Ircam documentation

- Research reports
- Musical works
- Software

# PatchWork

## PW-Script

*First English Edition, February 1996*

IRCAM Centre Georges Pompidou

This manual was written by Carlos Agon and Gérard Assayag, translated into English by J. Fineberg, and was produced under the editorial responsibility of Marc Battier, Marketing Office, Ircam.

Patchwork was conceived and programmed by
Mikael Laurson, Camilo Rueda, and Jacques Duthen.

The PW-Script library was conceived and programmed by Carlos Agon.

First English edition of the documentation, February 1996.
This documentation corresponds to version 1.1 of the library, and to version 2.1 or higher of PatchWork.


**Ircam**
**1, place Igor-Stravinsky**
**F-75004 Paris**
**Tel. (33) (1) 44 78 49 62**
**Fax (33) (1) 42 77 29 47**
**E-mail ircam-doc@ircam.fr**

# IRCAM Users' group

The use of this software and its documentation is restricted to members of the Ircam software users' group. For any supplementary information, contact:

Département de la Valorisation
Ircam
Place Stravinsky, F-75004 Paris

Tel. (1) 44 78 49 62
Fax (1) 42 77 29 47
E-mail: bousac@ircam.fr

Send comments or suggestions to the editor:

E-mail: bam@ircam.fr
Mail: Marc Battier,
Ircam, Département de la Valorisation
Place Stravinsky, F-75004 Paris

To see the table of contents of this manual, click on the Bookmark Button located in the Viewing section of the Adobe Acrobat Reader toolbar.

# Contents

# Résumé

Les applications Macintosh se caractérisent par leur interface graphique « amical » qui permet à l'utilisateur l'exécution de certaines actions de façon presque transparente. Pourtant cette dépendance entre l'application et son interface graphique rend très difficile l'exécution de tâches répétitives, de même que le contrôle et l'échange des données entre différentes applications. Pour résoudre ces problèmes Apple a proposé un protocole pour le pilotage des applications.

AppleScript est un nouveau composant du logiciel système Apple qui permet à l'utilisateur de contrôler toutes les opérations fournies par une application à travers un *script* plutôt que par l'habituelle interaction souris-clavier.

Un script est une liste de commandes écrites dans un langage symbolique simple et proche du langage naturel, stockées dans un fichier texte que l'on crée à l'aide de l'application Editeur de Script fournie par Apple. Un script peut aussi être créé grâce à la fonction `Enregistrement` de l'Editeur de Script. Dans ce cas, AppleScript enregistre les actions souris-clavier exécutées par l'utilisateur et crée automatiquement le script correspondant.

Pour être contrôlée via un script, une application doit explicitement intégrer cette possibilité dans son architecture. La plupart des applications produites aujourd'hui pour le Macintosh sont scriptables ou vont le devenir. De même, pour donner accès à la fonction Enregistrement, une application doit être *recordable*. Pour savoir si une application est scriptable, glissez son icône sur celle de l'Editeur de Script. Vous verrez alors apparaître, le cas échéant, une fenêtre contenant son *dictionnaire des commandes*.

Un script peut donc être utilisé pour piloter une application, combiner et intégrer les fonctionnalités de plusieurs applications et servir à la communication inter-application.

L'idée d'un protocole de communication utilisant des scripts peut se réduire à l'équation suivante :

```
script protocole = commandes + objets
```

Les objets correspondent aux entités manipulables au sein de votre applications (fenêtres, menus, objets graphiques etc.). Ils définissent un interface de programmation auquel vous avez accès au moyen d'un ensemble de commandes. L'ensemble de commandes doit être restreint et chaque commande doit pouvoir s'appliquer à des objets de différentes classes.

Conscients des évolutions du système Macintosh, des grandes possibilités offertes par les scripts et du fait que de

plus en plus d'applications se servent de telles possibilités, nous avons rendu PatchWork *scriptable* et *recordable* et nous y avons ajouté un nouveau module éditeur de scripts doté des même fonctionnalités que l'Editeur de Scripts d'Apple.
Nous commençons ce texte avec des définitions générales destinées à aider à la compréhension des nouveaux service offerts par PatchWork.

# Définitions générales

1. Un **script** est une collection de données qui engendre une séquence d'actions quand elle est exécutés.
2. Un script est écrit dans une **langage de scripts**. Apple fournit un langage de script standard qui sera utilisé tout au long de ce manuel.
3. Une application est **scriptable** si elle est capable de répondre aux différentes commandes envoyées par l'interprète de langage de scripts. PatchWork 2.5 est scriptable.
4. Une application est **recordable** si elle capable d'enregistrer, sous la forme de script, les différentes actions exécutées par l'utilisateur. PatchWork 2.5 est recordable.
5. On dit qu'une application **gère des scripts** si elle peut elle même fournir un interface, comparable à celui de l'Editeur de scripts, pour créer, éditer, compiler, exécuter des scripts. PatchWork 2.5 gère des scripts.

Pour une information plus détaillée, voir [App92, App93a, App93b].
A partir de la version 2.5, PatchWork est scriptable, recordable, et gère des scripts. Du fait de cette dernière fonctionnalité, un script peut apparaître sous la forme d'une boîte PatchWork dont l'évaluation déclenche l'exécution du script et ramène une valeur utilisable dans un patch. Par l'envoi de commandes à une autre application (éventuellement une autre image de PatchWork) qui peut être située sur une autre machine, PatchWork voit son champ d'application et son opérabilité nettement étendus.

# Plan du manuel

Dans le chapitre 2, nous exposons les configurations minimales matérielles et logicielles dont vous aurez besoin pour profiter de PW-Script. Si vous ne disposez pas des logiciels systèmes requis, PatchWork continuera à travailler de façon habituelle.

Quand on parle d'une application *scriptable*, on parle de deux parties :

- une structure hiérarchique d'*objets* propres à l'application ;
- un ensemble de commandes sur ces objets susceptibles d'être déclenchées à partir d'un script.

