# Package: ProTrackR (via r-universe)

September 12, 2024

**Type** Package

**Title** Manipulate and Play 'ProTracker' Modules

**Version** 0.4.3

**Date** 2024-02-15

**Author** Pepijn de Vries [aut, cre, dtc] (0000-0002-7961-6646)

**Maintainer** Pepijn de Vries <pepijn.devries@outlook.com>

**Description** 'ProTracker' is a popular music tracker to sequence music on a Commodore Amiga machine. This package offers the opportunity to import, export, manipulate and play 'ProTracker' module files. Even though the file format could be considered archaic, it still remains popular to this date. This package intends to contribute to this popularity and therewith keeping the legacy of 'ProTracker' and the Commodore Amiga alive.

**License** GPL (>= 3)

**LazyData** True

**Depends** audio, lattice, signal, tuneR (>= 1.0)

**Imports** graphics, methods, stats, utils, XML

**Suggests** AmigaFFH (>= 0.2.0)

**Encoding** UTF-8

**RoxygenNote** 7.2.3

**Roxygen** list(markdown = TRUE)

**NeedsCompilation** no

**URL** https://pepijn-devries.github.io/ProTrackR/, https://github.com/pepijn-devries/ProTrackR/

**BugReports** https://github.com/pepijn-devries/ProTrackR/issues

**Repository** https://pepijn-devries.r-universe.dev

**RemoteUrl** https://github.com/pepijn-devries/ProTrackR

**RemoteRef** master

**RemoteSha** 52af90e9580d34db3cb17b9d03010173c7b90001

# Contents

---

`appendPattern` *Append a PTPattern to a PTModule*

---

## Description

Appends a specified `PTPattern` to a `PTModule`.

## Usage

```
## S4 method for signature 'PTModule,PTPattern'
appendPattern(x, pattern)
```

## Arguments

| | |
|---|---|
| `x` | A `PTModule` object to which a `PTPattern` is to be appended. |
| `pattern` | A `PTPattern` object which is to be appended to the `PTModule` `x`. |

## Details

Depending on the trackerFlag, a ProTracker module can hold either 64 or 100 pattern tables. As long as the number of pattern tables is below this maximum, new pattern tables can be added to the module with this function.

The patternOrder table should hold the maximum index of the available pattern tables in a module, otherwise, the module is not valid. As the maximum index increases, by appending a pattern table, the patternOrder table should be updated. The appendPattern method does this automatically, by replacing the first non-unique index in the order table, outside the current order table's length, with the new maximum index. If this is not possible, the highest element in the order table is set to hold the maximum index.

## Value

Returns a PTModule, to which the PTPattern is appended.

## Note

As per ProTracker specification, the pattern indices stored in the PTModule and obtained with patternOrder start at 0. Whereas R starts indexing at 1. Beware of this discrepancy.

## Author(s)

Pepijn de Vries

## See Also

Other pattern.operations: MODPlugToPTPattern(), PTPattern-class, PTPattern-method, PTPatternToMODPlug(), deletePattern(), pasteBlock(), patternLength(), patternOrderLength(), patternOrder()

Other module.operations: PTModule-class, clearSamples(), clearSong(), deletePattern(), fix.PTModule(), modToWave(), moduleSize(), patternLength(), patternOrderLength(), patternOrder(), playMod(), playingtable(), rawToPTModule(), read.module(), trackerFlag(), write.module()

## Examples

```
data("mod.intro")

## append an empty pattern to mod.intro

mod.intro <- appendPattern(mod.intro, new("PTPattern"))

## append a copy of pattern # 1 (this is pattern #0 in the
## patternOrder table) to mod.intro

mod.intro <- appendPattern(mod.intro, PTPattern(mod.intro, 1))
```

`as.character` *Character representation of ProTrackR objects*

## Description

Create a `character` representation of `PTCell`, `PTTrack` or `PTPattern` objects.

## Usage

```
## S4 method for signature 'PTCell'
as.character(x)

## S4 method for signature 'PTTrack'
as.character(x)

## S4 method for signature 'PTPattern'
as.character(x)
```

## Arguments

| | |
|---|---|
| `x` | An object of any of the following classes: `PTCell`, `PTTrack` or `PTPattern`. |

## Details

A `PTCell` is an element of a `PTTrack` which in turn is an element of a `PTPattern`. A `PTPattern` tells a tracker which sample to play at which frequency on which of the four audio channels and with which effects. A `PTCell` in essence holds all this information as described at the documentation of the `PTCell`.

Data in these objects are stored in these objects in a `raw` form, to save working memory and to comply to the ProTracker file specifications. As the raw data is not easy to interpret, this method is provided to make your life (and the interpretation of the objects) easier.

This method generates a character representation of each of the three objects. These character representations can be coerced back to their original classes with the following methods: `PTCell-method`, `PTTrack-method` and `PTPattern-method`.

## Value

Returns a single character string when `x` is of class `PTCell`.

Returns a `vector` of length 64 of the type `character` when `x` is of class `PTTrack`.

Returns a 64 by 4 `matrix` of the type `character` when `x` is of class `PTPattern`.

## Author(s)

Pepijn de Vries

## See Also

Other character.operations: name, periodToChar(), rawToCharNull(), sampleRate

Other track.operations: PTTrack-method

## Examples

```
data("mod.intro")

as.character(   PTCell(mod.intro, 1, 1, 1))

as.character(PTTrack(mod.intro, 1, 1))

as.character(PTPattern(mod.intro, 1))
```

---

# as.raw *Extract and replace raw data*

---

## Description

Information of PTCell, PTTrack and PTPattern objects are stored as raw values. This method can be used to extract and replace this raw data. PTModule objects can also be converted to raw data but not replaced by it.

## Usage

```
## S4 method for signature 'PTCell'
as.raw(x)

## S4 replacement method for signature 'PTCell,raw'
as.raw(x) <- value

## S4 method for signature 'PTTrack'
as.raw(x)

## S4 replacement method for signature 'PTTrack,matrix'
as.raw(x) <- value

## S4 method for signature 'PTPattern'
as.raw(x)

## S4 replacement method for signature 'PTPattern,matrix'
as.raw(x) <- value

## S4 method for signature 'PTModule'
as.raw(x)
```

## Arguments

| | |
|---|---|
| x | A PTCell, PTTrack or PTPattern object, for which the raw data needs to extracted or replaced. A PTModule object is also allowed, but this object cannot be replaced. |
| value | raw data with which the raw data in object x needs to be replaced.<br>If x is a PTCell object, value should be a vector of four raw values (conform specifications provided at the documentation of the PTCell).<br>If x is a PTTrack object, value should be a 64 by 4 matrix holding raw values (conform specifications provided at the documentation of the PTTrack).<br>If x is a PTPattern object, value should be a 64 by 16 matrix holding raw values (conform specifications provided at the documentation of the PTPattern). |

## Details

A PTCell is an element of a PTTrack which in turn is an element of a PTPattern. A PTPattern tells a tracker which sample to play at which frequency on which of the four audio channels and with which effects. A PTCell in essence holds all this information as described at the documentation of the PTCell.

Data in these objects are stored in these objects in a raw form, to save working memory and to comply to the ProTracker file specifications (see documentation of each of these classes for more details). This method can be used to extract and replace raw data.

The PTModule object has a more complex structure but can also be converted into raw data (the way it would be stored in a ProTracker module file). However, this object cannot be replaced by raw data.

## Value

For as.raw, a length 4 vector, 64 by 4 matrix or a 64 by 16 matrix of raw data is returned, when x is of class PTCell, PTTrack or PTPattern, respectively.

If x is a PTModule object, the raw data returned will have the same format as the ProTracker file format.

For as.raw<-, a copy of object x is returned in which the raw data is replaced by value.

## Author(s)

Pepijn de Vries

## See Also

Other raw.operations: nybbleToSignedInt(), nybble(), rawToCharNull(), rawToPTModule(), rawToSignedInt(), rawToUnsignedInt(), signedIntToNybble(), signedIntToRaw(), unsignedIntToRaw()

## Examples

```
data("mod.intro")

## Get the raw data of the PTCell at
```

```
## pattern #1, track #1 and row #1
## of mod.intro:
as.raw(PTCell(mod.intro, 1, 1, 1))

## idem for PTTrack #1 of pattern #1:
as.raw(PTTrack(mod.intro, 1, 1))

## idem for PTPattern #1:
as.raw(PTPattern(mod.intro, 1))

## replace raw data of PTCell 1, 1, 1
## with that of PTCell 2, 1, 1:
as.raw(PTCell(mod.intro, 1, 1, 1)) <-
  as.raw(PTCell(mod.intro, 2, 1, 1))
```

---

`clearSamples` *Clear all samples from module*

---

## Description

Remove all `PTSamples` from a `PTModule` object.

## Usage

```
## S4 method for signature 'PTModule'
clearSamples(mod)
```

## Arguments

| | |
|---|---|
| `mod` | A `PTModule` object from which all samples needs to be removed. |

## Details

Conform the original ProTracker, this method removes all patterns `PTSamples` from a module. You keep all patterns (`PTPattern`) and `patternOrder` info.

## Value

Returns a copy of object `mod` in which all samples are removed.

## Author(s)

Pepijn de Vries

## See Also

Other module.operations: `PTModule-class`, `appendPattern()`, `clearSong()`, `deletePattern()`, `fix.PTModule()`, `modToWave()`, `moduleSize()`, `patternLength()`, `patternOrderLength()`, `patternOrder()`, `playMod()`, `playingtable()`, `rawToPTModule()`, `read.module()`, `trackerFlag()`, `write.module()`

## Examples

```
data(mod.intro)

## 'clear.mod' is a copy of 'mod.intro' without the
## samples. It still holds all pattern tables and
## pattern order info.
clear.mod <- clearSamples(mod.intro)
```

---

`clearSong` *Clear all pattern info from module*

---

## Description

Remove all patterns (`PTPattern`) and `patternOrder` info from a `PTModule` object.

## Usage

```
## S4 method for signature 'PTModule'
clearSong(mod)
```

## Arguments

| | |
|---|---|
| `mod` | A `PTModule` object from which all pattern (order) info needs to be removed. |

## Details

Conform the original ProTracker, this method removes all patterns (`PTPattern`) and `patternOrder` info from a module. You keep the audio `PTSamples`.

## Value

Returns a copy of object `mod` in which all pattern (order) info is removed.

## Author(s)

Pepijn de Vries

## See Also

Other module.operations: `PTModule-class`, `appendPattern()`, `clearSamples()`, `deletePattern()`, `fix.PTModule()`, `modToWave()`, `moduleSize()`, `patternLength()`, `patternOrderLength()`, `patternOrder()`, `playMod()`, `playingtable()`, `rawToPTModule()`, `read.module()`, `trackerFlag()`, `write.module()`

## Examples

```
data(mod.intro)

## 'clear.mod' is a copy of 'mod.intro' without the
## pattern (order) info. It still has the audio samples.
clear.mod <- clearSong(mod.intro)
```

deletePattern *Remove a PTPattern table from a PTModule object*

## Description

This method removes a PTPattern from a PTModule object and updates the patternOrder table accordingly.

## Usage

```
## S4 method for signature 'PTModule,numeric'
deletePattern(x, index)
```

## Arguments

| | |
|---|---|
| x | A PTModule from which a PTPattern needs to be removed. |
| index | A numeric index of the PTPattern table that needs to be removed. The index should be between 1 and patternLength. It's not possible to delete multiple patterns simultaneously with this method. A PTModule should always hold at least 1 pattern table, therefore, the last PTPattern table cannot be deleted. |

## Details

This method safely removes a PTPattern from a PTModule object, guarding the validity of the PTModule object. It therefore also updates the patternOrder table, by renumbering the indices listed there. The index of the removed object is replaced with a zero in the patternOrder table.

## Value

Returns a PTModule from which the selected PTPattern is deleted.

## Note

As per ProTracker specification, the pattern indices stored in the PTModule and obtained with patternOrder start at 0. Whereas R starts indexing at 1. Beware of this discrepancy.

## Author(s)

Pepijn de Vries

## See Also

Other pattern.operations: MODPlugToPTPattern(), PTPattern-class, PTPattern-method, PTPatternToMODPlug(), appendPattern(), pasteBlock(), patternLength(), patternOrderLength(), patternOrder()

Other module.operations: PTModule-class, appendPattern(), clearSamples(), clearSong(), fix.PTModule(), modToWave(), moduleSize(), patternLength(), patternOrderLength(), patternOrder(), playMod(), playingtable(), rawToPTModule(), read.module(), trackerFlag(), write.module()

### Examples

```
data("mod.intro")
print(mod.intro)

## delete pattern #2 from mod.intro:

mod.intro <- deletePattern(mod.intro, 2)
print(mod.intro)
```

---

## effect *Extract or replace effect/trigger codes*

---

### Description

The 3 right-hand symbols of a `character` representation of a `PTCell` represent an effect or trigger code. This method can be used to extract or replace this code.

### Usage

```
## S4 method for signature 'PTCell'
effect(x)

## S4 replacement method for signature 'PTCell,character'
effect(x) <- value
```

### Arguments

| | |
|---|---|
| `x` | A `PTCell` from which the effect code needs to be extracted. |
| `value` | A `character` string containing a three hexadecimal digit effect code. All hexadecimal codes are accepted, not all will produce meaningful effects. |

### Details

When a `PTCell` is represented by a `character` string, the last three symbols represent a hexadecimal effect or trigger code. In general the first of the three symbols indicates a type of effect or trigger, whereas the latter two generally indicate a magnitude or a position for effects and triggers.

Effects can for instance be volume or frequency slides. The codes can also affect the module tempo or cause position jumps.

When replacing this code, all three digit hexadecimal `character` strings are accepted, although not all codes will represent a valid effect or trigger. See https://wiki.openmpt.org/Manual:_Effect_Reference#MOD_Effect_Commands for a valid list of effect codes.

### Value

For `effect`, a `character` string with the three hexadecimal digit effect code will be returned.

For `effect<-`, a copy of object `x` with effect code `value` will be returned.

## Author(s)

Pepijn de Vries

## See Also

Other cell.operations: PTCell-class, PTCell-method, note(), sampleNumber()

## Examples

```
data("mod.intro")

## the PTCell in row #1, of pattern #1, track #1
## has effect code "A08", which is a volume slide down (0xA)
## with speed 0x8:
effect(PTCell(mod.intro, 1, 1, 1))

## this is how you can change an effect:
cell <- PTCell("C-2 01 000")
effect(cell) <- "C20"

## the above expression sets the volume (effect 0xC)
## to 50% (0x20 which is halve of the maximum 0x40)
```

---

# fineTune *Fine tune a PTSample*

---

## Description

Extract or replace the fine tune value of a PTSample.

## Usage

```
## S4 method for signature 'PTSample'
fineTune(sample)

## S4 replacement method for signature 'PTSample,numeric'
fineTune(sample) <- value
```

## Arguments

| | |
|---|---|
| sample | A PTSample for which the fine tune value needs to be extracted or replace. |
| value | A numeric value ranging from -8 up to 7, representing the fine tune. |

## Details

PTSamples can be tuned with their fine tune values. The values range from -8 up to 7 and affect the playback sample rate of specific notes (see period_table). This method can be used to extract this value, or to safely replace it.

### Value

For `fineTune` the fine tune value, represented by an `integer` value ranging from -8 up to 7, is returned.

For `fineTune<-` A `PTSample` `sample`, updated with the fine tune `value`, is returned.

### Author(s)

Pepijn de Vries

### See Also

Other sample.operations: `PTSample-class`, `PTSample-method`, `loopLength()`, `loopSample()`, `loopStart()`, `loopState()`, `name`, `playSample()`, `read.sample()`, `sampleLength()`, `volume()`, `waveform()`, `write.sample()`

### Examples

```
data("mod.intro")

## get the finetune of the first sample of mod.intro:

fineTune(PTSample(mod.intro, 1))

## Let's tweak the finetune of the first sample of
## mod.intro to -1:

fineTune(PTSample(mod.intro, 1)) <- -1
```

---

## fix.PTModule *Attempt to fix PTModule to ProTracker specs*

---

### Description

Try to fix non-valid `PTModule` objects in order to meet with ProTracker specs such that they pass validity tests.

### Usage

```
## S4 method for signature 'PTModule,logical'
fix.PTModule(mod, verbose = T)

## S4 method for signature 'PTModule,missing'
fix.PTModule(mod)
```

## Arguments

| | |
|---|---|
| mod | A PTModule object which needs fixing. |
| verbose | With the default value of TRUE, the method prints a progress report to the sink. When set to FALSE, the progress report is suppressed. |

## Details

Almost any file can be read as a PTModule object (using read.module) when validity is ignored and no unexpected end of file is reached. This package's object validity are very strickly testing for compliance with ProTracker specifications. As many modules could have been created with other trackers (which often will play just as well in ProTracker) it is desirable to convert such object to ProTracker specs. This method attempts to do so, by fixing each aspect, that is also tested in the object validity functions. Note that the attempts are no guarantee for success, and 'fixed' modules may not play as intended.

## Value

Returns a copy of object mod in which all non-conformaties are attempted to be fixed. (Attempted) fixes are listed printed in the progress report.

## Note

In the current version, pattern data itself is not checked for non-conformaties nor is it fixed.

## Author(s)

Pepijn de Vries

## See Also

Other module.operations: PTModule-class, appendPattern(), clearSamples(), clearSong(), deletePattern(), modToWave(), moduleSize(), patternLength(), patternOrderLength(), patternOrder(), playMod(), playingtable(), rawToPTModule(), read.module(), trackerFlag(), write.module()

## Examples

```
## Not run:
data("mod.intro")

## Let's do something illegal and destroy mod.intro:
mod.intro@pattern.order <- mod.intro@pattern.order[1:9]

## We should have used the 'patternOrder'-method to
## change the pattern order. Now we have broken the
## object:
validObject(mod.intro, TRUE)

## No worries, we can fix it:
mod.intro <- fix.PTModule(mod.intro)
```

```
## See, it's all OK again:
validObject(mod.intro, TRUE)

## End(Not run)
```

---

funk_table *ProTracker Funk Table*

---

## Description

Small list of numbers used by an obscure audio effect in ProTracker

## Format

A `numeric vector` of length 16 holding values to be used in ProTracker funk repeat effects.

## Details

This dataset is included for completeness sake. It is not yet used by any class, method or function in the ProTrackR package. It may very well be obsolete for recent ProTracker versions.

## References

https://fossies.org/linux/uade/amigasrc/players/tracker/eagleplayers/mod32_protracker/PTK_versions.txt

## Examples

```
data("funk_table")
```

---

loopLength *The loop length of a PTSample*

---

## Description

Extract or replace the loop length of a `PTSample`.

