# INScore
# OSC Messages Reference
# v.1.00

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

December 20, 2012

INScore was initiated in the Interlude project, funded by the French National Research Agency [ANR- 08-CORD-010].

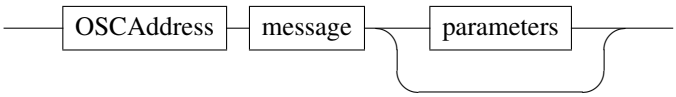
# Contents

# 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**

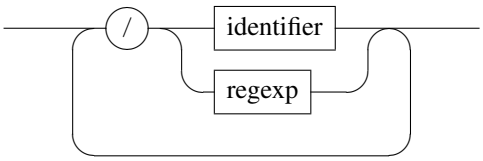
```
/ITL/scene/score x 0.5
```

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.

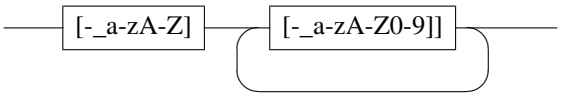
**NOTE** A valid address always starts with `/ITL` that is the application address and that is also used as a discriminant for incoming messages.

*OSCAddress*



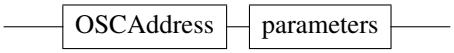

Identifiers may include letters, hyphen, underscore and numbers apart at first position (see lexical definition section 15.2 p.53).

*identifier*




Some specific nodes (like *signals* - see section 11.1.1) 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 14), **`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:

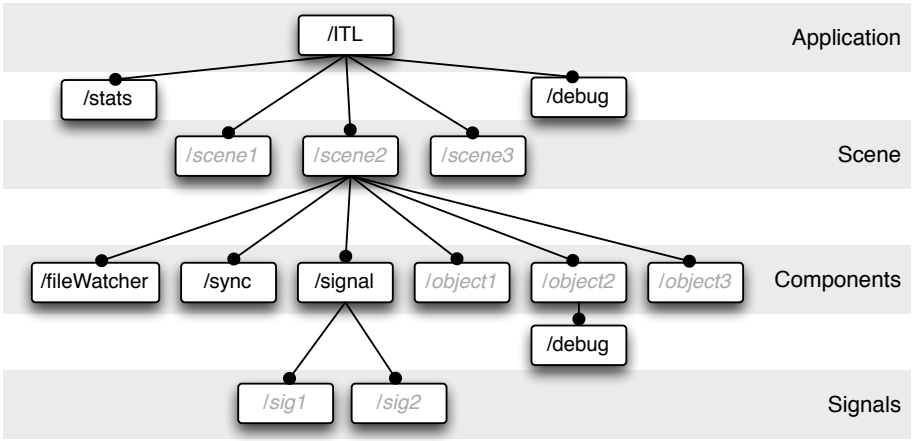



Figure 1.1: The OSC address space. Nodes in italic/blue are dynamic nodes.

OSC messages are accepted at any level of the hierarchy:

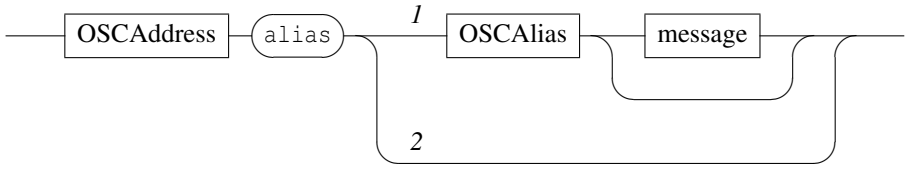
- **the application level** responds to messages for application management (udp ports management, loading files, query messages).
  It includes a static node named `stats` that collects information about incoming messages, and a static `debug` node that can be used to get debug information.
- **the scene level** contains *scores* that are associated to a window and respond to specific scene/window management messages.
- **the component level** contains the scene objects and two static nodes:
  - a *signal* node that may be viewed as a folder containing signals
  - a *sync* node, in charge of the synchronization messages

  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 11 p.35).

## 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`.

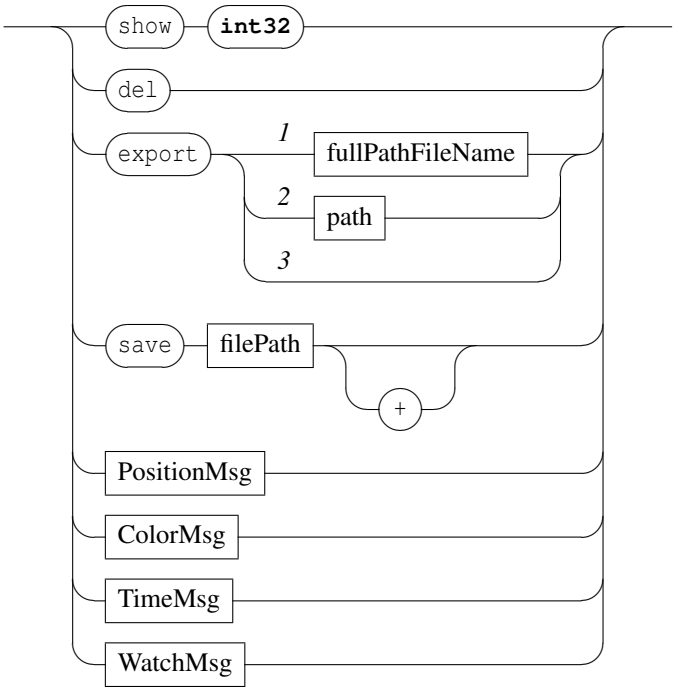
**NOTE** Regular expressions are not supported by the alias mechanism and could lead to unpredictable results.

# 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.
- `export`: exports an object to an image file.
  1) exports to a full path name. 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 `path/identifier.pdf`. When path is a relative path, exports to `rootPath/path/identifier.pdf`.
  3) exports to `rootPath/identifier.pdf`.
  When the destination file is not completely specified (third 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. 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.22 and p.26). 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 that 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 14 p.49).
- '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.12.
- 'WatchMsg' are described in section 13 p.43.

**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 uses the append mode:

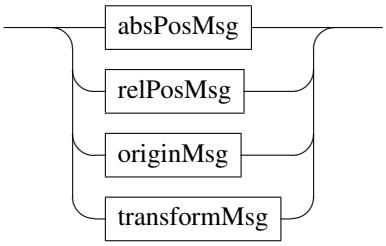
```
/ITL/scene save 'myScore.inscore'
/ITL/scene save 'myScore.inscore' '+'
```

Hiding an object:

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

## 2.1 Positioning

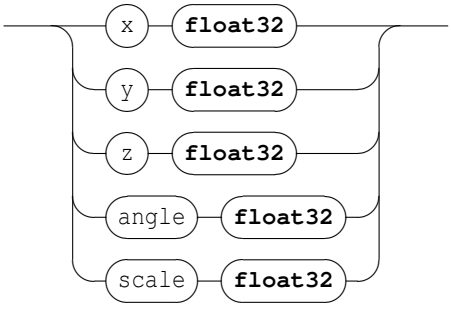
*PositionMsg*




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. The range is $[0,\infty[$. `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. The range is $[0,\infty[$. Default scale is `1`.

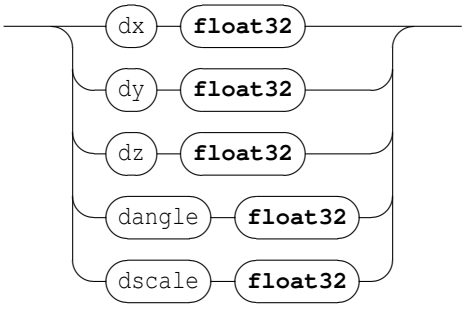
**EXAMPLE**

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.
- `dscale` is similar to `scale` but the parameters represents a scale multiplying factor.

**EXAMPLE**

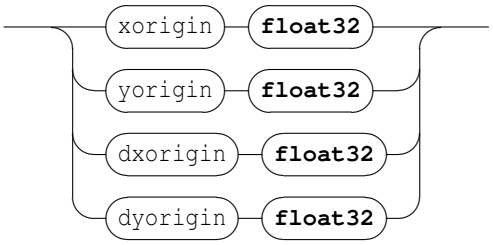
Relative displacement of an object:

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

### 2.1.3 Components origin

The origin of a component is the point $(x_o, y_o)$ such that the $(x, y)$ coordinates and the $(x_o, y_o)$ 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**

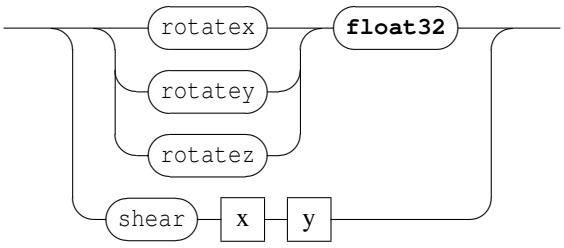
Setting an object graphic origin to the top left corner.

```
/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**

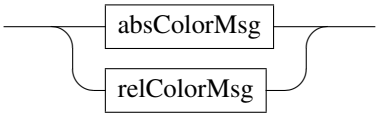
Rotating an object graphic on the `z` axis.