Dans le chapitre 3, nous donnons la référence des *objets* définis par PatchWork.

Le chapitre 4 présente la liste des *commandes*.

Le chapitre 5 est présenté sous la forme d'un tutoriel montrant comment enregistrer des scripts sous PatchWork.

Dans le chapitre 6 nous présentons la nouvelle librarie AAC contenant le module d'édition de scripts interne à PatchWork.

La bonne utilisation des nouvelles fonctionnalités PatchWork dépend en grande partie de votre imagination ; nous vous remercions de nous communiquer des problèmes rencontrés, de même que les travaux réalisés avec PW script que vous jugerez d'intérêt général.

# 1 Introduction

Macintosh applications are characterized by their "friendly" graphical user interface, which allows the user to perform many actions in an almost transparent way. However, this interdependence between the application and its graphic interface makes it very difficult to perform repetitive tasks or to control the exchange of data between different applications. To resolve this problem Apple has established a protocol for controlling applications.

AppleScript is a new part of the Apple system software which allows the user to direct all operation that are possible within a given application through the use of a script, in place of the normal mouse or keyboard operations.

A script is a list of commands written in a symbolic language that is simple and close to natural language. These scripts are stored as text files which are created with the help of the Script Editor application, supplied by Apple. A script may also be created with the function Record of the Script Editor. In this case, AppleScript records the manipulations the user performs on the keyboard or with the mouse and automatically creates the corresponding script.

In order to be controlled by a script, an application must have that possibility explicitly integrated into its architecture. Most current Macintosh applications are either already scriptable will soon become so. Additionally, to permit access to the Record function, an application must be recordable. To see if an application is scriptable, drag its icon onto that of the Script Editor. If the application is scriptable, a window will appear with the application's dictionary of commands.

A script may, thus, be used to control an application, to combine and integrate the functions of multiple applications and to facilitate inter-application communication.

The idea of a communication protocol using scripts may be reduced to the following equation:

```
script protocol = commands + objets
```

The objets correspond to entities that may be manipulated within you applications (windows, menus, graphic objets, etc.). They define a programming interface through which you will have access to a group of commands. The group of commands must be limited and each command must be applicable to the objects of different classes.

Aware of the evolutions within the Macintosh system, great possibilities are offered by scripts and since more and more applications take advantage of these possibilities, we have made PatchWork scriptable and recordable we have added a new script editor module possessing the same functionality as Apple's Script Editor.

We will begin this text with some general definitions which should help the reader to understand the new possibilities offered by PatchWork.

# General Definitions

1. A script is a collection of data which produces a series of actions when it is executed.
2. A script is written in script language. Apple provides a standard script language which will be used throughout this manual.
3. An application is scriptable if it is capable of responding to different commands transmitted through script language. PatchWork 2.5 is scriptable.
4. An application is recordable if it is capable of recording, in the form of a script, the different actions performed by the user. PatchWork 2.5 is recordable.
5. An application may be said to be capable of managing scripts if it provides its own interface, comparable to that of Apple's Script Editor, to create, edit, compile and execute scripts. PatchWork 2.5 can manage scripts.

For more details, see [App92, App93a, App93b].

Starting from version 2.5, PatchWork is scriptable, recordable, and manages scripts. Because of that last functionality, a script may appear in the form of a PatchWork module; the evaluation of which triggers the execution of the script and returns a value which may be used in a patch. By sending commands to another application (eventually another PatchWork image) which may even be on another machine, PatchWork sees its possible uses and functions greatly expanded.

# Guide to this Manual

In Chapter 2, we will give details concerning the minimum hardware and software configurations which are necessary to use PW-Script. If you lack the required system software, PatchWork will continue to function as before.

In speaking of a scriptable application, two parts are concerned:

- an hierarchic structure of objets specific to the application ;
- a group of commands which may be applied to these objects that can be triggered from a script.

In Chapter 3, we will provide a reference for the *objects* defined by PatchWork.

Chapter 4 presents the list of *commands*.

Chapter 5 is a tutorial showing how to record scripts in PatchWork.

Chapter 6 presents the new AAC library, which contains PatchWork's internal script editor module.

To make proper use of PatchWork's new functions will greatly depend on the user's imagination. We would appreciate your communicating to us any problems you might find and, also, any work you may do with PW-Script that you think might be of general interest.

# 2 Configuration

Here is the configuration needed to run PW-Script.

- Macintosh System version 7.5 or higher
- 8 MB of RAM minimum
- AppleScript® installed (AppleScript if furnished with System 7.5. The user's manual and the command language documentation may be obtained from your Apple dealer).

To get the most out of PW-script, we recommend obtaining the manual for AppleScript. It contains descriptions of the syntax and commands of this language.

# 3 PatchWork Object Classes

A scriptable application possesses two types of components : a hierarchy of objects belonging to the application and a set of commands which may be triggered from a script.

PatchWork objects contain both the internal information used for the application itself and the external methods allowing them to be accessed and manipulated by outside applications. A PatchWork object is defined as part of a class. A PatchWork class is defined by : its *name*, its *properties* and its *elements*.

The *properties* are attributes that are specific to the object, for example: name, size, etc.

The *elements* are subsidiary objects contained within the reference object. For example, the class `chord` is defined as:

| | | |
|---|---|---|
| Name | `Chord` | |
| Properties | `Time` | (the attack time of the `chord`) |
| Elements | `Notes` | |

The value of a property is both unique and obligatory, Whereas an object may have 0 or many elements associated with the same key.

The PatchWork object classes may be divided into two large groups: functional object classes and musical object classes.

# The Functional Object Classes

The functional object classes allow the control of the execution of different actions within PatchWork.