## Usage

```
## S4 method for signature 'PTSample'
loopLength(sample)

## S4 replacement method for signature 'PTSample'
loopLength(sample) <- value
```

## Arguments

| | |
|---|---|
| sample | A PTSample for which the loop length needs to be extracted or replace. |
| value | An even numeric value giving the loop length in samples ranging from 2 up to 131070 (It can be 0 when the sample is empty). The sum of the loopStart and loopLength should not exceed the sampleLength.<br>Use a value of either character "off" or logical FALSE, in order to turn off the loop all together. |

## Details

PTSamples can have loops, marked by a starting position and length of the loop (in samples), for more details see the PTSample. This method can be used to extract the loop length or safely replace its value.

## Value

For loopLength the loop length (in samples), represented by an even integer value ranging from 0 up to 131070, is returned.

For loopLength<- A PTSample sample, updated with the loop length value, is returned.

## Author(s)

Pepijn de Vries

## See Also

Other loop.methods: loopSample(), loopStart(), loopState()

Other sample.operations: PTSample-class, PTSample-method, fineTune(), loopSample(), loopStart(), loopState(), name, playSample(), read.sample(), sampleLength(), volume(), waveform(), write.sample()

## Examples

```
data("mod.intro")

## get the loop length of the
## first sample of mod.intro:

loopLength(PTSample(mod.intro, 1))

## Let's change the length of
## the loop to 200

loopLength(PTSample(mod.intro, 1)) <- 200

## Let's turn off the loop all together:

loopLength(PTSample(mod.intro, 1)) <- FALSE
```

loopSample *Looped waveform of a sample*

## Description

Generate a looped waveform of a PTSample object.

## Usage

```
## S4 method for signature 'PTSample'
loopSample(sample, times, n_samples)
```

## Arguments

| | |
|---|---|
| sample | A PTSample object that needs to be looped. |
| times | A positive integer value indicating the number of times a sample loop should be repeated. This argument is ignored if n_samples is specified. |
| n_samples | A positive integer value indicating the desired length of the looped waveform in number of samples. This argument overrules the times argument. |

## Details

For playing routines, it can be useful to generate repeats of a sample loop. This method returns the waveform of a PTSample where the loop is repeated times`' times or has a length of n_samples‘’.

## Value

Returns a waveform represented by a numeric vector of values ranging from 0 up to 255. Has a length of n_samples when that argument is specified.

## Author(s)

Pepijn de Vries

## See Also

Other loop.methods: loopLength(), loopStart(), loopState()

Other sample.operations: PTSample-class, PTSample-method, fineTune(), loopLength(), loopStart(), loopState(), name, playSample(), read.sample(), sampleLength(), volume(), waveform(), write.sample()

## Examples

```
data("mod.intro")

## Loop sample number 4 10 times:
wform <- loopSample(PTSample(mod.intro, 4), times = 10)
plot(wform, type = "l")

## Loop sample number 4, such that its
## final length is 5000 samples:
wform <- loopSample(PTSample(mod.intro, 4), n_samples = 5000)
plot(wform, type = "l")
```

---

`loopStart` *The loop start position of a PTSample*

---

## Description

Extract or replace the loop start position of a `PTSample`.

## Usage

```
## S4 method for signature 'PTSample'
loopStart(sample)

## S4 replacement method for signature 'PTSample'
loopStart(sample) <- value
```

## Arguments

| | |
|---|---|
| `sample` | A `PTSample` for which the loop start position needs to be extracted or replace. |
| `value` | An even `numeric` value giving the loop starting position in samples ranging from 0 up to 131070. The sum of the `loopStart` and `loopLength` should not exceed the `sampleLength`.<br>Use a `value` of either `character` `"off"` or `logical` `FALSE`, in order to turn off the loop all together. |

## Details

`PTSample`s can have loops, marked by a starting position and length of the loop (in samples), for more details see the `PTSample`. This method can be used to extract the loop starting position or safely replace its value.

## Value

For `loopStart` the loop start position (in samples), represented by an even `integer` value ranging from 0 up to 131070, is returned.

For `loopStart<-` A `PTSample` `sample`, updated with the loop start position "value'', is returned.

### Author(s)

Pepijn de Vries

### See Also

Other sample.operations: PTSample-class, PTSample-method, fineTune(), loopLength(), loopSample(), loopState(), name, playSample(), read.sample(), sampleLength(), volume(), waveform(), write.sample()

Other loop.methods: loopLength(), loopSample(), loopState()

### Examples

```
data("mod.intro")

## get the loop start position of the
## first sample of mod.intro:

loopStart(PTSample(mod.intro, 1))

## Let's change the starting position of
## the loop to 500

loopStart(PTSample(mod.intro, 1)) <- 500

## Let's turn off the loop all together:

loopStart(PTSample(mod.intro, 1)) <- FALSE
```

---

## loopState *Get PTSample loop state*

---

### Description

Determines whether a loop is specified for a PTSample object.

### Usage

```
## S4 method for signature 'PTSample'
loopState(sample)
```

### Arguments

| | |
|---|---|
| sample | A PTSample object for which the loop state needs to be determined. |

### Details

The loop state is not explicitly stored in a PTSample object. It can be derived from the loopStart position and loopLength. This method is provided as a convenient method to get the state. Use either loopStart or loopLength to change the state.

## Value

Returns a logical value indicating whether a loop is (TRUE) or isn't (FALSE) specified for the sample.

## Author(s)

Pepijn de Vries

## See Also

Other loop.methods: loopLength(), loopSample(), loopStart()

Other sample.operations: PTSample-class, PTSample-method, fineTune(), loopLength(), loopSample(), loopStart(), name, playSample(), read.sample(), sampleLength(), volume(), waveform(), write.sample()

## Examples

```
data("mod.intro")

## Get the loop status of sample number 1
## (it has a loop):
loopState(PTSample(mod.intro, 1))

## Get the loop status of sample number 2
## (it has no loop):
loopState(PTSample(mod.intro, 2))
```

---

# mod.intro *Example of a PTModule object*

---

## Description

A PTModule object included in the package as example.

## Format

A PTModule object containing 4 PTSample objects (and 27 empty PTSample objects, adding up to the 31 samples a PTModule should hold) and 4 PTPattern objects.

## Details

This PTModule object is based on an original ProTracker module file I've composed in the late nineteen nineties. It is used as example for many of the ProTrackR methods and you can use it to test your own code. It can also be exported back to the original ProTracker module file by using write.module.

## Author(s)

Pepijn de Vries

## Examples

```
data("mod.intro")
print(mod.intro)
plot(mod.intro)

## Not run:
playSample(mod.intro)

## Save as an original module file,
## which can be played with ProTracker (or several modern audio players):
write.module(mod.intro, "intro.mod")

## End(Not run)
```

---

modArchive *ModArchive helper functions*

---

## Description

https://ModArchive.org is on of the largest online archive of module files. These functions will assist in accessing this archive.

## Usage

```
modArchive.info(mod.id, api.key)

modArchive.download(mod.id, ...)

modArchive.search.mod(
  search.text,
  search.where = c("filename_or_songtitle", "filename_and_songtitle", "filename",
    "songtitle", "module_instruments", "module_comments"),
  format.filter = c("unset", "669", "AHX", "DMF", "HVL", "IT", "MED", "MO3", "MOD",
    "MTM", "OCT", "OKT", "S3M", "STM", "XM"),
  size.filter = c("unset", "0-99", "100-299", "300-599", "600-1025", "1025-2999",
    "3072-6999", "7168-100000"),
  genre.filter = "deprecated",
  page,
  api.key
)

modArchive.request.count(api.key)

modArchive.max.requests(api.key)

modArchive.view.by(
  view.query,
```

```
  view.by = c("view_by_list", "view_by_rating_comments", "view_by_rating_reviews",
     "view_modules_by_artistid", "view_modules_by_guessed_artist"),
  format.filter = c("unset", "669", "AHX", "DMF", "HVL", "IT", "MED", "MO3", "MOD",
     "MTM", "OCT", "OKT", "S3M", "STM", "XM"),
  size.filter = c("unset", "0-99", "100-299", "300-599", "600-1025", "1025-2999",
     "3072-6999", "7168-100000"),
   page,
   api.key
)

modArchive.search.genre(
  genre.filter = c("unset", "Alternative", "Gothic", "Grunge", "Metal - Extreme",
    "Metal (general)", "Punk", "Chiptune", "Demo Style", "One Hour Compo", "Chillout",
     "Electronic - Ambient", "Electronic - Breakbeat", "Electronic - Dance",
    "Electronic - Drum and Bass", "Electronic - Gabber", "Electronic - Hardcore",
     "Electronic - House", "Electronic - IDM", "Electronic - Industrial",
     "Electronic - Jungle", "Electronic - Minimal", "Electronic - Other",
     "Electronic - Progressive", "Electronic - Rave", "Electronic - Techno",

     "Electronic (general)", "Trance - Acid", "Trance - Dream", "Trance - Goa",
    "Trance - Hard", "Trance - Progressive", "Trance - Tribal", "Trance (general)",
    "Big Band", "Blues", "Jazz - Acid", "Jazz - Modern", "Jazz (general)", "Swing",
    "Bluegrass", "Classical", "Comedy", "Country", "Experimental", "Fantasy", "Folk",
    "Fusion", "Medieval", "New Ages", "Orchestral", "Other", "Piano", "Religious",
    "Soundtrack", "Spiritual", "Video Game", "Vocal Montage", "World", "Ballad", "Disco",
     "Easy Listening",
      "Funk", "Pop - Soft", "Pop - Synth", "Pop (general)",
    "Rock - Hard", "Rock - Soft", "Rock (general)", "Christmas", "Halloween", "Hip-Hop",
     "R and B", "Reggae", "Ska", "Soul"),
  format.filter = c("unset", "669", "AHX", "DMF", "HVL", "IT", "MED", "MO3", "MOD",
     "MTM", "OCT", "OKT", "S3M", "STM", "XM"),
  size.filter = c("unset", "0-99", "100-299", "300-599", "600-1025", "1025-2999",
     "3072-6999", "7168-100000"),
   page,
   api.key
)

modArchive.search.artist(search.artist, page, api.key)

modArchive.search.hash(search.hash, api.key)

modArchive.random.pick(
   genre.filter = c("Alternative", "Gothic", "Grunge", "Metal - Extreme",
    "Metal (general)", "Punk", "Chiptune", "Demo Style", "One Hour Compo", "Chillout",
     "Electronic - Ambient", "Electronic - Breakbeat", "Electronic - Dance",
    "Electronic - Drum and Bass", "Electronic - Gabber", "Electronic - Hardcore",
     "Electronic - House", "Electronic - IDM", "Electronic - Industrial",
     "Electronic - Jungle", "Electronic - Minimal", "Electronic - Other",
```

```
    "Electronic - Progressive", "Electronic - Rave", "Electronic - Techno",
    "Electronic (general)",
     "Trance - Acid", "Trance - Dream", "Trance - Goa",
   "Trance - Hard", "Trance - Progressive", "Trance - Tribal", "Trance (general)",
   "Big Band", "Blues", "Jazz - Acid", "Jazz - Modern", "Jazz (general)", "Swing",
   "Bluegrass", "Classical", "Comedy", "Country", "Experimental", "Fantasy", "Folk",
   "Fusion", "Medieval", "New Ages", "Orchestral", "Other", "Piano", "Religious",
   "Soundtrack", "Spiritual", "Video Game", "Vocal Montage", "World", "Ballad", "Disco",
    "Easy Listening", "Funk", "Pop - Soft",
     "Pop - Synth", "Pop (general)",
   "Rock - Hard", "Rock - Soft", "Rock (general)", "Christmas", "Halloween", "Hip-Hop",
    "R and B", "Reggae", "Ska", "Soul"),
 format.filter = c("unset", "669", "AHX", "DMF", "HVL", "IT", "MED", "MO3", "MOD",
   "MTM", "OCT", "OKT", "S3M", "STM", "XM"),
 size.filter = c("unset", "0-99", "100-299", "300-599", "600-1025", "1025-2999",
   "3072-6999", "7168-100000"),
 api.key
)
```

## Arguments

| | |
|---|---|
| `mod.id` | An `integer` code used as module identifier in the ModArchive database. A `mod.id` can be obtained by performing a search with `modArchive.search.mod`. When downloading a module, make sure that the identifier represents a MOD file, as other types will result in an error. |
| `api.key` | Most ModArchive functions require a personal secret API key. This key can be obtained from the ModArchive forum. See 'ModArchive API Key' section below for instructions on how to obtain such a key. |
| `...` | arguments that are passed on to `read.module`. |
| `search.text` | A `character` string to be used as terms to search in the ModArchive. |
| `search.where` | A `character` string indicating where in the module files to search for the `search.text`. See usage section for the available options. |
| `format.filter` | File format filter to be used in a search in the ModArchive. See the usage section for all possible options. Default is "unset" (meaning that it will search for any file format). Note that only the 'MOD' format is supported by this package. |
| `size.filter` | File size filter to be used in a search in the ModArchive. Needs to be a `character` string representation of a file size category as specified on ModArchive.org. See the usage section for all possible options. Default is "unset" (meaning that it will search for any file size). Note that the maximum file size of a module is approximately 4068 kilobytes, meaning that the largest file size category is irrelevant for 'MOD' files. Also note that the category names are inconsistant, these are the literal catagories used by ModArchive |
| `genre.filter` | Genre filter to be used in some of the overviews from the ModArchive. Needs to be a `character` string representation of a genre as specified on ModArchive.org. See the usage section for all possible options. This argument is deprecated in the function `modArchive.search` since ProTrackR version 0.3.4, other functions will still accept this argument. |

`page` Many of the ModArchive returns paginated tables. When this argument is omitted, the first page is returned. Use an integer value to return a specific page. The total number of pages of a search or view is returned as an attribute to the returned base::data.frame.

`view.query` A query to be used in combination with the `view.by` argument. Use the queries in combination with `view.by` as follows:

- `view_by_list`: Use a single capital starting letter to browse modules by name
- `view_by_rating_comments`: Provide a (user) rating by which you wish to browse the modules
- `view_by_rating_reviews`: Provide a (reviewer) rating by which you wish to browse the modules
- `view_modules_by_artistid`: Provide an artist id number for whom you wish to browse his/her modules
- `view_modules_by_guessed_artist`: Provide an artist guessed name for whom you wish to browser his/her modules

`view.by` Indicate how the `modArchive.view.by` function should sort the overview tables of modules. See 'usage' section for the possible options.

`search.artist` A character string representing the (guessed) artist name or id number that you ar looking for in the archive.

`search.hash` The MD5 hash code of the specific module you are looking for. See `https://modarchive.org/?xml-api-usage-level3` for details.

## Details

The `modArchive.info` function will retrieve info on a specific module from the ModArchive. The `modArchive.search.mod`, `modArchive.search.genre` and `modArchive.search.hash` functions can be used to find specific modules in the archive. Use `modArchive.random.pick` to get module info on a random module in the archive.

Use the `modArchive.view.by` function to browse the archive by specific aspects. Note that the ModArchive also contains file formats other than ProTracker's MOD format. This package can only handle the MOD format.

The `modArchive.download` function will download a module from the archive.

Use `modArchive.search.artist` to find artist details in the archive.

Use `modArchive.request.count` to determine how many request you have made in the current month with the specified key (see `ModArchive API key' section for details)`. Use modArchive.max.requeststo deter
API key' section for details).

## Value

`modArchive.info`, `modArchive.search.genre`, `modArchive.search.hash`, `modArchive.random.pick` and `modArchive.view.by` will return a data.frame containing information on modules in the ModArchive. Note that this data.frame is formatted differently since ProTrackR 0.3.4, which may cause backward compatibility issues.

`modArchive.download` will download a module and return it as a `PTModule` object.

`modArchive.search.artist` will return a `data.frame` containing information on artists on the ModArchive.

`modArchive.request.count` returns the number of ModArchive API request that are left for this month, for the provided key.

`modArchive.max.requests` returns the maximum monthly requests for the provided key.

## ModArchive API key

Since ProTrackR 0.3.4, the ModArchive helper functions have changed. In earlier version, a labile html scraper was used, in 0.3.4 and later, this is replaced by functions that more robustly use the Application Programming Interface (API) provided by ModArchive. There are some downsides to this new approach: a personal API key needs to be obtained from the ModArchive team; and the ProTrackR package relies on yet another package (XML) to parse the XML files that are returned by the API.

So why is this switch? Well, first of all, this approach is better supported by ModArchive. The personal API key is used to avoid excessive access by imposing a monthly request limit (keep in mind that ModArchive provides free services and is run by volunteers). The upside is that the XML files are a lot lighter than the html files returned by the regular website. Therefore, the new functions are faster, and they reduce the load on the ModArchive servers. The XML files also allow for easier access to more of the ModArchive functionality as implemented in the ModArchive helper functions described here.

So how do you get your personal API key? First, you need to register at the ModArchive Forums. Then follow the instructions provided in this topic on the forum. For more info, see also the API page on ModArchive.

If you want to search for module files without an API key, one could make use of to the modLand collection instead.

## Author(s)

Pepijn de Vries

## Examples

```
## Not run:
## most of the example below will fail as they require a
## real modArchive API key. The key used in these example
## is just a dummy. See details on how to get a key
## in the section 'ModArchive API Key' in the manual.

## Search for the module that is also used as
## an example in this package:
search.results <- modArchive.search.mod("*_-_intro.mod",
                                        size.filter = "0-99",
                                        format.filter = "MOD",
                                        api.key = "<your key here>")

## apparently there are multiple modules in
## database that have '_-_intro' in their
## file name or title. Select the wanted
```

```
## module from the list (the one with the
## word 'protrackr' in the instrument names):
search.select <- subset(search.results,
                        grepl("protrackr", search.results$instruments))

## get the same details, but now only for
## the specific module based on its ModArchive ID:
modArchive.info(search.select$id, api.key = "<your key here>")

## download the selected module from ModArchive.org:
mod <- modArchive.download(search.select$id)

## here's a randomly picked module from the ModArchive:
info.random <- modArchive.random.pick(api.key = "<your key here>")

## use modArchive.view.by to list the 2nd page
## of MOD files that start with the letter 'A'
info.list  <- modArchive.view.by("A", "view_by_list", "MOD",
                                 page = 2,
                                 api.key = "<your key here>")

## list the modules of the artist with id number 89200:
artist.mods <- modArchive.view.by("89200", "view_modules_by_artistid",
                                  format.filter = "MOD",
                                  api.key = "<your key here>")

## here's how you can list MOD files of a
## specific genre:
list.genre  <- modArchive.search.genre("Chiptune", "MOD",
                                       api.key = "<your key here>")

## get module info for a specific hash code
mod.hash    <- modArchive.search.hash("8f80bcab909f700619025bd7f2975749",
                                      "<your key here>")

## find modarchive artist info, search for artist name
## or artist id:
artist.list <- modArchive.search.artist("89200",
                                        api.key = "<your key here>")

## How many requests did I make this month?:
modArchive.request.count("<your key here>")

## How many requests am I allowed to make each month?:
modArchive.max.requests("<your key here>")

## End(Not run)
```

---

modLand *ModLand helper functions*

---

## Description

https://modland.com is on of the largest online archive of module files. These functions will assist in accessing this archive.