```
/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

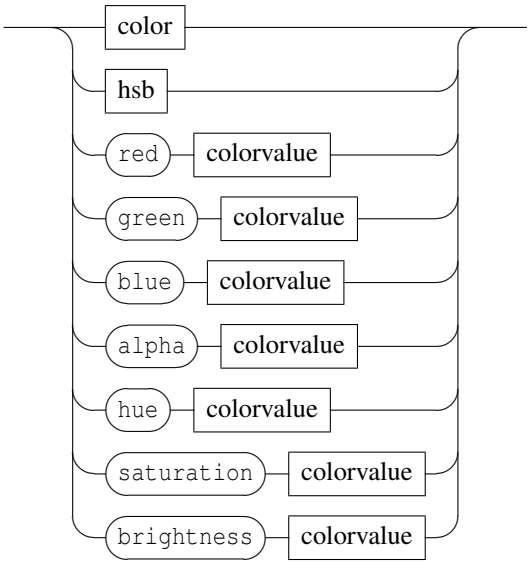
*ColorMsg*




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

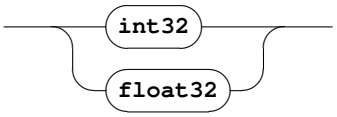
### 2.3.1 Absolute color messages

*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.

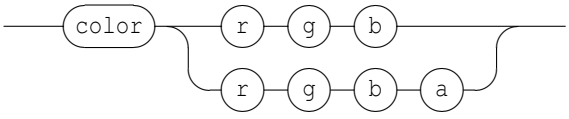
**EXAMPLE**

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*



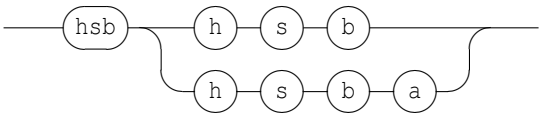
| Component | integer range | float range |
|---|---|---|
| `red` [R] | [0,255] | [-1,1] |
| `green` [G] | [0,255] | [-1,1] |
| `blue` [B] | [0,255] | [-1,1] |
| `alpha` [A] | [0,255] | [-1,1] |
| `hue` [H] | [0,360] | [-1,1] mapped to [-180,180] |
| `saturation` [S] | [0,100] | [-1,1] |
| `brightness`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