**Class** `patch`

| | |
|---|---|
| Name | `patch` |
| Properties | `name` |
| Elements | `box` |

**Syntax**

`patch` *name*

*name* is the 'name' of the patch. In order to eliminate all ambiguity, the full file access path may be given.

**Examples**

```
patch "myPatch"
patch "HD:patches:myPatch.pw"
```

A patch objet contains elements of the type: `box`.

# List of object classes

**Class** box

| | |
|---|---|
| Name | box |
| Properties | class |
| | name |
| | position |
| Elements | input |
| | menu |

**Syntax**

box name [of patch name]

**Examples**

```
box "chordseq"
box "chordseq" of patch "myPatch"
```

PatchWork automatically gives a different name to each box of the same class, so as to avoid having two boxes with the same name in the same patch. If there is no reference to a specific patch object, it is assumed that the reference is to the active patch. Boxes created with the menu option **Lisp Function...** are part of the class funlisp and carry the name of the function (**list**, **first**, etc.). The position of a box is a point {x , y} where x and y are the respective horizontal and vertical coordinates within the patch window. An objet box contains inputs and sometimes menus.

**Class** input

| | |
|---|---|
| Name | input |

**Syntax**

input *number* of *reference-to-an-objet-box*

**Examples**

```
input 1 of box "chordseq"
input 2 of box "chordseq" of patch "HD:patches:myPatch.pw"
```

**Class** menu

| | |
|---|---|
| Name | menu |
| Properties | name |
| Elements | menu |

**Syntax**

menu *name* of *reference-to-an-objet-box*

**Examples**

```
menu "save" of box "chordseq"
menu "save" of box "chordseq" of patch "HD:patches:myPatch.pw"
```

See the command: `command`.

Menus containing sub-menus have been provided for, although no current Patchwork box makes use of them..

**Class** `library`

Name `library`
Properties `name`

**Syntax**

library *name*

**Examples**

```
library "HD:PW 2.1:user library:aac.lib"
```

See the command: load.

**Class** `abstract`

Name abstract
Properties name

**Syntax**

```
abstract name
```

**Examples**

```
abstract "HD:work:music-abstractions"
```

*name* is a path specifying a directory containing abstractions or else sub-directories containing abstractions. These abstractions are loaded in the form of hierarchical sub-menus within the menu **user-lib**. See the command: `load`.

**Class** `config`

Name `config`

**Syntax**

```
config
```

A configuration (a set of libraries and preferences to be pre-loaded when PatchWork is opened) is referred to by the word `config`. See the command: `save`.

# The Musical Object Classes

The musical object classes gives access to the results that have been created by the evaluation of a patch. They are obtained as the result of a command eval sent to an object of the class box, or they may be extracted from another musical object through an index.

| **Class** | `chord-line` |
|---|---|
| Name | `chord-line` |
| Properties | `ID`, a non-editable identifier |
| | `length`, the number of `chord`s in the `chord-line` |
| Elements | `chord` |

`Chord-lines` are generally produced by evaluating a box of the class `chord`**seq** or **multiseq**. The elements of a `chord-line` are elements of the class `chord`. In order to refer to a `chord-line`, one must evaluate the box which calculates it, using the command `eval`.

```
eval box "chordseq"
```

If the objet `chord-line` is to be used more than once, it is possible to lock the box which calculates it:

```
lock box "chordseq"
eval box "chordseq"
....
eval box "chordseq"
```

Another technique is to make use of the affectation instructions of script language:

```
set myChordLine to (eval box "chordseq")
```

The variable *my*`Chord`*Line* may then be used to refer to the objet `chord-line`.

In the case of a box from the class **multiseq**, an index must be used to extract each `chord-line`.

```
item 1 of (eval box "multiseq")
item 3 of (eval box "multiseq")
```

| **Class** | `chord` |
|---|---|
| Name | `chord` |
| Properties | `ID`, a non-editable identifier |
| | `time` |
| | `length`, the number of notes in the `chord` |
| Elements | `note` |

A `chord` objet may be referenced by evaluating a box which delivers a `chord` as its result:

```
eval box "chord"
```

or by its index within another object, for example:

```
chord 2 of (eval box "chordseq")
set polyphony to (eval box "multiseq")
```

```
chord 5 of item 3 of polyphony
```

The property time gives the attack time of the `chord`:

```
time of chord 2 of myChordLine
```

**Class** `note`

| | |
|---|---|
| Name | `note` |
| Properties | `ID`, a non-editable identifier<br>`pitch`<br>`duration`<br>`velocity`<br>`channel`<br>`time` |

A `note` objet may be referenced by evaluating a box which delivers a note as its result:

```
eval box "mk-note"
```

or by its index within another object, for example:

```
note 5 of chord 2 of item 1 of (eval box "multiseq")
```

The different properties may be accessed by name within script language.

```
pitch of note 1 of (eval box "chord")
duration of note 3 of (eval box "chord")
```

**Class** `measure-line`

| | |
|---|---|
| Name | `measure-line` |
| Properties | `ID`, a non-editable identifier |
| Elements | `measure` |

`Measure-lines` are generally produced by evaluating a box of the class **rtm** or **poly-rtm**.