## Usage

```
modLand.search.mod(search.text)

modLand.download.mod(
  format,
  author,
  title,
  mirror = c("modland.com", "ftp.modland.com", "antarctica.no", "ziphoid.com",
    "exotica.org.uk"),
  ...
)
```

## Arguments

| | |
|---|---|
| search.text | A single length character vector, containing search text. Provided search pattern is searched in all fields (mod format, author and title). Prefixes can be added to keywords for inclusive or exclusive searches. For details see https://www.exotica.org.uk/wiki/Modland#Searching. Note that modLand contains a wide range of tracker files, only mod-files are supported by the ProTrackR package. It is therefore advisable to add the keyword 'mod' to the search string. |
| format | A single length character vector, indicating the tracker file format. "Protracker'' is the option that is most likely to work in this package. |
| author | A single length character vector, indicating the module author name. Can be obtained from a modLand.search.mod. |
| title | A single length character vector, indicating the module title. Can be obtained from a modLand.search.mod. |
| mirror | A single length character vector. Should contain one of the mirrors listed in the 'usage' section. Select a mirror site from which the module file needs to be downloaded. |
| ... | Argument that are passed on to read.module. |

## Details

Like the https://modarchive.org, modland provides access to a large collection of module files. Compared to the modArchive, modLand provides limited searching features. However, it does not require an API key.

The functions documented here are provided as a convenience and depend on third party services. Note that continuity of these services cannot be guaranteed.

Use modLand.search.mod to search through the modLand collection.

Use modLand.download.mod to download a specific mod file as an S4 object.

## Value

modLand.search.mod returns a data.frame. The data.frame contains a search result in each row. The data.frame contains a number of columns, each containing character strings. The column title' contains the mod file name; The column named author' contains the author name; the column named format' contains the tracker file format (only ``Protracker' is supported by this package); The collumn collect' contains modLand collections in which the mod is included; the c contains a download link for the ogg'-file generated on the modLand server from the mod file. Note that ogg-fil Land.download.mod' to download the mod file.

modLand.download.mod attempts to download the specified mod file and return it as a PTModule object. It will throw errors when the mod file is not available or when there are network problems...

## Author(s)

Pepijn de Vries

## Examples

```
## Not run:
## Search for a funky tune:

modland <- modLand.search.mod("elekfunk mod")

## The ogg file can be downloaded (in this case to the tempdir()),
## but it is not supported by the ProTrackR package...

utils::download.file(modland$url[1], tempdir())

## Instead, use the following approach to download the module:

mod <- modLand.download.mod(modland$format[1],
                            modland$author[1],
                            modland$title[1])


## End(Not run)
```

---

MODPlugToPTPattern *Convert MODPlug pattern into a PTPattern object*

---

## Description

Convert pattern data from text or clipboard, originating from the modern MODPlug tracker and convert it into a PTPattern or PTBlock object.

## Usage

```
MODPlugToPTPattern(text = NULL, what = c("PTPattern", "PTBlock"))
```

## Arguments

| | |
|---|---|
| text | A vector of characters, representing MOD pattern data obtained from Open-MPT. If set to NULL (default), the text will be read from the system's clipboard. |
| what | A character string that indicates what type of object should be returned. Can be "PTPattern" or "PTBlock". |

## Details

The Open MODPlug Tracker (https://openmpt.org) is a modern music tracker that is for free. It too can handle ProTracker modules. This function assists in moving pattern data from Open MPT to R.

Simply select and copy the pattern data to the system's clipboard and use this function to import it to R as a PTPattern or PTBlock object.

## Value

Depending on the value of the argument what, it will return either a PTPattern or PTBlock object.

## Author(s)

Pepijn de Vries

## See Also

Other MODPlug.operations: PTPatternToMODPlug()

Other pattern.operations: PTPattern-class, PTPattern-method, PTPatternToMODPlug(), appendPattern(), deletePattern(), pasteBlock(), patternLength(), patternOrderLength(), patternOrder()

## Examples

```
## Not run:
## This is what Mod Plug Pattern data looks like on
## the system's clipboard:
modPlugPattern <- c("ModPlug Tracker MOD",
                    "|C-601...A08|C-602...C40|A#403...F06|A#504......",
                    "|...01...A08|C-602...C30|........A01|........A02",
                    "|...01...A08|C-602......|........A01|C-604......",
                    "|...........|C-602......|........A02|........A02",
                    "|...01...A08|C-602......|........120|D-604......",
                    "|...........|A#504...C08|........A02|........A02",
                    "|...01...A08|C-602......|........220|D#604......",
                    "|...........|A#504...C08|........A01|........A02",
                    "|...01...A08|C-602......|........A01|F-604......",
                    "|...........|A#604...C08|........A01|........A02",
                    "|...01...A08|C-602......|........A01|D#604......",
                    "|...........|G-604...C08|........A01|........A02",
                    "|G-601......|C-602......|........A01|D-604......",
                    "|........A08|F-604...C08|...........|........A02",
                    "|F-601......|C-602......|...........|C-604......",
                    "|........A08|A#504...C08|...........|........A02",
```

```
                    "|C-601...A08|C-602...C40|A#403...F06|A#504......",
                    "|...01...A08|C-602...C30|........A01|........A02",
                    "|...01...A08|C-602......|........A01|D-604......",
                    "|...........|C-602......|........A02|........A02",
                    "|...01...A08|C-602......|........120|F-504......",
                    "|...........|A#504...C08|........A02|........A02",
                    "|...01...A08|C-602......|........220|G-504......",
                    "|...........|A#504...C08|........A01|........A02",
                    "|...01...A08|C-602......|........A01|A#504......",
                    "|...........|A#604...C08|........A01|........A01",
                    "|...01...A08|C-602......|........A01|...........",
                    "|...........|G-604...C08|........A01|........A01",
                    "|G-501......|C-602......|........A01|...........",
                    "|........A08|F-504...C08|...........|........A01",
                    "|A-501......|C-602......|...........|...........",
                    "|........A08|G-504...C08|...........|........A01",
                    "|E-601...A08|C-602...C40|D-503......|D-604......",
                    "|...01...A08|C-602...C30|........A01|........A02",
                    "|...01...A08|C-602......|........A01|E-604......",
                    "|...........|C-602......|........A02|........A02",
                    "|...01...A08|C-602......|........126|F#604......",
                    "|...........|D-604...C08|........A02|........A02",
                    "|...01...A08|C-602......|........226|G-604......",
                    "|...........|E-604...C08|........A01|........A02",
                    "|...01...A08|C-602......|........A01|A-604......",
                    "|...........|D-604...C08|........A01|........A02",
                    "|...01...A08|C-602......|........A01|G-604......",
                    "|...........|D-604...C08|........A01|........A02",
                    "|B-601......|C-602......|........A01|F#604......",
                    "|........A08|D-604...C08|...........|........A02",
                    "|A-601......|C-602......|...........|E-604......",
                    "|........A08|E-504...C08|...........|........A02",
                    "|D-601...A08|C-602...C40|C-503......|C-604......",
                    "|...01...A08|C-602...C30|........A01|........A02",
                    "|...01...A08|C-602......|........A01|D-604......",
                    "|...........|C-602......|........A02|........A02",
                    "|...01...A08|C-602......|........12B|E-604......",
                    "|...........|G-604...C08|........A02|........A02",
                    "|...01...A08|C-602......|........22B|F-604......",
                    "|...........|G-604...C08|........A01|........A02",
                    "|...01...A08|C-602......|........A01|G-604......",
                    "|...........|E-604...C08|........A01|........A02",
                    "|...01...A08|C-602......|........A01|F-604......",
                    "|...........|C-604...C08|........A01|........A02",
                    "|A-601......|C-602......|........A01|E-604......",
                    "|........A08|G-604...C08|...........|........A02",
                    "|G-601......|F-604...C08|...........|D-604......",
                    "|........A08|C-604...C08|...........|........A02")

## You could read it directly from the clipboard,
## by leaving text NULL (default). Here we provide
## the text specified above:
pat <- MODPlugToPTPattern(modPlugPattern, "PTPattern")
```

```
## look it is a "PTPattern" object now:
class(pat)

## we can also only import the first 10 lines as a
## PTBlock:
blk <- MODPlugToPTPattern(modPlugPattern[1:10], "PTBlock")

## End(Not run)
```

---

modToWave *Convert a PTModule object into an audio Wave object*

---

## Description

Converts a PTModule object into a tuneR::Wave object, which can be played, further analysed, modified and saved.

## Usage

```
## S4 method for signature 'PTModule'
modToWave(
  mod,
  video = c("PAL", "NTSC"),
  target.rate = 44100,
  target.bit = 16,
  stereo.separation = 1,
  low.pass.filter = TRUE,
  tracks = 1:4,
  mix = TRUE,
  ...
)
```

## Arguments

| | |
|---|---|
| mod | An object of class PTModule |
| video | The video mode of a Commodore Amiga affects timing routines and the playback sample rate. This mode can be specified with this argument and is represented by a character string that can have either the value 'PAL' or 'NTSC'. PAL is used by default. |
| target.rate | A positive integer sample rate for the target Wave. Should be at least 2000. Default value is 44100 Hz, which is conform CD quality. 22050 Hz will also produce a decent sound quality and saves you some working memory. |
| target.bit | Number of bits for the target Wave. Should be a numeric value of either 8, 16, 24 or 32. Default is 16, which is conform CD quality (the quality doesn't really improve at higher bit values, as the original samples are of 8 bit quality). |

`stereo.separation`
: A `numeric` value between 0 and 1. When set to 1 (default), stereo channels (Amiga channels 1 and 4 on left, and channels 2 and 3 on right) are completely separated. When set to less than 1, stereo channels are mixed, where the number gives the fraction of separation of the channels. When set to 0, both channels are completely mixed and a mono `Wave` is returned.

`low.pass.filter`
: A `logical` value indicating whether low pass filters should be applied when generating wave data. The Commodore Amiga had hardware audio filters. One (low pass 6 db/Oct tuned at 4.9 kHz) that filters all audio and one (low pass 12 db/Oct tuned at approximately 3.3 kHz) that can be turned on and off at will with effect command E00/E01 (see also ProTrackR documentation, section on effect commands). These filters are only applied when the `low.pass.filter` argument is set to `TRUE` and the `target.rate` is set to values > 4.9 kHz. If you don't want to simulate this typical Amiga sound, turn the filters off to save processing time.

`tracks`
: Either `logical` or `numeric` values indicating which of the 4 `PTTrack`s are to be converted. By default all 4 tracks are selected.

`mix`
: A logical value indicating whether the 4 Amiga channels should be mixed to the 2 (stereo) output channels. When set to `TRUE` (default) a stereo `tuneR::Wave` object is returned. When set to `FALSE` a multi-channel `tuneR::WaveMC` object is returned. The `stereo.separation` argument is ignored in the latter case.

`...`
: Additional arguments that are passed to `playingtable`.

## Details

Before the `PTModule` object can be converted into a `tuneR::Wave` object, the rows of the `PTPattern` objects in the module need to be put in the right order. This method does that by calling `playingtable`.

Once the rows of the pattern tables are in the right order, all selected `PTTrack` objects of the module are looped by this function and the routines described below are applied to each track.

On the Commodore Amiga the chip responsible for audio output (Paula), the audio playback of samples can be controlled by the user in two ways: the playback rate of the sample can be changed by specifying 'period' values (see e.g. `periodToSampleRate`) and specifying a volume which is linearly scaled between 0 (silent) and 64 (maximum).

So, for each track, the correct period and volume values are determined based on the note, effect command and sample information in the module.

Then, the `PTSample` objects are resampled, using the period values and volume values as determined in the previous step.

Next audio filters are applied to mimic original Commodore Amiga sound. Finaly, the wave data for each separate track is mixed to one (mono) or two (stereo) of the output channels.

Converting ProTracker modules into wave objects can be time consuming. The time required to convert an object obviously depends on your machine's capacities and the length of the module but also the complexity of the module. To speed up the conversion you could reduce the target sample rate or turn off the low pass filter. On modern machines, the time required for conversion should generally be less than the playback time of the module.

You can save the resulting `tuneR::Wave` object by calling `tuneR::writeWave`.

## Value

A tuneR::Wave object, generated from the mod object is returned. A tuneR::WaveMC object is returned when the mix argument is set to FALSE.

## Note

As audio can be mixed with this package at frequencies much greater than the Commodore Amiga's audio output rate, some aliasing of the sound could occur. This results in high frequency audio, that would not be produced on an Amiga. The current version of this package does not filter out these artefacts. This should not be a problem if you're not concerned with producing an accurate Amiga timbre.

## Author(s)

Pepijn de Vries

## See Also

Other module.operations: PTModule-class, appendPattern(), clearSamples(), clearSong(), deletePattern(), fix.PTModule(), moduleSize(), patternLength(), patternOrderLength(), patternOrder(), playMod(), playingtable(), rawToPTModule(), read.module(), trackerFlag(), write.module()

## Examples

```
## Not run:
data(mod.intro)
wav <- modToWave(mod.intro)

## End(Not run)
```

---

# moduleSize *Get module file size*

---

## Description

Get the file size in bytes of a PTModule object, when it is to be saved as an original module file with write.module.

## Usage

```
## S4 method for signature 'PTModule'
moduleSize(x)
```

## Arguments

| | |
|---|---|
| x | A PTModule object for which the file size is to be calculated. |

## Details

The ProTracker module has a 1084 byte sized header containing all (meta) information on the patterns, their order and the audio samples. Each pattern holds exactly 1 Kb of information and the length of the audio samples corresponds with the size in bytes, as they are of 8 bit quality in mono. This function calculates the file size of the PTModule object when it is to be saved with write.module.

## Value

Returns potential uncompressed module file size in bytes represented by a number of class object_size.

## Author(s)

Pepijn de Vries

## See Also

Other module.operations: PTModule-class, appendPattern(), clearSamples(), clearSong(), deletePattern(), fix.PTModule(), modToWave(), patternLength(), patternOrderLength(), patternOrder(), playMod(), playingtable(), rawToPTModule(), read.module(), trackerFlag(), write.module()

## Examples

```
## Calculate the file size for the example module 'mod.intro':

data("mod.intro")
moduleSize(mod.intro)

## Note that this is not the same as the size the object
## requires in R working memory:

object.size(mod.intro)

## In working memory it takes more memory to store the module, than in a
## file. This is because the S4 structure of the object consumes some
## memory. In addition, samples are of 8 bit quality, corresponding with
## a byte per sample. In the PTSample object it is stored as a
## vector of integer values. In R, integer values are 32 bit, which
## costs 4 times as much memory as the original 8 bit.
```

---

# name *Obtain or replace the name of a PTModule or PTSample*

---

## Description

The name of both a PTModule and PTSample are stored as raw data. This method returns the name as a character string, or it can be used to assign a new name to a PTModule or PTSample.

## Usage

```
## S4 method for signature 'PTSample'
name(x)

## S4 replacement method for signature 'PTSample,character'
name(x) <- value

## S4 method for signature 'PTModule'
name(x)

## S4 replacement method for signature 'PTModule,character'
name(x) <- value
```

## Arguments

| | |
|---|---|
| x | A PTModule or a PTSample object for which to obtain or replace the name. |
| value | A character string which should be used to replace the name of PTModule or PTSample x. |

## Details

The name of a PTModule and PTSample is stored as a vector of raw data with a length of 20 or 22 respectively. This method provides the means for getting the name as a character string or to safely redefine the name of a PTModule or PTSample object. To do so, the provided name (value) is converted to a raw vector of length 20 or 22 respectively. Long names may therefore get clipped.

## Value

For name, the name of the PTModule or PTSample object as a character string is returned.

For name<-, object x with an updated name is returned.

## Author(s)

Pepijn de Vries

## See Also

Other character.operations: as.character(), periodToChar(), rawToCharNull(), sampleRate

Other sample.operations: PTSample-class, PTSample-method, fineTune(), loopLength(), loopSample(), loopStart(), loopState(), playSample(), read.sample(), sampleLength(), volume(), waveform(), write.sample()

## Examples

```
data("mod.intro")

## get the name of mod.intro:
name(mod.intro)
```

```
## I don't like the name, let's change it:
name(mod.intro) <- "I like this name better"

## Note that the provided name was too long and is truncated:
name(mod.intro)

## print all sample names in the module:
unlist(lapply(as.list(1:31), function(x)
  name(PTSample(mod.intro, x))))
```

---

`note` *Extract or replace a note*

---

## Description

Obtain a note from a period value or extract or replace a note of a `PTCell` object.

## Usage

```
## S4 method for signature 'numeric'
note(x)

## S4 method for signature 'PTCell'
note(x)

## S4 replacement method for signature 'PTCell,character'
note(x) <- value
```

## Arguments

| | |
|---|---|
| `x` | Either a (`vector` of) numeric value(s), representing a period value. It can also be a `PTCell` object. |
| `value` | A `character` string representing the chromatic scale note with wich the current note needs to be replaced. Should have any of the folling values: `"C-"`, `"C#"`, `"D-"`, `"D#"`, `"E-"`, `"F-"`, `"F#"`, `"G-"`, `"G#"`, `"A-"`, `"A#"`, `"B-"`, or `"--"`. Right-hand dashes can be omitted from these strings. Both upper and lower case are accepted.<br>If an `octave` is not yet specified for `PTCell` `x`, it will be set to 1.<br>Assigning a value of `"--"` will remove both the note and octave from object `x`. |

## Details

Period values are used by ProTracker to set a playback sample rate and in essence determine the key in which a sound is played. This method can be used to obtain the note (key) associated with a period value (according to the ProTracker `period_table`, assuming zero `fineTune`). If the period value is not in the `period_table`, the note associated with the period closest to this value in the table is returned.

The note can also be obtained or replaced for a `PTCell` object.

## Value

For `note`, a `character` string representing the note is returned.

For `note<-`, a copy of `PTCell` object `x` in which the note is replaced by `value` is returned.

## Author(s)

Pepijn de Vries

## See Also

Other period.operations: `noteToPeriod()`, `octave()`, `periodToChar()`, `period_table`, `sampleRate`

Other note.and.octave.operations: `noteToPeriod()`, `noteUp()`, `octave()`, `periodToChar()`, `sampleRate`

Other cell.operations: `PTCell-class`, `PTCell-method`, `effect()`, `sampleNumber()`

## Examples

```
data("mod.intro")

## get the note of PTCell at pattern #3, track #2,
## row #1 from mod.intro (which is note "C-"):

note(PTCell(mod.intro, 1, 2, 3))

## replace the note of PTCell at pattern #3, track #2,
## row #1 from mod.intro with "A-":

note(PTCell(mod.intro, 1, 2, 3)) <- "A-"

## get the notes associated with the period
## values 200 up to 400:

note(200:400)
```

---

# noteToPeriod *Extract period value for a specific note*

---

## Description

Extracts the ProTracker period value for a specific note.