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

`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

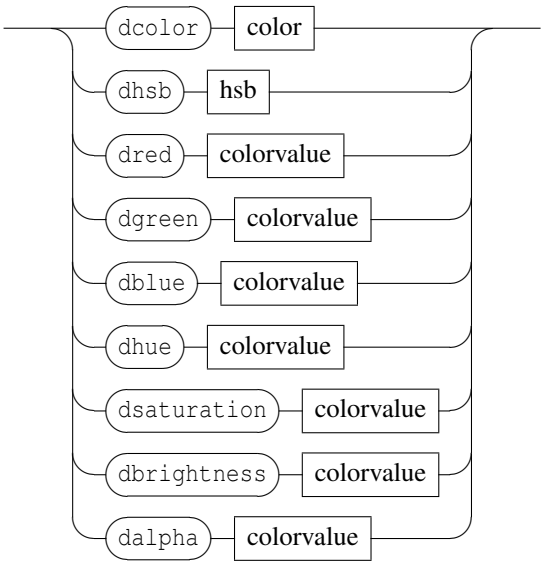
*hsb*




`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
```

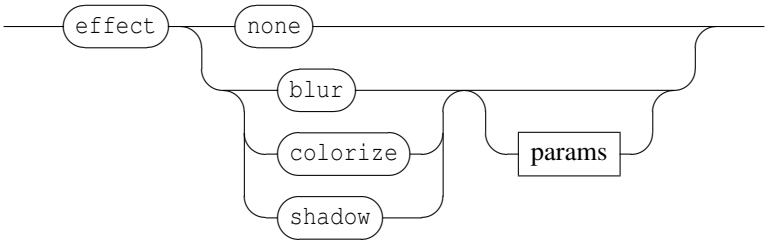
will increase the red component by 10, the blue component by 5, and decrease the transparency by 10.

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

## 2.4 The 'effect' messages

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

*effectMsg*

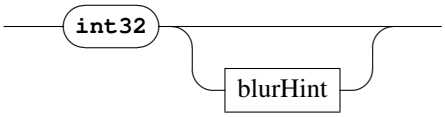



- `none`: removes any effect set on the target object.
- `blur, colorize, shadow`: sets the corresponding effect. An effect always replaces any previous effect. The effect name is followed by optional specific effects parameters.

**NOTE** **A**n effect affects the target object but also all the target slaves.

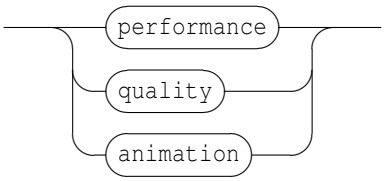
### 2.4.1 The blur effect

*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`.

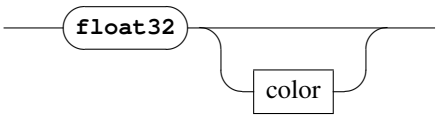
**EXAMPLE**

Setting a 8 pixels effect on `myObject`

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

### 2.4.2 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 2.3 p.8) by default, the color value is light blue (0, 0, 192).

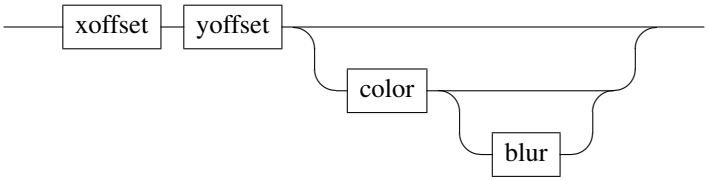
**EXAMPLE**

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

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

### 2.4.3 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 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.

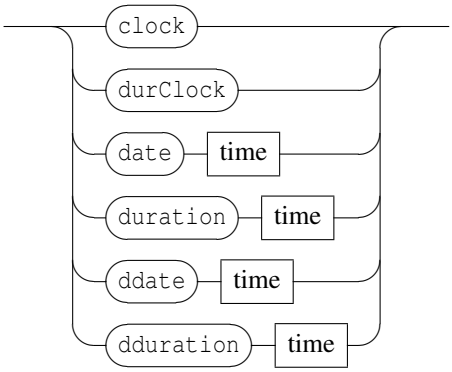
```
/ITL/scene/myObject effect shadow 10 10 100 100 100 50 8
```

# Chapter 3

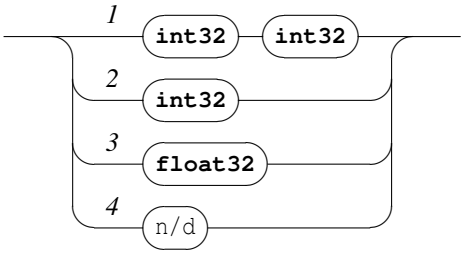
# Time management messages

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.

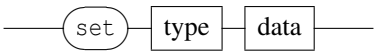
```
/ITL/scene/myObject clock
/ITL/scene/myObject ddate '1/96'
```

# Chapter 4

# 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.

**EXAMPLE**

Setting the content of a text object.

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

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

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

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

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

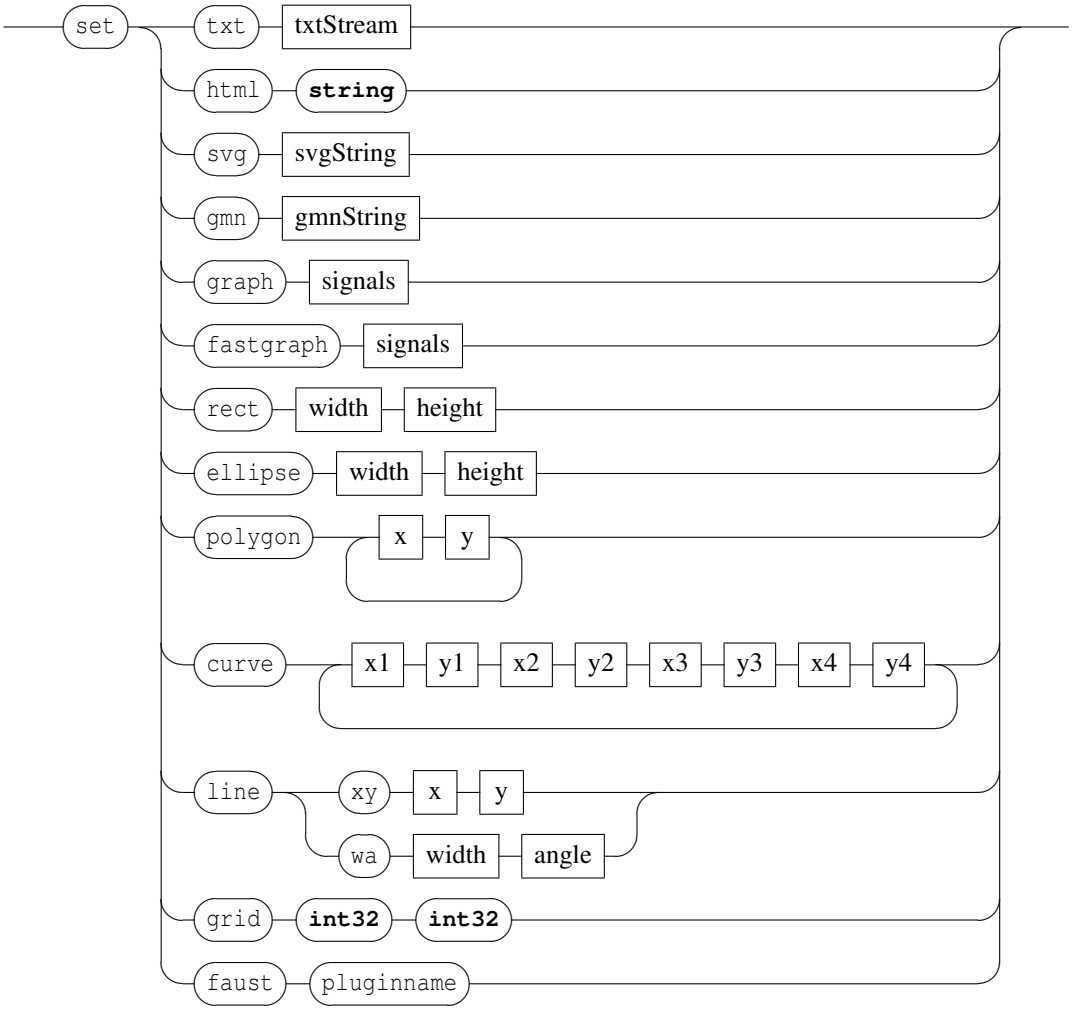
Creating a line specified using width and angle.

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

## 4.1 Inline components

Format of the `set` message is:

*setMsg*




- `txt`: a textual component.
- `html`: an html component defined by an HTML string.
- `gmn`: a Guido score defined by a GMN string.
- `svg`: SVG graphics defined by a SVG string.
- `graph`: graphic of a signal. See section 11 p.35 for details about the `graph` objects data.
- `fastgraph`: fast rendering graphic signal. See also section 11 p.35.
- `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

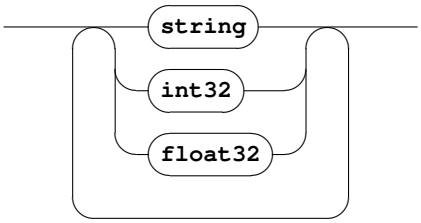
angle. The point form is used to compute a line from (0,0) to (x,y), which is next drawn centered on the scene.

- `grid`: a white transparent object that provides a predefined time to graphic mapping (see section 6.4 p.21 for more details and section 9 p.28 for time to graphic relations). The parameters are int32 values representing the number of columns and rows.

**NOTE** The default position of any component is `[0,0]`. Objects are drawn centered on their position.

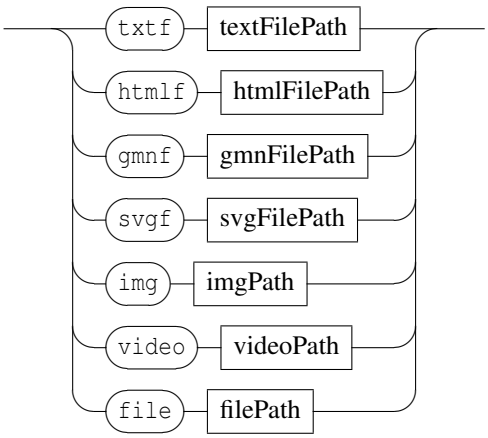
Text may be specified by a single quoted string or using an arbitrary count of parameters that are converted to a single string with a space used as separator.

*txtStream*




## 4.2 File based components

*setFile*




- `txtf`: a textual component defined by a file.
- `htmlf`: an html component defined by an HTML file.
- `gmnf`: a Guido score defined by a GMN file.
- `svgf`: vectorial graphics defined by a SVG file.
- `img`: an image file based component. The image format is infered from the file extension.
- `video`: a video file based component. The video format is infered from the file extension. Note that navigation through the video is made using its `date`.
- `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 4.1).

**See also:** the application `rootPath` message (section 7 p.22) for file based objects.

**EXAMPLE**

Creating an image.

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

Using the `file` type.

```
/ITL/scene/myObject set file "myImage.png"
will be translated into
/ITL/scene/myObject set img "myImage.png"
```

Table 4.1: File extensions supported by the `file` translation scheme.

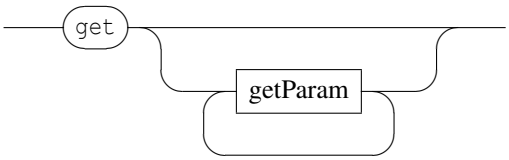
| file extension | translated type |
|---|---|
| .txt .text | txtf |
| .htm .html | htmlf |
| .gmn .xml | gmnf |
| .svg | svgf |
| .jpg .jpeg .png .gif .bmp .tiff | img |
| .avi .wmv .mpg .mpeg .mp4 | video |

# Chapter 5

# 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 section **??** p.**??**).

**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!"
```

**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 drew.

# Chapter 6

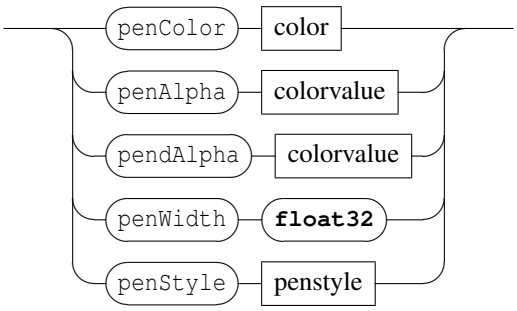
# Type specific messages

Some of the messages are specific to the component `type`.

## 6.1 Pen control

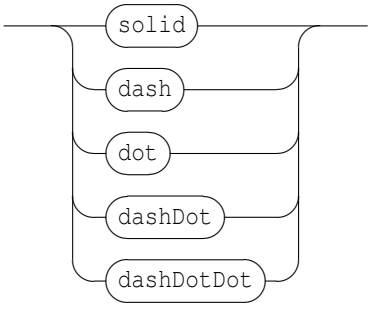
Specific pen messages accepted by the components types `rect | ellipse | polygon | curve | line | graph | fast graph | grid`.

*penMsg*




- `penColor` controls the pen color. The color should be given in the RGBA space. The default value is opaque black (0 0 0 255).
- `penAlpha, pendAlpha` controls the pen transparency only. See section 2.3 p.8 for the expected
- `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).
- `penStyle` controls the pen style.

*penstyle*



The pen style default value is `solid`.

**EXAMPLE**

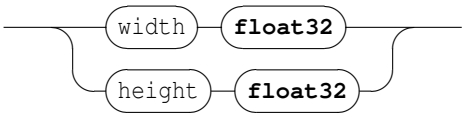
Setting a rectangle border width and color:

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

## 6.2 Width and height control

Specific width and height messages accepted by the components types `rect | ellipse | graph | fastgraph | grid`.

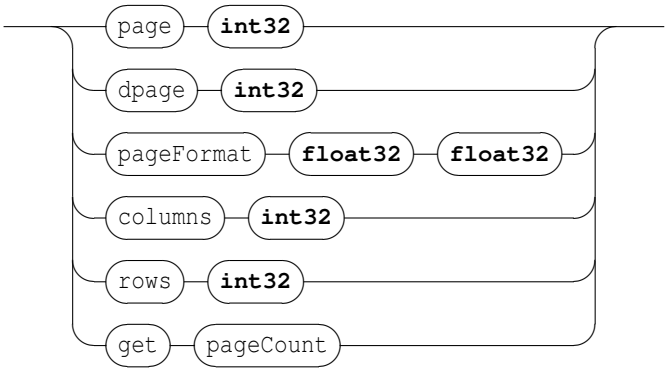
*widthMsg*




## 6.3 Symbolic score management

Messages accepted by the components types `gmn | 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.