A measure-line objet may be referenced by:

```
eval box "rtm"
item 1 of (eval box "poly-rtm")
```

**Class** `measure`

| | |
|---|---|
| Name | `measure` |
| Properties | `ID`, a non-editable identifier<br>`signature`<br>`tempo` |
| Elements | `beat` |

A measure objet may be referenced by its index within a `measure-line` object:

```
measure 2 of (eval box "rtm")
measure 2 of item 2 of (eval box "poly-rtm")
```

The property signature gives a list {a,b} where a is the number of pulsations and b is the value of each pulsation (i.e. {3, 4} to express a measure in 3/4). Tempo also produces a list, for example {62,4} signifies a tempo of quarter note equals 62:

```
signature of measure 2 of (eval box "rtm")
tempo of measure 2 of (eval box "rtm")
```

| **Class** | `beat` |
|---|---|
| Name | `beat` |
| Elements | `chord` |
| Properties | `ID`, a non-editable identifier |
| Length | the number of `chords` in the beat |

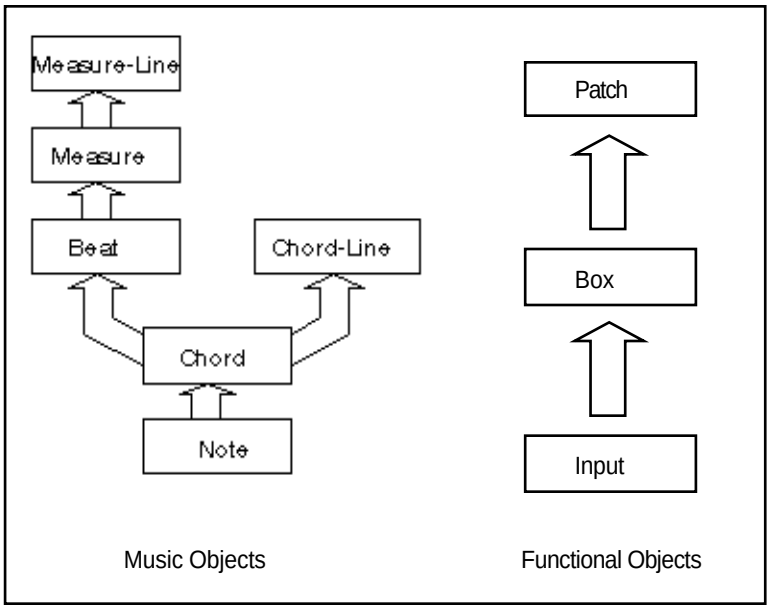
A `beat` object may be referenced by its index within a measure object:

```
beat 3 of measure 2 of (eval box "rtm")
```

The property `chord` gives access to the `chord` associated to a `beat`.

```
chord 2 of beat 3 of measure 5 of item 2 of (eval box "poly-rtm")
```

The following chart shows the organization of the functional and musical object classes. The arrows signify « is an element of... ».



# 4 List of Commands

The functional and musical objects which were described in Chapter 3 are used as the destination of messages which may be sent from a script. These messages are written in the form of commands whose arguments are references to those objects.

## Close

Closes a patch window.

**Syntax**

`close` *`Patch-reference`*

**Parameter**

*Patch-reference* a reference to an object of the class patch

**Output Result**

None

**Example**

```
close patch ""HD:patches:myPatch.pw"
close patch -- close the active patch
```

# Command

Execute a command contained within the local menu of a PatchWork box.

**Syntax**

```
command menu  [file name  string ]
```

**Parameter**

| | |
|---|---|
| *menu* | a reference to an object of the class menu |
| *string* | an optional file name |

**Output Result**

None

**Example**

```
command menu "chord object" of box "chord"
command menu "save chord" of box "chord" file name "HD:work:myChord"
command menu "x points" of box "bpf"
command menu "save" of box "bpf" file name "mybpf"
```

# Connect

Connect the output of one box to the input of another.

**Syntax**

connect *Box* to *Input*

**Parameter**

| | |
|---|---|
| *Box* | a reference to an object of the class box |
| *Input* | a reference to an object of the class input |

**Output Result**

None

**Example**

```
connect box "mk-chord" to input 1 of box "chord"
```

# Copy

Copy the selected boxes.

**Syntax**

copy [selection]

**Parameter**

None

**Output Result**

None

**Example**

```
copy
copy selection
```

# Delete

Delete the selected boxes.

**Syntax**

```
delete [selection]
```

**Parameter**

None

**Output Result**

None

**Example**

```
delete
delete selection
```

# Duplicate

Duplicate the selected boxes.

**Syntax**

```
duplicate [selection]
```

**Parameter**

None

**Output Result**

None

**Example**

```
duplicate
duplicate [selection]
```

# Eval

Trigger the evaluation of the referenced box.

**Syntax**

`eval` *`Box`*

**Parameter**

| | |
|---|---|
| *`Box`* | a reference to an object of the class `box` |

**Output Result**

The value returned by the evaluation of the box

**Example**

```
eval box "chordseq"
eval box "poly-rtm" of patch "myPatch"
set myVar to (eval box "chordseq") -- place the result of the evaluation
in a variable myvar
```

# Extend

Add an input to an extendible box.

**Syntax**

```
extend Box
```

**Parameter**

| | |
|---|---|
| *Box* | a reference to an object of the class *box* |

**Output Result**

A reference to the *box* object.

**Example**

```
extend box "rtm"
```

# Get

Return the value of the referenced object.

**Syntax**

```
get Object
```

**Parameter**

| | |
|---|---|
| *Object* | a reference to an objet or the property of an object |

**Output Result**

The value of the object or the property

**Example**

```
get input 2 of box "chordseq"
get pitch of note 2 of chord 1 of beat 2 of measure 3 of (eval box "rtm")
get box "g*"
```

# Load

Loads a library or a group of abstractions.

**Syntax**

`load` *`string`*

**Parameter**

| | |
|---|---|
| *`string`* | a file or folder name |

**Output Result**

None

**Example**

```
load library "HD:PW 2.1:user library:kant.lib"
load abstract "HD:abstracts:music-abstractions"
```

# Lock

Lock a box.

**Syntax**

lock *Box*

**Parameter**

| | |
|---|---|
| *Box* | a reference to an object of the class *box* |

**Output Result**

None

**Example**

```
lock box "chordseq"
```

# Make

Create a new objet.

**Syntax**

```
make new  [of class  class ] [with name   string ] [at  point]
```

Parameter

| | |
|---|---|
| *new* | The class of the object to be created (`patch`, `box` or `abstract`) |
| *class* | The class of the box to be created |
| *string* | The name to be given to the objet |
| *point* | the initial position for the object in the case of a `box` or an `abstract` |

Output Result

a reference to the new objet.