## Usage

```
noteToPeriod(note = "C-3", finetune = 0)
```

### Arguments

| | |
|---|---|
| note | character string representing a note and octave for which the ProTracker period value needs to be determined |
| finetune | integer value ranging from -8 up to 7. A value used to tune an audio sample. |

### Details

ProTracker uses a period_table to link period values to certain octaves and notes. This function serves to look up corresponding period values for specific notes and octaves.

### Value

Returns the numeric ProTracker period value for a corresponding note, octave and fineTune(). Returns 0 if a note could not be found in the table.

### Author(s)

Pepijn de Vries

### See Also

Other period.operations: note(), octave(), periodToChar(), period_table, sampleRate

Other note.and.octave.operations: noteUp(), note(), octave(), periodToChar(), sampleRate

### Examples

```
## Determine the period value corresponding with note 'A-3':
noteToPeriod("A-3")

## get the period values for notes 'A-3' and 'A#3' with finetune at -1:
noteToPeriod(c("A-3", "A#3"), -1)

## get the period values for note 'A-3' with finetune at 0 and 1:
noteToPeriod("A-3", 0:1)
```

---

## noteUp *Raise or lower notes and octaves*

---

### Description

Methods to raise or lower notes in PTCell, PTTrack and PTPattern objects.

## Usage

```
## S4 method for signature 'PTCell'
noteUp(x, sample.nr = "all")

## S4 method for signature 'PTCell'
noteDown(x, sample.nr = "all")

## S4 method for signature 'PTCell'
octaveUp(x, sample.nr = "all")

## S4 method for signature 'PTCell'
octaveDown(x, sample.nr = "all")

## S4 method for signature 'PTTrack'
noteUp(x, sample.nr = "all")

## S4 method for signature 'PTTrack'
noteDown(x, sample.nr = "all")

## S4 method for signature 'PTTrack'
octaveUp(x, sample.nr = "all")

## S4 method for signature 'PTTrack'
octaveDown(x, sample.nr = "all")

## S4 method for signature 'PTPattern'
noteUp(x, sample.nr = "all")

## S4 method for signature 'PTPattern'
noteDown(x, sample.nr = "all")

## S4 method for signature 'PTPattern'
octaveUp(x, sample.nr = "all")

## S4 method for signature 'PTPattern'
octaveDown(x, sample.nr = "all")
```

## Arguments

| | |
|---|---|
| x | A PTCell, PTTrack or PTPattern object for which the notes need to be lowered or raised. |
| sample.nr | A single positive integer value, or a vector of positive integers, listing the indices of samples, for which the notes need to be lowered or raised. A character string equal to "all" is also allowed (this is in fact the default), in which case notes of all sample indices are raised or lowered. |

## Value

Returns an object of the same class as object `x`, in which the notes for samples selected with `sample.nr` are raised or lowered.

In case raised or lowered notes would lead to notes that are out of ProTracker's range, the returned notes remain unchanged.

## Author(s)

Pepijn de Vries

## See Also

Other note.and.octave.operations: `noteToPeriod()`, `note()`, `octave()`, `periodToChar()`, `sampleRate`

## Examples

```
## raise note from C-2 to C#2:
noteUp(PTCell("C-2 01 000"))


## lower note from C-2 to B-1:
noteDown(PTCell("C-2 01 000"))


## raise note from octave 2 to octave 3:
octaveUp(PTCell("C-2 01 000"))


## lower note from octave 2 to octave 1:
octaveDown(PTCell("C-2 01 000"))


data("mod.intro")

## Raise the notes of all cells in pattern
## number 2 of mod.intro:
noteUp(PTPattern(mod.intro, 2))

## Raise only the notes of sample number 4
## in pattern number 2 of mod.intro:
noteUp(PTPattern(mod.intro, 2), 4)

## Raise only the notes of samples number 2 and 4
## in pattern number 2 of mod.intro:
noteUp(PTPattern(mod.intro, 2), c(2, 4))
```

nybble *Get the high or low nybble of a raw value*

## Description

Get the high or low nybble of a raw value and return as integer value [0,15].

## Usage

```
nybble(raw_dat, which = c("low", "high"))

loNybble(raw_dat)

hiNybble(raw_dat)
```

## Arguments

| | |
|---|---|
| raw_dat | A vector of class raw from which the high or low nybble value needs to be extracted. |
| which | A character string indicating whether the high or low nybble should be returnd. It should either be "low" (default) or "high". |

## Details

A raw is basically a byte, composed of 8 bits (zeros and ones). A nybble is a 4 bit value. Hence, a raw value (or byte) is composed of two nybbles. The leftmost nybble of a raw value is refered to as the high nybble, the rightmost nybble is referred to as the low nybble. These functions return either the high or low nybbles of raw data as integer values [0,15]. As ProTracker stores some information as nybbles this function can be used to retrieve this info.

## Value

A vector of the same length as raw_dat holding integer values.

## Author(s)

Pepijn de Vries

## See Also

Other nybble.functions: nybbleToSignedInt(), signedIntToNybble()

Other raw.operations: as.raw(), nybbleToSignedInt(), rawToCharNull(), rawToPTModule(), rawToSignedInt(), rawToUnsignedInt(), signedIntToNybble(), signedIntToRaw(), unsignedIntToRaw()

Other integer.operations: nybbleToSignedInt(), rawToSignedInt(), rawToUnsignedInt(), signedIntToNybble(), signedIntToRaw(), unsignedIntToRaw(), waveform()

## Examples

```
## this will return 0x0f:
hiNybble(as.raw(0xf3))

## which is the same as:
nybble(as.raw(0xf3), "high")

## this will return 0x03:
loNybble(as.raw(0xf3))

## which is the same as:
nybble(as.raw(0xf3), "low")
```

---

# nybbleToSignedInt *Get signed integer values from nybbles*

---

## Description

Get signed integer values from one or more nybble.

## Usage

```
nybbleToSignedInt(raw_dat, which = c("low", "high"))
```

## Arguments

| | |
|---|---|
| raw_dat | raw data (either a single value or a vector), from which a nybble will be extracted and converted. |
| which | A character string indicating whether the "low" (default) or "high" nybble of raw_dat needs to be converted into a signed integer. |

## Details

Nybbles are 4 bit values, where each byte (8 bits) holds two nybbles. A high nybble (left-hand side of a byte) and a low nybble (right-hand side of a byte). This function extracts a nybble from raw data and converts it into a signed integer value ranging from -8 up to 7.

## Value

Returns integer values of the same length as raw_dat, ranging from -8 up to 7.

## Author(s)

Pepijn de Vries

## See Also

Other nybble.functions: nybble(), signedIntToNybble()

Other raw.operations: as.raw(), nybble(), rawToCharNull(), rawToPTModule(), rawToSignedInt(), rawToUnsignedInt(), signedIntToNybble(), signedIntToRaw(), unsignedIntToRaw()

Other integer.operations: nybble(), rawToSignedInt(), rawToUnsignedInt(), signedIntToNybble(), signedIntToRaw(), unsignedIntToRaw(), waveform()

## Examples

```
## generate some raw data:

rdat <- as.raw(255*runif(100))

## get signed integers of low nybbles:

sintl <- nybbleToSignedInt(rdat)

## get signed integers of high nybbles:

sinth <- nybbleToSignedInt(rdat, "high")
```

---

# octave *Extract or replace an octave*

---

## Description

Obtain an octave number from a period value or extract or replace a note of a PTCell object.

## Usage

```
## S4 method for signature 'numeric'
octave(x)

## S4 method for signature 'PTCell'
octave(x)

## S4 replacement method for signature 'PTCell,numeric'
octave(x) <- value
```

## Arguments

| | |
|---|---|
| x | Either a (vector of) numeric value(s), representing a period value. It can also be a PTCell object. |

value — A numeric value representing the octave number with which that of object x needs to be replaced. 0, 1 and 3 are valid octave numbers. Use zero to disable both the note and octave for object x.

Note that the octave can only be set for PTCells for which a note is already defined.

## Details

Period values are used by ProTracker to set a playback sample rate and in essence determine the key and octave in which a sound is played. This method can be used to obtain the octave number associated with a period value (according to the ProTracker period_table, assuming zero fineTune). If the period value is not in the period_table, the octave number associated with the period closest to this value in the table is returned.

The octave number can also be obtained or replaced for a PTCell object.

## Value

For octave, a numeric value representing the octave number is returned.

For octave<-, a copy of PTCell object x in which the octave number is replaced by value is returned.

## Author(s)

Pepijn de Vries

## See Also

Other period.operations: noteToPeriod(), note(), periodToChar(), period_table, sampleRate

Other note.and.octave.operations: noteToPeriod(), noteUp(), note(), periodToChar(), sampleRate

## Examples

```
data("mod.intro")

## get the octave number of PTCell at pattern #3, track #2,
## row #1 from mod.intro (which is number 3):

octave(PTCell(mod.intro, 1, 2, 3))

## replace the octave number of PTCell at pattern #3, track #2,
## row #1 from mod.intro with 2:

octave(PTCell(mod.intro, 1, 2, 3)) <- 2

## get the octave numbers associated with the period
## values 200 up to 400:

octave(200:400)
```

pasteBlock *Paste a block of PTCell data into a PTPattern*

## Description

Paste a block of PTCell data into a PTPattern at a specified location.

## Usage

```
## S4 method for signature 'PTPattern,matrix,numeric,numeric'
pasteBlock(pattern, block, row.start, track.start)
```

## Arguments

| | |
|---|---|
| pattern | A PTPattern object into which the block needs to be pasted. |
| block | A PTBlock holding the PTCell data that needs to be pasted into the pattern. |
| row.start | A positive integer value (ranging from 1 up to 64) indicating the starting position (row) in the pattern to paste the block into. |
| track.start | A positive integer value (ranging from 1 up to 4) indicating the starting position (track) in the pattern to paste the block into. |

## Details

A PTBlock is not a formal S4 class. It is a matrix where each element holds a list of a single PTCell object. As explained at the PTBlock method documentation, this allows for a flexible approach of manipulating PTCell objects. The pasteBlock method allows you to paste a PTBlock back into a PTPattern.

The PTBlock will be pasted at the specified location and will span the number of tracks and rows that are included in the PTBlock. The PTCells in the pattern will be replaced by those of the block. Elements of the bock that are out of the range of the pattern are not included in the pattern.

## Value

Returns a copy of pattern into which block is pasted.

## Author(s)

Pepijn de Vries

## See Also

Other block.operations: PTBlock()

Other pattern.operations: MODPlugToPTPattern(), PTPattern-class, PTPattern-method, PTPatternToMODPlug(), appendPattern(), deletePattern(), patternLength(), patternOrderLength(), patternOrder()

## Examples

```
data("mod.intro")

block <- PTBlock(PTPattern(mod.intro, 1), 1:16, 1)

## Do some operations using lapply (the effect
## code is set to "C10"):
block <- matrix(lapply(block, function(x) {(effect(x) <- "C10"); x}),
                nrow(block), ncol(block), byrow = TRUE)

## Paste block back on the same position:
PTPattern(mod.intro, 1) <-
  pasteBlock(PTPattern(mod.intro, 1), block, 1, 1)

## You can also paste the block anywhere you like:
PTPattern(mod.intro, 1) <-
  pasteBlock(PTPattern(mod.intro, 1), block, 49, 2)
```

---

# patternLength *Get the number of PTPattern tables in a PTModule*

---

## Description

Get the number of PTPattern tables in a PTModule object.

## Usage

```
## S4 method for signature 'PTModule'
patternLength(x)
```

## Arguments

| | |
|---|---|
| x | A PTModule object for which the number of PTPattern tables need to be returned. |

## Details

The number of PTPattern tables in a PTModule object should range from 1 up to either 64 or 100. The maximum depends on the trackerFlag of the PTModule object.

## Value

Returns a numeric value representing the number of PTPattern tables in object x.

## Author(s)

Pepijn de Vries

## See Also

Other pattern.operations: MODPlugToPTPattern(), PTPattern-class, PTPattern-method, PTPatternToMODPlug(), appendPattern(), deletePattern(), pasteBlock(), patternOrderLength(), patternOrder()

Other module.operations: PTModule-class, appendPattern(), clearSamples(), clearSong(), deletePattern(), fix.PTModule(), modToWave(), moduleSize(), patternOrderLength(), patternOrder(), playMod(), playingtable(), rawToPTModule(), read.module(), trackerFlag(), write.module()

## Examples

```
data("mod.intro")

## Get the number of pattern tables in mod.intro:
patternLength(mod.intro)
```

---

# patternOrder *Get the pattern order table*

---

## Description

The pattern order table is a vector of numeric indices of PTPattern tables, which determines in which order the patterns need to be played. This method returns this vector.

## Usage

```
## S4 method for signature 'PTModule'
patternOrder(x, full = FALSE)

## S4 replacement method for signature 'PTModule,ANY,numeric'
patternOrder(x, full = FALSE) <- value
```

## Arguments

| | |
|---|---|
| x | A PTModule object for which the pattern order table needs to be returned or modified. |
| full | A logical value indicating whether the full (TRUE, default), or only the visible (FALSE) part of the pattern order table should be returned. This argument will also affect how new pattern order tables are assigned (see value). |
| value | A numeric vector (maximum length: 128) holding PTPattern indices minus 1 for the new pattern order table.<br>When full = TRUE, the vector will be padded with zeros to a length of 128, and the patternOrderLength will be set to the length of value. When full = FALSE, value will only repplace the part of the order table up to the length of value. The remainder of the table is not changed. The patternOrderLength is also not modified in this case. |

## Details

The actual length of the vector containing the pattern order is 128 as per ProTracker standards. Only part of this vector is 'visible' and will be used to determine in which order pattern tables are to be played. This method can be used to return either the visible or full (all 128) part of the table. It can also be used to assign a new patter order table.

Note that PTPattern indices start at 0, as per ProTracker standards, whereas R start indices at 1. Hence, add 1 to the indices obtained with patternOrder, in order to extract the correct PTPattern from a PTModule.

The maximum index plus 1 in the full pattern order table should equal the number of pattern tables (see patternLength) in the PTModule. Is you assign a new pattern order, with a lower maximum, PTPattern objects will get lost (see also examples)!

## Value

For patternOrder, a vector of numeric PTPattern indices is returned.

For patternOrder<-, an updated version of object x is returned, in which the pattern order table is modified based on value.

## Note

The maximum number of PTPatterns cannot exceed either 64 or 100 (depending on the trackerFlag). This means that values in the order table should also not exceed these values minus 1.

## Author(s)

Pepijn de Vries

## See Also

Other pattern.operations: MODPlugToPTPattern(), PTPattern-class, PTPattern-method, PTPatternToMODPlug(), appendPattern(), deletePattern(), pasteBlock(), patternLength(), patternOrderLength()

Other module.operations: PTModule-class, appendPattern(), clearSamples(), clearSong(), deletePattern(), fix.PTModule(), modToWave(), moduleSize(), patternLength(), patternOrderLength(), playMod(), playingtable(), rawToPTModule(), read.module(), trackerFlag(), write.module()

## Examples

```
data("mod.intro")

## get the visible part of the patternOrder table:
patternOrder(mod.intro)

## get the full patternOrder table:
patternOrder(mod.intro, full = TRUE)

## add 1 to get extract the right PTPattern from
## mod.intro:
first.pattern.played <-
  (PTPattern(mod.intro, patternOrder(mod.intro)[1] + 1))
```

```
## set a different playing order:
patternOrder(mod.intro) <- c(0:3, 0:3, 0:3)

## The assignment above uses a value that
## longer than the patternOrderLength.
## This means that a part ends up in the
## 'invisible' part of the order table:
patternOrder(mod.intro)
patternOrder(mod.intro, full = TRUE)

## Let's do the same assignment, but update
## the visible part of the table as well:
patternOrder(mod.intro, full = TRUE) <- c(0:3, 0:3, 0:3)

## note that the maximum of the order table plus 1
## equals the patternLength of mod.intro (always the case
## for a valid PTModule object):
max(patternOrder(mod.intro, full = TRUE) + 1) ==
  patternLength(mod.intro)

## Let's do something dangerous. If the replacement
## indices do not hold a maximum value that equals
## the patternLength minus 1, PTPatterns will get lost,
## in order to maintain the validity of mod.intro:
patternOrder(mod.intro) <- rep(0, 12)
```

---

`patternOrderLength` *Get the length of the pattern order table*

---

## Description

The pattern order table is a `vector` of `numeric` indices of `PTPattern` tables, which determines in which order the patterns need to be played. This method returns the visible length of this `vector`.

## Usage

```
## S4 method for signature 'PTModule'
patternOrderLength(x)

## S4 replacement method for signature 'PTModule,numeric'
patternOrderLength(x) <- value
```

## Arguments

| | |
|---|---|
| `x` | A `PTModule` object for which the length of the visible part of the pattern order table is to be returned. |
| `value` | A `numeric` value which is to be used to set the visible length of the pattern order table. |

## Details

The actual length of the `vector` containing the pattern order is 128 as per ProTracker standards. Only part of this `vector` is 'visible' and will be used to determine in which order pattern tables are to be played. The length returned by this method is the length of this visible part of the pattern order table. The length of this visible part can also be set with this method.

## Value

For `patternOrderLength` the visible length of the pattern order table of `PTModule` `x` is returned as a `numeric` value, ranging from 1 up to 128.

For `patternOrderLength<-` an updated version of object `x` is returned, in which the visible length of the pattern order table is set to `value`. Note that this does not change the pattern order table itself, only which part is 'visible'.

## Author(s)

Pepijn de Vries

## See Also

Other pattern.operations: `MODPlugToPTPattern()`, `PTPattern-class`, `PTPattern-method`, `PTPatternToMODPlug()`, `appendPattern()`, `deletePattern()`, `pasteBlock()`, `patternLength()`, `patternOrder()`

Other module.operations: `PTModule-class`, `appendPattern()`, `clearSamples()`, `clearSong()`, `deletePattern()`, `fix.PTModule()`, `modToWave()`, `moduleSize()`, `patternLength()`, `patternOrder()`, `playMod()`, `playingtable()`, `rawToPTModule()`, `read.module()`, `trackerFlag()`, `write.module()`

## Examples

```
data("mod.intro")

## get the length of the pattern order table:
patternOrderLength(mod.intro)

## set the length of the pattern order table to 1:
patternOrderLength(mod.intro) <- 1

## note that the pattern order table remained intact:
patternOrder(mod.intro, full = TRUE)
```

---

# paula_clock *Paula clock table*

---

## Description

Table that provides audio output frequencies for the Commodore Amiga original chipset.

