Package 'AmigaFFH'

January 20, 2025

Type Package

Title Commodore Amiga File Format Handler

Version 0.4.7

Author Pepijn de Vries [aut, cre, dtc]

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

Description Modern software often poorly support older file formats. This package intends to handle many file formats that were native to the antiquated Commodore Amiga machine. This package focuses on file types from the older Amiga operating systems (<= 3.0). It will read and write specific file formats and coerces them into more contemporary data.

Depends R (>= 2.10), tuneR (>= 1.0)

Imports grDevices, methods, utils, vctrs

Suggests adfExplorer (>= 2.0.0), ProTrackR (>= 0.3.4), testthat (>= 3.0.0)

License GPL-3

LazyData True

Encoding UTF-8

RoxygenNote 7.2.3

NeedsCompilation no

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

BugReports https://github.com/pepijn-devries/AmigaFFH/issues

Config/testthat/edition 3

Repository CRAN

Date/Publication 2025-01-08 09:10:08 UTC

Contents

`AmigaBasic` *The S3 AmigaBasic class*

Description

A class that represents the content of Amiga Basic files.

Details

Amiga Basic is a BASIC-style programming language that was shipped with early Commodore Amiga machines. It requires an interpreter to run an Amiga Basic script. The AmigaFFH package does not interpret Amiga Basic scripts. It does allow for encoding and decoding scripts in the binary format in which it was originally stored on the Amiga. Amiga Basic scripts were stored as encoded binaries instead of ASCII text files in order to save (at the time precious) memory and disk space.

Amiga Basic binary files start with a file header (as an identifier) and is followed by each line of the script as binary data. The `AmigaBasic`-class object stores each line of the script as a `list` item as a `vector` of `raw` data. Use `as.character()` and `as.AmigaBasic()` to switch between `character` data and `AmigaBasic`-class objects.

Note

Although there is ample reference material on the Amiga BASIC language, there is no documentation available on the script file storage format. The implementation in the AmigaFFH package is all the result of painstaking reverse engineering on my part. Consequently the Amiga Basic file encoders and decoders implemented here may not be infallible.

Author(s)

Pepijn de Vries

References

https://en.wikipedia.org/wiki/AmigaBASIC

See Also

Other AmigaBasic.operations: `AmigaBasic.reserved()`, `AmigaBasicBMAP`, `[.AmigaBasic()`, `as.AmigaBasicBMAP()`, `as.AmigaBasic()`, `as.character()`, `check.names.AmigaBasic()`, `names.AmigaBasic()`, `rawToAmigaBasicBMAP()`, `rawToAmigaBasic()`, `read.AmigaBasicBMAP()`, `read.AmigaBasic()`, `write.AmigaBasic()`

Examples

```
## This creates an AmigaBasic-class object:
bas <- as.AmigaBasic("PRINT \"hello world!\"")

## This will decode the object as plain text:
as.character(bas)
```

AmigaBasic-files *'demo.bas', 'r_logo.shp' and 'ball.shp' as example files for AmigaBasic and AmigaBasicShape objects*

Description

'demo.bas', 'r_logo.shp' and 'ball.shp' as example files for `AmigaBasic()` and `AmigaBasicShape()` objects

Format

See `AmigaBasic()` and `AmigaBasicShape()` for more information about the format.

Details

The 'r_logo.shp' and 'ball.shp' files are formatted such that they can be read with `read.AmigaBasicShape()`. They serve as an example of the `AmigaBasicShape()` class, where the first represents a blitter object, and the latter a sprite.

The 'demo.bas' file is an example of a binary encoded Amiga Basic script. It can be read with `read.AmigaBasic()`. The script demonstrates how the shape files could be used in Amiga Basic.

Examples

```
read.AmigaBasic(system.file("demo.bas", package = "AmigaFFH"))
read.AmigaBasicShape(system.file("ball.shp", package = "AmigaFFH"))
read.AmigaBasicShape(system.file("r_logo.shp", package = "AmigaFFH"))
```

`AmigaBasic.reserved` *List Amiga Basic reserved words.*

Description

Obtain a list of reserved Amiga Basic words. These words are not allowed as names of variables or labels in Amiga Basic.

Usage

```
AmigaBasic.reserved()
```

Details

This function will return a full list of reserved Amiga Basic words. This list does not serve as a manual for basic (for that purpose consult external resources). This list is meant to consult when choosing label names in Amiga Basic code. These reserved words are not allowed as names.

Value

Returns a `vecor` of `character` strings of reserved Amiga Basic words.

Author(s)

Pepijn de Vries

See Also

Other AmigaBasic.operations: `AmigaBasicBMAP`, `AmigaBasic`, `[.AmigaBasic()`, `as.AmigaBasicBMAP()`, `as.AmigaBasic()`, `as.character()`, `check.names.AmigaBasic()`, `names.AmigaBasic()`, `rawToAmigaBasicBMAP()`, `rawToAmigaBasic()`, `read.AmigaBasicBMAP()`, `read.AmigaBasic()`, `write.AmigaBasic()`

Examples

```
AmigaBasic.reserved()
```

AmigaBasicBMAP *The S3 AmigaBasicBMAP class*

Description

A class that represents the content of Amiga Basic BMAP files.

Details

The Amiga operating system made use of library files to execute specific (repetitive/routine) tasks. Amiga Basic was also able to call such routines from library files. In order to do so, it required a 'bmap' file for each library. This file contains a map of the library where it specifies: the name of routine; the 'Library Vector Offset' (explained below); and used CPU registers (explained below).

The 'Library Vector Offset' is an offset to the base address of a library in memory. This offsets indicates where a specific executable routine starts. The CPU registers are used to (temporary) store (pointers to) input data used by the routine. The BMAP file thus lists which CPU registers are used by specified routines.

Author(s)

Pepijn de Vries

References

https://en.wikipedia.org/wiki/AmigaOS#Libraries_and_devices

See Also

Other AmigaBasic.operations: `AmigaBasic.reserved()`, `AmigaBasic`, `[.AmigaBasic()`, `as.AmigaBasicBMAP()`, `as.AmigaBasic()`, `as.character()`, `check.names.AmigaBasic()`, `names.AmigaBasic()`, `rawToAmigaBasicBMAP()`, `rawToAmigaBasic()`, `read.AmigaBasicBMAP()`, `read.AmigaBasic()`, `write.AmigaBasic()`

AmigaBasicShape *The S3 AmigaBasicShape class*

Description

A class that represents the file format used by Amiga Basic to store bitmap graphics: blitter objects and sprites.

Details

Amiga Basic used a specific format to store bitmap images that could be displayed using Basic code. Both sprites and blitter objects can be stored and used. This class is used to represent such files.

Author(s)

Pepijn de Vries

See Also

Other AmigaBasicShape.operations: `rasterToAmigaBasicShape()`, `read.AmigaBasicShape()`, `write.AmigaBasicShape()`

Examples

```
ball   <- read.AmigaBasicShape(system.file("ball.shp", package = "AmigaFFH"))
r_logo <- read.AmigaBasicShape(system.file("r_logo.shp", package = "AmigaFFH"))

plot(ball)
plot(r_logo)
```

`AmigaBitmapFont` *The S3 AmigaBitmapFont and AmigaBitmapFontSet classes*

Description

A comprehensive representation of monochromous Amiga bitmap fonts.

Details

Nowadays fonts are represented by vector graphics an computer systems. On the original Commodore Amiga, the screen resolution, system memory and cpu speed were limited. On those systems, it was more efficient to use bitmap images to represent the glyphs in fonts. The `AmigaBitmapFontSet` and `AmigaBitmapFont` classes can be used to represent Amiga bitmap fonts.

The Commodore Amiga had a directory named 'FONTS' located in the root, where (bitmap) fonts were stored. Font sets were stored under the font name with a *.font extension. Files with the *.font extension did not contain the bitmap images of the font. Rather the *.font file contained information on which font heights (in pixels) are available, in addition to some other meta-information.

The bitmap images were stored in separate files for each individual height. The `AmigaBitmapFontSet` is an S3 class that forms a comprehensive format (named `list`) to represent the *.font files. The `AmigaBitmapFont` is an S3 class is a comprehensive format (named `list`) that represent each font bitmap and glyph information. The `AmigaBitmapFontSet` objects will hold one or more `AmigaBitmapFont` objects.

The `AmigaBitmapFont` and `AmigaBitmapFontSet` objects are essentially named `lists`. Their structure and most important elements are described below. Although it is possible to replace elements manually, it is only advisable when you know what you are doing as it may break the validity of the font.

AmigaBitmapFontSet

- `fch_FileID`: A `factor` with levels 'FontContents', 'TFontContents' and 'ScalableOutline'. It specifies the type of font. Currently only the first level is supported.
- `fch_NumEntries`: number of font heights available for this font. It should match with the length of `FontContents`. Do not change this value manually.
- `FontContents`: This is a `list` with bitmap entries for each specific font height (in pixels). The name of each element in this list is 'pt' followed by the height. Each element in this list holds the elements:
  - Miscellaneous: Miscellaneous information from the *.font file
    - `fc_FileName`: This element represents the filename of the nested font bitmap images. Note that it should be a valid Commodore Amiga filename. It is best to modify this name using `fontName()`. Note that this field could cause problems as Commodore Amiga filenames can contain characters that most modern platforms would not allow (such as the question mark).
    - `BitmapFont`: This element is of type `AmigaBitmapFont` and is structured as described in the following section. The information in this element is no longer part of the *.font file.

AmigaBitmapFont

Information represented by a `AmigaBitmapFont` is not stored in *.font files. Rather it is stored in sub-directories of the font in separate files. It has the following structure:

- Miscellaneous: Elements with information on the font properties and style, and also relative file pointers.
- `glyph.info`: A `data.frame` containing glyph info with information for specific glyphs on each row. Each row matches with a specific ASCII code, ranging from `tf_LoChar` up to `tf_HiChar`. There is an additional row that contains information for the default glyph that is out of the range of the `tf_LoChar` and `tf_HiChar`. The `data.frame` thus has `2 + tf_HiChar - tf_LoChar` rows. This table is used to extract and plot a glyph from the `bitmap` image correctly.
- `bitmap`: Is a monochromous bitmap image of all the font's glyphs in a single line. It is a simple `raster` object (see `grDevices::as.raster()`) with an additional attribute 'palette', which lists the two colours in the image. In this palette, the first colour is the background colour and the second colour is interpreted as the foregroundcolour.

Useful functions

For importing and exporting the following functions are useful: `read.AmigaBitmapFont()`, `read.AmigaBitmapFontSet()`, `write.AmigaBitmapFont()` and `write.AmigaBitmapFontSet()`.

The following generic functions are implemented for these objects: `plot()`, `print`, `as.raster()` and `as.raw()`.

Use `c()` to combine one or more `AmigaBitmapFont` objects into a `AmigaBitmapFontSet`.

Author(s)

Pepijn de Vries

References

http://amigadev.elowar.com/read/ADCD_2.1/Libraries_Manual_guide/node03E0.html http://amigadev.elowar.com/read/ADCD_2.1/Libraries_Manual_guide/node03DE.html http://amigadev.elowar.com/read/ADCD_2.1/Libraries_Manual_guide/node05BA.html

See Also

Other AmigaBitmapFont.operations: availableFontSizes(), c(), fontName(), font_example, getAmigaBitmapFont(), rasterToAmigaBitmapFont(), rawToAmigaBitmapFontSet(), rawToAmigaBitmapFont(), read.AmigaBitmapFontSet(), read.AmigaBitmapFont(), write.AmigaBitmapFont()

Other raster.operations: as.raster.AmigaBasicShape(), bitmapToRaster(), dither(), index.colours(), rasterToAmigaBasicShape(), rasterToAmigaBitmapFont(), rasterToBitmap(), rasterToHWSprite(), rasterToIFF()

Examples

```
## 'font_example' is an example of the AmigaBitmapFontSet object:
data(font_example)

## An AmigaBitmapFont object can also be extracted from this object:
font_example_9 <- getAmigaBitmapFont(font_example, 9)

## the objects can be printed, plotted, converted to raw data or a raster:
print(font_example)
plot(font_example)
font_example_raw    <- as.raw(font_example)
font_example_raster <- as.raster(font_example)

## You can also format text using the font:
formated_raster     <- as.raster(font_example, text = "Foo bar", style = "bold")
plot(font_example, text = "Foo bar", style = "underlined", interpolate = FALSE)
```

AmigaIcon *The S3 AmigaIcon class*

Description

A comprehensive representation of an Amiga Workbench icon file.

Details

Files, directories and other similar objects were depicted as icons on the Amiga Workbench (the Amiga's equivalent of what is now mostly known as the computer's desktop). Icons were actually separate files with the exact same name as the file or directory it represents, except for an additional '.info' extension.

In addition of being a graphical representation of files or directories, icon files also contained additional information about the file. It could for instance indicate which tool would be required to open the file.

The classic Amiga Workbench icon file has a rather complex structure as it is basically a dump of how it is stored in memory. As a result it contains many memory pointers that are really not necassary to store in a file.

The S3 AmigaIcon class is used to represent these complex files as a named `list`. The elements in that `list` have mostly identical names as listed in the document at the top referenced below. The names are usually self-explanatory, but the referred documents can also be consulted to obtain more detailed information with respect to each of these elements. As pointed out earlier, not all elements will have a meaningful use.

It is possible to change the values of the list, but not all values may be valid. Note that they will not be fully checked for validity. Invalid values may result in errors when writing to a binary file using `write.AmigaIcon()`, or may simply not work properly on an Amiga or in an emulator.

The original '.info' file could be extended with NewIcon or with an OS3.5 `IFFChunk()` data, that allowed for icons with larger colour depths. These extensions are currently not implemented.

Use `simpleAmigaIcon()` for creating a simple `AmigaIcon` object which can be modified. Use `read.AmigaIcon()` to read, and `write.AmigaIcon()` to write workbench icon files (*.info). With `rawToAmigaIcon()` and `as.raw()` `AmigaIcon` can be coerced back and forth from and to its raw (binary) form.

Author(s)

Pepijn de Vries

References

http://www.evillabs.net/index.php/Amiga_Icon_Formats http://fileformats.archiveteam.org/wiki/Amiga_Workbench_icon http://amigadev.elowar.com/read/ADCD_2.1/Libraries_Manual_guide/node0241.html http://amigadev.elowar.com/read/ADCD_2.1/Includes_and_Autodocs_3._guide/node05D6.html

See Also

Other AmigaIcon.operations: `rawToAmigaIcon()`, `read.AmigaIcon()`, `simpleAmigaIcon()`, `write.AmigaIcon()`

---

amiga_display_keys *A list of special display modes*

Description

A list of special display modes on the Amiga and corresponding `raw` keys.

Format

A `data.frame` with 2 columns:

- The column named 'mode': a `factor` reflecting a display mode, monitor or bitwise mask
- The column named 'code': vector of 4 `raw` values as used by the Amiga to reflect specific display modes

Details

This table show specific special display modes and to which Amiga monitors they relate. The raw codes can be used to interpret specific display modes as listed in amiga_display_modes(). This information is used to interpret IFFChunk() objects of type 'CAMG'. It is also used to interpret ILBM images and creating IFF files from raster images.

References

https://wiki.amigaos.net/wiki/Display_Database#ModeID_Identifiers

http://amigadev.elowar.com/read/ADCD_2.1/AmigaMail_Vol2_guide/node00FD.html

Examples

```
data("amiga_display_keys")
```

---

amiga_display_modes *A table of display modes on the Amiga and corresponding* raw *codes*

Description

A table of display modes on the Amiga and corresponding raw codes representing these modes.