Example

```
make new patch
make  box of class "chordseq"
make  box of class "chordseq" with name  "mychordseq" at {50,50}
make abstract -- create an abstraction containing the selected boxes.
make box of class "funlisp" with name "list"-(create a "lisp" box)
```

# Move

Move a box to a new position.

Syntax

```
move Box  to newposition
```

Parameter

| | |
|---|---|
| *Box* | a reference to an object of the class `box` |
| *newposition* | the new coordinates of the box |

Output Result

None

Example

```
move box "chordseq" to {100, 500}
```

# Open

Open a patch or a box.

**Syntax**

```
open Object
```

**Parameter**

| | |
|---|---|
| *Object* | a reference to an object of the class `patch` or `box` |

**Output Result**

None

**Example**

```
open patch  "HD:patches:myPatch.pw"
open box "chordseq"
```

# Options

Change PatchWork's global options.

This command performs the same functions as the menu **PWoper>Global options**.

**Syntax**

```
options [scale string] [approximation string] [play options string]  [eva-
luation string]
```

**Parameter**

| | |
|---|---|
| *string* | the selected option |

**Output Result**

None

**Example**

```
options scale "C-major"
options approximation "quarter tone"
option play options "Pitch Bend"
```

# Paste

Paste the boxes previously copied into the current patch window.

**Syntax**

```
paste [selection]
```

**Parameter**

None

**Output Result**

None

**Example**

```
paste
paste selection
```

# Play

Play a musical objet.

**Syntax**

```
play object
```

**Parameter**

| | |
|---|---|
| *object* | a reference to an object of the class `box` or a musical object obtained by evaluating a PatchWork box. |

**Output Result**

None

**Example**

```
play box "chord"
play box "rtm" of patch "myPatch"
play chord 3 of item 2 of (eval box "chordseq")
play chord 2 of beat 3 of measure 5 of item 2 of (eval box "poly-rtm")
```

# Print

Print the active window.

**Syntax**

```
print  [selection]
```

**Parameter**

None

**Output Result**

None

**Example**

```
print
print selection
```

# Rename

Change the name of a box.

**Syntax**

```
rename Box  as NewName
```

**Parameter**

| | |
|---|---|
| *Box* | a reference to an object of the class `box` |
| *NewName* | the new name |

**Output Result**

The referenced box

**Example**

```
rename box "chordseq" as "ch1"
```

# Save

Save a patch or a configuration.

**Syntax**

```
save [Patch] [as NewName] [with mn]
save config
```

**Parameter**

| | |
|---|---|
| *Patch* | a reference to an object of the class `patch` or `config` |
| *NewName* | the name of the file |
| *with mn* | option allowing the contents of the musical notation editor to be saved with the patch |

**Output Result**

None

**Example**

```
save as "HD:patches:new-patch.pw"-- save the active patch
save "old-patch.pw" as "new-patch.pw" with mn
save config-- save the configuration of libraries and preferences
```

# Select

Graphically select a box or list of boxes, or activate a patch window.

**Syntax**

```
select parameter [with shift]
```

Parameter

| | |
|---|---|
| *parameter* | A reference or list of references to an object or several objects of the class box, or a reference to an object of the class `patch`. |
| *shift* | simulates the shift key to allow new elements to be added to the current selection. |

**Output Result**

The referenced object.

**Example**

```
select patch "myPatch.pw"
select box "chordseq" with shift
select {box "chordseq", box "rtm", box "cons"}
```

# Set

Affect a value to an objet. This may be used to give a value to a temporary variable or to set an input value for a box prior to evaluation.

**Syntax**

```
set     directParameter
to      value
```

**Parameter**

| | |
|---|---|
| *directParameter* | the object to be modified |
| *value* | the new value |

**Output Result**

None

Example

```
set mybox to select box "chordseq"
set input 1 of box "chordseq" to {6000, 6300, 6500}
```

# Stop

To stop a command play that is in progress.

**Syntax**

```
stop parameter
```

**Parameter**

| | |
|---|---|
| *parameter* | A reference to the object to be stopped |

**Output Result**

None

**Example**

```
stop box "chordseq"
```

# Unconnect

Sever the connection between two boxes.

**Syntax**

```
unconnect    Input
```

**Parameter**

| | |
|---|---|
| *Input* | a reference to an object of the type `input` |

**Output Result**

None

**Example**

```
unconnect input 2 of box "chordseq"
```

# Unlock

Unlock a box.