### Format

a `data.frame` with two columns:

- 'frequency' A `numeric` value representing Paula's output rate in Hz.
- 'video' A `character` string representing the two video modes.

### Details

Paula was one of the custom chips on the original Commodore Amiga. This chip was dedicated (amongst other tasks) to controlling audio playback. The chip's output rate depended on the video mode used: either 'PAL' or 'NTSC'. This table provides the output rate for both video modes that can be used in calculating sample rates.

### References

https://en.wikipedia.org/wiki/Original_Chip_Set#Paula

### Examples

```
data("paula_clock")
```

---

## periodToChar *Get the note and octave from period table*

---

### Description

These functions return the note and octave that is closest to the provided period value.

### Usage

```
periodToChar(period)
```

### Arguments

| | |
|---|---|
| `period` | `integer` value of a period value. |

### Details

ProTracker uses a period_table to link period values to certain octaves and notes. This function serves to look up corresponding notes and octaves for specific period values.

### Value

`periodToChar` returns a `character` representing the combination of octave and note that is closest to `period` in the ProTracker period table.

### Author(s)

Pepijn de Vries

## See Also

Other character.operations: `as.character()`, `name`, `rawToCharNull()`, `sampleRate`

Other period.operations: `noteToPeriod()`, `note()`, `octave()`, `period_table`, `sampleRate`

Other note.and.octave.operations: `noteToPeriod()`, `noteUp()`, `note()`, `octave()`, `sampleRate`

## Examples

```
## Note C# in octave 3 is closest to a period of 200 in the table:
periodToChar(200)
## try with a range of period values:
periodToChar(200:400)
```

---

`period_table` *ProTracker Period Table*

---

## Description

Table of ProTracker period values and corresponding, octave, tone and fine tune

## Format

a `data.frame` with fourteen columns:

- The column named 'octave': `integer` value [1,3]
- The column named 'finetune': `integer` value [-8, 7] used to tune a sample
- The columns named 'C-' to 'B-': represent the twelve (semi)tones. The values in these columns are the period values for the corresponding tone, octave and finetune.

## Details

Table of ProTracker period values used in calculating the playback sampling rate of samples for specific tones. These are the values that are actually used by ProTracker, they cannot be calculated directly due to undocumented rounding inconsistencies. This lookup table is therefore a requirement.

## See Also

Other period.operations: `noteToPeriod()`, `note()`, `octave()`, `periodToChar()`, `sampleRate`

## Examples

```
data("period_table")
```

playingtable *Generate a table for playing a PTModule object*

## Description

This method generates a table (`data.frame`) in which information from the pattern tables are put in the right order and in a comprehensive format.

## Usage

```
## S4 method for signature 'PTModule'
playingtable(
  mod,
  starting.position = 1,
  max.duration = 2 * 60,
  speed = 6,
  tempo = 125,
  video = c("PAL", "NTSC"),
  play.once = T,
  verbose = T
)
```

## Arguments

| | |
|---|---|
| `mod` | An object of class `PTModule`. |
| `starting.position` | A `numeric` starting position index. Determines where in the `patternOrder` table of the module to start generating the playingtable. |
| `max.duration` | A `numeric` value indicating the maximum length in seconds of the pattern information returned. By default set to 120 seconds (2 minutes). As some modules can be very long, or contain infinite loops or position jumps, the maximum duration is required to break out of the routine for generating the table. |
| `speed` | Default speed to use when it is not specified in the pattern data. See ProTrackR documentation for more info on `speed' and tempo'`. |
| `tempo` | Default tempo to use when it is not specified in the pattern data. See ProTrackR documentation for more info on `speed' and tempo'`. |
| `video` | The video mode of a Commodore Amiga affects timing routines. This mode can be specified with this argument and is represented by a `character` string that can have either the value 'PAL' or 'NTSC'. PAL is used by default. |
| `play.once` | A `logical` value. When set to `TRUE`, the routine will stop adding data to the table when the starting position (`starting.position`) is reach once again. Warning: may not work correctly when the last pattern contains a pattern break. Will be overruled when the `maximum.duration` is reached before the end of the song. |
| `verbose` | A `logical` value. Suppresses a progress report from being printed to the `base::sink` when set to `FALSE`. The default value is `TRUE`. |

## Details

This method generates a table (data.frame) in which information from the pattern tables (PTPattern) are put in the right order, taking into account pattern breaks, position jumps and pattern loops (see also ProTrackR documentation, section on effect commands). The information is put in a comprehensive format in a data.frame, with the following columns:

**row** Row number index of the original PTPattern object.

**filter** A logical value indicating whether the Amiga hardware audio filter was either turned on or off using effect command E00/E01 (see also ProTrackR documentation, section on effect commands).

**speed** Number of 'ticks' per row as set with the Fxy effect commands in the module.

**tempo** The tempo as specified by the Fxy commands in the module.

**delay** The delay that should be applied to the row as specified with the EEx effect command in the module.

**effect.track1..4** The effect code (raw) as specified in each of the 4 tracks in the module.

**effect.mag.track1..4** The effect magnitude (raw) as specified for each of the 4 tracks in the module.

**sample.nr.track1..4** The sample index number (numeric) as specified for each of the 4 tracks in the module.

**note.track1..4** The note (factor) as specified for each of the four tracks in the module.

**position** The positions index number (numeric) from the patternOrder table in the module.

**duration** Playback duration of the corresponding row in seconds.

**cum_duration** Cumulative playback duration of the corresponding row in seconds.

## Value

Returns a data.frame with pattern rows put in the right order. Information contained in the returned table is described in the 'Details' section

## Author(s)

Pepijn de Vries

## See Also

Other module.operations: PTModule-class, appendPattern(), clearSamples(), clearSong(), deletePattern(), fix.PTModule(), modToWave(), moduleSize(), patternLength(), patternOrderLength(), patternOrder(), playMod(), rawToPTModule(), read.module(), trackerFlag(), write.module()

## Examples

```
## Not run:
data(mod.intro)
pt <- playingtable(mod.intro)

## End(Not run)
```

playMod *Play PTModule objects*

## Description

Converts PTModule objects into audio tuneR::Waves, and plays them.

## Usage

```
## S4 method for signature 'PTModule'
playMod(mod, wait = T, ...)
```

## Arguments

| | |
|---|---|
| mod | A PTModule object to be played. |
| wait | A logical value. When set to TRUE the playing routine will wait with executing any code until the playing is finished. When set to FALSE, subsequent R code will be executed while playing. |
| ... | Arguments that are passed on to modToWave. |

## Details

Unfortunately, it was not feasible to create a routine that can directly interpret PTModule objects and play them simultaneously. Instead, the audio first needs to be rendered after which it can be played. This method therefore first calls modToWave and then playWave. Rendering may take some time and requires some balance between speed, quality and accuracy. See the documentation of the modToWave method for the control you have on these aspects.

## Value

A tuneR::Wave object, generated from the mod object, is returned.

## Author(s)

Pepijn de Vries

## See Also

Other play.audio.routines: playSample(), playWave()

Other module.operations: PTModule-class, appendPattern(), clearSamples(), clearSong(), deletePattern(), fix.PTModule(), modToWave(), moduleSize(), patternLength(), patternOrderLength(), patternOrder(), playingtable(), rawToPTModule(), read.module(), trackerFlag(), write.module()

## Examples

```
## Not run:
data("mod.intro")

## play the module and capture the audio Wave
wav <- playMod(mod.intro)

## End(Not run)
```

---

playSample *Play audio samples*

---

## Description

Method to play PTSamples or all such samples from PTModule objects as audio.

## Usage

```
## S4 method for signature 'PTSample'
playSample(x, silence = 0, wait = T, note = "C-3", loop = 1, ...)

## S4 method for signature 'PTModule'
playSample(x, silence = 0, wait = T, note = "C-3", loop = 1, ...)
```

## Arguments

| | |
|---|---|
| x | Either a PTSample or a PTModule object. In the latter case, all samples in the module will be played in order. |
| silence | Especially for short samples, the audio::play routine can be a bit buggy: playing audible noise, ticks or parts from other samples at the end of the sample. By adding silence after the sample, this problem is evaded. Use this argument to specify the duration of this silence in seconds. When, x is a PTModule object, the silence will also be inserted in between samples. |
| wait | A logical value. When set to TRUE the playing routine will wait with executing any code until the playing is finished. When set to FALSE, subsequent R code will be executed while playing. |
| note | A character string specifying the note to be used for calculating the playback sample rate (using noteToSampleRate). It should start with the note (ranging from A' up to G') optionally followed by a hash sign (#') if a note is sharp (or a dash (-') if it's not) and finally the octave number (ranging from 1 up to 3). A valid notation would for instance be 'F#3'. The fineTune as specified for the sample will also be used as an argument for calculating the playback rate. A custom finetune can also be passed as an argument to noteToSampleRate. |
| loop | A positive numeric indicating the duration of a looped sample in seconds. A looped sample will be played at least once, even if the specified duration is less than the sum of loopStart position and the loopLength. See loopStart and loopLength for details on how to set (or disable) a loop. |

... Further arguments passed on to noteToSampleRate. Can be used to change the video mode, or finetune argument for the call to that method.

## Details

This method plays PTSamples and such samples from PTModule objects, using the audio::play method from the audio package. Default fineTune and volume as specified for the PTSample will be applied when playing the sample.

## Value

Returns nothing but plays the sample(s) as audio.

## Author(s)

Pepijn de Vries

## See Also

Other sample.operations: PTSample-class, PTSample-method, fineTune(), loopLength(), loopSample(), loopStart(), loopState(), name, read.sample(), sampleLength(), volume(), waveform(), write.sample()

Other sample.rate.operations: sampleRate

Other play.audio.routines: playMod(), playWave()

## Examples

```
## Not run:
data("mod.intro")

## play all samples in mod.intro:
playSample(mod.intro, 0.2, loop = 0.5)

## play a chromatic scale using sample number 3:
for (note in c("A-2", "A#2", "B-2", "C-3", "C#3",
               "D-3", "D#3", "E-3", "F-3", "F#3",
               "G-3", "G#3"))
{
  playSample(PTSample(mod.intro, 3), note = note, silence = 0.05, loop = 0.4)
}

## play the sample at a rate based on a specific
## video mode and finetune:
playSample(PTSample(mod.intro, 3), video = "NTSC", finetune = -5)

## End(Not run)
```

playWave *Play Wave objects*

## Description

Use the command line audio::play function from the audio package to play tuneR::Wave objects.

## Usage

```
## S4 method for signature 'Wave'
playWave(wave, wait = T)

## S4 method for signature 'WaveMC'
playWave(wave, wait = T)
```

## Arguments

| | |
|---|---|
| wave | An object of class tuneR::Wave or tuneR::WaveMC. Note that the playing routine implemented here can only play stereo waves. Multi-channel waves are therefore converted to stereo before playing. |
| wait | A logical value. When set to TRUE the playing routine will wait with executing any code until the playing is finished. When set to FALSE, subsequent R code will be executed while playing. |

## Details

As the tuneR package play-function relies on external players, this method is provided as a convenient approach to play samples in the R console, using the audio package. Wave objects are played at the rate as specified in the object. Of course you can also play the Wave objects with the tuneR implementation of tuneR::play, by calling tuneR::play(wave).

## Value

Returns an audio::$.audioInstance.

## Author(s)

Pepijn de Vries

## See Also

Other play.audio.routines: playMod(), playSample()

## Examples

```
## Not run:
data(mod.intro)

## PTSample objects can also be
## played with this function as they
## are a child of the Wave object:
playWave(PTSample(mod.intro, 2))

## End(Not run)
```

---

plot *Plot a PTModule object*

---

## Description

Plots the waveforms of the (non-empty) PTSamples in a PTModule object.

## Usage

```
## S4 method for signature 'PTModule,missing'
plot(x, y, plot.loop.positions = T, ...)
```

## Arguments

| | |
|---|---|
| x | A PTModule object for which the waveforms of the PTSamples need to be plotted. |
| y | missing. Argument from the generic plotting method, don't use. |
| plot.loop.positions | A logical value indicating whether loop positions need to be visualised. For looped samples, the starting and ending positions are marked by a vertical green and red line, respectively. |
| ... | Arguments that are passed on to lattice::xyplot. |

## Details

A plotting routine based on the lattice::xyplot from the lattice-package. Plots each (non-empty) waveform in a separate panel. Use arguments of the lattice::xyplot function to customise the plot.

## Value

Returns an object of class trellis. See documentation of lattice::xyplot for more details.

## Author(s)

Pepijn de Vries

## Examples

```
## get the example PTModule provided with the ProTrackR package
data("mod.intro")

## The most basic way to plot the module samples:
plot(mod.intro)

## By using xyplot arguments, we can make it look nicer:
plot(mod.intro, type = "l", layout = c(1,4),
     scales = list(x = list(relation = "free")))
```

---

print *Print ProTrackR objects*

---

## Description

A method to print ProTrackR S4 class objects.

## Usage

```
## S4 method for signature 'PTCell'
print(x, ...)

## S4 method for signature 'PTTrack'
print(x, ...)

## S4 method for signature 'PTPattern'
print(x, ...)

## S4 method for signature 'PTSample'
print(x, ...)

## S4 method for signature 'PTModule'
print(x, ...)
```

## Arguments

| | |
|---|---|
| x | Either a PTModule, PTPattern, PTTrack, PTCell or PTSample object. |
| ... | further arguments passed to or from other methods |

## Value

Depending on the class of x, returns either nothing (NULL) or a character representation of object x.

## Author(s)

Pepijn de Vries

### Examples

```
data("mod.intro")
print(mod.intro)
print(PTPattern(mod.intro, 1))
print(PTTrack(mod.intro, 1, 1))
print(PTCell   (mod.intro, 1, 1, 1))
print(PTSample (mod.intro, 1))
```

---

## proTrackerVibrato *Get the vibrato table used by ProTracker*

---

### Description

Gets the vibrato table as used by ProTracker in vibrato effects.

### Usage

```
proTrackerVibrato(x)
```

### Arguments

| | |
|---|---|
| x | integer representing the table index ranging from 0 up to 31. Values outside this range can be used, but will produce results that are not valid in the context of ProTracker. |

### Details

As the old Commodore Amiga computer didn't have built-in mathematical functions, many programs on that machine used their own data tables. As did ProTracker for vibrato effects for which a sine function was used. As there was no sine function that could be called, sine values were stored in a table.

This function returns the integer sine values (ranging from 0 up to 255) as a function of the table index (ranging from 0 up to 31).

### Value

Returns an integer sine value ranging from 0 up to 255 when a valid table index (x) is provided. It will otherwise return a sine value ranging from -255 up to 255.

### Author(s)

Pepijn de Vries

### Examples

```
## this will return the table as used in ProTracker
proTrackerVibrato(0:31)
```

PTBlock *Select and copy a range of PTCells into a PTBlock*

## Description

Select and copy a range of PTCells from a PTPattern into a PTBlock. This allows a more flexible approach to select and modify PTCells and paste the modified cells back into a PTPattern.

## Usage

```
## S4 method for signature 'PTPattern,numeric,numeric'
PTBlock(pattern, row, track)
```

## Arguments

| | |
|---|---|
| pattern | A PTPattern object from which the PTBlock needs to be selected. |
| row | A numeric index or indices of rows that needs to be copied from the pattern into the PTBlock. |
| track | A numeric index or indices of tracks that needs to be copied from the pattern into the PTBlock. |

## Details

Most objects in this ProTrackR package are very strict in the operations that are allowed, in order to guarantee validity and compatibility with the original ProTracker. This makes those objects not very flexible.

This PTBlock is not a formal S4 object, in fact you can hardly call it an object at all. It is just a matrix, where each element holds a list with a single PTCell.

This matrix is very flexible and makes it easier to select and modify the cells. This flexibility comes at a cost as validity is only checked at the level of the PTCells. The PTBlock can be pasted back into a PTPattern with the pasteBlock method. At which point validity will be checked again. If your modifications resulted in violation of ProTracker standards, you should not be able to paste the block into a pattern.

## Value

Returns a matrix from the selected rows and tracks from the pattern. Each element in the matrix is a list holding a single PTCell.

## Author(s)

Pepijn de Vries

## See Also

Other block.operations: pasteBlock()

## Examples

```
data("mod.intro")

## in most ProTrackR methods you can only select a single row or track.
## with a PTBlock your selection is more flexible.

## select rows 4 up to 8 and tracks 2 up to 4, from the first
## pattern table in mod.intro:

block <- PTBlock(PTPattern(mod.intro, 1), 4:8, 2:4)

## 'block' is now a matrix with in each a list with a PTCell.
## These can now easily be accessed and modified:

cell1 <- block[1, 1][[1]]

print(cell1)
```

---

# PTCell-class *The PTCell class*

---

## Description

The PTCell class is the smallest possible element of a PTPattern table. It holds all information on which note to play, at which frequency, with which effect and what kind of triggers or jumps should be applied.

## Details

The PTCell class consists of a vector of four raw values, as specified in the 'Slots' section. A cell will tell which PTSample is to be played at which frequency (corresponding to a note and octave). If no octave or note is specified, nothing will be played, or if a sample was started to play on the same PTTrack, this sample will continue playing. The PTCell can also hold effect() codes which can be used to add audio effects to the sample being played, change the speed/tempo at which patterns are played, or trigger jumps to other positions within a PTPattern or to other positions in the patternOrder table.

## Slots

data A vector of class raw of length 4. The raw data is stored identical to the way it is stored in a ProTracker module file. The character representation is easier to understand, and with the ProTrackR package it shouldn't be necessary to manipulate the raw data directly.

The structure is illustrated with an example. Let's start with a character representation of a PTCell as an example: "C-3 1B A08". The left-hand part of this string shows that this cell will play note "C" in octave 3. The middle part shows that PTSample number 0x1B = 27 will be played. The right-hand part of the string shows that effect "A08" will be applied (which is a volume slide down).

The raw representation of this example would be "10 d6 ba 08", or when I replace the actual values with symbols: "sp pp se ee". Where "ss" represents the sample number, "eee" represents the effect() code and "ppp" represents the period value. The correct note and octave can be derived by looking up the period value in the period_table (which is also implemented in the following methods: note(), octave() and periodToChar()). The period value 0x0d6 = 214 corresponds with note "C" in octave 3.

## Author(s)

Pepijn de Vries

## See Also

Other cell.operations: PTCell-method, effect(), note(), sampleNumber()

## Examples

```
data("mod.intro")

## get the PTCell from mod.intro at
## PTPattern #1, PTTrack #1 and row #1:

cell <- PTCell(mod.intro, 1, 1, 1)

## get the note of this cell:
note(cell)

## get the octave of this cell:
octave(cell)

## get the sampleNumber of this cell:
sampleNumber(cell)

## get the effect code of this cell:
effect(cell)

## get the raw data of this cell:
as.raw(cell)

## get the character representation of this cell:
as.character(cell)
```

---

# PTCell-method *Coerce to or replace PTCell*

---

## Description

This method will coerce a set of objects to a PTCell object. It can also be used to select specific cells from PTModule, PTPattern and PTTrack objects and replace the selected PTCell.