Format

A data.frame with 4 columns:

- The column named 'DISPLAY_MODE': a factor reflecting the display mode
- The column named 'DISPLAY_MODE_ID': A list containing a vector of 4 raw values as used by the Amiga to reflect specific display modes. These raw values are usually also stored with bitmap images in the Interchange File Format in a IFFChunk() called 'CAMG'.
- The column named 'MONITOR_ID': A character string identifying the monitor that could display the specific mode.
- The column named 'CHIPSET': a factor identifying the minimal chip set that was required to display the specific mode. OCS is the original chip set; ECS is the Enhanced Chip Set. AGA is the Advanced Graphics Architecture chip set (in some countries known as just Advanced Architecture). AGA could also display OCS and ECS modes, ECS could also display OCS modes, OCS could only display OCS modes.

Details

This table contains most display modes that were available on the Amiga. It also contains raw codes that were used to represent these modes. The table also contains the hardware monitors that could display the specific modes, and the minimal chip set that was required for the display mode. This data is used to interpret IFFChunk() objects of type 'CAMG'. It is also used to interpret ILBM images and creating IFF files from raster images.

References

https://wiki.amigaos.net/wiki/Display_Database#ModeID_Identifiers

http://amigadev.elowar.com/read/ADCD_2.1/AmigaMail_Vol2_guide/node00FD.html

Examples

```
data("amiga_display_modes")
```

---

amiga_monitors *A list of Amiga monitors*

Description

This table lists Amiga monitors and corresponding `raw` codes that represent these monitors.

Format

A `data.frame` with 2 columns:

- The column named 'MONITOR_ID': a `factor` representing an Amiga monitor
- The column named 'CODE': A `list` containing a `vector` of 4 `raw` values as used by the Amiga to represent a specific monitor.

Details

This table contains monitors that were compatible with the Amiga. It also contains `raw` codes that were used to represent them. This data is used to interpret `IFFChunk()` objects of type 'CAMG'. It is also used to interpret ILBM images and creating IFF files from raster images.

References

https://wiki.amigaos.net/wiki/Display_Database#ModeID_Identifiers

Examples

```
data("amiga_monitors")
```

amiga_palettes *Commonly used palettes on the Commodore Amiga*

Description

`amiga_palettes` is a named list, where each element represents a commonly used palette on the Commodore Amiga.

Format

A named list with the following elements:

- `wb.os1`: A `vector` of 4 colours that were used as the default palette of the Workbench on Amiga OS 1.x.
- `wb.os2`: A `vector` of 8 colours. The first 4 colours are the default colours of a standard Workbench on Amiga OS 2.x. The latter 4 are additional colours used by the Workbench expansion MagicWB.
- `spr.os1`: A `vector` of 3 colours that were used by default for a mouse pointer sprite on Amiga OS 1.x.
- `spr.os2`: A `vector` of 3 colours that were used by default for a mouse pointer sprite on Amiga OS 2.x.

Details

Some files that contain bitmap images with an indexed palette did not store the palette in the same file. Amiga Workbench icons (`AmigaIcon()`) for instance only store the index values of the palette, but not the palette itself. `amiga_palettes` therefore provides some commonly used palettes on the Amiga, such that these files can be interpreted.

Examples

```
data("amiga_palettes")
```

as.AmigaBasic *Coerce raw or character data to an AmigaBasic class object*

Description

Coerce raw or character data to an `AmigaBasic()` S3 class object

Usage

```
as.AmigaBasic(x, ...)
```

Arguments

x	x should be a vector of raw data or character strings. When x is raw data, it is interpreted as if it where from an Amiga Basic binary encoded file. When x is a vector of character strings, each element of the vector should represent one line of Basic code. Each line should not contain line break or other special characters, as this will result in errors. The text should represent valid Amiga Basic syntax. The syntax is only checked to a limited extent as this package does not implement an interpreter for the code.
...	Currently ignored.

Details

Convert text to an AmigaBasic() S3 class object. The text should consist of valid Amiga BASIC syntaxis. This function does not perform a full check of the syntaxis, but will break on some fundamental syntaxis malformations

Value

Returns an AmigaBasic() class object based on x.

Author(s)

Pepijn de Vries

References

https://en.wikipedia.org/wiki/AmigaBASIC

See Also

Other AmigaBasic.operations: AmigaBasic.reserved(), AmigaBasicBMAP, AmigaBasic, [.AmigaBasic(), as.AmigaBasicBMAP(), as.character(), check.names.AmigaBasic(), names.AmigaBasic(), rawToAmigaBasicBMAP(), rawToAmigaBasic(), read.AmigaBasicBMAP(), read.AmigaBasic(), write.AmigaBasic()

Other raw.operations: as.raw.AmigaBasic(), colourToAmigaRaw(), packBitmap(), rawToAmigaBasicBMAP(), rawToAmigaBasicShape(), rawToAmigaBasic(), rawToAmigaBitmapFontSet(), rawToAmigaBitmapFont(), rawToAmigaIcon(), rawToHWSprite(), rawToIFFChunk(), rawToSysConfig(), simpleAmigaIcon()

Examples

```
## An AmigaBasic object can be created from text.
## Note that each line of code is a seperate element
## in the vector:
bas <- as.AmigaBasic(c(
  "CLS ' Clear the screen",
  "PRINT \"Hello world!\" ' Print a message on the screen"
))

## Let's make it raw data:
```

```
bas.raw <- as.raw(bas)

## We can also use the raw data to create an Amiga Basic object:
## Note that this effectively the same as calling 'rawToAmigaBasic'
bas <- as.AmigaBasic(bas.raw)
```

`as.AmigaBasicBMAP` *Coerce raw or named list to an AmigaBasicBMAP class object*

Description

Coerce `raw` or named `list` to an `AmigaBasicBMAP()` class object

Usage

```
as.AmigaBasicBMAP(x)
```

Arguments

`x`	When `x` is a `vector` of `raw` data, it needs to be structured as it would be when stored in a binary file (see `read.AmigaBasicBMAP()`). `x` can also be a named `list`, where the name of each element corresponds with a routine in the library. Each element should than consist of a `list` with 2 elements: The first should be named `libraryVectorOffset'` and should hold the numeric`offset of the routine in the libra` and should contain a `vector` of `raw` values refering to CPU registers used by the routine (see details).

Details

An Amiga Basic BMAP file maps the offset of routines in Amiga libraries. This function converts the raw format in which it would be stored as a file into a comprehensive S3 class object. It can also convert a named list into an S3 class object. See `Arguments'` and Examples' sections on how to format this list.

Value

Returns a `AmigaBasicBMAP()` based on `x`

Author(s)

Pepijn de Vries

See Also

Other AmigaBasic.operations: `AmigaBasic.reserved()`, `AmigaBasicBMAP`, `AmigaBasic`, `[.AmigaBasic()`, `as.AmigaBasic()`, `as.character()`, `check.names.AmigaBasic()`, `names.AmigaBasic()`, `rawToAmigaBasicBMAP()`, `rawToAmigaBasic()`, `read.AmigaBasicBMAP()`, `read.AmigaBasic()`, `write.AmigaBasic()`

Examples

```
## For the dos.library, the start of the bmap list would look like:
dos.list <- list(
  xOpen = list(
    libraryVectorOffset = -30,
    registers = as.raw(2:3)
  ),
  xClose = list(
    libraryVectorOffset = -36,
    registers = as.raw(2)
  ),
  xRead = list(
    libraryVectorOffset = -42,
    registers = as.raw(2:4)
  )
)

## Note that the list above is incomplete, the dos.library holds more routines than shown here.
## This merely serves as an example.
## This list can be converted to an S3 class as follows:
dos.bmap <- as.AmigaBasicBMAP(dos.list)
```

---

as.character *Coerce an AmigaBasic class object to its character representation*

Description

Coerce an AmigaBasic()-class object to its character representation

Usage

```
## S3 method for class 'AmigaBasic'
as.character(x, ...)
```

Arguments

x	An AmigaBasic() class object that needs to be coerced to its character representation.
...	Currently ignored.

Details

Amiga Basic files are encoded in a binary format and are also stored as such in AmigaBasic()-class objects. Use this function to convert these objects into legible character data.

Value

A vector of character strings, where each element of the vector is a character representation of a line of Amiga Basic code stored in x.

Author(s)

Pepijn de Vries

See Also

Other AmigaBasic.operations: AmigaBasic.reserved(), AmigaBasicBMAP, AmigaBasic, [.AmigaBasic(), as.AmigaBasicBMAP(), as.AmigaBasic(), check.names.AmigaBasic(), names.AmigaBasic(), rawToAmigaBasicBMAP(), rawToAmigaBasic(), read.AmigaBasicBMAP(), read.AmigaBasic(), write.AmigaBasic()

Examples

```
## First create an Amiga Basic object:
bas <- as.AmigaBasic("PRINT \"Hello world!\"")

## now convert the object back into text:
bas.txt <- as.character(bas)
```

---

as.raster.AmigaBasicShape

Convert AmigaFFH objects into grDevices raster images

Description

Convert AmigaFFH objects that contain bitmap images into grDevices raster images.

Usage

```
## S3 method for class 'AmigaBasicShape'
as.raster(x, selected = c("bitmap", "shadow", "collision"), ...)

## S3 method for class 'AmigaBitmapFont'
as.raster(x, text, style, palette, ...)

## S3 method for class 'AmigaBitmapFontSet'
as.raster(x, text, style, palette, ...)

## S3 method for class 'hardwareSprite'
as.raster(x, background = "#AAAAAA", ...)

## S3 method for class 'IFFChunk'
as.raster(x, ...)

## S3 method for class 'AmigaIcon'
as.raster(x, selected = FALSE, ...)
```

Arguments

x	Object that needs to be converted into a grDevices raster. It can be an IFFChunk() containing an interleaved bitmap image (ILBM) or animation (ANIM), a hardwareSprite(), an AmigaBitmapFont() object or an AmigaBitmapFontSet() object.
selected	When x is an object of class AmigaIcon(), selected can be used to select a specific state. When set to TRUE, the raster of the AmigaIcon() will be based on the 'selected' state of the icon. Otherwise it will be based on the deselected state (default). When x is an AmigaBasicShape() class object, selected can be used to select a specific layer of the shape to plot, which can be one of "bitmap" (default), "shadow" or "collision".
...	Currently ignored.
text	Text (a character string) to be formated with x (when x is an AmigaBitmapFont() or an AmigaBitmapFontSet().
style	Argument is only valid when x is an AmigaBitmapFont() or an AmigaBitmapFontSet(). No styling is applied when missing or NULL. One or more of the following styles can be used 'bold', 'italic or 'underlined'.
palette	Argument is only valid when x is an AmigaBitmapFont() or an AmigaBitmapFontSet(). Should be a vector of two colours. The first is element is used as background colour, the second as foreground. When missing, transparent white and black are used.
background	Use the argument background to specify a background colour in case x is a hardwareSprite().

Details

Images on the Amiga were stored as bitmap images with indexed colour palettes. This was mainly due to hardware and memory limitations. Bitmap images could also be embedded in several file types. This method can be used to convert AmigaFFH objects read from such files into grDevices raster images (grDevices::as.raster()).

Value

Returns a grDevices raster image (grDevices::as.raster()) based on x. If x is an animation (IFFChunk() of type ANIM), a list of raster objects is returned.

Author(s)

Pepijn de Vries

See Also

Other raster.operations: AmigaBitmapFont, bitmapToRaster(), dither(), index.colours(), rasterToAmigaBasicShape(), rasterToAmigaBitmapFont(), rasterToBitmap(), rasterToHWSprite(), rasterToIFF()

Other raster.operations: AmigaBitmapFont, bitmapToRaster(), dither(), index.colours(), rasterToAmigaBasicShape(), rasterToAmigaBitmapFont(), rasterToBitmap(), rasterToHWSprite(), rasterToIFF()

Other iff.operations: IFFChunk-class, WaveToIFF(), getIFFChunk(), interpretIFFChunk(), rasterToIFF(), rawToIFFChunk(), read.iff(), write.iff()

Other raster.operations: AmigaBitmapFont, bitmapToRaster(), dither(), index.colours(), rasterToAmigaBasicShape(), rasterToAmigaBitmapFont(), rasterToBitmap(), rasterToHWSprite(), rasterToIFF()

Other raster.operations: AmigaBitmapFont, bitmapToRaster(), dither(), index.colours(), rasterToAmigaBasicShape(), rasterToAmigaBitmapFont(), rasterToBitmap(), rasterToHWSprite(), rasterToIFF()

Examples

```
## load an IFF file
example.iff <- read.iff(system.file("ilbm8lores.iff", package = "AmigaFFH"))

## The file contains an interleaved bitmap image that can be
## converted into a raster:
example.raster <- as.raster(example.iff)

## the raster can be plotted:
plot(example.raster)

## note that the IFFChunk can also be plotted directly:
plot(example.iff)

## Hardware sprites can also be converted into raster images.
## Let's generate a 16x16 sprite with a random bitmap:
spr <- new("hardwareSprite",
           VStop = 16,
           bitmap = as.raw(sample.int(255, 64, replace = TRUE)))

## now convert it into a raster image.
## as the background colour is not specified for hardware
## sprite, we can optionally provide it here.
spr.raster <- as.raster(spr, background = "green")

## AmigaBasicShape objects can also be converted into rasters:
ball <- read.AmigaBasicShape(system.file("ball.shp", package = "AmigaFFH"))
ball.rst <- as.raster(ball)
```

---

as.raw.AmigaBasic *Convert AmigaFFH objects into raw data*

Description

Convert AmigaFFH objects into raw data, as they would be stored in the Commodore Amiga's memory or files.

Usage

```
## S3 method for class 'AmigaBasic'
as.raw(x, ...)

## S3 method for class 'AmigaBasicShape'
as.raw(x, ...)

## S3 method for class 'AmigaBasicBMAP'
as.raw(x)

## S3 method for class 'AmigaBitmapFont'
as.raw(x, ...)

## S3 method for class 'AmigaBitmapFontSet'
as.raw(x, ...)

## S3 method for class 'AmigaTimeVal'
as.raw(x, ...)

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

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

## S3 method for class 'IFF.ANY'
as.raw(x, ...)

## S3 method for class 'SysConfig'
as.raw(x, ...)

## S3 method for class 'AmigaIcon'
as.raw(x, ...)
```

Arguments

x	An AmigaFFH object that needs to be converted into raw data. See usage section for all supported objects.
...	Arguments passed on to IFFChunk-method() when x is of class IFF.ANY.

Details

Objects originating from this package can in some cases be converted into raw data, as they would be stored on an original Amiga. See the usage section for the currently supported objects.

Not all information from x may be included in the raw data that is returned, so handle with care.

As this package grows additional objects can be converted with this method.

Value

Returns a `vector` of `raw` data based on `x`.

Author(s)

Pepijn de Vries

See Also

Other raw.operations: `as.AmigaBasic()`, `colourToAmigaRaw()`, `packBitmap()`, `rawToAmigaBasicBMAP()`, `rawToAmigaBasicShape()`, `rawToAmigaBasic()`, `rawToAmigaBitmapFontSet()`, `rawToAmigaBitmapFont()`, `rawToAmigaIcon()`, `rawToHWSprite()`, `rawToIFFChunk()`, `rawToSysConfig()`, `simpleAmigaIcon()`

Examples

```
## read an IFF file as an IFFChunk object:
example.iff <- read.iff(system.file("ilbm8lores.iff", package = "AmigaFFH"))

## This will recreate the exact raw data as it was read from the file:
example.raw <- as.raw(example.iff)
```

---

availableFontSizes *Get available font sizes from an AmigaBitmapFontSet*

Description

Get available font sizes (height) from an `AmigaBitmapFontSet()` in pixels.

Usage

```
availableFontSizes(x)
```

Arguments

`x`	An `AmigaBitmapFontSet()` for which the available font sizes (height) in number of pixels need to be obtained.

Details

An `AmigaBitmapFontSet()` can hold bitmaps of multiple font sizes. Use this function to obtain the available size from such a set.

