.1.1 p.54): it accepts data stream as input both in learning and following modes. It implements a specific set of events related to gesture following and can generate message streams parametrized with the gesture follower current state.

A gesture follower is setup to handle a given count of gestures, which are actually denoted by streams of float vectors. We'll refer to the size of the float vector as the *gesture dimension*. For example, the dimension of a gesture captured from x, y and z accelerometers is 3.

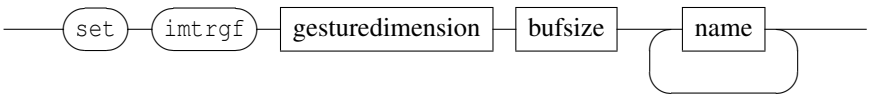
A gesture follower operates in two distinct phases: a *learning phase* where it actually stores the gestures data, and a *following phase* where it tries to match incoming data to the stored gestures data. When not learning nor following, we'll talk of an idle phase.

In the *following phase*, the system maintains a list of likelihood for the learned gestures, a list of positions in the gestures and a list of speeds representing how fast the gestures are made. Of course, the higher the likelihood, the more these data are meaningfull. It's the user responsability to decide on the meaningfull likelihood threshold value. Interaction events are triggered only in the *following phase* and for meaningfull likelihoods.

18.2.2 Messages

A gesture follower is created in a scene using the `imtrgf` type. It has a graphic appearance that may be used for debug purpose but it is hidden by default.

gesturefollower



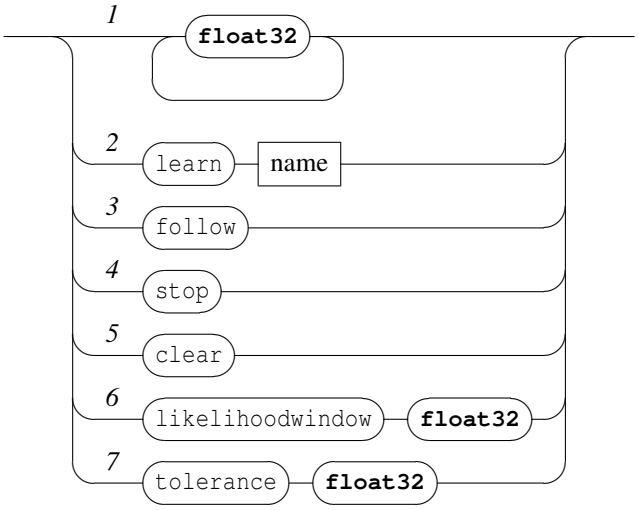
The parameters are:

- `gesturedimension`: the size of the gestures data vector.
- `bufsize`: the size of the gesture data storage.
- `name`: a list of names to be used to refer to the learned gestures.

NOTE

A gesture follower is created with a fixed count of gestures that can be learned and decoded. These gestures are named gestures and can be addressed at `/ITL/`*`scene`*`/`*`myfollower`*`/`*`gesturename`* where the part in italic are user defined names and where `myfollower` is a gesture follower.

gesturefollower



- **[1]** input data into the gesture follower. The data are interpreted according to the current operating mode i.e. learning, following or idle.
- **[2]** starts to learn the gesture designated by *name*. Actually records the next input data to the gesture.
- **[3]** starts following i.e. trying to match the next input data to the recorded gestures.
- **[4]** stops learning or following. Actually puts the system in idle phase.
- **[5]** clear all the gestures data. This is equivalent to send the `clear` message to all the gestures.
- **[6]** sets the size of the window that contains the history of the likelihoods. May be viewed as how fast the likelihoods will change.
- **[7]** sets the follower tolerance.

EXAMPLE

Creating a gesture follower for 3 dimensional data and a typical learning sequence:

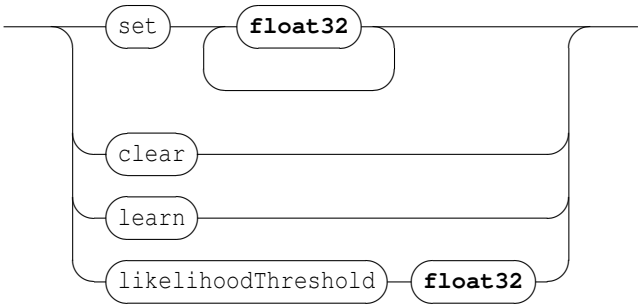
```
/ITL/scene/gf set imtrgf 3 1000 gestureA gestureB gestureC gestureD ;
/ITL/scene/gf learn gestureA ;
/ITL/scene/gf 0.1 0.5 -0.2 ...  0.7; !  the data size must be a multiple of 3
/ITL/scene/gf stop;
```

18.2.3 Gestures management

Messages can also be sent to gestures i.e. to addresses in the form `/ITL/scene/myfollower/gesturename` where `myfollower` is a gesture follower.
A gesture could be in two states:

- an active state: when its likelihood is greater or equal to the likelihood threshold.
- an idle state: when its likelihood is lower than the likelihood threshold.