## Usage

```
## S4 method for signature 'raw,missing,missing,missing'
PTCell(x)

## S4 method for signature 'character,missing,missing,missing'
PTCell(x)

## S4 method for signature 'PTModule,numeric,numeric,numeric'
PTCell(x, row, track, pattern)

## S4 replacement method for signature 'PTModule,numeric,numeric,numeric,PTCell'
PTCell(x, row, track, pattern) <- value

## S4 method for signature 'PTPattern,numeric,numeric,missing'
PTCell(x, row, track)

## S4 replacement method for signature 'PTPattern,numeric,numeric,missing,PTCell'
PTCell(x, row, track) <- value

## S4 method for signature 'PTTrack,numeric,missing,missing'
PTCell(x, row)

## S4 replacement method for signature 'PTTrack,numeric,missing,missing,PTCell'
PTCell(x, row) <- value
```

## Arguments

| | |
|---|---|
| x | Object (any of raw data, a character string, a PTTrack, a PTPattern or a PTModule) to coerce to a PTCell. See details below for the required format of x. |
| row | When x is a PTTrack, a PTPattern, or a PTModule, provide an index [1,64] of the row that needs to be coerced to a PTCell. |
| track | When x is a PTPattern, or a PTModule, provide an index [1,4] of the track that needs to be coerced to a PTCell. |
| pattern | When x is a PTModule, provide an index of the pattern that needs to be coerced to a PTCell. Note that ProTracker uses indices for patterns that start at zero, whereas R uses indices that start at one. Hence add one to an index obtained from a PTModule object (e.g., x$pattern.order) |
| value | An object of PTCell with which the PTCell object at the specified indices in object x needs to be replaced. |

## Details

Method to coerce x to class PTCell.

When x is raw data, it should consist of a vector of 4 elements, formatted as specified in the PTCell.

When x is a character string, it should be formatted as follows: "NNO SS EEE", where NN is the note (for instance "C-" or "A#", where the dash has no particular meaning and may be omitted, the hash sign indicates a sharp note). Use a dash if the cell holds no note. O is the octave (with a value of 0, or a dash, if a note is missing, otherwise any of 1, 2 or 3). SS is the sample index number, formatted as two hexadecimal digits (for example 1A). EEE is a three hexadecimal digit effect or trigger code (for more details see the PTCell). The method is not case sensitive, so you can use both upper and lower case. White spaces are ignored, you can use as many as you would like. A correct character input for x would be for example: "A#2 01 A0F". A blank character representation would look like this: "--- 00 000".

When x is of class PTTrack, PTPattern, or PTModule, the PTCell at the specified indices (row, track and pattern) is returned, or can be replaced.

## Value

When PTCell is used, a PTCell object based on x is returned.

When PTCell<- is used, object x is returned in which the selected PTCell is replaced with value.

## Author(s)

Pepijn de Vries

## See Also

Other cell.operations: PTCell-class, effect(), note(), sampleNumber()

## Examples

```
## This will create an empty PTCell (equivalent
## to new("PTCell"):
PTCell(raw(4))

## Use a character representation to create
## a new PTCell object. A cell with note
## B in octave 2, sample number 10 and with
## effect '105':
cell <- PTCell("B-2 0A 105")

data("mod.intro")

## replace PTCell at pattern number 1, track
## number 2, and row number 3:
PTCell(mod.intro, 3, 2, 1) <- cell
```

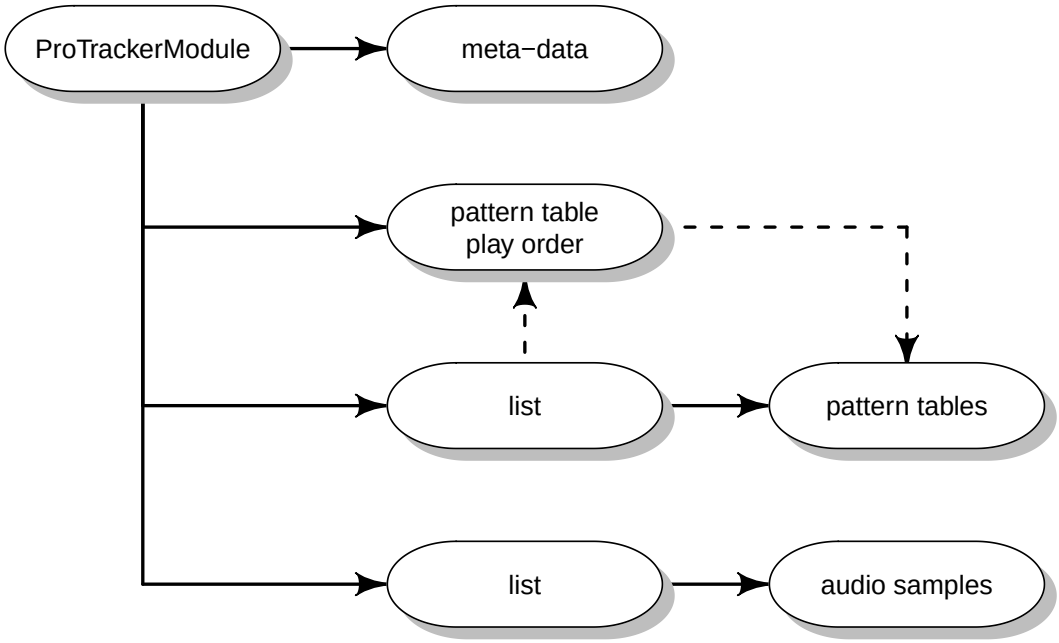
`PTModule-class` *The PTModule class*

## Description

The PTModule class provides a container to store and modify and use ProTracker module files.

## Details

MOD is a computer file format used primarily to represent music. A MOD file contains a set of instruments in the form of samples, a number of patterns indicating how and when the samples are to be played, and a list of what patterns to play in what order. The simplified structure of a module class is visualised in the scheme below. Details are given in the slot descriptions below.




This class is designed to hold all relevant information of a ProTracker module (MOD) for which ProTracker 2.3a documentation was used. The ProTrackR package may be compatible with earlier or later versions, but this was not tested. Use `read.module` and `write.module` to import and export objects of class `PTModule`.

## Slots

`name` A `vector` of length 20 of class `raw`, representing the name of the `PTModule`. The name of a module can be extracted or replaced with the `name` method.

`pattern.order` A `vector` of length 128 of class `raw`. The `raw` values represent the indices of `PTPattern` tables and indicate in which order these patterns need to be played. Note that the `raw` values are conform the indices used in ProTracker, starting at zero. In R, indices of objects start at one. Users need to compensate for this discrepancy theirselves.

The pattern order table can be extracted or replaced with the `patternOrder` method.

`pattern.order.length` A single value of class `raw`. Indicates the length of the visible (and playable) part of the pattern order table.

Use the `patternOrderLength` method to extract or replace the length of a pattern order table of a module.

`tracker.byte` A single `raw` value. Gives an indication of which Tracker was used to produce a module file. In ProTracker modules, this byte is set to 0x7f, which is also used in `PTModule` objects. This value should not be changed.

`tracker.flag` A `vector` of length 4 of class `raw`, indicates the version of a module, which basically reflects how many patterns the module can hold. For details, and extracting and replacing this flag see the `trackerFlag` method.

`samples` List of length 31 of class "`PTSample`".

`patterns` List of class "`PTPattern`" (the pattern tables). The list should have at least 1 element, and can have a maximum of 64 or 100 elements (depending on the state of the `trackerFlag`).

## Author(s)

Pepijn de Vries

## References

https://en.wikipedia.org/wiki/MOD_(file_format)

https://wiki.multimedia.cx/index.php?title=Protracker_Module

## See Also

Other module.operations: `appendPattern()`, `clearSamples()`, `clearSong()`, `deletePattern()`, `fix.PTModule()`, `modToWave()`, `moduleSize()`, `patternLength()`, `patternOrderLength()`, `patternOrder()`, `playMod()`, `playingtable()`, `rawToPTModule()`, `read.module()`, `trackerFlag()`, `write.module()`

## Examples

```
## create an empty PTModule class object:
mod.empty <- new("PTModule")

## get an example PTModule class object
## provided with the ProTrackR package:
data("mod.intro")
```

`PTPattern-class` *The PTPattern class*

## Description

The `PTPattern` (or simply pattern) is a table that determines which samples are played at which notes in which octave, in which order and with which effects.

## Details

When a `PTPattern` table (or simply pattern) is played, each of the 64 rows (see the green mark in the illustration below for an example of a row) are played subsequently at a specified speed/tempo.

Note that ProTracker uses row indices that start at zero. However, this package uses indices starting at one, conform R language definitions.

| row index | track 1 | track 2 | track 3 | track 4 |
|---|---|---|---|---|
| 1 | C-3 01 A08 | C-3 02 C40 | A#1 03 F06 | A#2 04 000 |
| 2 | --- 01 A08 | C-3 02 C30 | --- 00 A01 | --- 00 A02 |
| 3 | --- 01 A08 | C-3 02 000 | --- 00 A01 | C-3 04 000 |
| 4 | --- 00 000 | C-3 02 000 | --- 00 A02 | --- 00 A02 |
| 5 | --- 01 A08 | C-3 02 000 | --- 00 120 | D-3 04 000 |
| 6 | --- 00 000 | A#2 04 C08 | --- 00 A02 | --- 00 A02 |
| 7 | --- 01 A08 | C-3 02 000 | --- 00 220 | D#3 04 000 |
| 8 | --- 00 000 | A#2 04 C08 | --- 00 A01 | --- 00 A02 |
| ... | ... | ... | ... | ... |
| 61 | A-3 01 000 | C-3 02 000 | --- 00 A01 | E-3 04 000 |
| 62 | --- 00 A08 | G-3 04 C08 | --- 00 000 | --- 00 A02 |
| 63 | G-3 01 000 | F-3 04 C08 | --- 00 000 | D-3 04 000 |
| 64 | --- 00 A08 | C-3 04 C08 | --- 00 000 | --- 00 A02 |

The table has four columns (see the purple outline in the illustration above as an example of a column), representing the four audio channels (`PTTrack`) of the Commodore Amiga. Samples listed in the same row at different tracks will be played simultaneously.

An element at a specific row and track will be referred to as a `PTCell` (or simply cell). The cell determines which sample needs to be played at which note and octave and what kind of `effect` or trigger should be applied.

With the `PTPattern-method`, objects can be coerced to a pattern table. This method can also be used to extract or replace patterns in `PTModule` objects.

## Slots

`data` A `matrix` (64 rows, 16 columns) of class `raw`. Each row contains the `raw` concatenated data of 4 `PTCell` objects, representing each of the 4 audio channels/tracks (as each `PTCell` object holds 4 `raw` values, each row holds 4 x 4 = 16 `raw` values). The `raw` data is formatted conform the specifications given in the `PTCell` documentation.

## Author(s)

Pepijn de Vries

## See Also

Other pattern.operations: `MODPlugToPTPattern()`, `PTPattern-method`, `PTPatternToMODPlug()`, `appendPattern()`, `deletePattern()`, `pasteBlock()`, `patternLength()`, `patternOrderLength()`, `patternOrder()`

---

# PTPattern-method *Coerce to or replace PTPattern*

---

## Description

This method will coerce a set of objects to a `PTPattern` object. It can also be used to select specific patterns from `PTModule` objects and replace the selected `PTPattern`.

## Usage

```
## S4 method for signature 'raw,missing'
PTPattern(x)

## S4 method for signature 'matrix,missing'
PTPattern(x)

## S4 method for signature 'PTModule,numeric'
PTPattern(x, pattern)

## S4 replacement method for signature 'PTModule,numeric,PTPattern'
PTPattern(x, pattern) <- value
```

## Arguments

| | |
|---|---|
| `x` | Object (any of `raw` data, a 64 by 16 `matrix` of `raw` data, a 64 by 4 `matrix` of `character` strings, or a `PTModule`) to coerce to a `PTPattern`. See details below for the required format of `x`. |
| `pattern` | When `x` is a `PTModule`, provide an index of the pattern that needs to be coerced to a `PTPattern`. Note that ProTracker uses indices for patterns that start at zero, whereas R uses indices that start at one. Hence add one to an index obtained from a `PTModule` object (e.g., `x$pattern.order`). |

value — An object of PTPattern with which the PTPattern object at the specified index in object x needs to be replaced.

## Details

Method to coerce x to class PTPattern.

When x is a 64 by 16 matrix of raw data, each row implicitly represents the PTCell objects of each of the four tracks. Each PTCell consists of four raw values. The values in each row are formatted accordingly, where the values of the cells of each track are concatenated. See PTCell documentation for more details on the raw format of a PTCell object.

When x is a 64 by 16 matrix of character representations of PTCell objects, the character representation must be conform the specifications as documented at the PTCell.

When x is of class PTModule, the PTPattern at the specified index (pattern) is returned, or can be replaced.

## Value

When PTPattern is used, a PTPattern object based on x is returned.

When PTPattern<- is used, object x is returned in which the selected PTPattern is replaced with value.

## Author(s)

Pepijn de Vries

## See Also

Other pattern.operations: MODPlugToPTPattern(), PTPattern-class, PTPatternToMODPlug(), appendPattern(), deletePattern(), pasteBlock(), patternLength(), patternOrderLength(), patternOrder()

## Examples

```
## This will create an 'empty' PTPattern with
## all 0x00 values, which is equivalent to
## new("PTPattern"):
PTPattern(as.raw(0x00))

## Create a PTPattern based on repeated
## PTCell character representations:
pat <- PTPattern(matrix("F#2 1A 20A", 64, 4))

data("mod.intro")

## Replace the first pattern in the patternOrder
## table in mod.intro with 'pat' (don't forget to
## add one (+1) to the index):
PTPattern(mod.intro,
          patternOrder(mod.intro)[1] + 1) <- pat
```

PTPatternToMODPlug *Convert PTPattern data into a MODPlug pattern*

## Description

Use a PTPattern or PTBlock to create a pattern table with a MODPlug flavour.

## Usage

```
PTPatternToMODPlug(x, to.clipboard = T)
```

## Arguments

| | |
|---|---|
| x | Either a PTPattern object or a PTBlock object from which an Open MODPlug Tracker pattern should be created. |
| to.clipboard | A logical value, indicating whether the result should be copied to the system's clipboard (TRUE) or should be returned as a vector of characters (FALSE). |

## Details

The Open MODPlug Tracker (https://openmpt.org) is a modern music tracker that is for free. It too can handle ProTracker modules. This function assists in moving pattern data from R to Open MPT.

## Value

Returns an invisible NULL when argument to.clipboard is set to TRUE. Returns an Open MODPlug Tracker flavoured pattern table as a vector of characters when it is set to FALSE.

## Author(s)

Pepijn de Vries

## See Also

Other MODPlug.operations: MODPlugToPTPattern()

Other pattern.operations: MODPlugToPTPattern(), PTPattern-class, PTPattern-method, appendPattern(), deletePattern(), pasteBlock(), patternLength(), patternOrderLength(), patternOrder()

## Examples

```
## Not run:
## get some pattern data

pattern <- PTPattern(mod.intro, 1)

## Now create a MODPlug pattern from this.
## The result is placed on the system clipboard.
```

```
## You can check by pasting it into a text
## editor, or better yet, the MODPlug Tracker.

PTPatternToMODPlug(pattern)

## If you want to handle the pattern data
## in R:

patModPlug <- PTPatternToMODPlug(pattern, F)

## We can do the same with a block:

block <- PTBlock(pattern, 1:10, 2:3)
PTPatternToMODPlug(block)

## End(Not run)
```

---

PTSample-class *The PTSample class*

---

## Description

This class holds audio fragments with meta-information, to be used in PTModule objects.

## Details

This class holds audio fragments with meta-information (so-called samples), to be used in PTModule objects. This class extends the tuneR::Wave class from tuneR::tuneR. It therewith inherits all properties and cool methods available from the tuneR::tuneR package. This allows you, for instance, to generate power spectra (tuneR::powspec) of them. You can also plot the waveform with the plot-Wave method. See tuneR::tuneR for all possibilities with tuneR::Wave objects. If you want you can also explicitly coerce PTSample to tuneR::Wave objects like this: `as(new("PTSample"), "Wave")`.

The PTSample class has some slots that are additional to the tuneR::Wave class, as ProTracker requires additional information on the sample with respect to its name, fine tune, volume and loop positions. The PTSample class restricts the enherited tuneR::Wave class such that it will only hold 8 bit, mono, pcm waves with a maximum of `2*0xffff = 131070` samples, as per ProTracker standards. The length should always be even.

PTSamples can be imported and exported using the read.sample and write.sample methods respectively. tuneR::Wave objects and raw data can be coerced to PTSamples with the PTSample-method.

## Slots

name A vector of length 22 of class raw, representing the name of the PTSample. It is often used to include descriptive information in a PTModule. The name of a sample can be extracted or replaced with the name method.

`finetune` Single value of class `raw`. The `loNybble` of the `raw` value, represents the sample fine tune value ranging from -8 up to 7. This value is used to tweak the playback sample rate, in order to tune it. Negative values will lower the sample rate of notes, positive values will increase the sample rate of notes. Period values corresponding to specific notes and fine tune values are stored in the `period_table`. The fine tune value can be extracted or replace with the `fineTune` method.

`volume` Single value of class `raw`. The raw data corresponds with the default playback volume of the sample. It ranges from 0 (silent) up to 64 (maximum volume). The volume value can be extracted or replaced with the `volume` method.

`wloopstart` A `vector` of length 2 of class `raw`. The `raw` data represent a single unsigned number representing the starting position of a loop in the sample. It should have a value of `0` when there is no loop. Its value could range from `0` up to `0xffff`. To get the actual position in bytes the value needs to be multiplied with 2 and can therefore only be can only be even. The sum of the loop start position and the loop length should not exceed the `sampleLength`. Its value can be extracted or replaced with the `loopStart` method.

`wlooplen` A `vector` of length 2 of class `raw`. The `raw` data represent a single unsigned number representing the length of a loop in the sample. To get the actual length in bytes, this value needs to be multiplied by 2 and can therefore only be even. It should have a value of `2` when there is no loop. Its value could range from `2` up to `2*0xffff` (= `131070`) and can only be even (it can be `0` when the sample is empty). The sum of the loop start position and the loop length should not exceed the `sampleLength`. Its value can be extracted or replaced with the `loopLength` method.

`left` Object of class `numeric` representing the waveform of the left channel. Should be `integer` values ranging from 0 up to 255. It can be extracted or replaced with the `waveform` method.

`right` Object of class `numeric` representing the right channel. This slot is inherited from the `tuneR::Wave` class and should be `numeric(0)` for all `PTSamples`, as they are all mono.