Value

Returns a `vector` of `numeric` values specifying the available font sizes (height in pixels) for `x`.

Author(s)

Pepijn de Vries

See Also

Other AmigaBitmapFont.operations: AmigaBitmapFont, c(), fontName(), font_example, getAmigaBitmapFont(), rasterToAmigaBitmapFont(), rawToAmigaBitmapFontSet(), rawToAmigaBitmapFont(), read.AmigaBitmapFontSet(), read.AmigaBitmapFont(), write.AmigaBitmapFont()

Examples

```
data(font_example)

## The example font holds two font sizes (8 and 9):
availableFontSizes(font_example)
```

---

bitmapToRaster *Convert an Amiga bitmap image into a raster*

Description

Amiga images are usually stored as bitmap images with indexed colours. This function converts raw Amiga bitmap data into raster data (grDevices::as.raster()).

Usage

```
bitmapToRaster(
  x,
  w,
  h,
  depth,
  palette = grDevices::gray(seq(0, 1, length.out = 2^depth)),
  interleaved = TRUE
)
```

Arguments

x	a vector of raw values, representing bitmap data.
w	Width in pixels of the bitmap image. Can be any positive value. However, bitmap data is 'word' aligned on the amiga. This means that the width of the stored bitmap data is a multiple of 16 pixels. The image is cropped to the width specified here.
h	Height in pixels of the bitmap image.
depth	The colour depth of the bitmap image (i.e., the number of bit planes). The image will be composed of 2^depth indexed colours.
palette	A vector of 2^depth colours, to be used for the indexed colours of the bitmap image. By default, a grayscale palette is used. When explicitly set to NULL, this function returns a matrix with palette index values.
interleaved	A logical value, indicating whether the bitmap is interleaved. An interleaved bitmap image stores each consecutive bitmap layer per horizontal scanline.

Details

Bitmap images stored as raw data, representing palette index colours, can be converted into raster data (grDevices::as.raster()). The latter data can easily be plotted in R. It is usually not necessary to call this function directly, as there are several more convenient wrappers for this function. Those wrappers can convert specific file formats (such as IFF ILBM and Hardware Sprites, see as.raster()) into raster objects. This function is provided for completeness sake (or for when you want to search for images in an amiga memory dump).

Value

Returns a raster object (as.raster()) as specified in the grDevices() package. Unless, palette is set to NULL, in which case a matrix with numeric palette index values is returned.

Author(s)

Pepijn de Vries

See Also

Other raster.operations: AmigaBitmapFont, as.raster.AmigaBasicShape(), dither(), index.colours(), rasterToAmigaBasicShape(), rasterToAmigaBitmapFont(), rasterToBitmap(), rasterToHWSprite(), rasterToIFF()

Examples

```
## first load an example image:
example.iff <- read.iff(system.file("ilbm8lores.iff", package = "AmigaFFH"))

## get the raw bitmap data, which is nested in the InterLeaved BitMap (ILBM)
## IFF chunk as the BODY:
bitmap.data <- interpretIFFChunk(getIFFChunk(example.iff, c("ILBM", "BODY")))

## In order to translate the bitmap data into a raster object we need
## to know the image dimensions (width, height and colour depth). This
## information can be obtained from the bitmap header (BMHD):

bitmap.header <- interpretIFFChunk(getIFFChunk(example.iff, c("ILBM", "BMHD")))

## First the bitmap data needs to be unpacked as it was stored in a compresssed
## form in the IFF file (see bitmap.header$Compression):

bitmap.data <- unPackBitmap(bitmap.data)

## It would also be nice to use the correct colour palette. This can be obtained
## from the CMAP chunk in the IFF file:

bitmap.palette <- interpretIFFChunk(getIFFChunk(example.iff, c("ILBM", "CMAP")))

example.raster <- bitmapToRaster(bitmap.data,
                                 bitmap.header$w,
                                 bitmap.header$h,
```

```
                                bitmap.header$nPlanes,
                                bitmap.palette)

## We now have a raster object that can be plotted:

plot(example.raster, interpolate = FALSE)
```

---

c *Combine multiple AmigaFFH objects*

Description

Use this function to correctly combine one or more `AmigaBitmapFont()` class objects into a single `AmigaBitmapFontSet()` class object, or to combine multiple `AmigaBasic()` class objects.

Usage

```
## S3 method for class 'AmigaBasic'
c(...)

## S3 method for class 'AmigaBitmapFont'
c(..., name = "font")
```

Arguments

`...`	Either `AmigaBasic()` or `AmigaBitmapFont()` class objects. In case of `AmigaBitmapFont()` objects: Each `AmigaBitmapFont()` object should have a unique Y-size.
`name`	This argument is only valid when `...` are one or more `AmigaBitmapFont()` class objects. A `character` string specifying the name that needs to be applied to the font set. When unspecified, the default name 'font' is used. Note that this name will also be used as a file name when writing the font to a file. So make sure the name is also a valid file name. This will not be checked for you and may thus result in errors.

Details

In case `...` are one or more `AmigaBasic()` class objects:

`AmigaBasic()` class objects are combined into a single `AmigaBasic()` class object in the same order as they are given as argument to this function. for this purpose the lines of Amiga Basic codes are simply concatenated.

In case `...` are one or more `AmigaBitmapFont()` class objects:

`AmigaBitmapFontSet()` class objects can hold multiple `AmigaBitmapFont()` class objects. Use this method to combine font bitmaps into such a font set. Make sure each bitmap represents a unique font height (in pixels). When heights are duplicated an error will be thrown.

You can also specify a `name` for the font, that will be embeded in the object. As this name will also be used as a file name when writing the font to a file, make sure that it is a valid filename.

Value

Returns an AmigaBitmapFontSet() in which the AmigaBitmapFont() objects are combined. Or when AmigaBasic() objects are combined, an AmigaBasic() object is returned in which the lines of Amiga Basic code are combined.

Author(s)

Pepijn de Vries

See Also

Other AmigaBitmapFont.operations: AmigaBitmapFont, availableFontSizes(), fontName(), font_example, getAmigaBitmapFont(), rasterToAmigaBitmapFont(), rawToAmigaBitmapFontSet(), rawToAmigaBitmapFont(), read.AmigaBitmapFontSet(), read.AmigaBitmapFont(), write.AmigaBitmapFont()

Examples

```
data(font_example)

## first get some AmigaBitmapFont objects:
font8 <- getAmigaBitmapFont(font_example, 8)
font9 <- getAmigaBitmapFont(font_example, 9)

## now bind these bitmaps again in a single set
font.set <- c(font8, font9, name = "my_font_name")

## Amiga Basic codes can also be combined:
bas1 <- as.AmigaBasic("LET a = 1")
bas2 <- as.AmigaBasic("PRINT a")
bas  <- c(bas1, bas2)
```

---

check.names.AmigaBasic

Check Amiga Basic label/variable names for validity

Description

Check Amiga Basic label/variable names for validity

Usage

```
check.names.AmigaBasic(x, ...)
```

Arguments

x	A vector of character strings that need to be checked
...	Currently ignored.

Details

Names for variables and labels should adhere to the following rules in Amiga Basic:

- Length of the names should be in the range of 1 up to 255 character
- Names cannot be `AmigaBasic.reserved()` words
- Names should only contain alphanumeric characters or periods and should not contain special characters (i.e., reserved for type definition, such as dollar- or percentage sign)
- Names should not start with a numeric character

This function tests names against each of these criteria.

Value

A `data.frame` with `logical` values with the same number of rows as the length of `x`. Columns in the data.frame corresponds with the criteria listed in the details. `FALSE` for invalid names.

Author(s)

Pepijn de Vries

See Also

Other AmigaBasic.operations: `AmigaBasic.reserved()`, `AmigaBasicBMAP`, `AmigaBasic`, `[.AmigaBasic()`, `as.AmigaBasicBMAP()`, `as.AmigaBasic()`, `as.character()`, `names.AmigaBasic()`, `rawToAmigaBasicBMAP()`, `rawToAmigaBasic()`, `read.AmigaBasicBMAP()`, `read.AmigaBasic()`, `write.AmigaBasic()`

Examples

```
## These are valid names in Amiga Basic:
check.names.AmigaBasic(c("Foo", "Bar"))

## Reserved words and repeated names are not allowed:

check.names.AmigaBasic(c("Print", "Foo", "Foo"))
```

---

colourToAmigaRaw *Convert colours to Amiga compatible raw data or vice versa*

Description

Convert colours to Amiga compatible raw data or vice versa, such that it can be used in graphical objects from the Commodore Amiga.

Usage

```
colourToAmigaRaw(
  x,
  colour.depth = c("12 bit", "24 bit"),
  n.bytes = c("2", "3")
)

amigaRawToColour(
  x,
  colour.depth = c("12 bit", "24 bit"),
  n.bytes = c("2", "3")
)
```

Arguments

x	In the case amigaRawToColour is called, x should be a vector of raw data. The length of this vector should be a multiple of 2 (when n.bytes = "2") or 3 (when n.bytes = "3"). When colourToAmigaRaw is called, x should be a character strings representing a colour.
colour.depth	A character string: "12 bit" (default) or "24 bit". The first should be used in most cases, as old Amigas have a 12 bit colour depth.
n.bytes	A character string: "2" or "3". The number of bytes that is used or should be used to store each colour.

Details

On the original Commodore Amiga chipset, graphics used indexed palettes of 12 bit colours. Colours are specified by their RGB (Red, Green and Blue) values, each component requiring 4 bits (with corresponding values ranging from 0 up to 15). Data structures on the Amiga were WORD (2 bytes) aligned. Colours are therefore typically stored in either 2 bytes (skipping the first four bits) or 3 bytes (one byte for each value).

These functions can be used to convert R colours into the closest matching Amiga colour in a raw format, or vice versa. Note that later Amiga models with the advanced (graphics) architecture (known as AA or AGA) allowed for 24 bit colours.

Value

In the case amigaRawToColour is called, a (vector of) colour character string(s) is returned. When colourToAmigaRaw is called, raw representing the colour(s) specified in x is returned.

Author(s)

Pepijn de Vries

See Also

Other raw.operations: as.AmigaBasic(), as.raw.AmigaBasic(), packBitmap(), rawToAmigaBasicBMAP(), rawToAmigaBasicShape(), rawToAmigaBasic(), rawToAmigaBitmapFontSet(), rawToAmigaBitmapFont(), rawToAmigaIcon(), rawToHWSprite(), rawToIFFChunk(), rawToSysConfig(), simpleAmigaIcon()

Examples

```
## Let's create some Amiga palettes:
colourToAmigaRaw(c("red", "navy blue", "brown", "#34AC5A"))

## let's do the reverse.
## this is white:
amigaRawToColour(as.raw(c(0x0f, 0xff)))

## this is white specified in 3 bytes:
amigaRawToColour(as.raw(c(0xf0, 0xf0, 0xf0)), n.bytes = "3")

## lower nybbles are ignored, you will get a warning when it is not zero:
# amigaRawToColour(as.raw(c(0xf0, 0xf0, 0x0f)), n.bytes = "3")
```

`deltaFibonacciCompress` *(De)compress 8-bit continuous signals.*

Description

Use a lossy delta-Fibonacci (de)compression to continuous 8-bit signals. This algorithm was used to compress 8-bit audio wave data on the Amiga.

Usage

```
deltaFibonacciCompress(x, ...)

deltaFibonacciDecompress(x, ...)
```

Arguments

`x`	A `vector` of `raw` data that needs to be (de)compressed.
`...`	Currently ignored.

Details

This form of compression is lossy, meaning that information and quality will get lost. 8-bit audio is normally stored as an 8-bit signed value representing the amplitude at specific time intervals. The delta-Fibonacci compression instead stores the difference between two time intervals (delta) as a 4-bit index. This index in turn represents a value from the Fibonacci series (hence the algorithm name). The compression stores small delta values accurately, but large delta values less accurately. As each sample is stored as a 4-bit value instead of an 8-bit value, the amount of data is reduced with almost 50\

The algorithm was first described by Steve Hayes and was used in 8SVX audio stored in the Interchange File Format (IFF). The quality loss is considerable (especially when the audio contained many large deltas) and was even in the time it was developed (1985) not used much. The function is provided here for the sake of completeness. The implementation here only compresses 8-bit data, as for 16-bit data the quality loss will be more considerable.

Value

Returns a `vector` of the resulting (de)compressed `raw` data.

Author(s)

Pepijn de Vries

References

https://en.wikipedia.org/wiki/Delta_encoding

http://amigadev.elowar.com/read/ADCD_2.1/Devices_Manual_guide/node02D6.html

Examples

```
## Let's get an audio wave from the ProTrackR package, which we
## can use in this example:
buzz     <- ProTrackR::PTSample(ProTrackR::mod.intro, 1)

## Let's convert it into raw data, such that we can compress it:
buzz.raw <- as.integer(ProTrackR::waveform(buzz) - 128) |>
  bitwAnd(0xFF) |>
  as.raw()

## Let's compress it:
buzz.compress <- deltaFibonacciCompress(buzz.raw)

## Look the new data uses less memory:
length(buzz.compress)/length(buzz.raw)

## The compression was lossy, which we can examine by decompressing the
## sample again:
buzz.decompress <- deltaFibonacciDecompress(buzz.compress)

## And turn the raw data into numeric data:
buzz.decompress <-
  ifelse(buzz.decompress > 0x7f, as.integer(buzz.decompress) - 256L,
         as.integer(buzz.decompress))

## Plot the original wave in black, the decompressed wave in blue
## and the error in red (difference between the original and decompressed
## wave). The error is actually very small here.
plot(ProTrackR::waveform(buzz) - 128, type = "l")
lines(buzz.decompress, col = "blue")
buzz.error <- ProTrackR::waveform(buzz) - 128 - buzz.decompress
lines(buzz.error, col = "red")

## this can also be visualised by plotting the orignal wave data against
## the decompressed data (and observe a very good correlation):
plot(ProTrackR::waveform(buzz) - 128, buzz.decompress)

## Let's do the same with a sample of a snare drum, which has larger
```

```
## delta values:
snare.drum <- ProTrackR::PTSample(ProTrackR::mod.intro, 2)

## Let's convert it into raw data, such that we can compress it:
snare.raw <- as.integer(ProTrackR::waveform(snare.drum) - 128L) |>
  bitwAnd(0xFF) |>
  as.raw()

## Let's compress it:
snare.compress <- deltaFibonacciCompress(snare.raw)

## Decompress the sample:
snare.decompress <- deltaFibonacciDecompress(snare.compress)

## And turn the raw data into numeric data:
snare.decompress <-
  ifelse(snare.decompress > 0x7f, as.integer(snare.decompress) - 256L,
         as.integer(snare.decompress))

## Now if we make the same comparison as before, we note that the
## error in the decompressed wave is much larger than in the previous
## case (red line):
plot(ProTrackR::waveform(snare.drum) - 128, type = "l")
lines(snare.decompress, col = "blue")
snare.error <- ProTrackR::waveform(snare.drum) - 128 - snare.decompress
lines(snare.error, col = "red")

## this can also be visualised by plotting the orignal wave data against
## the decompressed data (and observe a nice but not perfect correlation):
plot(ProTrackR::waveform(snare.drum) - 128, snare.decompress)
```

`dither` *Image dithering*

Description

Dither is an intentional form of noise applied to an image to avoid colour banding when reducing the amount of colours in that image. This function applies dithering to a grDevices `raster` image.