**EXAMPLE**

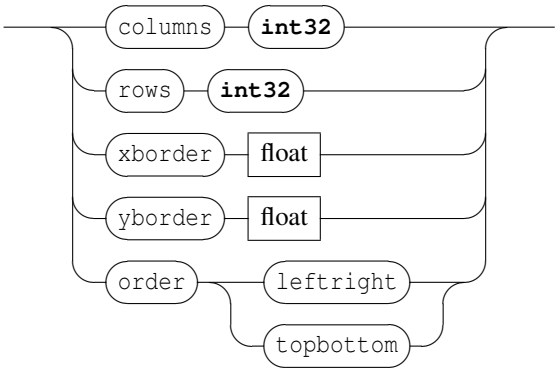
Displaying a multi-pages score on two pages starting at page 3:

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

## 6.4 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**

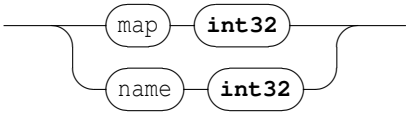
Creating a 10 x 10 grid organized from top to bottom with a border:

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

## 6.5 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 boolean value. Default is 0.
- `name` is used to display both the object name and bounding box. The parameter is a boolean value. Default is 0.

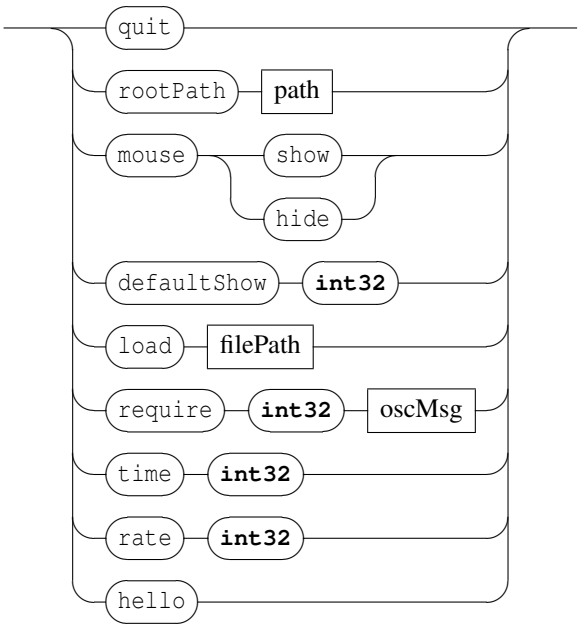
# Chapter 7

# Application messages

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

## 7.1 Application management

*ITLMsg*




- `quit`: requests the client application to quit.
- `rootPath`: *rootPath* of an Interlude 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.

- `require`: check for a version number equal or greater to the number given as argument. The version number should be encoded as an integer value (for example: 76 for version 0.76, 112 for version 1.12). An OSC message is associated to the require message with the same syntax and semantic described in section 13 p.43. This message is triggered when the check fails.
- `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 current rate is internally added to the current time by the time task.
- `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.

**EXAMPLE**

when sending the message:

```
/ITL hello
```

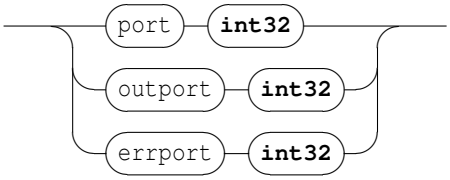
the application will answer with the following message:

```
/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.

## 7.2 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.

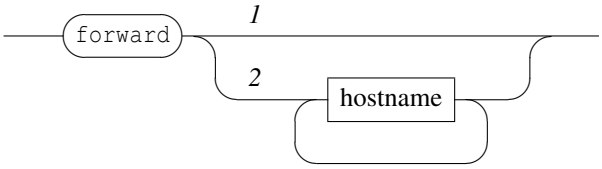
**NOTE**

Error messages are sent as a single string.

## 7.3 Messages forwarding

The messages handled by the application can be forwarded to arbitrary remote hosts using the `forward` message. The `forward` message itself can't be forwarded.

*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 current application listening port number is used.

**EXAMPLE**

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

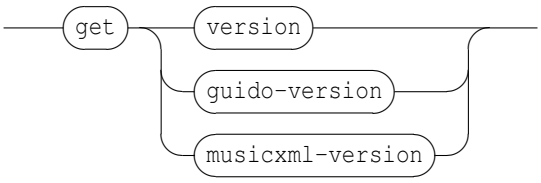
Forwards messages to `host1.adomain.org` using the current application listening port number and to `host2.adomain.org` on port number 5100.

**WARNING:** forwarding messages to the application host results in an infinite loop with unpredictable results. Thus it is not recommended to use the local network broadcast address for forwarding.

## 7.4 Application level queries

The application supports the `get` messages for its parameters (see section 5 p.18). 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
```

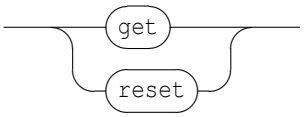
will give the following as output:

```
/ITL version 1.00
```

## 7.5 The 'stats' and 'debug' nodes

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

*ITLStats*



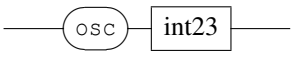

- `get` gives the count of handled messages at OSC and UDP levels: the UDP count indicates the count of messages received from the network, the OSC count includes the UDP count and the messages received internally.
- `reset` resets the counters to zero. Note that querying the `stats` node increments at least the OSC the counter.

**EXAMPLE**

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

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

*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.

**EXAMPLE**

Error messages generated on error port in debug mode:

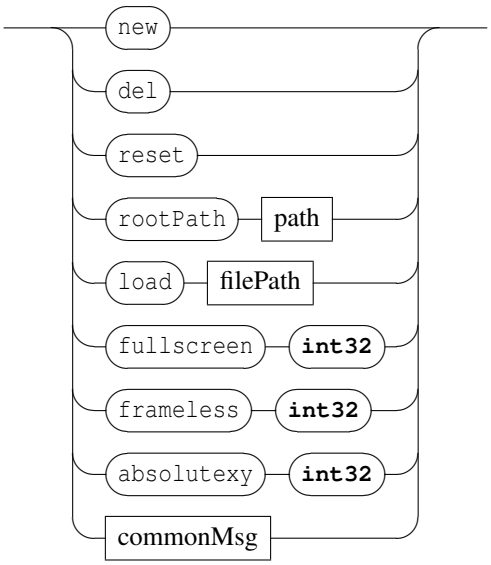
```
error:  incorrect OSC address:  /ITL/stat
error:  incorrect parameters:  /ITL/scene/foo unknown 0.1
error:  incorrect parameters:  /ITL/scene/foo x "incorrectType"
```

# Chapter 8

# 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. It handles the following messages:

*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).
- `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.
- `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`.

- `commonMsg`: a scene support the common graphic attributes. See section 2 p.4.