**Syntax**

```
unlock Box
```

**Parameter**

| | |
|---|---|
| *Box* | a reference to an object of the type `box` |

**Output Result**

None

**Example**

```
unlock box "chordseq"
```

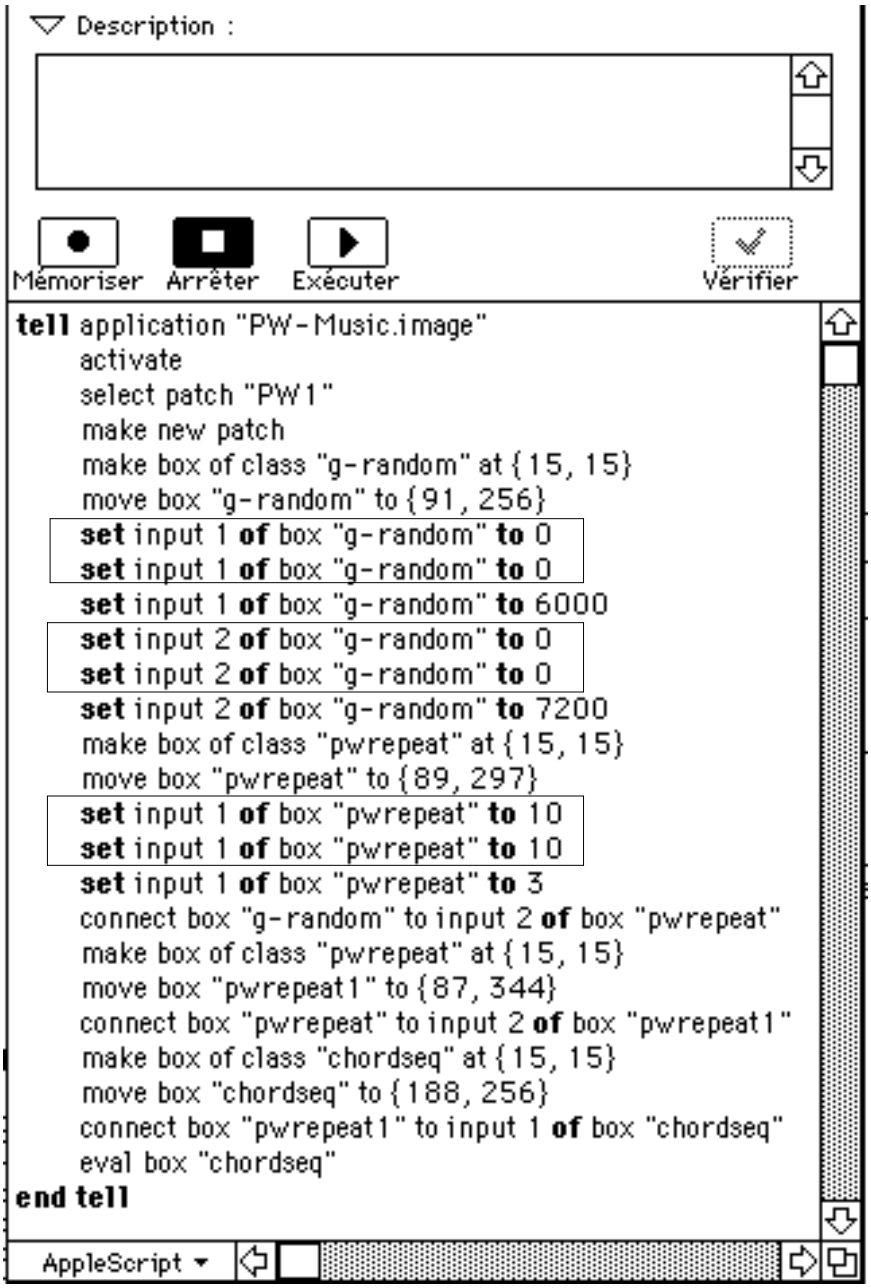
# 5 Tutorial: example of Recordable PatchWork

A script may be created by typing it directly in the Script Editor or by recording it directly. Here is an example in which a small patch is created by recording:

- Open PatchWork
- Open Apple's Script Editor application
- Click on the record button `Memorize` (*`Mémoriser`*)
- Return to PatchWork
- Create a new Patch window
- Add a box **g-random**
- Place the value 6000 in the input 1 and the value 7200 in the input 2
- Add a box **pwrepeat**
- Place the value 3 in the input 1
- Connect the output of **g-random** to the input 2 of **pwrepeat**
- Add a second box **pwrepeat**. (note that PatchWork automatically renames it **pwrepeat1**)
- Connect the output of pwrepeat to the input 2 of **pwrepeat1**
- Add a box **chordseq**
- Connect the output of pwrepeat1 o the input 1 de **chordseq**
- Evaluate the **chordseq**
- Return to the Script Editor application
- Press the button `Stop (`*`Arrêter`*`)`

The following figure shows the obtained script. The boxed instructions are redundant: they are the result of the double click performed in PatchWork to change an input. You can delete them without any risk.

Press the `execute` (*`Exécuter`*)button. You will see that PatchWork faithfully reproduces the recorded actions.




Immediately prior to the last line (`End tell`) type the following instructions.

```
open box "chordseq"
play box "chordseq"
```

Press once again on the execute button. The commands `open` and `play` are not recordable: you must add them by hand in your scripts.

Save your script with the option `compiled script`. Subsequently, double clicking on the icon of your script or dragging it onto the Script Editor application icon will be sufficient to open and execute your script. If you would like to construct a "silent" mini-application which performs the same task, save your script with the options `application` and `don't show start up screen` (In French: *`ne pas afficher l'écran de démarrage`*).

When writing your own scripts, don't forget to place all your commands between the following lines:

```
tell application "pw-music.image"
activate
...
end tell
```

In this way all the commands will be properly directed towards PatchWork. A good technique is to start by creating the skeleton of a script by recording it directly and then to complete it with manual editing.

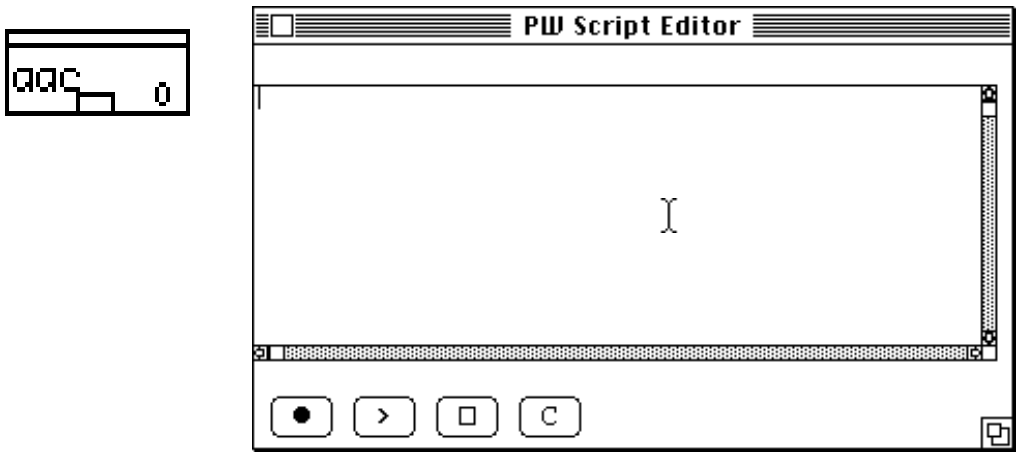
The AppleScript language contains all sorts of control instructions (affectations, loops, tests) which permit the construction of real and powerful applications. Contact your local Apple distributor to obtain the necessary documentation. There are programs (e.g. Frontmost from Software Designs Unl.) which allow a graphic interface to be associated to a script and also the construction of complete and personalized applications.