Usage

```
dither(x, method, ...)

## S3 method for class 'raster'
dither(
  x,
 method = c("none", "floyd-steinberg", "JJN", "stucki", "atkinson", "burkse", "sierra",
    "two-row-sierra", "sierra-lite"),
  palette,
```

```
  mode = c("none", "HAM6", "HAM8"),
  ...
)

## S3 method for class 'matrix'
dither(
  x,
 method = c("none", "floyd-steinberg", "JJN", "stucki", "atkinson", "burkse", "sierra",
    "two-row-sierra", "sierra-lite"),
  palette,
  mode = c("none", "HAM6", "HAM8"),
  ...
)
```

Arguments

x	Original image data that needs to be dithered. Should be a raster object (grDevices::as.raster()), or a matrix of character string representing colours.
method	A character string indicating which dithering method should be applied. See usage section for all possible options (Note that the "JJN" is the Jarvis, Judice, and Ninke algorithm). Default is "none", meaning that no dithering is applied.
...	Currently ignored.
palette	A palette to which the image should be dithered. It should be a vector of character strings representing colours.
mode	A character string indicating whether a special Amiga display mode should be used when dithering. By default 'none' is used (no special mode). In addition, 'HAM6' and 'HAM8' are supported. See rasterToBitmap() for more details.

Details

The approaches implemented here all use error diffusion to achieve dithering. Each pixel is scanned (from top to bottom, from left to right), where the actual colour is sampled and compared with the closest matching colour in the palette. The error (the differences between the actual and used colour) is distributed over the surrounding pixels. The only difference between the methods implemented here is the way the error is distributed. The algorithm itself is identical. For more details consult the listed references.

Which method results in the best quality image will depend on the original image and the palette colours used for dithering, but is also a matter of taste. Note that the dithering algorithm is relatively slow and is provided in this package for your convenience. As it is not in the main scope of this package you should use dedicated software for faster/better results.

Value

Returns a matrix with the same dimensions as x containing numeric index values. The corresponding palette is returned as attribute, as well as the index value for the fully transparent colour in the palette.

Author(s)

Pepijn de Vries

References

R.W. Floyd, L. Steinberg, *An adaptive algorithm for spatial grey scale*. Proceedings of the Society of Information Display 17, 75-77 (1976).

J. F. Jarvis, C. N. Judice, and W. H. Ninke, *A survey of techniques for the display of continuous tone pictures on bilevel displays*. Computer Graphics and Image Processing, 5:1:13-40 (1976).

https://en.wikipedia.org/wiki/Floyd-Steinberg_dithering

https://tannerhelland.com/4660/dithering-eleven-algorithms-source-code/

See Also

Other colour.quantisation.operations: `index.colours()`

Other raster.operations: `AmigaBitmapFont`, `as.raster.AmigaBasicShape()`, `bitmapToRaster()`, `index.colours()`, `rasterToAmigaBasicShape()`, `rasterToAmigaBitmapFont()`, `rasterToBitmap()`, `rasterToHWSprite()`, `rasterToIFF()`

Examples

```
## first: Let's make a raster out of the 'volcano' data, which we can use in the example:
volcano.raster <- as.raster(t(matrix(terrain.colors(1 + diff(range(volcano)))[volcano -
  min(volcano) + 1], nrow(volcano))))

## let's dither the image, using a predefined two colour palette:
volcano.dither <- dither(volcano.raster,
                         method = "floyd-steinberg",
                         palette = c("yellow", "green"))

## Convert the indices back into a raster object, such that we can plot it:
volcano.dither <- as.raster(apply(volcano.dither, 2, function(x) c("yellow", "green")[x]))
par(mfcol = c(1, 2))
plot(volcano.raster, interpolate = FALSE)
plot(volcano.dither, interpolate = FALSE)

## results will get better when a better matching colour palette is used.
## for that purpose use the function 'index.colours'.
```

`fontName` *Extract or replace a font name*

Description

Extract or replace a font name from an `AmigaBitmapFontSet()` object.

Usage

```
fontName(x)

fontName(x) <- value
```

Arguments

x	An AmigaBitmapFontSet() for which the font name needs to be changed.
value	A character string specifying the name you wish to use for the font.

Details

The name of a font is embeded at multiple locations of an AmigaBitmapFontSet() object. This function can be used to extract or replace the font name correctly. This is also the name that will be used when writing the font to a file with write.AmigaBitmapFontSet().

Value

Returns the font name. In case of the replace function, a copy of x is returned with the name replaced by 'value'.

Author(s)

Pepijn de Vries

See Also

Other AmigaBitmapFont.operations: AmigaBitmapFont, availableFontSizes(), c(), font_example, getAmigaBitmapFont(), rasterToAmigaBitmapFont(), rawToAmigaBitmapFontSet(), rawToAmigaBitmapFont(), read.AmigaBitmapFontSet(), read.AmigaBitmapFont(), write.AmigaBitmapFont()

Examples

```
data(font_example)

## show the name of the example font:
fontName(font_example)

## This is how you change the name into "foo"
fontName(font_example) <- "foo"

## see it worked:
fontName(font_example)
```

font_example *An example object for the AmigaBitmapFontSet class*

Description

An example object for the AmigaBitmapFontSet() class used in examples throughout this package. It also contains a nested AmigaBitmapFont() class objects, which can be obtain by using getAmigaBitmapFont(font_example, 9).

Format

font_example is an AmigaBitmapFontSet() object. For details see the object class documentation.

Details

The font_example contains a font that was designed as an example for this package. It holds bitmap glyphs for 8 and 9 pixels tall characters.

See Also

Other AmigaBitmapFont.operations: AmigaBitmapFont, availableFontSizes(), c(), fontName(), getAmigaBitmapFont(), rasterToAmigaBitmapFont(), rawToAmigaBitmapFontSet(), rawToAmigaBitmapFont(), read.AmigaBitmapFontSet(), read.AmigaBitmapFont(), write.AmigaBitmapFont()

Examples

```
data("font_example")
```

getAmigaBitmapFont *Extract a specific AmigaBitmapFont from a AmigaBitmapFontSet*

Description

Extract a specific AmigaBitmapFont() from a AmigaBitmapFontSet().

Usage

```
getAmigaBitmapFont(x, size)
```

Arguments

x	An AmigaBitmapFontSet() object, from which the specific AmigaBitmapFont() object needs to be extracted.
size	A single numeric value specifying the desired font size in pixels. Use availableFontSizes() to get available sizes.

Details

An AmigaBitmapFontSet() object can hold one or more bitmaps for specific font sizes (heights). Use this function to obtain such a specific AmigaBitmapFont().

Value

Returns an AmigaBitmapFont() of the requested size. An error is thrown when the requested size is not available.

Author(s)

Pepijn de Vries

See Also

Other AmigaBitmapFont.operations: AmigaBitmapFont, availableFontSizes(), c(), fontName(), font_example, rasterToAmigaBitmapFont(), rawToAmigaBitmapFontSet(), rawToAmigaBitmapFont(), read.AmigaBitmapFontSet(), read.AmigaBitmapFont(), write.AmigaBitmapFont()

Examples

```
data(font_example)

## get the font object for the first available size:
font <- getAmigaBitmapFont(font_example,
                           availableFontSizes(font_example)[1])
```

---

getIFFChunk *Get a specific IFFChunk nested inside other IFFChunks*

Description

IFFChunk()s can be nested in a tree-like structure. Use this method to get a specific chunk with a specific label.

Usage

```
## S4 method for signature 'IFFChunk,character,integer'
getIFFChunk(x, chunk.path, chunk.number)

## S4 method for signature 'IFFChunk,character,missing'
getIFFChunk(x, chunk.path, chunk.number)

## S4 replacement method for signature 'IFFChunk,character,missing,IFFChunk'
getIFFChunk(x, chunk.path, chunk.number = NULL) <- value

## S4 replacement method for signature 'IFFChunk,character,integer,IFFChunk'
getIFFChunk(x, chunk.path, chunk.number = NULL) <- value
```

Arguments

x	An IFFChunk() object from which the nested IFFChunk() should be extracted an returned.
chunk.path	A vector of 4 character long strings of IFF chunk labels, specifying the path of the target IFF chunk. For example: c("ILBM", "BODY") means, get the "BODY" chunk from inside the "ILBM" chunk.
chunk.number	A vector of the same length as chunk.path, with integer index numbers. Sometimes a chunk can contain a list of chunks with the same label. With this argument you can specify which element should be returned. By default (when missing), the first element is always returned.
value	An IFFChunk() with which the target chunk should be replaced. Make sure that value is of the same chunk.type as the last chunk specified in the chunk.path.

Details

IFFChunk objects have 4 character identifiers, indicating what type of chunk you are dealing with. These chunks can be nested inside of each other. Use this method to extract specific chunks by referring to there respective identifiers. The identifiers are shown when calling print on an IFFChunk(). If a specified path doesn't exist, this method throws a 'subscript out of range' error.

Value

Returns an IFFChunk() object nested inside x at the specified path. Or in case of the replace method the original chunk x is returned with the target chunk replaced by value.

Author(s)

Pepijn de Vries

See Also

Other iff.operations: IFFChunk-class, WaveToIFF(), as.raster.AmigaBasicShape(), interpretIFFChunk(), rasterToIFF(), rawToIFFChunk(), read.iff(), write.iff()

Examples

```
## load an IFF file
example.iff <- read.iff(system.file("ilbm8lores.iff", package = "AmigaFFH"))

## Get the BMHD (bitmap header) from the ILBM (interleaved bitmap) chunk:
bmhd <- getIFFChunk(example.iff, c("ILBM", "BMHD"))

## This is essentially doing the same thing, but we now explicitly
## tell the method to get the first element for each specified label:
bmhd <- getIFFChunk(example.iff, c("ILBM", "BMHD"), c(1L, 1L))

## Let's modify the bitmap header and replace it in the parent IFF chunk.
bmhd.itpt <- interpretIFFChunk(bmhd)
```

```
## Let's disable the masking, the bitmap will no longer be transparent:
bmhd.itpt$Masking <- "mskNone"
bmhd <- IFFChunk(bmhd.itpt)

## Now replace the header from the original iff with the modified header:
getIFFChunk(example.iff, c("ILBM", "BMHD")) <- bmhd
```

`hardwareSprite-class` *The hardwareSprite class*

Description

An S4 class object that represent graphical objects known as hardware sprites on the Commodore Amiga.

Details

Amiga hardware supported sprites, which are graphical objects that could be moved around the display and independently from each other. Basic sprites were 16 pixels wide and any number of pixels high and were composed of four colours, of which one is transparent.

More complex sprites could be formed by linking separate sprites together. That way, sprites could become wider, or be composed of more colours. Such extended sprites are currently not supported by this package.

A well known example of hardware sprite on the Commodore Amiga is the mouse pointer.

This object simply holds the basic information belonging to hardware sprite. Use `as.raster()` to convert it to a raster which is a more useful graphical element in R.

Slots

`VStart` The vertical starting position of a sprite.

`HStart` The horizontal starting position of a sprite.

`VStop` The vertical stopping position of a sprite. The height of a sprite should be given by `VStart - VStop`.

`control.bits` 8 `logical` values used for extending sprites. The values are stored in this objects but extending sprites is not (yet) supported.

`bitmap` Interleaved bitmap data containing information on the pixel colour numbers of the sprite.

`colours` A vector of the 3 colours used for the sprite.

`end.of.data` Sprite data can be followed by another sprite. It is terminated with two WORDS equalling zero (`raw(4)`). Repeated sprite data is currently not supported.

Author(s)

Pepijn de Vries

References

http://amigadev.elowar.com/read/ADCD_2.1/Hardware_Manual_guide/node00AE.html

Examples

```
## This generates a sprite of a single line (16x1 pixels) with an empty bitmap:
new("hardwareSprite")

## This generates a sprite of a single line (16x1 pixels) where
## the bitmap contains some coloured pixels:
new("hardwareSprite", bitmap = as.raw(c(0x01,0x02,0x03,0x04)))

## This generates a sprite of 16x16 pixels:
new("hardwareSprite",
    VStop = 16,
    bitmap = as.raw(sample.int(255, 64, replace = TRUE)))
```

IFFChunk-class *A class structure to represent IFF files*

Description

An S4 class structure to represent data stored in the Interchange File Format (IFF).

Details

The Interchange File Format (IFF) was introduced in 1985 by Electronic Arts. This format stores files in standardised modular objects, called 'chunks'. At the start of each chunk it is specified what type of data can be expected and what the size of this data is. This was a very forward thinking way of storing data, similar structures are still used in modern file formats (such as PNG images and XML files).

Although the IFF format is still in use, and new standardised chunk types can still be registered, this package will focus on the older chunk types that were primarily used on the Commodore Amiga (OS <= 3.0). IFF files could contain any kind of information. It could contain bitmap images, but also audio clips or (formatted) texts.

The `IFFChunk` class is designed such that it theoretically can hold any type of IFF data. This package will mostly focus on the early IFF file types (i.e., IFF chunks as originally registered by Electronic Arts). IFF files are read by this package in a none lossy way (`read.iff()`), such that all information is preserved (even if it is of an unknown type, as long as the chunk identifier is 4 characters long).

This means that the object needs to be interpreted in order to make sense out of it (`interpretIFFChunk()`). This interpretation returns simplified interpretations of class `IFF.ANY` when it is supported (see `IFFChunk-method()` for supported chunk types). Note that in the interpretation process (meta-)information may get lost. converting `IFF.ANY` objects back into `IFFChunk()` objects (if possible) could therefore result in an object that is different from then one stored in the original file and could even destroy the correct interpretation of IFF objects. IFF files should thus be handled with care.

Slots

chunk.type A four character long code reflecting the type of information represented by this chunk.

chunk.data A list that holds either one or more valid IFFChunks or a single vector of raw data. This data can only be interpreted in context of the specified type or in some cases information from other IFFChunks.

Author(s)

Pepijn de Vries

References

https://wiki.amigaos.net/wiki/IFF_Standard

https://wiki.amigaos.net/wiki/IFF_FORM_and_Chunk_Registry

https://en.wikipedia.org/wiki/Interchange_File_Format

See Also

Other iff.operations: WaveToIFF(), as.raster.AmigaBasicShape(), getIFFChunk(), interpretIFFChunk(), rasterToIFF(), rawToIFFChunk(), read.iff(), write.iff()

Examples

```
## load an IFF file
example.iff <- read.iff(system.file("ilbm8lores.iff", package = "AmigaFFH"))

## 'example.iff' is of class IFFChunk:
class(example.iff)

## let's plot it:
plot(example.iff)

## The default constructor will create an empty FORM:
new("IFFChunk")

## The constructor can also be used to create simple chunks:
new("IFFChunk",
    chunk.type = "TEXT",
    chunk.data = list(charToRaw("A simple chunk")))
```

IFFChunk-method *Coerce to and create IFFChunk objects*

Description

Convert IFF.ANY objects (created with interpretIFFChunk()) into IFFChunk() objects. A basic IFFChunk() can also be created with this method by providing the chunk type name.

Usage

```
IFFChunk(x, ...)

## S3 method for class 'character'
IFFChunk(x, ...)

## S3 method for class 'IFF.FORM'
IFFChunk(x, ...)

## S3 method for class 'IFF.BODY'
IFFChunk(x, ...)

## S3 method for class 'IFF.ANNO'
IFFChunk(x, ...)

## S3 method for class 'IFF.AUTH'
IFFChunk(x, ...)

## S3 method for class 'IFF.CHRS'
IFFChunk(x, ...)

## S3 method for class 'IFF.NAME'
IFFChunk(x, ...)

## S3 method for class 'IFF.TEXT'
IFFChunk(x, ...)

## S3 method for class 'IFF.copyright'
IFFChunk(x, ...)

## S3 method for class 'IFF.CHAN'
IFFChunk(x, ...)

## S3 method for class 'IFF.VHDR'
IFFChunk(x, ...)

## S3 method for class 'IFF.8SVX'
IFFChunk(x, ...)

## S3 method for class 'IFF.ILBM'
IFFChunk(x, ...)

## S3 method for class 'IFF.CMAP'
IFFChunk(x, ...)

## S3 method for class 'IFF.BMHD'
IFFChunk(x, ...)
```

```
## S3 method for class 'IFF.CAMG'
IFFChunk(x, ...)

## S3 method for class 'IFF.CRNG'
IFFChunk(x, ...)

## S3 method for class 'IFF.ANIM'
IFFChunk(x, ...)

## S3 method for class 'IFF.ANHD'
IFFChunk(x, ...)

## S3 method for class 'IFF.DLTA'
IFFChunk(x, ...)

## S3 method for class 'IFF.DPAN'
IFFChunk(x, ...)
```

Arguments

x	An S3 class IFF.ANY object that needs to be coerced into an IFFChunk-class() object. IFF.ANY objects are created with the interpretIFFChunk() method. x can also be a character string of a IFF chunk type (e.g., "FORM" or "BMHD"). In that case an IFFChunk() object of that type is created with some basic content.
...	Arguments passed onto methods underlying the interpretation of the specific IFF chunks. Allowed arguments depend on the specific type of IFF chunk that x represents.