**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
```

# Chapter 9

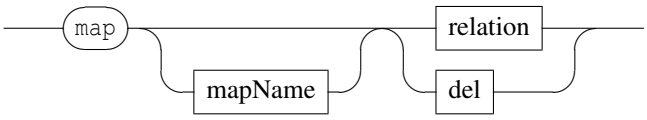
# Mapping graphic space to time space

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.

## 9.1 The 'map' message

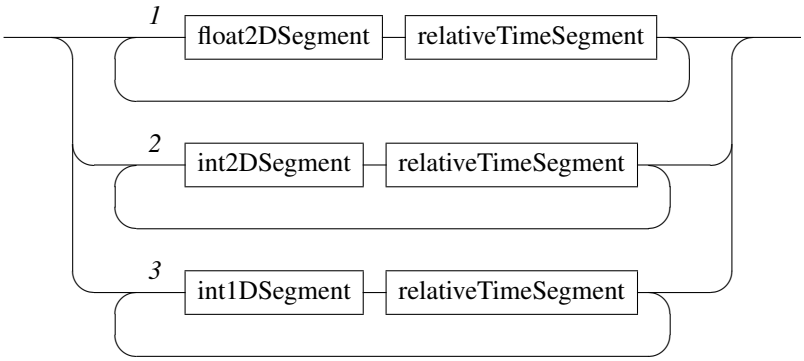
The `map` messages can be sent to any address with the form `/ITL/`*`scene`*`/`*`identifier`*. It is intended to describe the target object relation to time and sets a relation between an object segmentation and a time segmentation. The global form of the message is:

*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 9.1 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*.

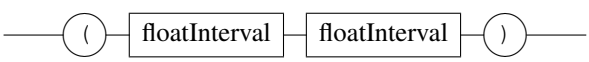
The `del` command deletes the mapping specified with `mapName`, or the *'empty name'* mapping if no map name is specified.

Table 9.1: Local segmentation type for each component

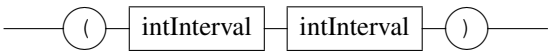
| component type | segmentation type |
|---:|---|
| `txt, txtf` | int2DSegments |
| `img` | int2DSegments |
| `rect, ellipse, polygon, curve` | float2DSegments |
| `graph` | int1DSegments |

*relativeTimeSegment*


*float2DSegment*

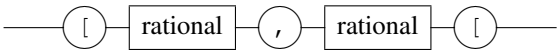



*int2DSegment*

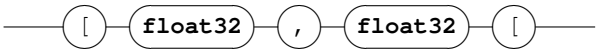



*int1DSegment*

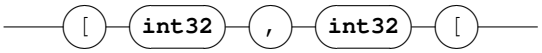

*relativeTimeInterval*



*floatInterval*

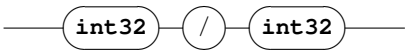



*intInterval*




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).

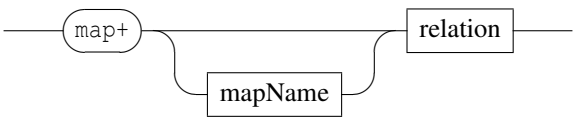
**NOTE ABOUT LOCAL SPACES**

- 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.

## 9.2 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.

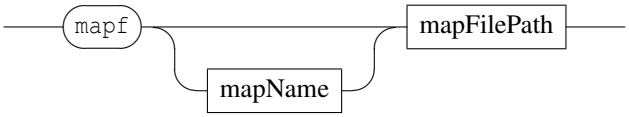
*mapAddMsg*




## 9.3 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.

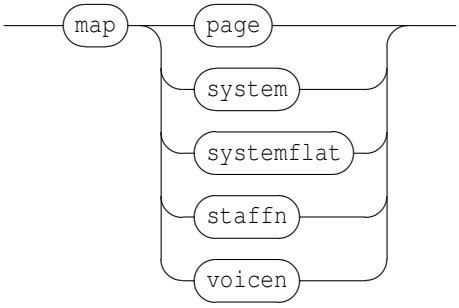
*mapfMsg*




## 9.4 Symbolic score mappings

Mapping between the graphic and time space is automatically computed for symbolic score `gun, 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)
- `staffn`: a staff level mapping: the staff number is indicated by `n`, a number between 1 and the score staves count.
- `voicen`: 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 $3^{rd}$ staff of a score:

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

Requesting the system mapping :

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

**NOTE**

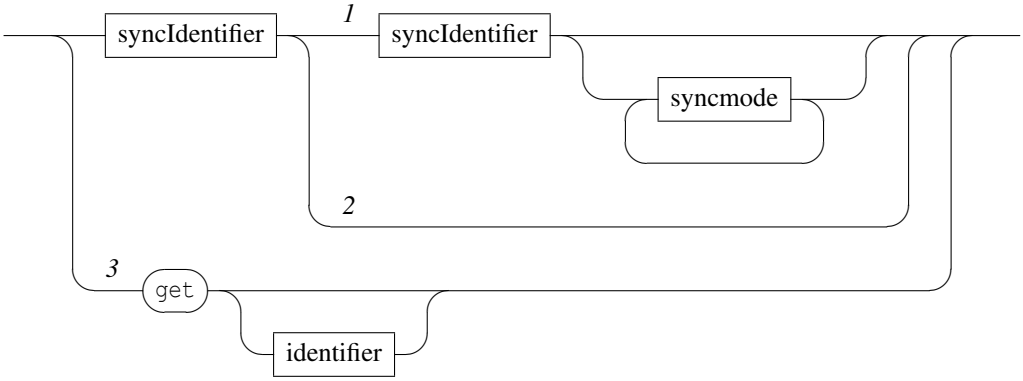
A voice may be distributed on several staves and thus a staff may contain several voices.

# Chapter 10

# Synchronization

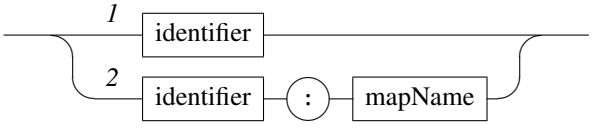
Synchronization between components is in charge of the static `sync` node, automatically embedded in each scene. Its address is `/ITL/`*`scene`*`/sync` and it supports messages to add or remove a master / slave relation between components or to query the synchronizations state.

*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` form without `master` removes a slave synchronization.
- **[3]** 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 a `scene` component or 2) the name of a `scene` component associated to a mapping name. Using the first form (i.e. without explicit mapping name), the system uses the default unnamed mapping (see section 9.1 p.28 mappings and named mappings).

Synchronization between components has no effect if any of the required mapping is missing (see table 9.1).

**NOTE**

A slave object can have only one master. Synchronizing a slave object to a different object replaces the previous relation with the new one.

**EXAMPLE**

Synchronizing two objects:

```
/ITL/scene/sync mySlave myMaster
```

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

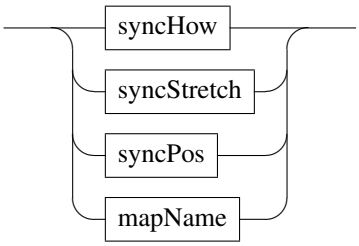
```
/ITL/scene/sync mySlave myMaster:system
```

## 10.1 Synchronization modes

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

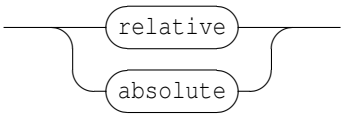
- `A` position (x) is modified to match the `B` time position corresponding to `A` date.
- depending on the optional `syncStretch` option, `A` width and/or height is modified to match the corresponding `B` dimension (see below).
- depending on the optional `syncPos` option, `A` vertical position (y) 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*




### 10.1.1 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: 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.

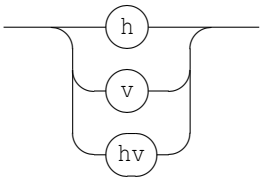
**EXAMPLE**

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

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

### 10.1.2 Representing the objects 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.

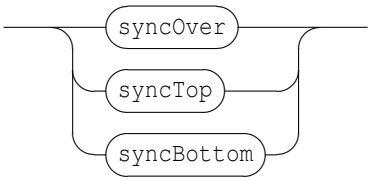
**EXAMPLE**

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
```

### 10.1.3 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`).

**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 11

# Signals and graphic signals

The graphic representation of a signal is approached with *graphic signals*. As illustrated in figure 11.1, 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*.

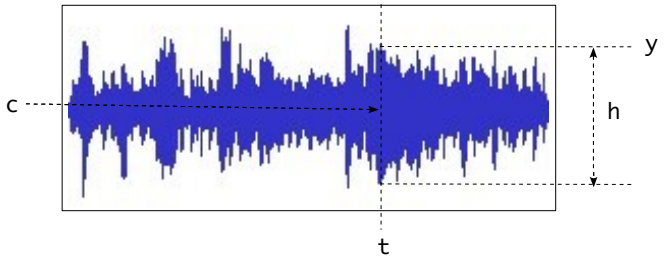



Figure 11.1: A simple graphic signal, defined at time t by a coordinate y, a thickness h and a color c

## 11.1 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 only the `get` message that gives the list of the defined signals.