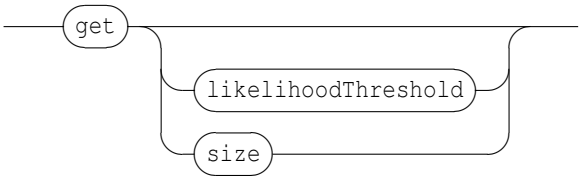
gesture



- `set`: sets the gesture data. This is equivalent to learn the corresponding data. The `set` message could be used to restored previously saved gesture data.
- `clear`: clears the gesture data.
- `learn`: puts the gestures follower in learning mode and starts learning the corresponding gesture. This is equivalent to send OSClearn *gesturename* to the parent gesture follower.
- `likelihoodThreshold`: sets the gesture likelihood threshold. The parameter is a float value in the range `[0,1]`. Default value is `0.5`.

Gestures supports also specific queries :

gestureget

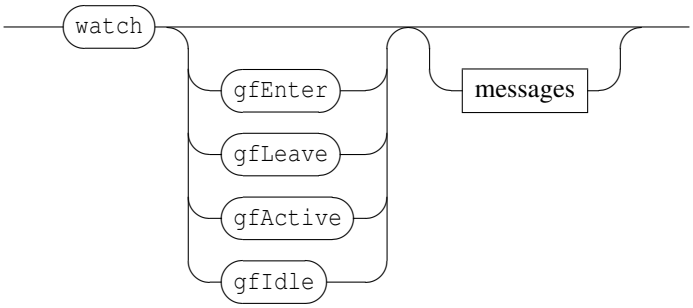


- `get`: without parameter, returns a set message when the gesture is not empty.
- `size`: gives the current size of the gesture, actually the number of recorded frames.

18.2.4 Events and interaction

Events are defined at gesture level and events management messages should be addressed to gestures.

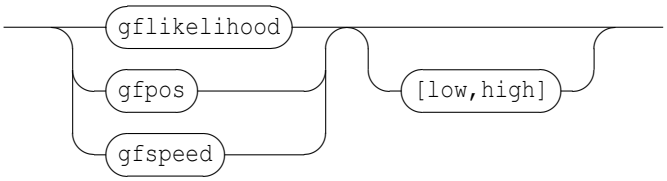
gestureevents



- `gfEnter` triggered when the gesture state changes from idle to active.
- `gfLeave` triggered when the gesture state changes from active to idle.
- `gfActive` triggered in active state each time the gesture likelihood is refreshed.
- `gfIdle` triggered in idle state each time the gesture likelihood is refreshed.

A message associated to a gesture supports the following specific variables:

gesturevariable



These variables support the scaling feature associated to position variables and described in section 15.2.1 p.65.

- `gflikelihood` indicates the current likelihood
- `gfpos` indicates the current position in the gesture
- `gfspeed` indicates the current gesture execution speed

NOTE

Variables described in section 15.2 p.65 may also be used but they are meaningless and contains default values.

18.2.5 Gesture Follower Appearance

A gesture follower object has a graphic appearance and supports all the standard objects properties, including mapping and synchronization. This graphic appearance is provided mainly for debug purpose and by default, the object is hidden. Figure 18.1 shows the gesture follower appearance in its different phases:

- when idle, the upper part of the graphic indicates the buffer state of the different gestures. It also includes the gestures likelihood threshold.
- when learning, a red frame and a grey background indicates that a learning a gesture is currently in progress. The gesture buffer state is refreshed while learning.
- when following, the upper part indicates each gesture current likelihood and the lower part indicates the current estimated positions.