# 6 The AAC Library

The AAC library (Application to Application Communication) allows the editing, recording and execution of scripts directly, from within PatchWork. It contains a single module, the box aac, which is a script editor. Evaluating this box in PatchWork triggers the evaluation of the script which it contains. As a script may return a value, this value is used as the evaluation output of the box aac and may be passed to other PatchWork modules, if the box **aac**'s output is connected to another boxes input. In this way, evaluating a patch containing a box aac may trigger the execution of a script which can send and receive messages from other applications (eventually on another machine, connected by a network, or another PatchWork image, or even the same image where the script is active).

To use this library, load the file `aac.lib`, which is located in the folder:

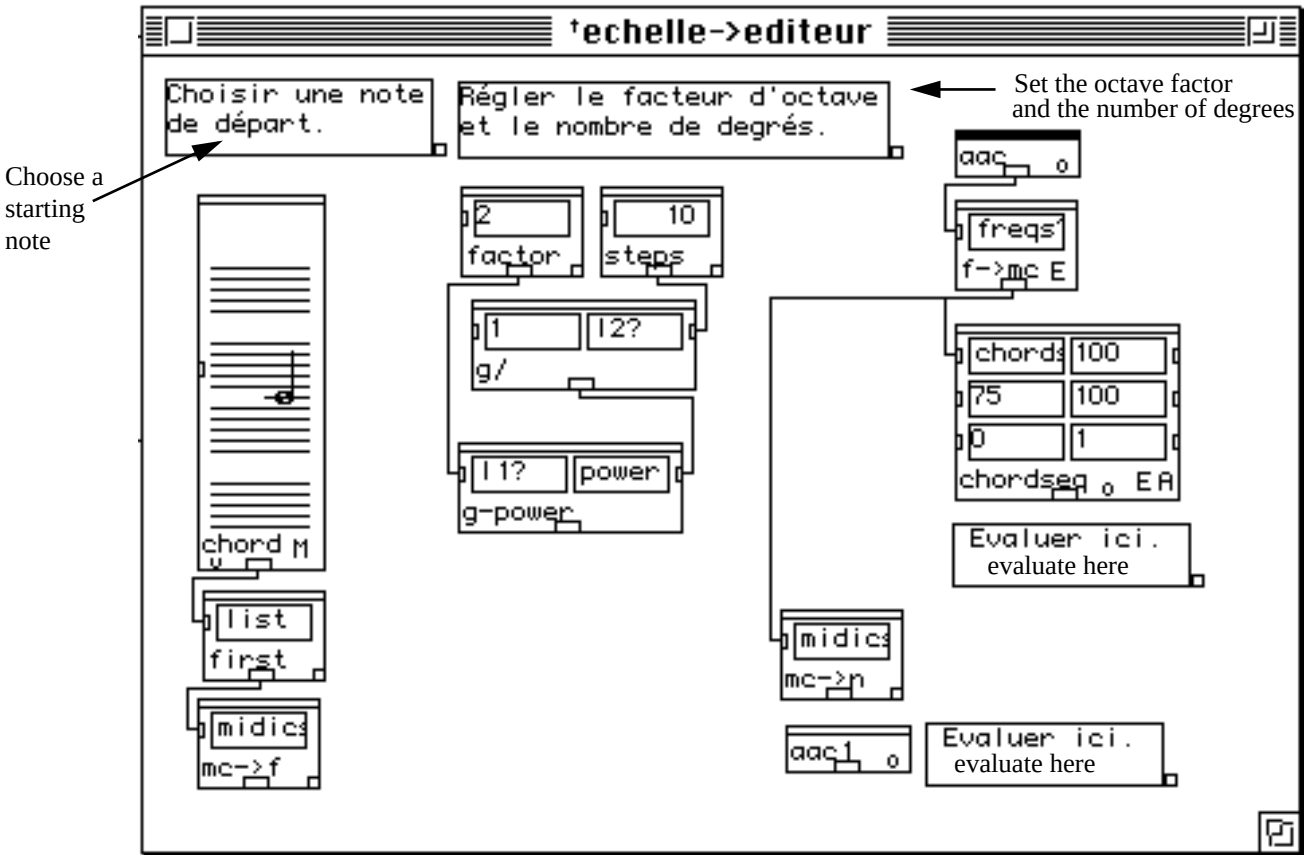
`...:pw 2.5:user-library:aac`, by using the menu **PWoper>load Library**. Add a box **aac** to the active window. Double-click on the box: an editor window will appear.