Details

IFF data is stored in a IFFChunk-class() object when read from an IFF file (read.iff()). These objects reflect the file structure well, but the data is stored as raw information. IFF files can contain a wide variety of information types, ranging from bitmap images to audio clips. The raw information stored in IFFChunk() objects can be interpreted into more meaningful representations that can be handled in R. This is achieved with the interpretIFFChunk() method, which returns IFF.ANY objects.

These IFF.ANY objects are a less strict representation of the IFF Chunk, but are easier to handle in R. The interpretation method is lossy and may not preserve all information in the IFF.ANY object. The IFFChunk-method() can coerce IFF.ANY back to the more strictly defined IFFChunk-class() objects. Be careful with conversions between IFFChunk-class() and IFF.ANY objects and vice versa, as information may get lost.

More detailed information about IFF chunks can be found in the IFF chunk registry (see references).

- IFF.FORM represents a FORM chunk, which is a container that can hold any kind of chunk. When interpreted, it is represented as a list, where each element is an interpreted chunk nested inside the FORM.
- IFF.BODY represents the actual data in an IFF file. However, without context this chunk cannot be interpreted and is therefore interpreted as a vector of raw data.

- IFF.ANIM represents an animation (ANIM) chunk. When interpreted, it will return a list where each element is an animation frame represented as an IFF.ILBM object. Each animation frame should be nested inside an ILBM chunk nested inside a FORM chunk, nested inside an ANIM chunk.
  - IFF.ANHD represents an ANimation HeaDer (ANHD) chunk. When interpreted, it returns a named list containing the following information:
    - operation is a character string indicating how the bitmap data for the animation frame is encoded. Can be one of the following: "standard", "XOR", "LongDeltaMode", "ShortDeltaMode", "GeneralDeltamode", "ByteVerticalCompression", "StereoOp5", or "ShortLongVerticalDeltaMode". Currently, only the ByteVerticalCompression is implemented in this package.
    - mask is a vector of 8 logical values. It is currently ignored.
    - w and h are positive numeric values, specifying the width and height of the frame (should be identical for all frames).
    - x and y are numeric values, specifying the plotting position for the frame.
    - abstime is a positive numeric value - currently unused - used for timing the frame relative to the time the first frame was displayed. In jiffies (1/60 sec).
    - reltime is a positive numeric value for timing the frame relative to time previous frame was displayed. In jiffies (1/60 sec).
    - interleave is currently unused. It should be set to 0.
    - pad0 is a padding byte (raw) for future use.
    - flags is a vector of 32 logical values. They contain information on how the bitmap data is stored.
    - pad1 are 16 padding bytes (raw) for future use.
  - IFF.DPAN represents an DPaint ANimation (DPAN) chunk. Some software will require this chunk to correctly derive the total number of frames in the animation. When interpreted, it will return a named list with the following elements:
    - version a numeric version number.
    - nframes a positive numeric value, indicating the number of frames in the animation.
    - flags a vector of 32 logical values. Ignored in this package as it was intended for future implementations.
  - IFF.DLTA represents a delta mode data chunk (DLTA). The first animation frame is stored as a normal InterLeaved BitMap (ILBM) image as described below. The following frames only store differences in bitmap data compared to the previous frames but is not interleaved. They are thus incorrectly embedded in an ILBM chunk (but is kept so for backward compatibility). When interpreted, a grDevices raster object is returned only showing the differences. It is not very meaningful to interpret these chunks on their own, but rather the entire parent ANIM chunk.
- IFF.ILBM represents InterLeaved BitMap (ILBM) chunks. It is interpreted here as a raster image (see grDevices::as.raster()). ILBM chunks are usually nested inside a FORM container.
  - IFF.BMHD represents the header chunk of a bitmap (BMHD), and should always be present (nested inside) an ILBM chunk. It is interpreted as a named list containing the following elements:

    * w and h are positive numeric values specifying the bitmap width and height in pixels. Note that the width can be any positive whole number, whereas the bitmap data always has a width divisible by 16.
    * x and y are numeric values specifying the plotting position relative to the top left position of the screen. Although required in the bitmap header. It is ignored in the interpretation of bitmap images.
    * nPlanes is a positive value indicating the number of bitplanes in the image. The number of colours in an image can be calculated as 2^nPlanes.
    * Masking indicates whether there are bitplanes that should be masked (i.e. are treated as transparent). It is a character string equalling any of the following: "mskNone", "mskHasMask", "mskHasTransparentColour", "mskLasso" or "mskUnknown". Only the first (no transparency) and third (one of the colours should be treated as transparent) id is currently interpreted correctly. The others are ignored. "mskUnknown" means that an undocumented mask is applied to the image.
    * Compression indicates whether the bitmap data is compressed. It is a character string that can equal any of the following: "cmpNone", "cmpByteRun1" or "cmpUnknown". The latter means an undocumented form of compression is applied and is currently ignored. In most cases bitmap data is compressed with the cmpByteRun1 algorithm (packBitmap()). In some cases, bitmap data is not compressed (cmpNone).
    * pad is a raw byte that is only used to align data. It is ignored in the interpretation.
    * transparentColour is a numeric value that indicates which colour number in the palette should be treated as fully transparent (when Masking equals "mskHasTransparentColour").
    * xAspect and yAspect or positive numeric values that indicate the aspect ratio of the pixels in the image. Amiga screen modes allowed for some extreme pixel aspect ratios. These values are used to stretch the image to their intended display mode.
    * pageWidth and pageHeight are positive numeric values indicating the size of the screen in which the image should be displayed. They are ignored in the interpretation of the image.

- IFF.CMAP represents the colour map (CMAP) or palette of a bitmap image. Although common, the chunk is optional and can be omitted from the parent ILBM chunk. It is interpreted as a vector of colours (i.e., a character string formatted as '#RRGGBB' or named colours such as 'blue').
- IFF.CAMG represents a chunk with information with respect to the display mode in which the bitmap image should be displayed. This information can be used to determine the correct pixel aspect ratio, or is sometimes required to correctly interpret the bitmap information. The IFF.CAMG chunk is interpreted as a named list containing the following elements:
    * monitor: a factor indicating the hardware monitor on which the image was created and should be displayed (see amiga_monitors()).
    * display.mode: a factor indicating the display mode in which the image should be displayed (see amiga_display_modes()).
- IFF.CRNG is an optional chunk nested in an ILBM chunk. It represents a 'colour range' and is used to cycle through colours in the bitmap's palette in order to achieve animation effects. It is interpreted as a named list with the following elements. This chunk is currently not used with the interpretation of ILBM images.
    * padding are two raw padding bytes and are ignored when interpreted.

    * rate is a numeric value specifying the rate at which the colours are cycled. The rate is in steps per second.
    * flags is a flag that indicates how colours should be cycled. It is a character string that can equal any of the following: "RNG_OFF", "RNG_ACTIVE", "RNG_REVERSE" or "RNG_UNKNOWN". When equal to the first, colours are not cycled. When equal to the second, colours are cycled. When equal to the third, colours are cycled in reverse direction. When equal to the latter, an undocumented form of cycling is applied.
    * low and high are numeric indices of colours between which should be cycled. Only colour from index low up to index high are affected.

- IFF.8SVX represents 8-bit sampled voice chunks (8SVX). The original Amiga supported 8-bit audio which could be stored using the IFF. 8SVX chunks can contain separate audio samples for each octave. 8SVX chunks are usually stored inside a FORM container. Its body chunk contains 8-bit PCM wave data that could be compressed. When the 8SVX chunk is interpreted with this package, a list is returned where each element represents an octave given as a tuneR::Wave() object. Possible chunks nested in 8SVX chunks and currently supported by this package are as follows.
  - IFF.VHDR represents voice header chunks (VHDR). It contains (meta-)information about the audio stored in the body of the parent 8SVX chunk. When interpreted, a named list is returned with the following elements:
    * oneShotHiSamples is a numeric value indicating how many samples there are in the audio wave of the first octave in the file, that should not be looped (repeated).
    * repeatHiSamples is a numeric value indicating how many samples there are in the audio wave of the first octave in the file, that should be looped (repeated).
    * samplesPerHiCycle is a numeric value specifying the number of samples per repeat cycle in the first octave, or 0 when unknown. The number of repeatHiSamples should be an exact multiple of samplesPerHiCycle.
    * samplesPerSec is a numeric value specifying the data sampling rate.
    * ctOctave a positive whole numeric value indicating how many octaves are included. In 8SVX files the audio wave is resampled for each octave. The wave data in the body starts with the audio sample in the highest octave (least number of samples). The data is then followed by each subsequent octave, where the number of samples increase by a factor of 2 for each octave.
    * sCompression is a character string indicating whether and how the wave data in the body is compressed. It can have one of the following values: "sCmpNone" (no compression), "sCmpFibDelta" (deltaFibonacciCompress()ion is applied), "sCmpUnknown" (an undocumented and unknown form of compression is applied).
    * volume is a numeric value between 0 (minimum) and 0x10000 (maximum) playback volume.
  - IFF.CHAN represents the channel chunk (CHAN). When interpreted it returns a named list with 1 named element: "channel". It's value can be one of the following character strings "LEFT", "RIGHT" or "STEREO". This indicates for how many (one or two) audio channels data is available in the body of the parent 8SVX chunk. It also indicates two which channels the audio should be played back.

- IFF.ANNO, IFF.AUTH, IFF.CHRS, IFF.NAME, IFF.TEXT and IFF.copyright are all unformatted text chunks that can be included optionally in any of the chunk types. Respectively, they represent an annotation, the author's name, a generic character string, the name of the work, generic unformatted text, and copyright text. They are interpreted as a character string.

Value

Returns an IFFChunk-class() representation of x.

References

https://wiki.amigaos.net/wiki/IFF_FORM_and_Chunk_Registry

Examples

```
## load an IFF file
example.iff <- read.iff(system.file("ilbm8lores.iff", package = "AmigaFFH"))

## interpret the IFF file (in some cases information
## will get lost in this step):
example.itpt <- interpretIFFChunk(example.iff)

## now coerce back to a formal IFFChunk class object.
## Only information in the interpreted object is used
## The coerced object may therefore depart from the
## original read from the file.
example.coerce <- IFFChunk(example.itpt)

## and indeed the objects are not identical, as shown below.
## In this case the difference is not disastrous, the order
## of the colours in the palette have shifted. But be careful
## with switching between formal IFFChunk objects and
## interpreted IFF.ANY objects.
identical(example.iff, example.coerce)

## It is also possible to create simple IFFChunk objects
## by providing the desired chunk type identifier as a
## character string.

## This creates a basic bitmap header:
bmhd <- IFFChunk("BMHD")

## This creates a basic colour palette:
cmap <- IFFChunk("CMAP")
```

---

ilbm8lores.iff *An example file of a bitmap image stored in the Interchange File Format*

Description

This file is provided to demonstrate the structure of an Interchange File Format and is used in several examples throughout this package.

Format

See IFFChunk-class() and references for more information about the Interchange File Format.

Details

The Interchange File Format stores information compartmentally in separate containers called 'chunks'. This file demonstrates how a bitmap image is stored in this format. In addition to the raw bitmap data, the file also contains meta-information on the bitmap dimensions, its colour palette and the display mode that should be used on an Amiga. See also interpretIFFChunk(), IFFChunk-class() and the example for bitmapToRaster().

References

https://en.wikipedia.org/wiki/Interchange_File_Format

https://wiki.amigaos.net/wiki/A_Quick_Introduction_to_IFF

Examples

```
filename <- system.file("ilbm8lores.iff", package = "AmigaFFH")
example.iff <- read.iff(filename)

## show the structure of the IFF file:
print(example.iff)
```

---

index.colours *Quantisation of colours and indexing a grDevices raster image*

Description

Converts an image represented by a grDevices `raster` object into a matrix containing numeric indices of a quantised colour palette.

Usage

```
index.colours(
  x,
  length.out = 8,
  palette = NULL,
  background = "#FFFFFF",
 dither = c("none", "floyd-steinberg", "JJN", "stucki", "atkinson", "burkse", "sierra",
    "two-row-sierra", "sierra-lite"),
  colour.depth = c("12 bit", "24 bit"),
  ...
)
```

Arguments

x	A raster object (grDevices::as.raster()), or a matrix containing character strings representing colours. x can also be a list of such matrices or rasters. All elements of this list should have identical dimensions. An overall palette will be generated for elements in the list.
length.out	A numeric value indicating the number of desired colours in the indexed palette. It can also be a character string indicating which special Amiga display mode should be used when indexing colours. 'HAM6' and 'HAM8' are supported. See rasterToBitmap() for more details on these special modes.
palette	A vector of no more than length.out colours, to be used for the bitmap image. When missing or set to NULL, a palette will be generated based on the provided colours in raster x. In that case, stats::kmeans() is used on the hue, saturation, brightness and alpha values of the colours in x for clustering the colours. The cluster centres will be used as palette colours.
background	On the Amiga, indexed images could not be semi-transparent. Only a single colour could be designated as being fully transparent. The "background'' argument should contain a background colour with which semi-transparent colours should be mixed, before colour quantisation. It is white by default.
dither	Dither the output image using the algorithm specified here. See the usage section for possible options. By default no dithering ("none") is applied. See dither() for more details.
colour.depth	A character string indicating the colour depth to be used. Can be either "12 bit" (default, standard on an Amiga with original chipset), or "24 bit". This argument is overruled when length.out is set to "HAM6" or "HAM8". In that case the colour depth linked to that special mode is used (12 bit for HAM6, 24 bit for HAM8).
...	Arguments that are passed onto stats::kmeans() (see palette argument).

Details

Determines the optimal limited palette by clustering colours in an image with stats::kmeans(). The result of the optimisation routine will depend on the randomly chosen cluster centres by this algorithm. This means that the result may slightly differ for each call to this function. If you want reproducible results, you may want to reset the random seed (set.seed()) before each call to this function.

Value

Returns a matrix with the same dimensions as x containing numeric index values. The corresponding palette is returned as attribute, as well as the index value for the fully transparent colour in the palette. When x is a list a list of matrices is returned.