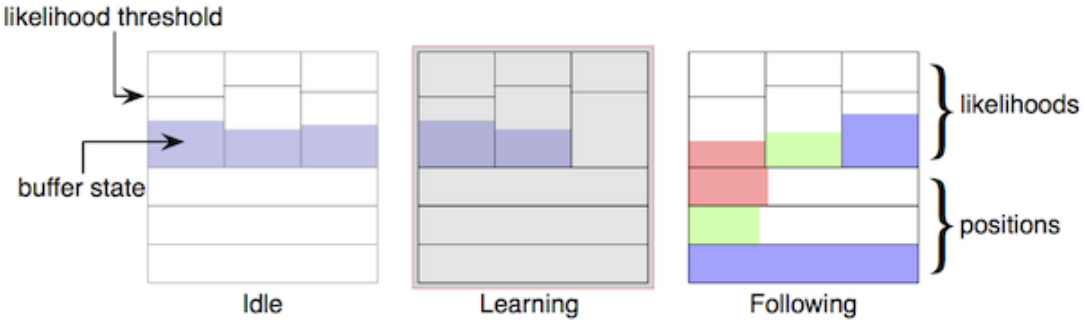


Figure 18.1: The gesture follower appearance in its different phases.

18.3 Httpd server plugin

INScore can embed Http server to expose real time screenshot image of a scene to the web. This feature is based on libmicrohttpd [2] and is available as a plugin that is included in the INScore distribution (version 1.11 or greater). The Url to get the image is the base url of the server.

18.3.1 Set Message

The http server object is created in a scene like other objects and served image of his scene.

httpdserver

- `port` http port used by the server.

EXAMPLE

```
/ITL/scene/server set httpd 8000;
```

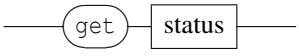
NOTE

If the http port is already used, the server cannot start.

18.3.2 Specific messages

The http server status can be delivered with a specific message.

httpdmessage



A string corresponding to the server status ("started" or "stopped") is return.

EXAMPLE

```
/ITL/scene/server get status;
```

[2] http://www.gnu.org/software/libmicrohttpd/

Chapter 19

Appendices

19.1 Grammar definition

```
//_____________________________________________________
// relaxed simple ITL format specification
//_____________________________________________________
start       : expr
            | start expr
            ;
//_____________________________________________________
// expression of the script language
//_____________________________________________________
expr        : message  ENDEXPR
            | variabledecl ENDEXPR
            | script
            | ENDSCRIPT
            ;
//_____________________________________________________
// javascript and lua support
//_____________________________________________________
script      : LUASCRIPT
            | JSCRIPT
            ;
//_____________________________________________________
// messages specification (extends osc spec.)
//_____________________________________expArg expression___________
message     : address
            | address params
            | address eval LEFTPAR messagelist RIGHTPAR
            | address eval variable
            ;
messagelist : message
            | messagelist messagelistseparator message
            ;
messagelistseparator  : COMMA
                      | COLON
                      ;
```

```
//_____________________________________________________
// address specification (extends osc spec.)
address     : oscaddress
            | relativeaddress
            | urlprefix oscaddress
            ;
oscaddress  : oscpath
            | oscaddress oscpath
            ;
relativeaddress : POINT oscaddress
            ;
oscpath     : PATHSEP identifier
            | PATHSEP WATCH
            | PATHSEP VARSTART varname
            ;
urlprefix   : hostname COLON UINT
            | IPNUM COLON UINT
            ;
hostname    : HOSTNAME
            | hostname POINT HOSTNAME
            ;
identifier  : IDENTIFIER
            | HOSTNAME
            | REGEXP
            ;
//_____________________________________________________
// parameters definitions
// eval need a special case since messages are expected as argument
eval        : EVAL
params      : param
            | variable
            | params variable
            | params param
            ;
variable    : VARSTART varname
            | VARSTART LEFTPAR message RIGHTPAR
            ;
param       : number
            | FLOAT
            | identifier
            | STRING
            | expression
            | LEFTPAR messagelist RIGHTPAR
            | script
            ;
//_____________________________________________________
// variable declaration
variabledecl : varname EQUAL params
            ;
varname     : IDENTIFIER
            | HOSTNAME
            ;
```

```
//_____________________________________________________
// misc
number      : UINT
            | INT
            ;
//_____________________________________________________
// expression declaration
expression  : EXPRESSION
            ;
```