**EXAMPLE**

Querying the signal node:

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

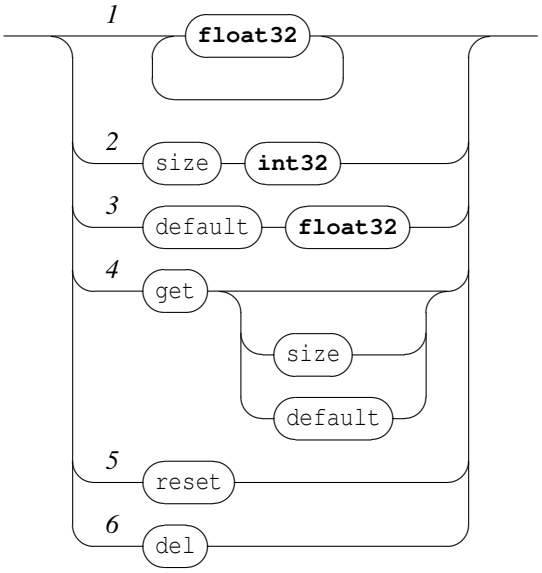
will give the enclosed signals definitions:

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

### 11.1.1 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):

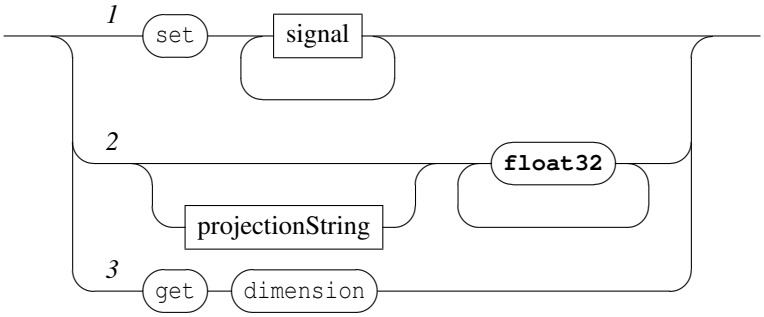
```
/ITL/scene/signal/mySig 0. 0.1 0.2 0.3 0.4 0.5 0.4 0.3 0.2 0.1 0. -0.1 -0.2
```

### 11.1.2 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.

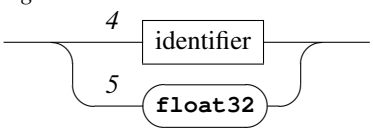
The format of the messages for parallel signals is the following:

*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 11.1.3 p.37.
- **[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**

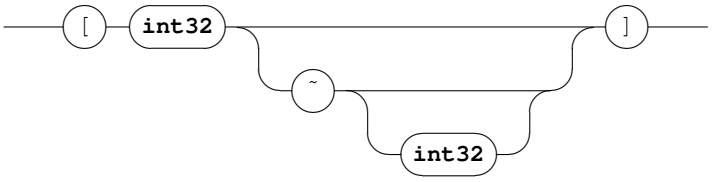
For a parallel signal:

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

### 11.1.3 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.

**EXAMPLE**

Sending data to the second component of a parallel signal:

```
/ITL/scene/signal/sig '[2]' 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 parallel signal:

```
/ITL/scene/signal/sig [2~] 0.1 0.2
```

is equivalent to the following messages (assuming that the signals names are 's2' and 's3'):

```
/ITL/scene/signal/sig [2] 0.1
/ITL/scene/signal/sig [3] 0.2
```

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

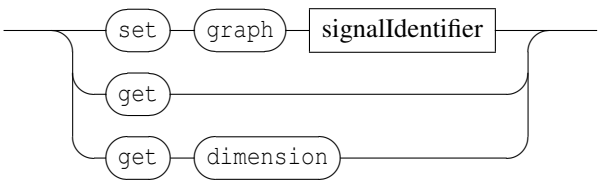
```
/ITL/scene/signal/s2 0.1
/ITL/scene/signal/s3 0.2
```

## 11.2 Graphic signals.

A graphic signal is created in the standard scene address space. A simple graphic signal is defined by a parallel signal controling the `y` value, the graphic 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 11.1.

A graphic signal responds to common component messages (section 2 p.4). In addition, it supports the following messages:

*graphicSignal*



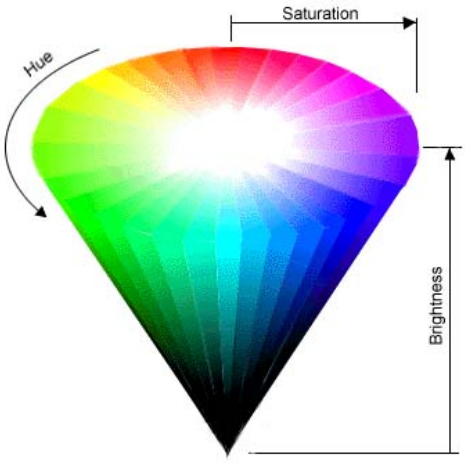



Figure 11.2: The HSB color space

Table 11.1: 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. |

- 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` message is the counterpart of the `set` message (see section 5 p.18).
- the `get dimension` message gives the number of graphic signals in parallel (see section 11.2.2 p.40).

### 11.2.1 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 11.2.

Table 11.2: 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 |

### 11.2.2 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.

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

# Chapter 12

# 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/INScore plugin is viewed as a parallel signal and thus it is created in the `signal` address space. Thus and similarly to signals, a FAUST plugin is associated to OSC addresses in the form `/ITL/`*`scene`*`/signal/`*`name`* where *`name`* is a user defined name.

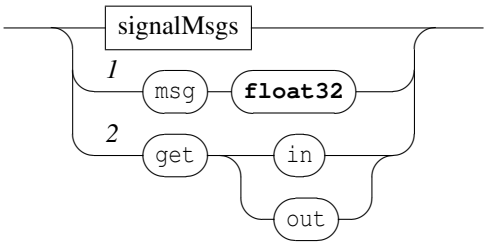
*faustprocessor*


A FAUST processor is created with a `set` message followed by the `faust` type, followed by the plugin path name. Thus the `faust` name is a reserved signal name.

## 12.1 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 program. 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, as defined by the FAUST program.
2 the `get` message is extended to query the FAUST processor: `in` and `out` give the number of input and output channels.

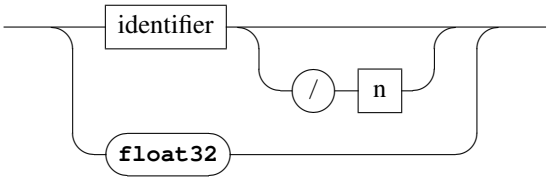
[1] http://faust.grame.fr

## 12.2 Feeding and composing FAUST processors

A FAUST processor accepts float values as input, which are taken as interleaved data.

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 11.1.2 is extended to support adressing single components of parallel signal as follows:

*signal*



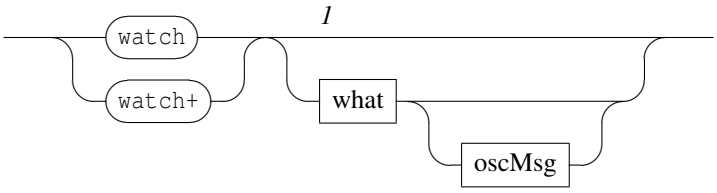

where *n* selects the signal #n of a parallel signal. Note that indexes start at 0.

# Chapter 13

# 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:

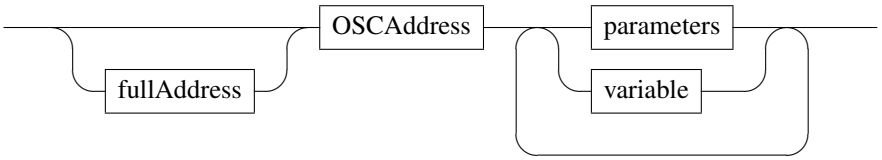
*interactMsg*



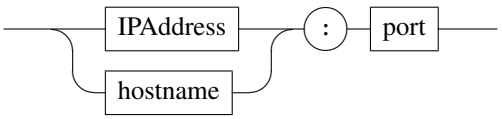

- `watch`: clears the messages associated to the event and when specified, associates the new message to the watched event. `watch` without arguments ([1]) clears all the messages associated to all events.
- `watch+`: adds a message to the watched event message list.

The associated OSC message is any valid OSC message (not restricted to the Interlude 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 variables (see section 13.3).

*oscMsg*



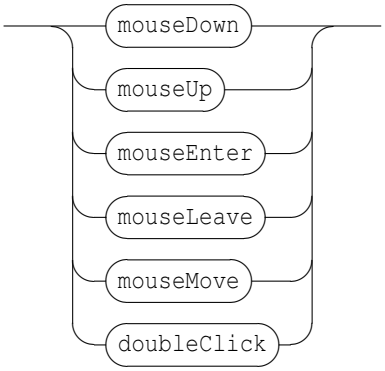

*fullAddress*



## 13.1 Interaction messages

Events currently *watchable* are:

*what*




**EXAMPLE**