`stereo` Object of class `logical` whether this is a stereo representation. This slot is inherited from the `tuneR::Wave` class. As `PTSamples` are always mono, this slot should have the value `FALSE`.

`samp.rate` Object of class `numeric` representing the sampling rate.

`bit` Object of class `numeric` representing the bit-wise quality. This slot is inherited from the `tuneR::Wave` class. As `PTSamples` are always of 8 bit quality, the value of this slot should always be 8.

`pcm` Object of class `logical` indicating whether wave format is PCM. This slot is inherited from the `tuneR::Wave` class, for `PTSamples` its value should always be `TRUE`.

## Author(s)

Pepijn de Vries

## See Also

Other sample.operations: `PTSample-method`, `fineTune()`, `loopLength()`, `loopSample()`, `loopStart()`, `loopState()`, `name`, `playSample()`, `read.sample()`, `sampleLength()`, `volume()`, `waveform()`, `write.sample()`

PTSample-method *Coerce to or replace PTSample*

## Description

This method will coerce a set of objects to a PTSample object. It can also be used to select specific samples from PTModule objects and replace the selected PTSample.

## Usage

```
## S4 method for signature 'Wave,missing'
PTSample(x)

## S4 method for signature 'raw,missing'
PTSample(x)

## S4 method for signature 'PTModule,numeric'
PTSample(x, index)

## S4 replacement method for signature 'PTModule,numeric,PTSample'
PTSample(x, index) <- value
```

## Arguments

| | |
|---|---|
| x | Object (any of class tuneR::Wave, a vector of raw data, or of class PTModule) that needs to be coerced to a PTSample object. In the latter case, the object can also be replaced. |
| index | A positive integer index of the sample in PTModule x that needs to be returned or replaced. |
| value | An object of PTSample with which the PTSample object at the specified index in object x needs to be replaced. |

## Details

Method to coerce x to class PTSample.

When x is a tuneR::Wave object, this method will not resample it. However, the sample rate will be adjusted and samples exeeding the maximum length of 2*0xffff = 131070 will be clipped to this maximum length. When x is a stereo sample, it will be converted to mono, by averaging the left and right channel.

When x is a vector of raw data, it will be truncated if the maximum length of 2*0xffff = 131070 is exceeded. The raw will be converted with rawToSignedInt in order to represent an 8 bit mono waveform.

As samples must have an even length (as per ProTracker specifications), a 0x00 value is appended if the length is odd.

When x is of class PTModule, the PTSample at the specified index is returned, or will be replaced.

## Value

When PTSample is used, a PTSample object based on x is returned.

When PTSample<- is used, object x is returned in which the selected PTSample is replaced with value.

## Author(s)

Pepijn de Vries

## See Also

Other sample.operations: PTSample-class, fineTune(), loopLength(), loopSample(), loopStart(), loopState(), name, playSample(), read.sample(), sampleLength(), volume(), waveform(), write.sample()

## Examples

```
## Create a raw data sine wave:
raw_sine <- signedIntToRaw(round(sin(2*pi*(0:275)/276)*127))

data("mod.intro")

## Replace sample number 1 from mod.intro
## with the sine wave:
PTSample(mod.intro, 1) <-
  PTSample(raw_sine)

## Note that the replacement above
## could also (maybe more efficiently)
## be done with the 'waveform' method

## Restore the loop in sample number 1:
loopLength(PTSample(mod.intro, 1)) <- 276
```

---

# PTTrack-class *The PTTrack class*

---

## Description

The four audio channels of the Commodore Amiga are represented as tracks (the PTTrack class) in a PTPattern.

## Details

The Commodore Amiga original chipset supported four audio channels. Meaning that audio could be played simultaneously and independently on each of these channels. Two channels (2 and 3) were hardware-mixed fully to the right stereo outputs and the other two (1 and 4) fully to the left stereo outputs.

This class represents such a single channel, reffered to as a track. A `PTPattern` is composed of four such channels. As a ProTracker pattern consists of 64 rows, a `PTTrack` object is also (implicitly) composed of 64 `PTCell` objects.

Use the `PTTrack-method` to construct or coerce objects to a `PTTrack-class` object, or to replace such an object.

## Slots

`data` A `matrix` (64 rows, 4 columns) of class `raw`. Each row implicetely represents a `PTCell` object, where the raw data is formatted as specified at the `PTCell` documentation. Use the `PTCell-method` to make an element of a `PTTrack` object explictly of class `PTCell`. Row numbers correspond with the row numbers of `PTPattern` objects.

## Author(s)

Pepijn de Vries

## Examples

```
data("mod.intro")

## Get track number 2 from pattern
## number 1 of mod.intro:
chan1 <- PTTrack(mod.intro, 2, 1)

## Create a blank track:
chan2 <- new("PTTrack")

## Get two more tracks:
chan3 <- PTTrack(mod.intro, 1, 2)
chan4 <- PTTrack(mod.intro, 4, 3)

## combine the four tracks in a
## new PTPattern:
patt <- PTPattern(cbind(
  as.character(chan1),
  as.character(chan2),
  as.character(chan3),
  as.character(chan4)
))
```

PTTrack-method *Coerce to or replace PTTrack*

## Description

This method will coerce a set of objects to a PTTrack object. It can also be used to select specific tracks from PTModule and PTPattern objects and replace the selected PTTrack.

## Usage

```
## S4 method for signature 'raw,missing,missing'
PTTrack(x)

## S4 method for signature 'matrix,missing,missing'
PTTrack(x)

## S4 method for signature 'character,missing,missing'
PTTrack(x)

## S4 method for signature 'PTModule,numeric,numeric'
PTTrack(x, track, pattern)

## S4 replacement method for signature 'PTModule,numeric,numeric,PTTrack'
PTTrack(x, track, pattern) <- value

## S4 method for signature 'PTPattern,numeric,missing'
PTTrack(x, track)

## S4 replacement method for signature 'PTPattern,numeric,missing,PTTrack'
PTTrack(x, track) <- value
```

## Arguments

| | |
|---|---|
| x | Object (any of raw data, a 64 by 4 matrix of raw data, a vector of character strings, a PTPattern or a PTModule) to coerce to a PTTrack. See details below for the required format of x |
| track | When x is a PTPattern, or a PTModule, provide an index [1,4] of the track that needs to be coerced to a PTTrack. |
| pattern | When x is a PTModule, provide an index of the pattern that needs to be coerced to a PTTrack. Note that ProTracker uses indices for patterns that start at zero, whereas R uses indices that start at one. Hence add one to an index obtained from a PTModule object (e.g., x$pattern.order) |
| value | An object of PTTrack with which the PTTrack object at the specified indices in object x needs to be replaced. |

## Details

Method to coerce x to class PTTrack.

When x is a 64 by 4 matrix of raw data, each row implicitly represents a PTCell object and should be formatted accordingly. See PTCell documentation for more details.

When x is a 64 element vector of character representation of PTCell objects, the character representation must be conform the specifications as documented at the PTCell.

When x is of class PTPattern, or PTModule, the PTTrack at the specified indices (track and pattern) is returned, or can be replaced.

## Value

When PTTrack is used, a PTTrack object based on x is returned.

When PTTrack<- is used, object x is returned in which the selected PTTrack is replaced with value.

## Author(s)

Pepijn de Vries

## See Also

Other track.operations: as.character()

## Examples

```
## This will create an 'empty' PTTrack with all nul
## values, which is equivalent to new("PTTrack"):
PTTrack(as.raw(0x00))

## This will generate a PTTrack from a repeated
## character representation of a PTCell:
chan <- PTTrack(rep("C-3 01 C20", 64))

data("mod.intro")

## This will replace the PTTrack at pattern
## number 1, track number 2 of mod.intro with chan:
PTTrack(mod.intro, 2, 1) <- chan
```

---

# rawToCharNull *Convert raw vectors into a character string*

---

## Description

A function that converts raw data into a character string.

## Usage

```
rawToCharNull(raw_dat)
```

## Arguments

| | |
|---|---|
| raw_dat | A vector of class raw to be converted into a character. |

## Details

The function rawToChar() will fail on vectors of raw data with embedded 0x00 data. This function will not fail on embedded 0x00 values. Instead, it will replace embedded 0x00 data with white spaces. Note that leading and trailing 0x00 data will be omitted from the result.

## Value

A character string based on the raw data

## Author(s)

Pepijn de Vries

## See Also

Other raw.operations: as.raw(), nybbleToSignedInt(), nybble(), rawToPTModule(), rawToSignedInt(), rawToUnsignedInt(), signedIntToNybble(), signedIntToRaw(), unsignedIntToRaw()

Other character.operations: as.character(), name, periodToChar(), sampleRate

## Examples

```
## generate some raw data with an embedded 0x00:
some.raw.data <- as.raw(c(0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x00,
                          0x77, 0x6f, 0x72, 0x6c, 0x64, 0x21))
## Not run:
## this will fail:
try(rawToChar(some.raw.data))

## End(Not run)

## this will succeed:
rawToCharNull(some.raw.data)
```

rawToPTModule *Convert a vector of raw data into a PTModule object*

## Description

This method treats a vector of raw data as if it where a file, and converts it into a PTModule object.

## Usage

```
## S4 method for signature 'raw'
rawToPTModule(x, ignore.validity = F)
```

## Arguments

| | |
|---|---|
| x | A vector of raw data, conform ProTracker file specs. |
| ignore.validity | A logical value. When set as TRUE this method will attempt to decode the raw data (x), even when it is invalid. When set to FALSE (default) validity is checked and an error is thrown when invalidity occurs. |

## Details

Data is read from a vector of raw data as if it where a file and converted into a PTModule object. This method can be useful for module files stored on virtual Amiga Disk Files (adf), which can be read as raw data, using the AmigaFFH package.

Use as.raw to achieve the inverse.

## Value

returns a PTModule object.

## Author(s)

Pepijn de Vries

## See Also

Other module.operations: PTModule-class, appendPattern(), clearSamples(), clearSong(), deletePattern(), fix.PTModule(), modToWave(), moduleSize(), patternLength(), patternOrderLength(), patternOrder(), playMod(), playingtable(), read.module(), trackerFlag(), write.module()

Other raw.operations: as.raw(), nybbleToSignedInt(), nybble(), rawToCharNull(), rawToSignedInt(), rawToUnsignedInt(), signedIntToNybble(), signedIntToRaw(), unsignedIntToRaw()

## Examples

```
## Not run:
## convert the example mod into raw data
data("mod.intro")
mod.raw <- as.raw(mod.intro)

## restore it as a PTModule-class object
mod.restored <- rawToPTModule(mod.raw)

## In this case the result is identical to the original:
identical(mod.restored, mod.intro)

## End(Not run)
```

---

`rawToSignedInt` *Convert a raw vector into signed integers (short)*

---

## Description

This function converts a vector of `raw` data into signed integer values.

## Usage

```
rawToSignedInt(raw_dat)
```

## Arguments

| | |
|---|---|
| `raw_dat` | A vector of `raw` data. |

## Details

This function converts a vector of `raw` data into signed integer values [-128,127]. To convert unsigned integers into raw data use `as.raw()`. For the inverse of this function see `signedIntToRaw()`.

## Value

A vector of the same length as `raw_dat`, holding signed integer values.

## Author(s)

Pepijn de Vries

## See Also

Other raw.operations: `as.raw()`, `nybbleToSignedInt()`, `nybble()`, `rawToCharNull()`, `rawToPTModule()`, `rawToUnsignedInt()`, `signedIntToNybble()`, `signedIntToRaw()`, `unsignedIntToRaw()`

Other integer.operations: `nybbleToSignedInt()`, `nybble()`, `rawToUnsignedInt()`, `signedIntToNybble()`, `signedIntToRaw()`, `unsignedIntToRaw()`, `waveform()`

## Examples

```
## generate some raw data:
some.raw.data <- as.raw(c(0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x90))

## convert the raw data into a vector of signed intgers:
rawToSignedInt(some.raw.data)
```

---

`rawToUnsignedInt` *Convert raw vector into a single unsigned integer value*

---

## Description

This function converts `raw` data into an unsigned integer

## Usage

```
rawToUnsignedInt(raw_dat)
```

## Arguments

| | |
|---|---|
| `raw_dat` | A vector of class `raw` to be converted into an unsigned integer |

## Details

This function converts a vector of raw data into a single unsigned integer. for conversion of raw data into a vector of unsigned integers `\[0,255\]` use `as.integer()`. For an inverse of this function see `unsignedIntToRaw()`.

## Value

A single unsigned integer value based on the provided `raw` data

## Author(s)

Pepijn de Vries

## See Also

Other raw.operations: `as.raw()`, `nybbleToSignedInt()`, `nybble()`, `rawToCharNull()`, `rawToPTModule()`, `rawToSignedInt()`, `signedIntToNybble()`, `signedIntToRaw()`, `unsignedIntToRaw()`

Other integer.operations: `nybbleToSignedInt()`, `nybble()`, `rawToSignedInt()`, `signedIntToNybble()`, `signedIntToRaw()`, `unsignedIntToRaw()`, `waveform()`

## Examples

```
## generate some raw data:
some.raw.data <- as.raw(c(0x01, 0x1e, 0x3f))

## convert raw data into an unsigned integer:
rawToUnsignedInt(some.raw.data)

## note the difference with
as.integer(some.raw.data)
```

---

`read.module` *Read a ProTracker module file*

---

## Description

Reads a ProTracker module file and coerces it to a `PTModule` object.

## Usage

```
## S4 method for signature 'character,logical'
read.module(file, ignore.validity = F)

## S4 method for signature 'ANY,missing'
read.module(file, ignore.validity = F)

## S4 method for signature 'ANY,missing'
read.module(file, ignore.validity = F)

## S4 method for signature 'ANY,logical'
read.module(file, ignore.validity = F)
```

## Arguments

| | |
|---|---|
| `file` | either a filename or a file connection, that allows reading binary data (see e.g., `base::file` or `base::url`). |
| `ignore.validity` | A `logical` value indicating whether the validity of the `PTModule` should be ignored. When set to `FALSE` (default), the validity of the read object is checked; an error is thrown when the object is not valid. When this argument is set to `TRUE`, the validity of the object will not be checked and a potentially invalid object is returned. As the validity check of `PTModule` objects is very strict, it can be useful to ignore this check. This way you can try to read a broken module file, try to fix it such that it becomes valid and save (with `write.module`) it again. |

## Details

The routine to read ProTracker modules is based on the referenced version of ProTracker 2.3A. This means that the routine may not be able to read files produced with later ProTracker versions, or earlier versions with back-compatibility issues. So far I've successfully tested this method on all modules I've composed with ProTracker version 2.3A (which I believe was one of the more popular versions of ProTracker back in the days).

It should also be able to read most of the .mod files in The Mod Archive.

## Value

Returns a `PTModule` object read from the provided ProTracker file

## Author(s)

Pepijn de Vries

## References

https://wiki.multimedia.cx/index.php?title=Protracker_Module

## See Also

Other io.operations: `read.sample()`, `write.module()`, `write.sample()`

Other module.operations: `PTModule-class`, `appendPattern()`, `clearSamples()`, `clearSong()`, `deletePattern()`, `fix.PTModule()`, `modToWave()`, `moduleSize()`, `patternLength()`, `patternOrderLength()`, `patternOrder()`, `playMod()`, `playingtable()`, `rawToPTModule()`, `trackerFlag()`, `write.module()`

## Examples

```
## Not run:

## first create an module file from example data:
data("mod.intro")
write.module(mod.intro, "intro.mod")

## read the module:
mod  <-  read.module("intro.mod")

## or create a connection yourself:
con  <- file("intro.mod", "rb")

## note that you can also read from URL connections!
mod2 <- read.module(con)

## don't forget to close the file:
close(con)

## End(Not run)
```

`read.sample` *Read an audio file and coerce to a PTSample object*

## Description

Reads audio files from "wav" and "mp3" files, using `tuneR::tuneR` methods. Commodore Amiga native formats "8svx" and "raw" can also be read.

## Usage

```
## S4 method for signature 'character'
read.sample(filename, what = c("wav", "mp3", "8svx", "raw"))
```

## Arguments

| | |
|---|---|
| `filename` | A `character` string representing the filename to be read. |
| `what` | A `character` string indicating what type of file is to be read. Can be one of the following: `"wav"` (default), `"mp3"`, `"8svx"` or `"raw"`. The `AmigaFFH` package needs to be installed in order to read 8svx files. |

## Details

This method provides a wrapper for the `tuneR::readWave` and `tuneR::readMP3` methods from `tuneR::tuneR`. It also provides the means to import audio from file formats native to the Commodore Amiga. Simple 8svx files (also known as "iff" files) can be read. This uses the `AmigaFFH::read.iff` method from the `AmigaFFH::AmigaFFH` package. It was also common practice to store audio samples as raw data on the Commodore Amiga, where each byte simply represented a signed integer value of the waveform.

All audio will be coerced to 8 bit mono with a maximum length of `2*0xffff = 131070` bytes (= samples) as per ProTracker standards.

## Value

Returns a `PTSample` object based on the file read.

## Note

As per ProTracker standards, a sample should have an even length (in bytes). If a sample file has an odd length, a `raw 0x00` value is added to the end.

## Author(s)

Pepijn de Vries

## See Also

Other sample.operations: PTSample-class, PTSample-method, fineTune(), loopLength(), loopSample(), loopStart(), loopState(), name, playSample(), sampleLength(), volume(), waveform(), write.sample()

Other io.operations: read.module(), write.module(), write.sample()

## Examples

```
## Not run:
data("mod.intro")

## create an audio file which we can then read:
write.sample(PTSample(mod.intro, 2), "snaredrum.iff", "8svx")

## read the created sample:
snare <- read.sample("snaredrum.iff", "8svx")
print(snare)

## End(Not run)
```

---

# resample *Resample data*

---

## Description

Resample numeric data to a different rate.

## Usage

```
resample(x, source.rate, target.rate, ...)
```

## Arguments

| | |
|---|---|
| x | A numeric vector that needs to be resampled. |
| source.rate | The rate at which x was sampled in Hz (or another unit, as long as it is in the same unit as target.rate). |
| target.rate | The desired target sampling rate in Hz (or another unit, as long as it is in the same unit as source.rate). |
| ... | Arguments passed on to stats::approx. To simulate the Commodore Amiga hardware, it's best to use 'method = "constant" for resampling 8 bit samples. |

## Details

This function resamples numeric data (i.e., audio data) from a source sample rate to a target sample rate. At the core it uses the stats::approx function.

## Value

Returns a resampled `numeric vector` of length `round(length(x) * target.rate / source.rate)` based on `x`.

## Author(s)

Pepijn de Vries

## Examples

```
some.data <- 1:100

## assume that the current (sample) rate
## of 'some.data' is 100, and we want to
## resample this data to a rate of 200:
resamp.data <- resample(some.data, 100, 200, method = "constant")
```

---

`sampleLength` *Get the length of a PTSample*

---

## Description

Gets the length (in samples = bytes) of an audio fragment stored as a `PTSample`.