Author(s)

Pepijn de Vries

See Also

Other colour.quantisation.operations: dither()

Other raster.operations: AmigaBitmapFont, as.raster.AmigaBasicShape(), bitmapToRaster(), dither(), rasterToAmigaBasicShape(), rasterToAmigaBitmapFont(), rasterToBitmap(), rasterToHWSprite(), rasterToIFF()

Examples

```
## first: Let's make a raster out of the 'volcano' data, which we can use in the example:
volcano.raster <- as.raster(t(matrix(terrain.colors(1 + diff(range(volcano)))[volcano -
  min(volcano) + 1], nrow(volcano))))

## This will create an image of the original raster using an indexed palette:
volcano.index <- index.colours(volcano.raster)

## The index values can be converted back into colours, using the palette:
volcano.index <- as.raster(apply(volcano.index, 2,
                                 function(x) attributes(volcano.index)$palette[x]))

## Create an indexed image using dithering
volcano.dith <- index.colours(volcano.raster, dither = "floyd-steinberg")
volcano.dith <- as.raster(apply(volcano.dith, 2,
                                function(x) attributes(volcano.dith)$palette[x]))

## plot the images side by side for comparison
par(mfcol = c(1, 3))
plot(volcano.raster, interpolate = FALSE)
plot(volcano.index, interpolate = FALSE)
plot(volcano.dith, interpolate = FALSE)
```

---

interpretIFFChunk *Interpret an IFFChunk object*

Description

IFFChunk()s represent the structure of the Interchange File Format well, but the iformation is stored as raw data. This method tries to interpret and translate the information into a more comprehensive format.

Usage

```
## S4 method for signature 'IFFChunk'
interpretIFFChunk(x, ...)
```

Arguments

x	An IFFChunk() object which needs to be interpreted.
...	Currently ignored.

Details

Interchange File Format chunks can hold any kind of information (images, audio, (formatted) text, etc.). This method will try to convert this information into something useful. Information may get lost in the translation, so be careful when converting back to an `IFFChunk-class()` object using `IFFChunk-method()`.

An error is thrown when the `IFFChunk()` object is currently not interpretable by this package. See `IFFChunk-method()` for an overview of currently supported IFF chunks. This list may increase while this package matures.

Value

If `x` is interpretable by this package an S3 class object of `IFF.ANY` is returned. The content of the returned object will depend on the type of `IFFChunk()` provided for `x`. The result can for instance be a `raster` image (`grDevices::as.raster()`), a list of audio `tuneR::Wave()`s, a `character` string or a named `list`.

Author(s)

Pepijn de Vries

See Also

Other iff.operations: `IFFChunk-class`, `WaveToIFF()`, `as.raster.AmigaBasicShape()`, `getIFFChunk()`, `rasterToIFF()`, `rawToIFFChunk()`, `read.iff()`, `write.iff()`

Examples

```
## load an IFF file
example.iff <- read.iff(system.file("ilbm8lores.iff", package = "AmigaFFH"))

## in this case, the file is a FORM container with a bitmap image, and a
## list with a raster object is returned when interpreted:
example.itpt <- interpretIFFChunk(example.iff)
class(example.itpt)
typeof(example.itpt)
class(example.itpt[[1]])

## Let's extraxt the bitmap header from the main chunk:
bmhd <- getIFFChunk(example.iff, c("ILBM", "BMHD"))

## When interpreted, a named list is returned with (meta-)information
## on the bitmap image:
bmhd.itpt <- interpretIFFChunk(bmhd)
class(bmhd.itpt)
typeof(bmhd.itpt)
print(bmhd.itpt)
```

`names.AmigaBasic` *Extract or replace variable and label names from Amiga Basic scripts*

Description

In the binary Amiga Basic files, names for labels and variables in the code are stored at the end of the file. In the encoded there is only a pointer to the index of the name in that list. Use this function to list, select or replace names included in the code

Usage

```
## S3 method for class 'AmigaBasic'
names(x)

## S3 replacement method for class 'AmigaBasic'
names(x) <- value
```

Arguments

`x`	An `AmigaBasic()`-class object for which to obtain or change variable and/or label names
`value`	A (`vector` of) `character` string of desired replacement variable/label names.

Details

Make sure that variable and label names are valid for the basic script (see check.names.AmigaBasic).

Value

A `vector` of `character` strings with label and variable names in the basic script. In case of the replacement method a `AmigaBasic()`-class with replaced names is returned.

Author(s)

Pepijn de Vries

See Also

Other AmigaBasic.operations: `AmigaBasic.reserved()`, `AmigaBasicBMAP`, `AmigaBasic`, `[.AmigaBasic()`, `as.AmigaBasicBMAP()`, `as.AmigaBasic()`, `as.character()`, `check.names.AmigaBasic()`, `rawToAmigaBasicBMAP()`, `rawToAmigaBasic()`, `read.AmigaBasicBMAP()`, `read.AmigaBasic()`, `write.AmigaBasic()`

Examples

```
## Let's create some Basic code with labels and variables:
bas <- as.AmigaBasic(c(
  "REM - This will loop forever...",
  "my.label:",
  "  my.variable% = 0",
  "  WHILE my.variable% < 10",
  "    my.variable% = my.variable% + 1",
  "  WEND",
  "  GOTO my.label"
))

## list the names in the script above:
names(bas)

## change the first name:
names(bas)[1] <- "better.label"
```

`packBitmap` *A routine to (un)pack bitmap data*

Description

A very simplistic lossless routine to (un)pack repetitive bitmap data. Often used in InterLeaved BitMap (ILBM) images in IFF containers (`IFFChunk()`).

Usage

```
packBitmap(x)

unPackBitmap(x)
```

Arguments

`x`	`raw` data, usually representing a (packed) bitmap.

Details

InterLeaved BitMap (ILBM) images on the Amiga often use a packing algorithm referred to as 'ByteRun1'. This routine was introduced first on the Macintosh where it was called PackBits. It is a form of run-length encoding and is very simple: when a specific byte is repeated in a bitmap, it is replaced by a (signed negative) byte telling how many times the following byte should be repeated. When a series of bytes are not repetitive, it is preceded by a (signed positive) byte telling how long the non repetitive part is.

Not very complicated, but for most images some bytes can be shaved off the file. This was very useful when everything had to be stored on 880 kilobyte floppy disks with little CPU time to spare. Note that the file size can also increase for (noisy) images.

This packing routine will pack the entire bitmap (x) at once. The IFF file format requires packing of bitmap data per scanline. This is done automatically by the rasterToIFF() function, which calls this packing routine per scanline.

Value

Returns packed or unpacked raw data, depending on whether packBitmap or unPackBitmap was called.

Author(s)

Pepijn de Vries

References

http://amigadev.elowar.com/read/ADCD_2.1/Devices_Manual_guide/node01C0.html

https://en.wikipedia.org/wiki/PackBits

See Also

Other raw.operations: as.AmigaBasic(), as.raw.AmigaBasic(), colourToAmigaRaw(), rawToAmigaBasicBMAP(), rawToAmigaBasicShape(), rawToAmigaBasic(), rawToAmigaBitmapFontSet(), rawToAmigaBitmapFont(), rawToAmigaIcon(), rawToHWSprite(), rawToIFFChunk(), rawToSysConfig(), simpleAmigaIcon()

Examples

```
## generate some random raw data:
dat.rnd <- as.raw(sample.int(10, 100, TRUE))

## try to pack it:
pack.rnd <- packBitmap(dat.rnd)

## due to the random nature of the source data
## the data could not be packed efficiently.
## The length of the packed data is close to
## the length of the original data:
length(pack.rnd) - length(dat.rnd)

## Now generate similar data but sort it
## to generate more repetitive data:
dat.srt  <- as.raw(sort(sample.int(10, 100, TRUE)))
pack.srt <- packBitmap(dat.srt)

## This time the packing routing is more successful:
length(pack.srt) - length(dat.srt)

## The original data can always be obtained
## from the packed data:
all(dat.rnd == unPackBitmap(pack.rnd))
all(dat.srt == unPackBitmap(pack.srt))
```

play *Playing Amiga audio data*

Description

A wrapper for tuneR package's tuneR::play() routine. Allowing it to play Amiga audio (for instance stored in an 8SVX Interchange File Format).

Usage

```
## S4 method for signature 'ANY'
play(object, player = NULL, ...)

## S4 method for signature 'IFFChunk'
play(object, player = NULL, ...)
```

Arguments

object	An IFFChunk-class() object that needs to be played. The IFFChunk() should be of type FORM, containing an 8SVX chunk, or an 8SVX itself. object can also be of class IFF.FORM or IFF.8SVX. See tuneR::play() for other objects that can be played.
player	Path to the external audio player. See tuneR::play() for more details.
...	Arguments passed onto the tuneR play() routine.

Details

A wrapper for tuneR package's tuneR::play() routine. It will try to play audio using an external audio player. When 8SVX audio is played, each octave is played separately. When a FORM container contains multiple 8SVX samples, they are also played successively.

Note that a separate package is developed to interpret and play ProTracker modules and samples (ProTrackR()).

Value

Returns a list of data returned by tuneR's tuneR::play(), for which the output is undocumented.

Author(s)

Pepijn de Vries

Examples

```
## First get an audio sample from the ProTrackR package
snare.samp <- ProTrackR::PTSample(ProTrackR::mod.intro, 2)

## Coerce it into an IFFChunk object:
snare.iff <- WaveToIFF(snare.samp)

## Play the 8SVX sample:
if (interactive()) {
  play(snare.iff)
}
```

---

plot.AmigaBasicShape *Plot AmigaFFH objects*

Description

Plot AmigaFFH objects using base plotting routines.

Usage

```
## S3 method for class 'AmigaBasicShape'
plot(x, y, ...)

## S3 method for class 'AmigaBitmapFont'
plot(x, y, ...)

## S3 method for class 'AmigaBitmapFontSet'
plot(x, y, ...)

## S3 method for class 'hardwareSprite'
plot(x, y, ...)

## S3 method for class 'IFFChunk'
plot(x, y, ...)

## S3 method for class 'IFF.FORM'
plot(x, y, ...)

## S3 method for class 'IFF.8SVX'
plot(x, y, ...)

## S3 method for class 'IFF.ILBM'
plot(x, y, ...)

## S3 method for class 'IFF.ANIM'
plot(x, y, ...)
```

```
## S3 method for class 'SysConfig'
plot(x, y, ...)

## S3 method for class 'AmigaIcon'
plot(x, y, asp = 2, ...)
```

Arguments

x	An AmigaFFH object to be plotted. See usage section for supported object classes. If x is an AmigaBitmapFont() or AmigaBitmapFontSet() class object, it will plot the full bitmap that is used to extract the font glyphs.
y	When x is an AmigaIcon() class object, y can be used as an index. In that case, when y=1 the first icon image is shown. When y=2 the selected icon image is shown. When x is an AmigaBitmapFontSet() class object, y can be used to plot the bitmap of a specific font height (y). When x is an AmigaBasicShape() class object, y can be used to select a specific layer of the shape to plot, which can be one of "bitmap", "shadow" or "collision".
...	Parameters passed onto the generic graphics plotting routine. When x is an AmigaBitmapFont() or an AmigaBitmapFontSet() object, '...' can also be used for arguments that need to be passed onto the as.raster() function.
asp	A numeric value indicating the aspect ratio for the plot. For many AmigaFFH, the aspect ratio will be based on the Amiga display mode when known. For AmigaIcon() objects a default aspect ratio of 2 is used (tall pixels). When x is an AmigaBitmapFont() or AmigaBitmapFontSet() object, an aspect ratio of 1 is used by default. When the TALLDOT flag is set for that font, the aspect ratio s multiplied by 2. When the WIDEDOT flag is set, it will be divided by 2. A custom aspect ratio can also be used and will override the ratios specified above.

Details

A plotting routine is implemented for most AmigaFFH objects. See the usage section for all supported objects.

Value

Returns NULL silently.

Author(s)

Pepijn de Vries

Examples

```
## load an IFF file
example.iff <- read.iff(system.file("ilbm8lores.iff", package = "AmigaFFH"))

## and plot it:
plot(example.iff)

## AmigaIcons can also be plotted:
plot(simpleAmigaIcon())

## As can the cursor from a SysConfig object:
plot(simpleSysConfig())

## As can Amiga fonts:
data(font_example)
plot(font_example)
plot(font_example, text = "foo bar", style = "underlined", interpolate = FALSE)

## As can AmigaBasicShapes:
ball <- read.AmigaBasicShape(system.file("ball.shp", package = "AmigaFFH"))
plot(ball)
```

---

rasterToAmigaBasicShape

Convert a grDevices raster object into an AmigaBasicShape class object.

Description

Convert a `raster()` object into an `AmigaBasicShape()` class object.

Usage

```
rasterToAmigaBasicShape(
  x,
  type = c("blitter object", "sprite"),
  palette,
  shadow,
  collision,
  ...
)
```

Arguments

x	A `raster()` class object to convert into a `AmigaBasicShape()` class obejct.
type	A `character` string indicating what type of graphic needs to be created: "`blitter object`" (default) or "`sprite`".

palette	A vector of character strings, where each element represents a colour. This palette is used to quantize the colours that occur in the raster x.
shadow	An optional layer that could be stored with the graphics. This layer could be used for specific shadow effects when blitting the graphics to the screen. It needs to be a raster() object consisting of the colours black (bit unset) and white (bit set). The raster needs to have the same dimensions as x. This layer will be omitted when this argument is omitted (or set to NULL).
collision	An optional layer that could be stored with the graphics. This layer could be used for collision detection between graphical objects. It needs to be a raster() object consisting of the colours black (bit unset) and white (bit set). The raster needs to have the same dimensions as x. This layer will be omitted when this argument is omitted (or set to NULL).
...	Arguments passed onto index.colours(). Can be used, for instance, to achieve specific dithering effects.

Details

This method can be used to turn any graphics into an AmigaBasicShape() class object. In order to do so, the colours of the input image (a raster() object) will be quantized to a limited palette. This palette can be forced as an argument to this function. Otherwise, it will be based on the input image.

Value

Returns an AmigaBasicShape() class object based on x.

Author(s)

Pepijn de Vries

See Also

Other AmigaBasicShape.operations: AmigaBasicShape, read.AmigaBasicShape(), write.AmigaBasicShape()

Other raster.operations: AmigaBitmapFont, as.raster.AmigaBasicShape(), bitmapToRaster(), dither(), index.colours(), rasterToAmigaBitmapFont(), rasterToBitmap(), rasterToHWSprite(), rasterToIFF()

Examples

```
## get a raster image:
ilbm <- as.raster(read.iff(system.file("ilbm8lores.iff", package = "AmigaFFH")))

## convert to an Amiga Basic blitter object:
bob <- rasterToAmigaBasicShape(ilbm, "blitter object")
```

`rasterToAmigaBitmapFont` *Convert a raster image into an AmigaBitmapFont*

Description

Convert a two-coloured `grDevices::as.raster()` image into an `AmigaBitmapFont()` class object.

Usage

```
rasterToAmigaBitmapFont(
  x,
  glyphs,
  default_glyph,
  baseline,
  glyph_width,
  glyph_space,
  glyph_kern,
  palette,
  ...
)
```

Arguments

`x`	A `raster` (see grDevices package) object composed of two colours only. Make sure that all glyphs (graphical representation of characters) are next to eachother on a single line. The height of this raster (in pixels) is taken automatically as font height.
`glyphs`	Specify which glyphs are included in the image `x` from left to right. It can be specified in one of the following ways: A single `character` string, where the length of the string (`nchar`) equals the number of displayed glyphs in `x`. A `vector` of `numeric` ASCII codes. The length of the `vector` should equal the number of displayed glyphs in `x`. A `list` of either `character` strings or `vector` of `numerics`. The length of the `list` should equal the number of displayed glyphs in `x`. Each element can represent multiple characters, meaning that the nth element of the list uses the nth glyph shown in `x` to represent all the characters included in that element. Note that Amiga bitmap fonts represent ASCII characters and may not include all special characters or symbols.
`default_glyph`	A single `character` or ASCII code (`numeric`) that should be used by default. This means that all characters that are not specified by `glyphs` will be represented by this `default_glyph`. `default_glyph` should be included in `glyphs`.
`baseline`	The baseline of the font, specified in number of pixels from the top (`numeric`). Should be a whole number between 0 and the font height (height of `x`) minus 1.

glyph_width	A numeric vector with the same number of elements or characters as used for glyphs. It specifies the width in pixels for each glyph reserved in the raster image x. They should be whole numbers greater or equal to 0.
glyph_space	A numeric vector with the same number of elements or characters as used for glyphs. It specifies the width in pixels for each glyph that should be used when formatting. text. Note that these values can be smaller or larger than the values specified for glyph_width. They should be whole numbers greater or equal to 0.
glyph_kern	Note that in Amiga bitmap fonts not the formal definition from typography is used for kerning. Here, kerning is used as the number of pixels the cursor should be moved forward or backward after typesetting a character. It should be a numeric vector with the same number of elements or characters as used for glyphs. It can hold both positive and negative values.
palette	A vector of two colours. Both colours should be in x. The first colour is used as background colour, the second as foreground colour. When missing, it will be checked whether x has a palette as attribute, and uses that. If that attribute is also missing, the palette will be guessed from x, where the most frequently occurring colour is assumed to be the background colour.
...	Currently ignored.