```
/ITL/scene/myObject watch mouseDown "/ITL/scene/myObject" "show" 0;
```

Request the object `myObject` to watch `mouseDown` events and send a message to itself.

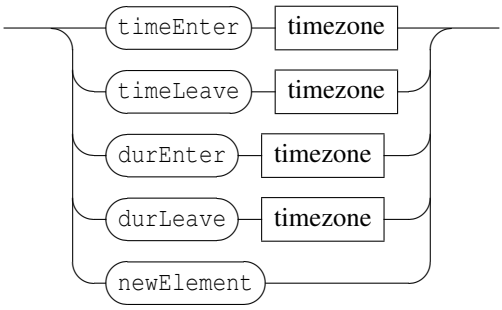
```
/ITL/scene/myObject watch mouseDown "host.domain.org:12100/an/address" "start";
```

Request the object `myObject` to watch `mouseDown` events and send a "start" message to host.domain.org on udp port 12100 to OSC address /an/address

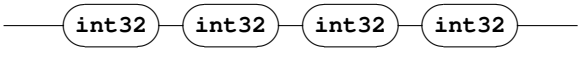
## 13.2 Notification messages

Events currently *watchable* are:

*what*




*timezone*




A time zone is defined by 2 dates expressed as rational values (i.e. with 4 integers).

- `timeEnter`, `timeLeave` are triggered when an object date is moved to or out of a watched `timezone`,
- `durEnter`, `durLeave` are triggered when an object duration is moved to or out of a watched `timezone` that should be viewed as a duration range.

**EXAMPLE**

a cursor that displays a given music page when it enters its time zone.

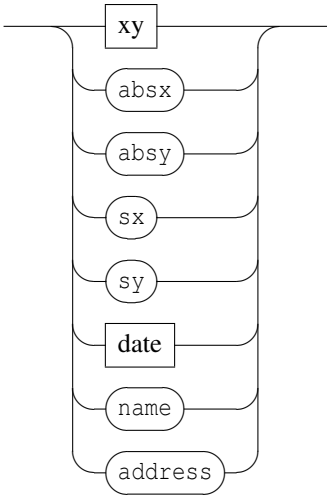
```
/ITL/scene/cursor watch 'timeEnter' 10 1 18 1 '/ITL/scene/score' 'set' 'img' 'page2.png';
```

The `newElement` event is supported at scene level only and triggered when a new element is added to the scene.

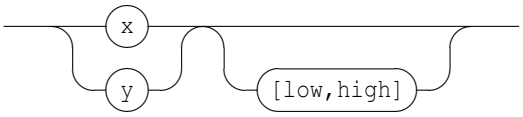
## 13.3 Variables

Variables denote values computed when an event is triggered. These values are send in place of the variable. A variable name starts with a '$' sign. Currently, the following variables are supported by mouse events:

*variable*

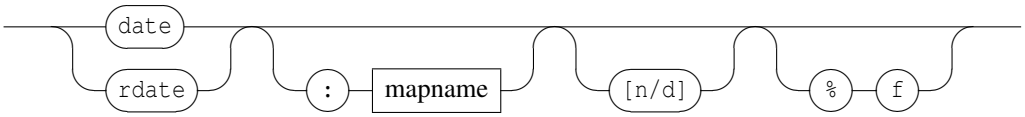



*xy*




- `$x` `$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.6). `$x` and `$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.
- `$name`, `$address`: replaced by the target object name or OSC address. Note that in case of `newElement` event, the target object is the new element.

*date*




- `$date`: denotes the object date corresponding to the mouse pointer position at the time of the event. It is optionnaly followed by a colon and the name of the mapping to be used to compute the date. The `$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.

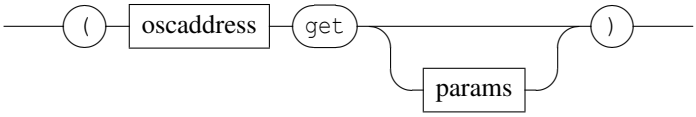
**NOTE** A variable can be used several times in a message, but several `$date` variables must always refer to the same mapping.

**`Warning:`** the interaction message set is provided for experiment. It is likely to change in a future version. Ideas, comments, suggestions are welcome for the design of a stable API.

## 13.4 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**

```
 /ITL/s/o1 watch 'mouseDown' '/ITL/s/o1' 'show' $('/ITL/s/o2' get 'show');
/ITL/s/o1 watch+ 'mouseDown' '/ITL/s/o2' 'show' $('/ITL/s/o1' get 'show');
/ITL/s/o2 watch 'mouseDown' '/ITL/s/o1' 'show' $('/ITL/s/o2' get 'show');
/ITL/s/o2 watch+ 'mouseDown' '/ITL/s/o2' 'show' $('/ITL/s/o1' get 'show')
```

When clicked, the objects `o1` and `o2` swap their visibility state.

## 13.5 OSC address variables

The OSC address parameter of a watch message supports the following variables:

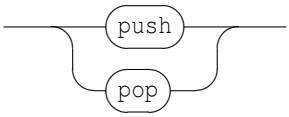
- `$self`: replaced by the object name at the time of the message setting.
- `$scene`: replaced by the scene name at the time of the message setting.

Note: the system doesn't check for the consistency of the variables placement.

## 13.6 States management

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

*stateMsg*




Actually every object embeds a stack to push and pop interaction states.

- `push`: push the current interaction state on top of the stack.
- `pop`: replace the current interaction state with the one poped from the top of the stack. Note that the effect of a pop message addressed to an empty stack is to clear the current interaction state.

## 13.7 File watcher

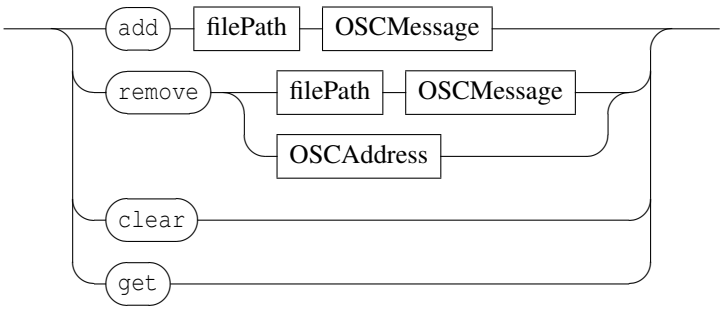
The `fileWatcher` is a component automatically embedded at scene level.
It receives messages at the address `/ITL/scene/fileWatcher`.

The `fileWatcher` service monitors files on demand; it is aware of file modifications and file removals (or renamings, which is equivalent). The `fileWatcher` works with associations between a file and an OSC message : when a file change occurs, the file watcher posts the associated OSC message.

The `fileWatcher` is controlled with 4 messages:

- the `add` message is used to add a new 'file' - 'OSC message' association ;
- the `remove` message is used to remove a 'file' - 'OSC message' association.
- the `clear` message is used to clear all existing associations.
- the `get` message is used to retrieve the current associations.

*fileWatcher*




where `OSCMessage` is any message complying to the format desscribed in section 1 p.1.

**EXAMPLE**

```
/ITL/scene/fileWatcher add 'fuga.gmn' '/ITL/scene/score' 'set' 'gmnf' 'fuga.gmn';
```

**NOTE** The `get` query returns a set of messages complying to the `add` message format.

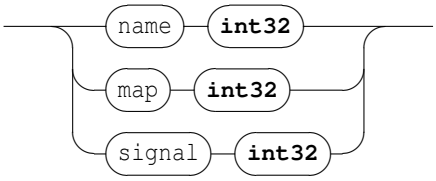
The `remove` message may take only 1 parameter, in which case the only parameter is an OSC address : it stops monitoring any file associated to an OSC message with this OSC address.

**NOTE** An OSC message can be associated to several files, and a file can be associated to several OSC messages. When a file has been removed (or renamed), the `fileWatcher` stops monitoring it.

## 13.8 Debug

Each object has a `debug` sub-node for debugging purposes. This `debug` virtual node has 3 flags, that can be actived or desactived with `0` or `1`:

*debug*




- When the `name` flag is on, each scene component displays its bounding rectangle and name.
- When the `map` flag is on, each scene component displays its mappings.
- When the `signal` flag is on, each object (even the scene or the application) will, according to its type, emit 'performance related' signals (or no signal). This 'performance signal' is specific to the type of object. The name of 'performance signal' is of the form `debug-`*objectName-SomeName*. Currently, only the `application`, the `scene` and `graph` objects emit 'performance signals'.

# Chapter 14

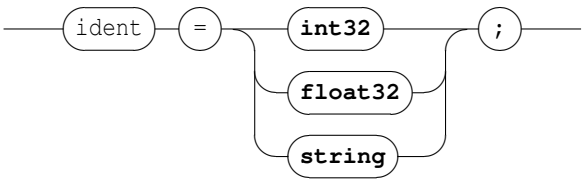
# Scripting

Since INScore saves its states as text files containing lists of OSC messages, it becomes natural to design INSCore scenes using these messages as a textual scripting language. For a better support of scripting, *variables* have been introduced to provide a better parameters control and programming languages have been embedded to generate messages algorithmically.

## 14.1 Variables

A variable may hold any value supported by the parameter types. **`string`** must be enclosed in simple or double quotes. It is declared as follows:

*variabledecl*




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.

Note that interaction messages may also use variables with similar reference forms ($ident). There is however no ambiguity concerning these references: interaction variables are always passed as strings and should be quoted, scripts variables as used without quotes.

**EXAMPLE**

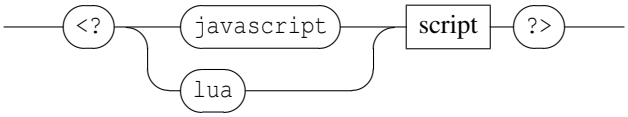
```
 x=0.5;
/ITL/scene/a x $x;
/ITL/scene/b x $x;
```

## 14.2 Languages

INScore supports Javascript and Lua as scripting languages. Javascript is embedded by default (using the v8 engine - see http://code.google.com/p/v8/. You need to recompile INScore to embed the Lua engine (http://www.lua.org/).

The principle of scripting using a programming language is the following: you can embed *javascript* of *lua* code in an INScore file in a way similar to scripts embedded into html files. This code is expected to produce INScore messages on output.

*lang*




The output of the script section is parsed like INScore messages.
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 for each scene. It allows scripts to reuse previously defined functions and thus design more structured scripts.

Note: the effect of 'load' message is actually to push the messages on the system stack for further evaluation. Thus when including a file containing javascript (or lua), the new language context won't be available to the next messages in the script. The workaround for structured script consists in designing a top level script that only contains 'load' messages.

**EXAMPLE**