19.2 Lexical tokens

```
//_____________________________________________________
// numbers
//_____________________________________________________
INT       a signed integer
UINT      an unsigned integer
FLOAT     a floating point number

//_____________________________________________________
// hosts addresses
//_____________________________________________________
// allowed character set for host names (see RFC952 and RFC1123)
HOSTNAME       : [-a-zA-Z0-9]+
IPNUM          :  {DIGIT}+"."{DIGIT}+"."{DIGIT}+"."{DIGIT}+

//_____________________________________________________
// OSC addresses
//_____________________________________________________
// allowed characters for identifiers
IDENTIFIER     : [_a-zA-Z][_a-zA-Z0-9]*
REGEXP           see OSC doc for regular expressions

//_____________________________________________________
// strings
//_____________________________________________________
STRING          : ("/"|("."."?"/")*)([^ \t\\/?:*><|"';=]+"/"?)+"."[_a-zA-Z0-9]+
                  or quoted strings that can include any character
                  quotes could be single (') or double quotes (")

//_____________________________________________________
// languages support
//_____________________________________________________
JSCRIPT        : <?javascript any javascript code ?>
LUASCRIPT      : <?lua any lua code ?>

//_____________________________________________________
// misc.
//_____________________________________________________
PATHSEP        : '/'
POINT          : '.'
```

```
VARSTART       : '$'
COLON          : ':'
COMMA          : ','
LEFTPAR        : '('
RIGHTPAR       : ')'
EQUAL          : '='
ENDEXPR        : ';'
ENDSCRIPT      : "__END__"
EVAL           : "eval"

//_____________________________________________________
// score expressions
//_____________________________________________________
EXPRESSION    expr( a valid score expression )
              see 'Score expressions grammar'
```

19.3 Score expressions grammar

```
//_____________________________________________________
// relaxed simple IExpression format specification
//_____________________________________________________
start : expression
;
//_____________________________________________________
//misc
identifier : IDENTIFIER
;
string : identifier
| STRING
| QUOTEDSTRING
;
variable : VARSTART identifier
;
//_____________________________________________________
// expression declaration
expression : EXPR_START operator exprArg exprArg EXPR_END
| EXPR_START exprArg EXPR_END
| EXPR_START operator variable EXPR_END
;
operator : identifier
| variable
;
exprArg : arg
| AMPERSAND arg
| TILDE arg
| expression
;
arg : string
| variable
;
```

```
//_____________________________________________
// special expression chars
//_____________________________________________
AMPERSAND      : '&'
TILDE          : '~'
```

Chapter 20

Changes list

20.1 Differences to version 1.15

- Carlito Regular open source font is embedded in the application ressources and used as a default font. See at https://fontlibrary.org/fr/font/carlito for more information.
- symbolic notation support extended with score expressions. See section 17 p.74.
- new `newData` event. See section 15.1.5 p.64.
- the javascript engine is shared between the application and the different scenes. Note that it may change a script behavior when exploiting the previous independance of the javascript engine environments.
- new javascript `osname` function that gives the current operating system name. See section 16.6.1 p.72.
- new javascript `osid` function that gives the current operating system as an id. See section 16.6.1 p.72.
- `rootPath` message can be called without parameter to clear a scene rootPath. See section 9 p.39.
- log window supports the `foreground` message. See section 8.4.3 p.37.
- user actions on windows are generating `foreground` messages.
- application quit when the last scene is closed (even when the log window is opened)
- new `lock` message supported by all objects to prevent an object deletion. See section 2 p.4.
- OSC output buffer has been enlarged to 32768. Note that sending large messages works on localhost but are likely to face the MTU on real network.
- crash bug corrected: outgoing OSC messages are now handling buffer overflow exceptions.
- support for multi touch events. See section 15.1.2 p.63.
- new `radialgraph` signal representation. See section 14.3 p.58.
- `httpd` object is visible as a qrcode giving the server url.
- `httpd` object is now part of the library (not a plugin any more) (not available on Windows, Android and iOS)
- `frameless` and `fullscreen` modes management revised at view level and are now now exclusive at model level
- String without spaces in INScore scripts no longer need to be quoted.

20.2 Differences to version 1.12

- new `frame` query method: `get frame` gives the coordinates of 4 points that represent the object frame, expressed in the scene local coordinates system and including all the graphic transformations (scaling, rotations on the 3 axis, shear etc.)
- pen messages are now accepted by all the components. Thie extension is provided to display any object bounding box. Note that for rects, ellipses etc. the previous behavior is preserved.
- `pianoroll` support. See section 5.2 p.18 and 7.4 p.30.
- Add web Api to expose inscore on the web with websocket or http.
- Add change tab on mobile with three digits gesture.
- add new object `filter` at application and scene level to filter forwarded messages.
- sending to broadcast address is enabled
- add `forward` and `filter` messages to the scene to handle messages forwarding at scene level. See section 10 p.42.
- default port to forward messages is now 7000.
- add new optional tab at startup with a menu for ios and android.
- add zoom and move capabilities at scene level using `scale`, OSCxorigin and OSCyorigin. This is intended to support two fingers gesture on mobile device.
- bug with lines corrected: a line in non-square parent was rotated when the parent's width was smaller than its height.
- bug with `eval` forwarding corrected: forwarded messages were triggering a syntax error due to a misinterpreted incorrect args count