Details

Create an AmigaBitmapFont() class object by providing a two-coloured raster image and specifying which characters are depicted by the image.

Value

Returns a AmigaBitmapFont() class object based on x.

Author(s)

Pepijn de Vries

See Also

Other AmigaBitmapFont.operations: AmigaBitmapFont, availableFontSizes(), c(), fontName(), font_example, getAmigaBitmapFont(), rawToAmigaBitmapFontSet(), rawToAmigaBitmapFont(), read.AmigaBitmapFontSet(), read.AmigaBitmapFont(), write.AmigaBitmapFont()

Other raster.operations: AmigaBitmapFont, as.raster.AmigaBasicShape(), bitmapToRaster(), dither(), index.colours(), rasterToAmigaBasicShape(), rasterToBitmap(), rasterToHWSprite(), rasterToIFF()

Examples

```
data("font_example")

## make a raster that we can use to create a bitmap font
font9.rast <- as.raster(getAmigaBitmapFont(font_example, 9))
```

```
## note the glyphs and the order in which they are included in
## the raster image:
plot(font9.rast)

## let's build a simple font, using only the first few glyphs
## in the raster:
font9 <- rasterToAmigaBitmapFont(
  ## 'x' needs the raster image:
  x             = font9.rast,

  ## 'glyphs' are the graphical representation of the characters
  ## that we will include in our font. We will only use the
  ## first 7 characters in the raster image:
  glyphs        = " !\"#$%&",

  ## We will use the '&' glyph to represent all characters that
  ## are not specified in the font:
  default_glyph = "&",

  ## The raster image is 9 pixels tall, as will be the font.
  ## Let's use 7 as the base (it needs to be less than the height)
  baseline      = 7,

  ## Let's define the width in pixels for each of the 7
  ## characters. This is their width in the raster image:
  glyph_width   = c(0, 1, 3, 6, 5, 5, 5),

  ## Let's define the space the character should take in pixels
  ## when it is used to format text:
  glyph_space   = c(4, 2, 4, 7, 6, 6, 6),

  ## the raster uses white as background colour and black as
  ## foreground:
  palette       = c("white", "black")
)

## note that for all characters that are not specified,
## the default glyph ('&') is used:
plot(font9, text = "!@#$%ABCD")

## Let's take a subset from the font's bitmap (rasteer):
font9abc.rast <- font9.rast[,263:282]

## as you can see this bitmap only contains the lowercase
## characters 'a', 'b', 'c', 'd' and 'e':
plot(font9abc.rast)

font9.abc <- rasterToAmigaBitmapFont(
  x             = font9abc.rast,
  ## Each glyph in the image can be represented by a single
  ## element in a list. By specifying multiple characters in
  ## each element, you can recycle a glyph to represent different
```

```
  ## characters. So in this case, the glyph 'a' is used for
  ## all the accented variants of the character 'a'.
  glyphs        = list("a\ue0\ue1\ue2\ue3\ue4\ue5",
                       "b",
                       "c\ua2\ue7",
                       "d",
                       "e\ue8\ue9\uea\ueb"),
  default_glyph = "c", ## 'c' is used as default glyph for all other characters
  baseline      = 7,
  glyph_width   = c(4, 4, 4, 4, 4),
  glyph_space   = c(5, 5, 5, 5, 5),
  palette       = c("white", "black")
)

## see what happens when you format text using the font we just created:
plot(font9.abc, text = "a\uE0\uE1\uE2\uE3\uE4\uE5\uA2\uE7\uE8\uE9\uEA\uEB, foo bar")
```

`rasterToBitmap` *Convert a grDevices* `raster` *object into binary bitmap data*

Description

Converts an image represented by a grDevices `raster` object into binary (Amiga) bitmap data.

Usage

```
rasterToBitmap(x, depth = 3, interleaved = TRUE, indexing = index.colours)
```

Arguments

`x`	A raster object created with `grDevices::as.raster()` which needs to be converted into bitmap data. It is also possible to let `x` be a matrix of `characters`, representing colours.
`depth`	The colour depth of the bitmap image. The image will be composed of `2^depth` indexed colours. `depth` can also be a `character` string "HAM6" or "HAM8" representing special Amiga display modes (see details).
`interleaved`	A `logical` value, indicating whether the bitmap needs to be interleaved. An interleaved bitmap image stores each consecutive bitmap layer per horizontal scanline.
`indexing`	A function that accepts two arguments: `x` (a grDevices `raster` object); `length.out`, a numeric value indicating the desired size of the palette (i.e., the number of colours). It should return a matrix with numeric palette indices (ranging from 1 up to the number of colours in the palette). The result should have an attribute named `palette' that contains the colours that correspond with the index numbers. The res` parent', with a single numeric value representing which colour in the palette should be treated as transparent (or `NA` when no transparency is required). By default the function `index.colours()` is used. You are free to provide a customised version of this function (see examples).

Details

Images represented by grDevices `raster` objects are virtually true colour (24 bit colour depth) and an alpha layer (transparency). On the early Amiga's the chipset (in combination with memory restrictions) only allowed images with indexed palettes. The colour depth was 12 bit with the original chipset and the number of colours allowed in a palette also depended on the chipset. This function will allow you to convert a `raster` object into binary bitmap data with an indexed palette. This means that the image is converted in a lossy way (information will be lost). So don't expect the result to have the same quality as the original image.

With the `depth` argument, the raster can also be converted to special mode bitmap images. One of these modes is the 'hold and modify' (HAM). In this mode two of the bitplanes are reserved as modifier switches. If the this switch equals zero, the remainder of the bitplanes are used as an index for colours in a fixed palette. If the switch equals 1, 2 or 3, the red, green or blue component of the previous is modified, using the number in the remainder of the bitplanes. So it holds the previous colour but modifies one of the colour components (hence the term 'hold and modify'.) Here only the HAM6 and the HAM8 mode are implemented. HAM6 uses 6 bitplanes and a 12 bit colour depth, HAM8 uses 8 bitplanes and a 24 bit colour depth.

The HAM mode was a special video modes supported by Amiga hardware. Normal mode bitmap images with a 6 bit depth would allow for a palette of 64 (2^6) colours, HAM6 can display 4096 colours with the same bit depth.

In addition to HAM6 and HAM8, sliced HAM (or SHAM) was another HAM variant. Using the coprocessor on the Amiga, it was possible to change the palette at specific scanlines, increasing the number of available colours even further. The SHAM mode is currently not supported by this package.

Value

The bitmap is returned as a `vector` of `logical` values. The `logical` values reflect the bits for each bitplane. The palette used for the bitmap is returned as attribute to the `vector`. There will also be an attribute called `transparent'. This will hold a numeric index corresponding with the colour in the palette th` when transparency is not used.

Author(s)

Pepijn de Vries

See Also

Other raster.operations: `AmigaBitmapFont`, `as.raster.AmigaBasicShape()`, `bitmapToRaster()`, `dither()`, `index.colours()`, `rasterToAmigaBasicShape()`, `rasterToAmigaBitmapFont()`, `rasterToHWSprite()`, `rasterToIFF()`

Examples

```
## first: Let's make a raster out of the 'volcano' data, which we can use in the example:
volcano.raster <- as.raster(t(matrix(terrain.colors(1 + diff(range(volcano)))[volcano -
  min(volcano) + 1], nrow(volcano))))

## convert the raster into binary (logical) bitmap data:
```

```
volcano.bm <- rasterToBitmap(volcano.raster)

## The palette for the indexed colours of the generated bitmap is returned as
## attribute. There is no transparency is the image:
attributes(volcano.bm)

## We can also include a custom function for colour quantisation. Let's include
## some dithering:
volcano.dither <- rasterToBitmap(volcano.raster,
                                 indexing = function(x, length.out) {
                                   index.colours(x, length.out,
                                                 dither = "floyd-steinberg")
                                 })

## You can also use a custom indexing function to force a specified palette,
## in this case black and white:
volcano.bw <- rasterToBitmap(volcano.raster,
                             indexing = function(x, length.out) {
                               index.colours(x, length.out,
                                             palette = c("black", "white"),
                                             dither = "floyd-steinberg")
                             })

## Make a bitmap using a special display mode (HAM6):
volcano.HAM <- rasterToBitmap(volcano.raster, "HAM6")
```

`rasterToHWSprite` *Convert a raster object into an hardwareSprite object*

Description

Convert a grDevices raster object into an Amiga hardwareSprite class object.

Usage

```
rasterToHWSprite(x, indexing = index.colours)
```

Arguments

`x`	A `grDevices()` raster object (`grDevices::as.raster()`) that needs to be converted into a `hardwareSprite()` class object. Note that a `hardwareSprite()` has a maximum width of 16 pixels. When `x` is wider, it will be cropped.
`indexing`	A function that accepts two arguments: `x` (a grDevices `raster` object); `length.out`, a numeric value indicating the desired size of the palette (i.e., the number of colours). It should return a matrix with numeric palette indices (ranging from 1 up to the number of colours in the palette). The result should have an attribute named `palette' that contains the colours that correspond with the index numbers. The res` parent', with a single numeric value representing which colour in the palette should be treated as transparent (or `NA` when no transparency is required). By default the function `index.colours()` is used.

Details

A grDevices() raster image can be converted into a hardwareSprite() class object with this function. For this purpose the any true-colour image will be converted to an indexed palette with 4 colours. The Amiga hardware sprite will reserve one of the colours as transparent. Thos function will use fully transparent colours in the original image (i.e., the alpha level equals 0) for this purpose. Or when the image has no fully transparent colours, it will use the most frequently occuring colour (at least when the default `indexing` function is used).

Value

Returns a hardwareSprite() class object based on `x`

Author(s)

Pepijn de Vries

See Also

Other raster.operations: AmigaBitmapFont, as.raster.AmigaBasicShape(), bitmapToRaster(), dither(), index.colours(), rasterToAmigaBasicShape(), rasterToAmigaBitmapFont(), rasterToBitmap(), rasterToIFF()

Other HWSprite.operations: rawToHWSprite()

Examples

```
## first create a raster object that can be used as input
## (making sure that the background is transparent):
rst <- as.raster(simpleSysConfig()$PointerMatrix, "#AAAAAA00")

## now turn it into a hardware sprite:
spr <- rasterToHWSprite(rst)

## and plot it as a check:
plot(spr)
```

---

rasterToIFF *Convert a grDevices raster image into an IFF formated bitmap image*

Description

Convert grDevices raster images (grDevices::as.raster()) into a formal IFFChunk() object, as an interleaved bitmap (ILBM) image.

Usage

```
rasterToIFF(
  x,
  display.mode = as.character(AmigaFFH::amiga_display_modes$DISPLAY_MODE),
  monitor = as.character(AmigaFFH::amiga_monitors$MONITOR_ID),
  anim.options,
  ...
)
```

Arguments

x	A raster object created with grDevices::as.raster() which needs to be converted into an IFF formated bitmap image. It is also possible to let x be a matrix of characters, representing colours.
display.mode	Specify the Amiga display mode that should be used. See amiga_display_modes() for all possible options. "LORES_KEY" is used by default, this is the lowest resolution possible on the Amiga.
monitor	The Amiga monitor on which the needs to be displayed. See amiga_monitors() for more details and posible options. By default "DEFAULT_MONITOR_ID" is used.
anim.options	Currently ignored. This argument will potentitally be implemented in future versions of this package. Currently, animations are always encoded with the "ByteVerticalCompression" in this package (when x is a list of raster objects).
...	Arguments passed on to rasterToBitmap().

Details

Convert any modern image into a interleaved bitmap (image) conform Interchange File Format (IFF) specifications. If your original image is in true colour (i.e., a 24 bit colour depth) it will be converted into a bitmap image with an indexed palette.

Value

Returns an IFFChunk() object holding an Interleaved Bitmap (ILBM) image based on x.

Author(s)

Pepijn de Vries

See Also

Other iff.operations: IFFChunk-class, WaveToIFF(), as.raster.AmigaBasicShape(), getIFFChunk(), interpretIFFChunk(), rawToIFFChunk(), read.iff(), write.iff()

Other raster.operations: AmigaBitmapFont, as.raster.AmigaBasicShape(), bitmapToRaster(), dither(), index.colours(), rasterToAmigaBasicShape(), rasterToAmigaBitmapFont(), rasterToBitmap(), rasterToHWSprite()

Examples

```
## first: Let's make a raster out of the 'volcano' data, which we can use in the example:
volcano.raster <- as.raster(t(matrix(terrain.colors(1 + diff(range(volcano)))[volcano -
  min(volcano) + 1], nrow(volcano))))

## Turning the raster into an IFFChunk object is easy:
volcano.iff <- rasterToIFF(volcano.raster)

## This object can be saved as an IFF file using write.iff

## in special modes HAM6 and HAM 8 higher quality images
## can be obtained. See 'rasterToBitmap' for more info on the
## special HAM modes.
volcano.ham <- rasterToIFF(volcano.raster, "HAM_KEY", depth = "HAM8")

## The result can be further improved by applying dithering
volcano.ham.dither <- rasterToIFF(volcano.raster, "HAM_KEY", depth = "HAM8",
  indexing = function(x, length.out) {
    index.colours(x, length.out, dither = "JJN", iter.max = 20)
  })
```

---

rawToAmigaBasic *Coerce raw data into an AmigaBasic class object*

Description

AmigaBasic() objects are comprehensive representations of binary-encode Amiga Basic scripts. Use this function to convert raw content from encoded Amiga Basic scripts to an AmigaBasic() object.

Usage

```
rawToAmigaBasic(x, ...)
```

Arguments

x	A vector of raw data that is to be converted into an AmigaBasic() class object.
...	Currently ignored.

Details

This function will convert raw data as stored in Amiga Basic files into its corresponding S3 AmigaBasic()-class object.

Value

An AmigaBasic() class object based on x.

Author(s)

Pepijn de Vries

See Also

Other AmigaBasic.operations: AmigaBasic.reserved(), AmigaBasicBMAP, AmigaBasic, [.AmigaBasic(), as.AmigaBasicBMAP(), as.AmigaBasic(), as.character(), check.names.AmigaBasic(), names.AmigaBasic(), rawToAmigaBasicBMAP(), read.AmigaBasicBMAP(), read.AmigaBasic(), write.AmigaBasic()

Other raw.operations: as.AmigaBasic(), as.raw.AmigaBasic(), colourToAmigaRaw(), packBitmap(), rawToAmigaBasicBMAP(), rawToAmigaBasicShape(), rawToAmigaBitmapFontSet(), rawToAmigaBitmapFont(), rawToAmigaIcon(), rawToHWSprite(), rawToIFFChunk(), rawToSysConfig(), simpleAmigaIcon()

Examples

```
## First create an AmigaBAsic object:
bas <- as.AmigaBasic("PRINT \"Hello world!\"")

## Make it raw:
bas.raw <- as.raw(bas)

## Now convert it back to an AmigaBasic object:
bas <- rawToAmigaBasic(bas.raw)
```

---

rawToAmigaBasicBMAP *Coerce raw data into an AmigaBasicBMAP class object*

Description

Coerce raw data into an AmigaBasicBMAP() class object

Usage

```
rawToAmigaBasicBMAP(x, ...)
```

Arguments

x	A vector of raw data that is to be converted into an AmigaBasicBMAP() class object.
...	Currently ignored.