```
/ITL load "javascript-code.inscore";
/ITL load "call-javascript.inscore";
```

# Chapter 15

# Appendices

## 15.1 Grammar definition

```
//_____________________________________________________
// relaxed simple INScore format specification
//_____________________________________________________
start       : expr
            | start expr
            ;

//_____________________________________________________
// expression of the script language
//_____________________________________________________
expr        : message  ENDEXPR
            | variabledecl ENDEXPR
            | script
            ;

//_____________________________________________________
// javascript and lua support
//_____________________________________________________
script      : LUASCRIPT
            | JSCRIPT
            ;
//_____________________________________________________
// messages specification (extends osc spec.)
//_____________________________________________________
message     : address
            | address params
            | address watchparams
            | address watchparams LEFTPAR messagelist RIGHTPAR
            | address watchparams script
            ;

messagelist : message
            | messagelist COMMA message
            ;
```

```
//_____________________________________________________
// address specification (extends osc spec.)
address     : oscaddress
            | urlprefix oscaddress
            ;

oscaddress  : oscpath
            | oscaddress oscpath
            ;

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
// watchparams need a special case since messages are expected as argument
//_____________________________________________________
watchparams : watchmethod
            | watchmethod params

params      : param
            | variable
            | params variable
            | params param
            ;

watchmethod : WATCH
            ;

variable    : VARSTART varname

param       : number
            | FLOAT
            | identifier
            | QUOTEDSTRING
            ;
```

```
//_____________________________________________________
// variable declaration
variabledecl : varname EQUAL params
             ;

varname      : IDENTIFIER
             | HOSTNAME
             ;

//_____________________________________________________
// misc
number       : UINT
             | INT
             ;
```

## 15.2 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

//_____________________________________________________
// parameters
//_____________________________________________________
QUOTEDSTRING     quotes could be single (') or double quotes (")
WATCH          : watch
               | watch+
               | require

//_____________________________________________________
// languages support
```

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

//_____________________________________________________
// misc.
//_____________________________________________________
PATHSEP         : '/'
POINT           : '.'
VARSTART        : '$'
COLON           : ':'
COMMA           : ','
LEFTPAR         : '('
RIGHTPAR        : ')'
EQUAL           : '='
ENDEXPR         : ';'
```

# Chapter 16

# Changes list

## 16.1 Differences to version 0.98

- new `rate` message at application level to control the time task rate (see section 7 p.22)
- new `frameless` message at scene level to switch to frameless or normal window (see section 8 p.26)

## 16.2 Differences to version 0.97

- new `fastgraph` object for graphic signals fast rendering (see section 4 p.14)
- $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 14.2 p.49)

## 16.3 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: *backslash* 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)

## 16.4 Differences to version 0.95

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

## 16.5 Differences to version 0.92

- new 'mouse' 'show/hide' message supported at application level (see section 7 p.22)

- graphic signal supports alpha messages at object level
- javascript and lua embedded and supported in inscore scripts (see section 14.2 p.49).
- bug correction in sync delete (introduced with version 0.90)

## 16.6 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.12).

## 16.7 Differences to version 0.90

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

## 16.8 Differences to version 0.82

- at application level: osc debug is now 'on' by default
- new scripting features (variables) (see section 14.1 p.49).
- 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

## 16.9 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

## 16.10 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

## 16.11 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

## 16.12 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

## 16.13 Differences to version 0.77

- guido system map extended: supports flat map or subdivided map (see section **??** p.**??**).
- new `shear` and `rotate` transformations messages (see section 2.2 p.7).
- 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 13.3 p.45).
- `reset` message addressed to a scene clears the scene rootPath

## 16.14 Differences to version 0.76

- get `guido-version` and `musicxml-version` messages supported by the application (see section 7 p.22).
- `save` message bug correction - introduced with version 0.70: only partial state of objects was saved
- `rootPath` message introduced at scene level (see section 8 p.26).
- 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

## 16.15 Differences to version 0.75

- new `require` message supported by the `/ITL` node (see section 7 p.22).
- new event named `newElement` supported at scene level (see section 13.2 p.44).
- new `name` and `address` variables (see section 13.3 p.45).
- 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`.

## 16.16 Differences to version 0.74

- new `map+` message (see section 9.2 p.30).
- the `click` and `select` messages are deprecated (but still supported). They will be removed in a future version (see section **??** p.**??**).

## 16.17 Differences to version 0.63

- new `dpage` message accepted by `gmn` objects (see section 6.3 p.20).
- x and y variables: automatic range type detection (int | float)
- set `txt` message: accepts polymorphic stream like parameters (see section 4 p.14).
- drag and drop files support in INScore viewer
- interaction variables extension: `$sx`, `$sy` variables added to support scene coordinate space (see section 13.3 p.45).
- automatic range mapping for `$x`, `$y` variables.
- new `$self` and `$scene` variables in the address field (see section 13.5 p.46).
- OSC identifiers characters set extended with '_' and '-' (see section 1 p.1).
- support for multiple scenes: `new`, `del` and `foreground` messages (see section 8 p.26).
- `load` message supported at scene level (see section 8 p.26).
- `get watch` implemented.
- `watch` message without argument to clear all the watched events (see section 13.3 p.45).
- 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.

## 16.18 Differences to version 0.60

- new 'mousemove' event (see section 13.1 p.44).
- interaction messages accept variables ($x, $y, $date...) (see section 13.3 p.45).
- SVG code and files support (see section 4.2 p.16).
- 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 4 p.14).
- new 'effect' message (section 2.4 p.10).
- 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.

## 16.19 Differences to version 0.55

- new 'xorigin' and 'yorigin' messages (section 2.1.3 p.6).
- new interaction messages set (section 13 p.43).
- 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,

## 16.20 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 9.1 p.28).
- format of sync messages extended to include map name (section 10 p.32).
- signal node: 'garbage' message removed
- new 'reset' message for the scene (/ITL/scene) (section 8 p.26).
- new 'version' message for the application (/ITL) (section 7 p.22).
- new 'reset' message for signals (section 11.1.1 p.35).
- 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

## 16.21 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 4 p.14).
- new `penStyle` message for vectorial graphics (section 6 p.19).
- new color messages `red, green, blue, alpha, dcolor, dred, dgreen, dblue` (section 2 p.4 and 2.1.2 p.6).
- color values for objects are bounded to [0,255]
- `get map` message behaves according to new map message (section 5 p.18).
- `get width` and `get height` is now supported by all objects (section 5 p.18).
- 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).

## 16.22 Differences to version 0.42

- `identifier` specification change (section 1 p.1).
- new application `hello` and `defaultShow` messages (section 7 p.22).
- new `load` and `save` messages (sections 7 p.22 and 2 p.4).
- `click` and `select` messages (section **??** p.**??**):
  - `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 4 p.14).
- `get` syntax change for the `scene` (section **??** p.**??**).
- `fileWatcher` messages completely redesigned (section 13.7 p.47).
- mappings can be identified by names (section 9.1 p.28).
- 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 (section **??** p.**??**).
- Guido mapping manages repeat bars.
- Graphic signals messages design (section 11.2 p.38).