20.3 Differences to version 1.08

- `line` objects: `color` message is now an alias of `penColor`.
- `foreground` method at scene level to put a scene window in foreground. See section 9 p.39.
- text items support font spec with new `fontSize`, `fontFamily`, `fontStyle` and `fontWeight` messages. See section 7.7 p.32.
- new `compatibility` method at application level, provided to preserve previous behaviors. See section 8.1 p.34.
- default size of guido item is increased: the ratio to the previous size is 8.
- force default size and font to text items in order to get equivelent rendering on different platforms (default to Arial 13px).
- new `arrows` attribute for `line` objects. See section 7.6 p.32.
- the `export` message supports multiple file paths. See section 2 p.4.
- new `exportAll` message to export an object with its children. See section 2 p.4.
- incoming messages buffer size increased to 10.000
- url support for inscore files (`load` message)
- new common queries (`get` message) : `count` and `rcount` that give the enclosed objects count and recursive count. The messages are supported at scene and application level as well. See section 9.2 p.40.
- new `memimg` object that capture the image of any object hierarchy including scenes. See section 5.6 p.21.
- supports relative OSC addresses that are evaluated in the context of the target object (i.e. a scene for

drag and drop operations, arbitrary objects with the `eval` method). See section 16.2 p.69.

- new `eval` method that takes a message list as argument, provided as a context for relative addresses evaluation. See section 4 p.16.
- new `httpd` object that implements an http server providing images of the scene to remote clients. See section 5.10 p.23 and section 18.3 p.85.
- new `websocket` object that implements a websocket server providing images of the scene to remote clients but also changes notifications. See section 5.10 p.23.
- Files objects can receive URL as path. See section 5.8 p.22.
- new intermediate object for the URL (waiting for the data to be downloaded to create the real object)
- new events associated to url based objects: success, error, cancel. See section 15.1.4 p.64.
- support for int values as parameters of the set method of rect shape and polygon objects
- the `clear` message addressed to a `gmnstream` object clears also the view. The change was not previously reflected until a new valid string was posted to the object.
- bug in export item corrected : child scaling was not applied.
- bug correction: for multiple exports, only the last one was done.
- bug in extended address support corrected: extended address was ignored for messages dropped to a scene .
- bug in window color corrected: black color was not correctly set due to an incorrect color information returned by Qt.
- bug with 'line' initialization corrected: wrong position and orientation with negative coordinates (was previously corrected but reintroduced), incorrect initialization in layers.

20.4 Differences to version 1.07

- new `__END__` marker supported to end a script parsing at arbitrary location (see section 16.1 p.69).
- when displaying the mapping, the map dates are not printed any more by default (due to size and collisions). The debug map parameter change from boolean to int value: 1 to activate the mapping display, 2 to have also the dates displayed (see section 7.8 p.33).
- the signal node is available at any level of the hierarchy (as well as the sync node)
- new `connect` and `disconnect` messages for the signal node to support signal connection to objects graphic attributes (see section 14.2 p.56).
- a slave can have several masters
- no more side effects for synchronized objects (position change, scaling)

20.5 Differences to version 1.06

- bug with 'line' initialization corrected: wrong position and orientation with negative coordinates.
- new `plugins` static node at application level to provide a user path to look for pugins (see section 8.4.4 p.38).
- explicit objects for musicxml scores (`musicxml` and `musicxmlf` types) (see section 5.1 p.17).
- new `faustdsp` object, charging libfaust as a plugin to compile faust DSP on-the-fly (see section 5.5 p.21).
- exception catched when sending osc messages: was a potential crash, e.g. in case of `get` message sent to a signal with a large buffer -> out of buffer memory
- new javascript 'post' function for posting delayed messages (see section 16.6.1 p.72)

- new `write` method supported by the 'log' window (see section 8.4.3 p.37)
- variable addresses are evaluated in message based variables
- supports relative rotations on x and y axis