## Usage

```
## S4 method for signature 'PTSample'
sampleLength(sample)
```

## Arguments

| | |
|---|---|
| `sample` | A `PTSample` object for which the length needs to be returned. |

## Details

`PTSample`s are 8 bit mono audio fragments. This method returns the length of this fragment expressed as number of samples (which also equals the number of bytes).

## Value

Returns a `numeric` value representing the number of samples (bytes) the `PTSample` object `sample` is composed of.

## Author(s)

Pepijn de Vries

## See Also

Other sample.operations: PTSample-class, PTSample-method, fineTune(), loopLength(), loopSample(), loopStart(), loopState(), name, playSample(), read.sample(), volume(), waveform(), write.sample()

## Examples

```
data("mod.intro")

## Show the length of the second sample in mod.intro
sampleLength(PTSample(mod.intro, 2))
```

---

# sampleNumber *Extract or replace a sample number*

---

## Description

Extract or replace a PTSample index number from a PTCell object.

## Usage

```
## S4 method for signature 'PTCell'
sampleNumber(x)

## S4 replacement method for signature 'PTCell,numeric'
sampleNumber(x) <- value
```

## Arguments

| | |
|---|---|
| x | A PTCell object from which the PTSample index number needs to be be extracted or replaced. |
| value | A numeric replacement value for the index. Valid indices range from 1 up to 31. A value of 0 can also be assigned, but will not play any sample. |

## Details

The PTSample index number in a PTCell object, indicates which sample from a PTModule object needs to be played. This method can be used to extract or replace this index from a PTCell object.

## Value

For sampleNumber, a numeric value representing the sample index number of object x is returned.

For sampleNumber<-, an copy of object x is returned in which the sample index number is replaced with value.

## Author(s)

Pepijn de Vries

## See Also

Other cell.operations: PTCell-class, PTCell-method, effect(), note()

## Examples

```
data("mod.intro")

## get the sample index number of PTCell at pattern #3,
## track #2, row #1 from mod.intro (which is 2):

sampleNumber(PTCell(mod.intro, 1, 2, 3))

## replace the sample index number of PTCell at pattern #3,
## track #2, row #1 from mod.intro with 1:

sampleNumber(PTCell(mod.intro, 1, 2, 3)) <- 1
```

---

# sampleRate *Calculate the sample rate for a note or period value*

---

## Description

Calculate the sample rate for a note or a ProTracker period value.

## Usage

```
noteToSampleRate(note = "C-3", finetune = 0, video = c("PAL", "NTSC"))

periodToSampleRate(period, video = c("PAL", "NTSC"))
```

## Arguments

| | |
|---|---|
| note | A character string representing a note for which the sample rate is to be calculated. |
| finetune | An integer value ranging from -8 up to 7. A value used to tune an audio sample. |
| video | The video mode used to calculate the sample rate. A character string that can have either the value 'PAL' or 'NTSC'. PAL is used by default. |
| period | A ProTracker integer value of a period value for which the sample rate is to be calculated. |

## Details

The timing on a Commodore Amiga depends on the video mode, which could be either 'PAL' or 'NTSC'. Therefore sample rates also depend on these modes. As the PAL is mostly used in Europe, and the Amiga was most popular in Europe, PAL is used by default.

## Value

Returns the sample rate in samples per seconds.

## Author(s)

Pepijn de Vries

## See Also

Other character.operations: `as.character()`, `name`, `periodToChar()`, `rawToCharNull()`

Other period.operations: `noteToPeriod()`, `note()`, `octave()`, `periodToChar()`, `period_table`

Other sample.rate.operations: `playSample()`

Other note.and.octave.operations: `noteToPeriod()`, `noteUp()`, `note()`, `octave()`, `periodToChar()`

## Examples

```
## calculate the sample rate for a ProTracker period value of 200
periodToSampleRate(200)

## calculate the sample rate for a sample at note 'A-3'
noteToSampleRate("A-3")

## note that the NTSC video system gives a slightly different rate:
noteToSampleRate("A-3", video = "NTSC")

## fine tuning a sample will also give a slightly different rate:
noteToSampleRate("A-3", finetune = -1)
```

---

# signedIntToNybble *Convert a signed integer to a nybble in raw data.*

---

## Description

This function converts a signed integer ranging from -8 up to 7 into either the high or low nybble of a byte, represented by `raw` data.

## Usage

```
signedIntToNybble(int_dat, which = c("low", "high"))
```

## Arguments

| | |
|---|---|
| `int_dat` | A single `integer` value or a `vector` of `integer` data ranging from -8 up to 7. |
| `which` | A character string indicating whether the nybble should be set to the `"low"` (default) or `"high"` position of the raw data that is returned. |

## Details

Nybbles are 4 bit values, where each byte (8 bits) holds two nybbles. A high nybble (left-hand side of a byte) and a low nybble (right-hand side of a byte). This function converts a signed `integer` value ranging from -8 up to 7 to a nybble and sets it as either a high or a low nybble in `raw` data.

## Value

Returns `raw` data of the same length as `int_dat`. The returned raw data holds either low or high nybbles (as specified by `which`) based on the provided signed `integers`.

## Author(s)

Pepijn de Vries

## See Also

Other nybble.functions: `nybbleToSignedInt()`, `nybble()`

Other raw.operations: `as.raw()`, `nybbleToSignedInt()`, `nybble()`, `rawToCharNull()`, `rawToPTModule()`, `rawToSignedInt()`, `rawToUnsignedInt()`, `signedIntToRaw()`, `unsignedIntToRaw()`

Other integer.operations: `nybbleToSignedInt()`, `nybble()`, `rawToSignedInt()`, `rawToUnsignedInt()`, `signedIntToRaw()`, `unsignedIntToRaw()`, `waveform()`

## Examples

```
## generate some integers in the right range:

dati <- sample(-8:7, 100, replace = TRUE)

## Set the low nybbles of rawl based on dati:

rawl <- signedIntToNybble(dati)

## Set the high nybbles of rawl based on dati:

rawh <- signedIntToNybble(dati, "high")
```

---

# signedIntToRaw *Convert signed integers (short) into a raw vector*

---

## Description

This function converts signed integer values into a vector of `raw` data.

## Usage

```
signedIntToRaw(int_dat)
```

### Arguments

| | |
|---|---|
| int_dat | A vector of integer values, ranging from -128 up to 127. |

### Details

This function converts signed integer values [-128,127] into a vector of raw data. The function will fail on values that are out of range (< -128 or > 127). To convert raw data into a vector of unsigned integers use as.integer(). For the inverse of this function see rawToSignedInt().

### Value

A vector of the same length as int_dat, holding raw data.

### Author(s)

Pepijn de Vries

### See Also

Other raw.operations: as.raw(), nybbleToSignedInt(), nybble(), rawToCharNull(), rawToPTModule(), rawToSignedInt(), rawToUnsignedInt(), signedIntToNybble(), unsignedIntToRaw()

Other integer.operations: nybbleToSignedInt(), nybble(), rawToSignedInt(), rawToUnsignedInt(), signedIntToNybble(), unsignedIntToRaw(), waveform()

### Examples

```
## generate some signed integers:
some.integers <- c(-100, 40, 0, 30, -123)

## convert the signed integers into a vector of raw data:
signedIntToRaw(some.integers)
```

---

## trackerFlag *Tracker flag indicating version compatibility*

---

### Description

Method to obtain a tracker flag, which indicates the version compatibility of a ProTracker module (PTModule object).

### Usage

```
## S4 method for signature 'PTModule'
trackerFlag(x)

## S4 replacement method for signature 'PTModule'
trackerFlag(x) <- value
```

## Arguments

| | |
|---|---|
| x | A PTModule object for which the flag needs to returned or replaced. |
| value | A character string representing the tracker flag with which that of object x needs to be replaced with. Should either be "M.K." or "M!K!". Note that if a current flag "M!K!" is replaced by "M.K.", PTPatterns may get lost as the latter supports less patterns. |

## Details

ProTrackR supports two tracker flags: "M.K." and "M!K!". M.K. are presumably the initials of programmers Mahony and Kaktus, unfortunately documentation on this matter is ambiguous. In any case, modules with the flag "M.K." can hold up to 64 patterns, whereas modules with the flag "M!K!" can hold up to 100 patterns. Use this method to obtain or replace the tracker flag of a PTModule.

## Value

For trackerFlag, the tracker flag of object x is returned.

For trackerFlag<-, a copy of object x with an updated tracker flag is returned.

## Author(s)

Pepijn de Vries

## See Also

Other module.operations: PTModule-class, appendPattern(), clearSamples(), clearSong(), deletePattern(), fix.PTModule(), modToWave(), moduleSize(), patternLength(), patternOrderLength(), patternOrder(), playMod(), playingtable(), rawToPTModule(), read.module(), write.module()

## Examples

```
data("mod.intro")

## the current trackerFlag of mod.intro is "M.K.",
## meaning that it can hold a maximum of 64 patterns:
trackerFlag(mod.intro)

patternOrder(mod.intro, full = TRUE) <- 0:63

## If we upgrade the trackerFlag of mod.intro to "M!K!"
## it can hold a maximum of 100 patterns!:
trackerFlag(mod.intro) <- "M!K!"

patternOrder(mod.intro, full = TRUE) <- 0:99

## Now let's do something dangerous:
## current flag is "M!K!", by setting it
## back to "M.K.", patterns 65:100 are lost...
trackerFlag(mod.intro) <- "M.K."
```

unsignedIntToRaw *Convert unsigned integer into a raw vector*

## Description

This function converts an unsigned integer into a vector of raw data.

## Usage

```
unsignedIntToRaw(int_dat, length.out = 1)
```

## Arguments

| | |
|---|---|
| int_dat | A single integer value. If a list or vector of values. is provided, only the first element is evaluated. Input data are converted to absolute integer values. |
| length.out | Required length of the vector that will hold the resulting. raw data. Defaults to 1. If the value of int_dat is to large to convert into raw data of length length.out, data will be clipped. |

## Details

This function converts an unsigned integer value into a vector (with a specified length, namely length.out) of raw data. For the inverse of this function use rawToUnsignedInt().

## Value

A vector of length length.out, holding raw data.

## Author(s)

Pepijn de Vries

## See Also

Other raw.operations: as.raw(), nybbleToSignedInt(), nybble(), rawToCharNull(), rawToPTModule(), rawToSignedInt(), rawToUnsignedInt(), signedIntToNybble(), signedIntToRaw()

Other integer.operations: nybbleToSignedInt(), nybble(), rawToSignedInt(), rawToUnsignedInt(), signedIntToNybble(), signedIntToRaw(), waveform()

## Examples

```
## generate some unsigned integer:
some.integer <- 43251

## convert the unsigned integer into raw data:
unsignedIntToRaw(some.integer, length.out = 4)

## Not run:
## note that the integer is too large to store as raw with length.out = 1:
unsignedIntToRaw(some.raw.data, length.out = 1)

## End(Not run)
```

---

# volume *Default playback volume of PTSample*

---

## Description

Extract or replace the default volume of a PTSample.

## Usage

```
## S4 method for signature 'PTSample'
volume(sample)

## S4 replacement method for signature 'PTSample,numeric'
volume(sample) <- value
```

## Arguments

| | |
|---|---|
| sample | A PTSample for which the default volume needs to be extracted or replace. |
| value | A numeric value ranging from 0 up to 64, representing the volume level. |

## Details

PTSamples have a default playback volume, ranging from 0 (silent) up to 64 (maximum volume). This method can be used to extract this value, or to safely replace it.

## Value

For volume the volume value, represented by an integer value ranging from 0 up to 64, is returned.

For volume<- A PTSample sample, updated with the volume value, is returned.

## Author(s)

Pepijn de Vries

## See Also

Other sample.operations: PTSample-class, PTSample-method, fineTune(), loopLength(), loopSample(), loopStart(), loopState(), name, playSample(), read.sample(), sampleLength(), waveform(), write.sample()

## Examples

```
data("mod.intro")

## get the volume of the first sample of mod.intro:

volume(PTSample(mod.intro, 1))

## Let's lower the volume of this sample to 32
## (or as a hexadecimal: 0x20):

volume(PTSample(mod.intro, 1)) <- 0x20
```

---

# waveform *Extract or replace a PTSample waveform*

---

## Description

Extract or replace the waveform of a PTSample object. The waveform is represented by a vector of numeric values ranging from 0 up to 255.

## Usage

```
## S4 method for signature 'PTSample'
waveform(sample, start.pos = 1, stop.pos = sampleLength(sample), loop = TRUE)

## S4 replacement method for signature 'PTSample'
waveform(sample) <- value
```

## Arguments

| | |
|---|---|
| sample | A PTSample object from which the waveform needs to be extracted or replaced. |
| start.pos | A numeric starting index, giving the starting position for the waveform to be returned. Default value is 1. This index should be greater than zero. |
| stop.pos | A numeric stopping index, giving the stopping position for the waveform to be returned. Default value is sampleLength(sample) This index should be greater than start.pos. |
| loop | A logical value indicating whether the waveform should be modulated between the specified loop positions (see loopStart and loopLength), or the waveform should stop at the end of the sample (padded with NA values beyond the sample length). Will do the first when set to TRUE and the latter when set to FALSE. |

value — A vector of numeric values ranging from 0 up to 255, representing the waveform that should be used to replace that of object sample. The length should be even and not exceed 2*0xffff = 131070. loopStart and loopLength will be adjusted automatically when they are out of range for the new waveform.

Use NA to generate an empty/blank PTSample object.

## Details

Sample waveforms are stored as 8 bit signed short integer values ranging from -128 up to +127 in original ProTracker files. However, as the PTSample class extends the tuneR::Wave class, the waveforms are represented by integer values ranging from 0 up to 255 in the ProTrackR package. As per ProTracker specifications, samples are of 8 bit mono quality and can only have an even length with a maximum of 2*0xffff = 131070. This method can be used to extract a waveform or replace it.

## Value

For waveform, the waveform of sample is returned as a vector of numeric values ranging from 0 up to 255. If loop is set to FALSE and the starting position is beyond the sample length, NA values are returned. If loop is set to TRUE and the starting position is beyond the sample loop (if present, see loopState), the waveform is modulated between the loop positions.

For waveform<-, a copy of object sample is returned in which the waveform has been replaced with value.

## Author(s)

Pepijn de Vries

## See Also

Other integer.operations: nybbleToSignedInt(), nybble(), rawToSignedInt(), rawToUnsignedInt(), signedIntToNybble(), signedIntToRaw(), unsignedIntToRaw()

Other sample.operations: PTSample-class, PTSample-method, fineTune(), loopLength(), loopSample(), loopStart(), loopState(), name, playSample(), read.sample(), sampleLength(), volume(), write.sample()

## Examples

```
data("mod.intro")

## Loop sample #1 of mod.intro beyond it's
## length of 1040 samples:
wav1 <- waveform(PTSample(mod.intro, 1),
                 1, 5000)

## get the waveform from sample #2
## of mod.intro:
wav2 <- waveform(PTSample(mod.intro, 2))

## create an echo effect using
```

```
## the extracted waveform:
wav2 <- c(wav2, rep(128, 1000)) +
        c(rep(128, 1000), wav2)*0.25 - 25

## assign this echoed sample to
## sample #2 in mod.intro:
waveform(PTSample(mod.intro, 2)) <- wav2

## Blank out sample #1 in mod.intro:
waveform(PTSample(mod.intro, 1)) <- NA
```

---

`write.module` *Export an PTModule object as a ProTracker module file*

---

## Description

Export an `PTModule` object as a ProTracker module file, conform ProTracker 2.3A specifications.

## Usage

```
## S4 method for signature 'PTModule,ANY'
write.module(mod, file)

## S4 method for signature 'PTModule,character'
write.module(mod, file)
```

## Arguments

| | |
|---|---|
| `mod` | A valid PTModule object to be saved as a ProTracker *.mod file |
| `file` | either a filename to write to, or a file connection, that allows to write binary data (see `base::file`). |

## Details

The routine to write ProTracker modules is based on the referenced version of ProTracker 2.3A. This means that the routine may not be able to write files that ar compatible with later or earlier ProTracker versions.

## Value

Writes to a module file but returns nothing.

## Author(s)

Pepijn de Vries

## References

https://wiki.multimedia.cx/index.php?title=Protracker_Module

## See Also

Other io.operations: read.module(), read.sample(), write.sample()

Other module.operations: PTModule-class, appendPattern(), clearSamples(), clearSong(), deletePattern(), fix.PTModule(), modToWave(), moduleSize(), patternLength(), patternOrderLength(), patternOrder(), playMod(), playingtable(), rawToPTModule(), read.module(), trackerFlag()

## Examples

```
## Not run:
## get the PTModule object provided with the ProTrackR package
data("mod.intro")

## save the object as a valid ProTracker module file:
write.module(mod.intro, "intro.mod")

## or create the connection yourself:
con <- file("intro2.mod", "wb")
write.module(mod.intro, con)

## don't forget to close the connection after you're done:
close(con)

## End(Not run)
```

---

# write.sample *Write a PTSample object to an audio file*

---

## Description

Write a PTSample as a "wav", "8svx" or "raw" audio file.

## Usage

```
## S4 method for signature 'PTSample,character'
write.sample(sample, filename, what = c("wav", "8svx", "raw"))
```

## Arguments

| | |
|---|---|
| sample | A PTSample object that needs to be exported to an audio file. |
| filename | A character string representing the filename to which the audio needs to be saved. |
| what | A character string indicating what type of file is to be exported. Can be one of the following: "wav" (default), "8svx" or "raw". The AmigaFFH package needs to be installed in order to write 8svx files. |

## Details

This method provides a wrapper for the tuneR::writeWave method from tuneR::tuneR. It also provides the means to export audio to file formats native to the Commodore Amiga. PTSamples can be exported as simple (uncompressed) 8svx files also known as "iff" files). In addition they can be exported as raw data, where each byte simply represents a signed integer value of the waveform.

## Value

Saves the audio to a file, but returns nothing.

## Author(s)

Pepijn de Vries

## See Also

Other sample.operations: PTSample-class, PTSample-method, fineTune(), loopLength(), loopSample(), loopStart(), loopState(), name, playSample(), read.sample(), sampleLength(), volume(), waveform()

Other io.operations: read.module(), read.sample(), write.module()

## Examples

```
## Not run:
data("mod.intro")

## Export the second sample of mod.intro as a wav file:
write.sample(PTSample(mod.intro, 2), "snaredrum.wav", "wav")

## Export the second sample of mod.intro as an 8svx file:
write.sample(PTSample(mod.intro, 2), "snaredrum.iff", "8svx")

## Export the second sample of mod.intro as a raw file:
write.sample(PTSample(mod.intro, 2), "snaredrum.raw", "raw")

## End(Not run)
```

# Index