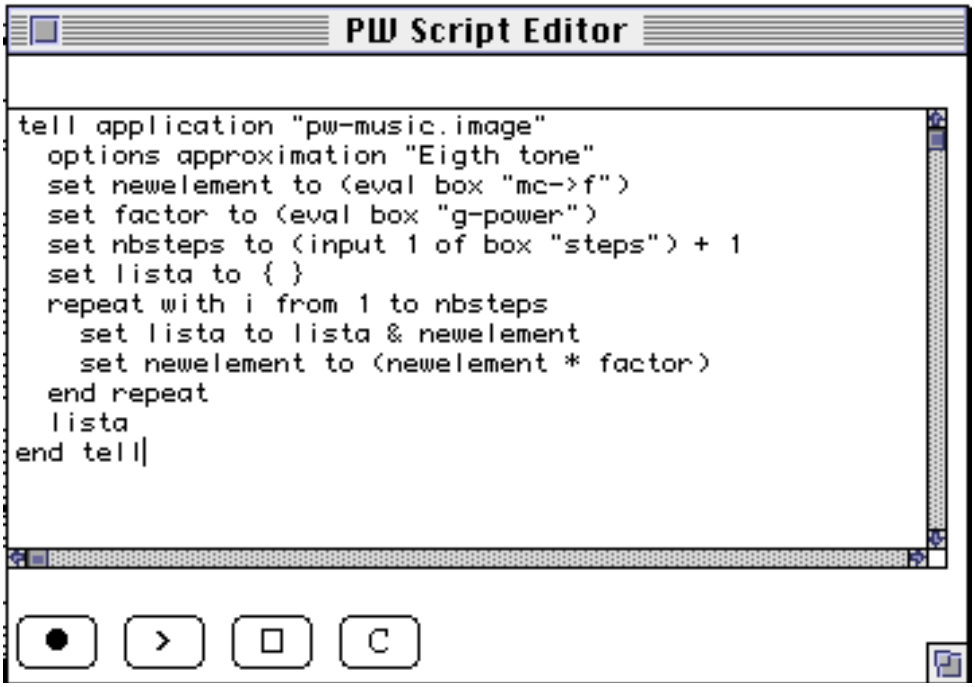

Record Play Stop Compile

Construct your script with the help of the `Record` button or simply by typing it in from the keyboard. Once you have constructed your script it is possible to check it for potential typing or syntactic mistakes by clicking on the `Compile` button. This button also prepares your script to be executed each time you evaluate the box **aac**. You can test your script by using the `Play` button.

Open the example patch `echelle>editeur.pw` in the examples folder of the AAC library.




This patch constructs a scale of equidistant pitches starting on the note shown in the `chord` box, with the number of steps shown in the box **steps** and continues until the frequency is equal to that of the firstnote multiplied by the factor in the box **factor**. The script contained in the **aac** box controls the various parts of the patch. The calculation is triggered by evaluating the **chordseq** box. Once the calculation is complete, open the **chordseq** box to see the result.




Annotated explanation of the script

```
tell application "pw-music.image"
```

Commands are addressed to the application PatchWork.

```
options approximation "Eighth tone"
```

The frequency domain results will be approximated and displayed to the nearest eighth tone.

```
set newelement to (eval box "mc->f")
```

The variable newelement is the frequency of the starting note

```
set factor to (eval box "g-power")
```

The variable factor is the frequency relationship between each degree of the scale

```
set nbsteps to (input 1 of box "steps") + 1
```

We recover the number of steps directly from the input of the **nbsteps** box

```
set lista to { }
```

We initialize an empty list

```
repeat with i from 1 to nbsteps
```

Loop from 1 to `nbsteps`

```
set lista to lista & newelement
```

We concatenate `newelement` to the end of the list

```
set newelement to (newelement * factor)
```

We multiply `newelement` by `factor`

```
end repeat
```

We close the loop

```
lista
```

The script returns the list of frequencies

```
end tell
```

The second script `aac1` shows an example of inter-application communication. To use this example you need to have Scriptable Text Editor, a small application given by Apple with AppleScript.

```
tell application "Scriptable Text Editor"
   activate
end tell
```

Activate the destination application.

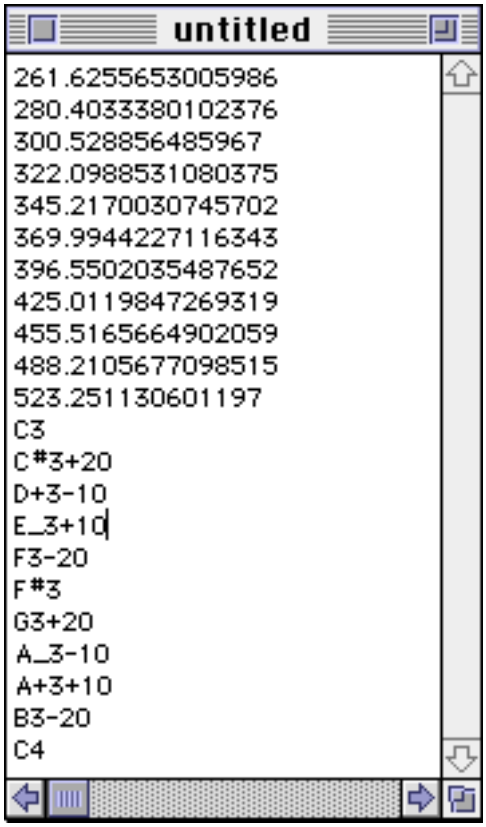
```
tell application "pw-music.image"
  set freqList to (eval box "aac")
  set noteList to (eval box "mc->n")
end tell
```

The aac script is triggered twice. It returns a list of frequencies which are placed in `freqList`, and, through the **mc>n** box, a list of note names, placed in `noteList`.

```
tell application "Scriptable Text Editor"
   repeat with aFreq in freqList
     make paragraph with data aFreq
   end repeat
   repeat with aNote in noteList
     make paragraph with data aNote
   end repeat
end tell
```

The two lists are traversed and each element is sent to the Scriptable Text Editor by the instruction `make paragraph...`

The application Scriptable Text Editor will, thus, display a window containing our scale in both frequency and symbolic form.




There are other examples to try in the folder `Examples` of the aac library.

Caution: Patches containing aac modules must be saved with the option **With MN** so as to save the scripts with the patch.

# 7 References

[App92]
Apple Events and AppleScript Programming Tutorial, Apple Computer, Inc. 1992.

[App93]
AppleScript Documentations,  Apple Computer, Inc, 1993.

[App93]
AppleScript Developper's Toolkit, Apple Computer, Inc, 1993.

# Index

## *W*