20.6 Differences to version 1.05

- `save` message can now take an optional list of attributes to be saved (see section 2 p.4)
- variables are now evaluated and expanded inside strings. Thus interaction variables can now be passed as argument of javascript functions.
- corrects musicxml-version output
- log window is put to front when the show menu is recalled
- object aliases are removed when the object is deleted

20.7 Differences to version 1.03

- incorrect error message for watch messages corrected
- new javascript `readfile` function (see section 16.6.1 p.72)
- log window is now available from the application 'Tools' menu
- new `brushStyle` attribute (see section 7.1 p.27)
- new `layer` object (see section 5.7 p.22)
- new `save` method specific to the log window: saves the window content to a file (see section 8.4.3 p.37)
- new `event` method supported at object level for UI events simulation
- new `del` watchable event: sent when deleting an object (see section 15.1.5 p.64)
- new `gmnstream` guido stream object (see section 5.1 p.17)

20.8 Differences to version 1.0

- log window utility provided as a new static node at application level (`/ITL/log`) (see section 8.4.3 p.37).
- new `systemCount` read only attribute for Guido scores (see section 7.3 p.29)
- IRCAM gesture follower support (see section 18.2 p.81)
- javascript engine is available at the static address /ITL/scene/javascript and can be activated using a 'run' method (see section 16.6.1 p.72)
- new `export` event (see section 15.1.5 p.64)
- new `endPaint` event at scene level (see section 15.1.5 p.64)
- new `windowOpacity` method at scene level (see section 9 p.39)
- bug correction: error messages not generated for dropped files (actually for the scene load method)
- bug correction: possible infinite loop in QStretchTilerItem::paint method
- bug correction: incorrect get alias output (all the aliases were dumped out in a single message)

20.9 Differences to version 0.98

- bug correction in streching very small objects (due to approximations)
- bug correction in $sx and $sy computation (xorigin and yorigin was not taken into account)
- new 'ticks' message at application level for querying or setting the current count of time tasks (see section 8.1 p.34)
- new 'time' message at application level for querying or setting the current time (see section 8.1 p.34)
- new 'forward' message at application level for messages forwarding to remote hosts (see section 8.1 p.34)
- new 'relative | absolute' synchronization mode (see section 13.1 p.50)
- 'rename' message not supported any more
- a scene accepts multiple dropped files
- significant extension and syntax changes in inscore script files (see Scripting documentation section 16 p.69)
- fileWatcher methods renamed and simplified (see section 15.4 p.68)
- 'click' and 'select' messages are not supported any more.
- new 'stats' virtual node at application level (address /ITL/stats), supports 'get' and 'reset' messages the node gives statistics about the incoming messages (see section 8.4.2 p.37)
- crash bug in signal creation corrected: a signal size created with an incorrect stream (e.g. a string value) was 0 and no buffer was allocated.
- extension of the time related events to duration: new 'durEnter' and 'durLeave' watchable events (see section 15.1.3 p.63)
- new 'absolutexy' message at scene level to switch to absolute coordinates (in pixels) (see section 9 p.39)
- new 'push' and 'pop' messages to store and restore current watched events and associated messages (see section 15.3 p.67)
- internal change: mappings are now implemented as a separable library strictly complying to the mappings formalism.
- new %f format for the date variable to request a float value (instead a rational value) (see section 15.2.2 p.65).
- dates may be specified as rational strings (see section 3 p.14).
- interaction messages are not any more generated when the date can't be resolved.
- new `rate` message at application level to control the time task rate (see section 8 p.34)
- new `frameless` message at scene level to switch to frameless or normal window (see section 9 p.39)

20.10 Differences to version 0.97

- new `fastgraph` object for graphic signals fast rendering (see section 5 p.17)
- $date variable overflow catched
- files dropped on application icon correctly opened when the application is not running
- supports drag and drop of textual osc message strings
- osc error stream normalized: the message address is 'error:' or 'warning:' followed by a single message string.
- javascript and lua support: a single persistent context is created at application level and for each scene. (see section 16.6 p.72)

20.11 Differences to version 0.96

- objects position, date and watched events preserved through type change
- bug in quantified dates corrected (null denominator set to the quantified value)
- new 'alias' message providing arbitrary OSC addresses support
- bug in parser corrected: \ escape only ' and " chars, otherwise it is literal
- guido score map makes use of the new guidolib extended mapping API for staff and system
- chords map correction (corrected by guido engine)

20.12 Differences to version 0.95

- switch to v8 javascript engine
- lua not embedded by default

20.13 Differences to version 0.92

- new 'mouse' 'show/hide' message supported at application level (see section 8 p.34)
- graphic signal supports alpha messages at object level
- javascript and lua embedded and supported in inscore scripts (see section 16.6 p.72).
- bug correction in sync delete (introduced with version 0.90)

20.14 Differences to version 0.91

- bug corrected: crash with messages addressed to a signal without argument
- date and duration messages support one arg form using 1 as implicit denominator value the one arg form accepts float values (see section 3 p.14).

20.15 Differences to version 0.90

- bug in sync management corrected (introduced with the new sync parsing scheme)

20.16 Differences to version 0.82

- at application level: osc debug is now 'on' by default
- new scripting features (variables) (see section 16.4 p.71).
- ITL file format change:
  - semicolon added at the end of each message
  - '//' comment not supported any more
  - '%' comment char replaced by '!'
  - new variables scripting features
  - single quote support for strings
  - messages addressed to sync node must use the string format
- new 'grid' object for automatic segmentation and mapping

20.17 Differences to version 0.81

- new Faust plugins for signals processing
- colors management change: all the color models (RGBA and HSBA) accept now float values that are interpreted in the common [-1,1] range. For the hue value, 0 always corresponds to 'red' whatever the scale used.
- stretch adjustment for video objects (corrects gaps in sync h mode)
- support for opening inscore files on the command line
- system mapping correction
- splash screen and about menu implemented by the viewer

20.18 Differences to version 0.80

- behavior change with synchronization without stretch: now the system looks also in the slave map for a segment corresponding to the master date.
- `$date` variable change: the value is now (0,0) when no date is available and `$date` is time shifted according to the object date.
- `date` message change: the date 0 0 is ignored

20.19 Differences to version 0.79

- corrects the map not saved by the `save` message issue
- corrects `get map` output: 2D segments were not correctly converted to string

20.20 Differences to version 0.78

- crash bug corrected for the 'save' message addressed to '/ITL'
- message policy change: relaxed numeric parameters policy (float are accepted for int and int for float)
- bug in `get watch` for time events corrected (incorrect reply)

Known issues:

- map not saved by the `save` message

20.21 Differences to version 0.77

- guido system map extended: supports flat map or subdivided map (see section 12.4 p.48).
- new `shear` and `rotate` transformations messages (see section 2.2 p.8).
- new `rename` message to change an object name (and thus its OSC address) (see section 2 p.4).
- relaxed bool parameter policy: objects accept float values for bool parameters
- automatic numbering of exports when destination file is not completely specified i.e. no name, no extension. (see section 2 p.4).
- quantification introduced to $date variable (see section 15.2 p.65).
- `reset` message addressed to a scene clears the scene rootPath

20.22 Differences to version 0.76

- get `guido-version` and `musicxml-version` messages supported by the application (see section 8 p.34).
- `save` message bug correction - introduced with version 0.70: only partial state of objects was saved
- `rootPath` message introduced at scene level (see section 9 p.39).
- scene name translation strategy change: only the explicit 'scene' name is translated by the scene load message handler into the current scene name, other names are left unchanged.
- bitmap copy adjustment in sync stretched mode is now only made for images

20.23 Differences to version 0.75

- new `require` message supported by the `/ITL` node (see section 8 p.34).
- new event named `newElement` supported at scene level (see section 15.1.3 p.63).
- new `name` and `address` variables (see section 15.2 p.65).
- new system map computation making use of the new slices map provided by the guidolib version 1.42
- INScore API: the `newMessage` method sets now the message src IP to localhost With the previous version and the lack of src IP, replies to queries or error messages could be sent to undefined addresses (and mostly lost).
- bug corrected with ellipse and rect : integer graphic size computation changed to float (prevents objects disappearance with small width or height)
- bug in scene export: left and right borders could be cut, depending on the scene size corrected by rendering the QGraphicsView container instead the QGraphicsScene
- crash bug with $date:name corrected: crashed when there is no mapping named `name`.

20.24 Differences to version 0.74

- new `map+` message (see section 12.2 p.47).
- the `click` and `select` messages are deprecated (but still supported). They will be removed in a future version.

20.25 Differences to version 0.63

- new `dpage` message accepted by `gmn` objects (see section 7.3 p.29).
- x and y variables: automatic range type detection (int | float)
- set `txt` message: accepts polymorphic stream like parameters (see section 5 p.17).
- drag and drop files support in INScore viewer
- interaction variables extension: `$sx`, `$sy` variables added to support scene coordinate space (see section 15.2 p.65).
- automatic range mapping for `$x`, `$y` variables.
- new `$self` and `$scene` variables in the address field (see section 15.2.5 p.67).
- OSC identifiers characters set extended with '_' and '-' (see section 1 p.1).
- support for multiple scenes: `new`, `del` and `foreground` messages (see section 9 p.39).
- `load` message supported at scene level (see section 9 p.39).

- `get watch` implemented.
- `watch` message without argument to clear all the watched events (see section 15.2 p.65).
- order of rendering and width, height update corrected (may lead to incorrect rendering)
- bug with gmn score corrected: missing update for page, columns and rows changes.
- package delivered with the Guido Engine version 1.41 that corrects minimum staves distance and incorrect mapping when optimum page fill is off.

20.26 Differences to version 0.60

- new 'mousemove' event (see section 15.1.1 p.62).
- interaction messages accept variables ($x, $y, $date...) (see section 15.2 p.65).
- SVG code and files support (see section 5.9 p.23).
- set line message change: the `x y` form is deprecated, it is replaced by the following forms: `'xy' x y` (equivalent to the former form) and `'wa' width angle` (see section 5 p.17).
- new 'effect' message (section 2.5 p.12).
- utf8 support on windows corrected
- transparency support for stretched synchronized objects corrected
- multiple application instances supported with dynamic udp port number allocation.
- command line option with –port portnumber option to set the receive udp port number at startup.

20.27 Differences to version 0.55

- new 'xorigin' and 'yorigin' messages (section 2.1.3 p.7).
- new interaction messages set (section 15 p.61).
- alpha channel handled by images and video
- bug correction in line creation corrected (false incorrect parameter returned)
- bug correction in line 'get' message handling
- memory leak correction (messages not deleted)

Known issues:

- incorrect graphic rendering when 'sync a b' is changed to 'sync b a' in the same update loop
- incorrect nested synchronization when master is horizontaly stretched,

20.28 Differences to version 0.53

- ITL parser corrected to support regexp in message string (used by messages addressed to sync node)
- format of mapping files and strings changed (section 12.1 p.45).
- format of sync messages extended to include map name (section 13 p.49).
- signal node: 'garbage' message removed
- new 'reset' message for the scene (/ITL/scene) (section 9 p.39).
- new 'version' message for the application (/ITL) (section 8 p.34).
- new 'reset' message for signals (section 14.1.1 p.54).
- bug parsing messages without params corrected

- slave segmentation used for synchronization
- new H synchronization mode (preserves slave segmentation)
- crash bug corrected for load message and missing ITL files

20.29 Differences to version 0.50

- Graphic signal thickness is now symmetrically drawn around y position.
- ITL file format supports regular expressions in OSC addresses.
- IP of a message sender is now used for the reply or for error reporting.
- new `line` object (section 5 p.17).
- new `penStyle` message for vectorial graphics (section 7 p.27).
- new color messages `red, green, blue, alpha, dcolor, dred, dgreen, dblue` (section 2 p.4 and 2.1.2 p.7).
- color values for objects are bounded to [0,255]
- `get map` message behaves according to new map message (section 6 p.25).
- `get width` and `get height` is now supported by all objects (section 6 p.25).
- bug in signal projection corrected (index 0 rejected)
- bug in signals default value delivery corrected
- new `pageCount` message for guido scores
- debug nodes modified state propagated to parent node (corrects the debug informations graphic update issue)
- rational values catch null denominator (to prevents divide by zero exceptions).

20.30 Differences to version 0.42

- `identifier` specification change (section 1 p.1).
- new application `hello` and `defaultShow` messages (section 8 p.34).
- new `load` and `save` messages (sections 8 p.34 and 2 p.4).
- `click` and `select` messages:
  - `rightbottom` and `leftbottom` modes renamed to `bottomright` and `bottomleft`
  - new `center` mode for the `click` message
  - query mode sent back with the reply both for `click` and `select` messages
- new `file`, `html` and `htmlf` types for the `set` message (section 5 p.17).
- `get` syntax change for the `scene`.
- `fileWatcher` messages completely redesigned.
- mappings can be identified by names (section 12.1 p.45).
- rect, ellipse, curve, line and polygon object support graphic to relative-time mapping
- new synchronization modes for Guido scores: voice1, voice2, ... , staff1, staff2, ... , system, page.
- Guido mapping manages repeat bars.
- Graphic signals messages design (section 14.3 p.58).

Index