Details

An Amiga Basic BMAP file maps the offset of routines in Amiga libraries. This function converts the raw format in which it would be stored as a file into a comprehensive S3 class object.

Value

An AmigaBasicBMAP() class object based on x.

Author(s)

Pepijn de Vries

See Also

Other AmigaBasic.operations: AmigaBasic.reserved(), AmigaBasicBMAP, AmigaBasic, [.AmigaBasic(), as.AmigaBasicBMAP(), as.AmigaBasic(), as.character(), check.names.AmigaBasic(), names.AmigaBasic(), rawToAmigaBasic(), read.AmigaBasicBMAP(), read.AmigaBasic(), write.AmigaBasic()

Other raw.operations: as.AmigaBasic(), as.raw.AmigaBasic(), colourToAmigaRaw(), packBitmap(), rawToAmigaBasicShape(), rawToAmigaBasic(), rawToAmigaBitmapFontSet(), rawToAmigaBitmapFont(), rawToAmigaIcon(), rawToHWSprite(), rawToIFFChunk(), rawToSysConfig(), simpleAmigaIcon()

Examples

```
## A small fragment of the dos.library BMAP would look like this:
dos.bmap <- as.AmigaBasicBMAP(list(
  xOpen = list(
    libraryVectorOffset = -30,
    registers = as.raw(2:3)
  ),
  xClose = list(
    libraryVectorOffset = -36,
    registers = as.raw(2)
  ),
  xRead = list(
    libraryVectorOffset = -42,
    registers = as.raw(2:4)
  )
))

## The raw representation would be
dos.bmap.raw <- as.raw(dos.bmap)

## And the reverse
rawToAmigaBasicBMAP(dos.bmap.raw)
```

---

rawToAmigaBasicShape *Coerce raw data into an AmigaBasicShape class object*

Description

Coerce raw data into an AmigaBasicShape()-class object

Usage

```
rawToAmigaBasicShape(x, palette)
```

Arguments

x	A vector of raw data that is to be converted into an AmigaBasicShape() class object.
palette	A vector of character strings, where each element represents a colour in the palette. This palette will be used to display the graphics (note that the raw format does not store the palette, but this S3 class does). When this argument is omitted a grey scale palette will be generated.

Details

AmigaBasicShape() objects are comprehensive representations of blitter and sprite graphics that can be used in AmigaBasic() scripts. Use this function to convert raw content to an AmigaBasicShape() object.

Value

returns an AmigaBasicShape()-class object.

Author(s)

Pepijn de Vries

See Also

Other raw.operations: as.AmigaBasic(), as.raw.AmigaBasic(), colourToAmigaRaw(), packBitmap(), rawToAmigaBasicBMAP(), rawToAmigaBasic(), rawToAmigaBitmapFontSet(), rawToAmigaBitmapFont(), rawToAmigaIcon(), rawToHWSprite(), rawToIFFChunk(), rawToSysConfig(), simpleAmigaIcon()

Examples

```
filename <- system.file("ball.shp", package = "AmigaFFH")

## read as binary:
con      <- file(filename, "rb")
ball.raw <- readBin(con, "raw", file.size(filename))
close(con)

## convert raw data into something useful:
ball     <- rawToAmigaBasicShape(ball.raw)

## A shortcut would be to call read.AmigaBasicShape
ball2    <- read.AmigaBasicShape(filename)
```

rawToAmigaBitmapFont *Coerce raw data into an AmigaBitmapFont class object*

Description

AmigaBitmapFont() objects are comprehensive representations of binary Amiga font subset files. The file name is usually simply a numeric number indicating the font height in pixels. Use this function to convert raw content from such a file to an AmigaBitmapFont() object.

Usage

```
rawToAmigaBitmapFont(x, ...)
```

Arguments

x	An AmigaBitmapFont() object which needs to be converted into raw data.
...	Currently ignored.

Details

This function converts raw data as stored in font bitmap files. These files are stored in subdirectories with the font's name and usually have the font height in pixels as file name. This function is effectively the inverse of as.raw().

Value

A vector of raw data representing x.

Author(s)

Pepijn de Vries

See Also

Other AmigaBitmapFont.operations: AmigaBitmapFont, availableFontSizes(), c(), fontName(), font_example, getAmigaBitmapFont(), rasterToAmigaBitmapFont(), rawToAmigaBitmapFontSet(), read.AmigaBitmapFontSet(), read.AmigaBitmapFont(), write.AmigaBitmapFont()

Other raw.operations: as.AmigaBasic(), as.raw.AmigaBasic(), colourToAmigaRaw(), packBitmap(), rawToAmigaBasicBMAP(), rawToAmigaBasicShape(), rawToAmigaBasic(), rawToAmigaBitmapFontSet(), rawToAmigaIcon(), rawToHWSprite(), rawToIFFChunk(), rawToSysConfig(), simpleAmigaIcon()

Examples

```
## first create raw data that can be converted into a AmigaBitmapFont
data(font_example)
font.raw <- as.raw(getAmigaBitmapFont(font_example, 9))

## Convert it back into an AmigaBitmapFont object:
font <- rawToAmigaBitmapFont(font.raw)
```

rawToAmigaBitmapFontSet *Coerce raw data into an AmigaBitmapFontSet class object*

Description

AmigaBitmapFontSet() objects are comprehensive representations of binary Amiga font files (*.font). Use this function to convert raw data from such a file to an AmigaBitmapFontSet object.

Usage

```
rawToAmigaBitmapFontSet(x, file)
```

Arguments

x	A vector of raw data that needs to be converted into an AmigaBitmapFontSet().
file	The raw version of the AmigaBitmapFontSet() does not contain the nested font bitmap images. In order to correctly construct an AmigaBitmapFontSet() the file location of the original *.font file is required in order to read and include the font bitmap image information. file should thus be a character string specifying the file location of the *.font file.

Details

This function converts raw data as stored in *.font files. The function also needs the file location, in order to load the nested bitmap images for each font height. This function is effectively the inverse of as.raw().

Value

Returns an AmigaBitmapFontSet() object.

Author(s)

Pepijn de Vries

See Also

Other AmigaBitmapFont.operations: AmigaBitmapFont, availableFontSizes(), c(), fontName(), font_example, getAmigaBitmapFont(), rasterToAmigaBitmapFont(), rawToAmigaBitmapFont(), read.AmigaBitmapFontSet(), read.AmigaBitmapFont(), write.AmigaBitmapFont()

Other raw.operations: as.AmigaBasic(), as.raw.AmigaBasic(), colourToAmigaRaw(), packBitmap(), rawToAmigaBasicBMAP(), rawToAmigaBasicShape(), rawToAmigaBasic(), rawToAmigaBitmapFont(), rawToAmigaIcon(), rawToHWSprite(), rawToIFFChunk(), rawToSysConfig(), simpleAmigaIcon()

Examples

```
data(font_example)

## First create raw font set data. Note that this raw data
## does not include the nested font bitmap images.
fontset.raw <- as.raw(font_example)

## Therefore it is necesary to have the entire font stored as files:
write.AmigaBitmapFontSet(font_example, tempdir())

font.restored <- rawToAmigaBitmapFontSet(fontset.raw, file.path(tempdir(), "AmigaFFH.font"))
```

---

rawToAmigaIcon *Coerce raw data into an AmigaIcon class object*

Description

`AmigaIcon()` objects are comprehensive representations of binary Amiga Workbench icon files (*.info). Use this function to convert `raw` data from such a file to an `AmigaIcon()` object.

Usage

```
rawToAmigaIcon(x, palette = NULL)
```

Arguments

`x`	A vector of `raw` data that needs to be converted into an S3 `AmigaIcon()` class object.
`palette`	Provide a palette (`vector` of colours) for the icon bitmap image. When set to `NULL` (default) the standard Amiga Workbench palette will be used.

Details

Icons files (*.info) were used as a graphical representations of files and directories on the Commodore Amiga. This function will convert the raw data from such files into a more comprehensive names list (see `AmigaIcon()`). Use `as.raw()` to achieve the inverse.

Value

Returns an `AmigaIcon()` class object based on `x`.

Author(s)

Pepijn de Vries

See Also

Other AmigaIcon.operations: AmigaIcon, read.AmigaIcon(), simpleAmigaIcon(), write.AmigaIcon()

Other raw.operations: as.AmigaBasic(), as.raw.AmigaBasic(), colourToAmigaRaw(), packBitmap(), rawToAmigaBasicBMAP(), rawToAmigaBasicShape(), rawToAmigaBasic(), rawToAmigaBitmapFontSet(), rawToAmigaBitmapFont(), rawToHWSprite(), rawToIFFChunk(), rawToSysConfig(), simpleAmigaIcon()

Examples

```
## generate a simple AmigaIcon object:
icon <- simpleAmigaIcon()

## convert it into raw data:
icon.raw <- as.raw(icon)

## convert the raw data back into an icon:
icon.restored <- rawToAmigaIcon(icon.raw)
```

---

rawToHWSprite *Convert raw data into an Amiga hardware sprite*

Description

Convert raw data structured conform a Commodore Amiga hardware sprite (see references) into a hardwareSprite() object.

Usage

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

## S4 method for signature 'raw,character'
rawToHWSprite(x, col)
```

Arguments

x	raw data structured as an Amiga hardware sprite (see references).
col	A vector of colours (character) to be used for the hardware sprite. Specify the three visible colours for the sprite. When missing some default colours (grayscale) will be used. The colours have to be provided separately as they are usually not stored together with the hardware sprite data.

Details

Information to set up a hardware sprite is stored as raw data on Commodore Amigas. This method can be used to convert this data into a hardwareSprite() object. This object can in turn be converted with as.raster() such that it can be plotted in R.

Value

Returns a hardwareSprite() object based on the provided raw data

Author(s)

Pepijn de Vries

References

http://amigadev.elowar.com/read/ADCD_2.1/Hardware_Manual_guide/node00B9.html

See Also

Other raw.operations: as.AmigaBasic(), as.raw.AmigaBasic(), colourToAmigaRaw(), packBitmap(), rawToAmigaBasicBMAP(), rawToAmigaBasicShape(), rawToAmigaBasic(), rawToAmigaBitmapFontSet(), rawToAmigaBitmapFont(), rawToAmigaIcon(), rawToIFFChunk(), rawToSysConfig(), simpleAmigaIcon()

Other HWSprite.operations: rasterToHWSprite()

Examples

```
## Let's generate a 16x16 sprite with a random bitmap:
dat <- as.raw(c(0x00, 0x00, 0x10, 0x00,
              sample.int(255, 64, replace = TRUE),
              0x00, 0x00, 0x00, 0x00))
## make it a hardware sprite object:
spr <- rawToHWSprite(dat)
## and plot it:
plot(spr, interpolate = FALSE)

## with some imagination when can make
## a more structured image:
dat <- as.raw(c(0x00, 0x00, 0x10, 0x00, 0x00, 0x00, 0xff, 0xf8,
                0x7f, 0x80, 0x80, 0x70, 0x7f, 0x00, 0xbe, 0xe0,
                0x7e, 0x00, 0x85, 0xc0, 0x7d, 0x80, 0x82, 0x40,
                0x6b, 0xc0, 0x95, 0xa0, 0x57, 0xe0, 0xa8, 0xd0,
                0x2f, 0xf0, 0xd1, 0x68, 0x4f, 0xf8, 0xb0, 0x34,
                0x07, 0xfc, 0xf8, 0x5a, 0x03, 0xfe, 0xe4, 0x0d,
                0x01, 0xfc, 0xc2, 0x12, 0x00, 0xf8, 0x81, 0x04,
                0x00, 0x70, 0x00, 0x88, 0x00, 0x20, 0x00, 0x50,
                0x00, 0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x00))
spr <- rawToHWSprite(dat, c("#EE4444", "#000000", "#EEEECC"))
plot(spr, interpolate = FALSE)
```

rawToIFFChunk *Coerce raw data to an IFFChunk class object*

Description

Coerce raw data, as it would be stored in the Interchange File Format (IFF), and convert it into an IFFChunk() class object.

Usage

```
## S4 method for signature 'raw'
rawToIFFChunk(x)
```

Arguments

x	A vector of raw data that needs to be converted into a IFFChunk() class object.

Details

This method should work for all IFF chunk types that are implemented in this package (see IFFChunk-method() for details). For non-implemented chunks this method may work properly as long as the chunks are nested inside a FORM type container chunk. This method is provided for your convenience, but it is recommended to import IFFChunk methods using the read.iff() function. Use as.raw() to achieve the inverse of this method.

Value

Returns an IFFChunk() class object based on x.

Author(s)

Pepijn de Vries

See Also

Other iff.operations: IFFChunk-class, WaveToIFF(), as.raster.AmigaBasicShape(), getIFFChunk(), interpretIFFChunk(), rasterToIFF(), read.iff(), write.iff()

Other raw.operations: as.AmigaBasic(), as.raw.AmigaBasic(), colourToAmigaRaw(), packBitmap(), rawToAmigaBasicBMAP(), rawToAmigaBasicShape(), rawToAmigaBasic(), rawToAmigaBitmapFontSet(), rawToAmigaBitmapFont(), rawToAmigaIcon(), rawToHWSprite(), rawToSysConfig(), simpleAmigaIcon()

Examples

```
## Get an IFFChunk object:
example.iff <- read.iff(system.file("ilbm8lores.iff", package = "AmigaFFH"))

## Coerce it to raw data:
example.raw <- as.raw(example.iff)
```

```
## Coerce raw data to IFF chunk:
example.iff.new <- rawToIFFChunk(example.raw)

## These conversions were non-destructive:
identical(example.iff, example.iff.new)
```

`rawToSysConfig` *Coerce raw data into a SysConfig class object*

Description

SysConfig objects are comprehensive representations of binary Amiga system-configuration files. Use this function to convert `raw` data from such a file to a SysConfig object.

Usage

```
rawToSysConfig(x)
```

Arguments

`x`	A vector of `raw` data that needs to be converted into an S3 SysConfig class object. It should have a length of at least 232. Although system-configurations can be extended, such extended files are not supported here.

Details

The Amiga used the system-configuration file to store certain system preferences in a binary file. With this function such `raw` data can be converted into a more comprehensive SysConfig object. Use `as.raw()` to achieve the inverse.

Value

Returns a SysConfig class object based on `x`.

Author(s)

Pepijn de Vries

See Also

Other SysConfig.operations: `SysConfig`, `read.SysConfig()`, `simpleSysConfig()`, `write.SysConfig()`

Other raw.operations: `as.AmigaBasic()`, `as.raw.AmigaBasic()`, `colourToAmigaRaw()`, `packBitmap()`, `rawToAmigaBasicBMAP()`, `rawToAmigaBasicShape()`, `rawToAmigaBasic()`, `rawToAmigaBitmapFontSet()`, `rawToAmigaBitmapFont()`, `rawToAmigaIcon()`, `rawToHWSprite()`, `rawToIFFChunk()`, `simpleAmigaIcon()`

Examples

```
if (requireNamespace("adfExplorer", quietly = TRUE)) {
  library(adfExplorer)

  ## get the system-configuration from the adfExplorer example disk:
  disk <- connect_adf(
    system.file("example.adz", package = "adfExplorer")
  )
  virtual_file_con <- adf_file_con(disk, "devs/system-configuration")
  sc <- readBin(virtual_file_con, "raw", 1024)
  close(disk)

  ## This will get you the raw data from the file:
  typeof(sc)

  ## Convert the raw data to a more comprehensive named list (and S3 SysConfig class):
  sc <- rawToSysConfig(sc)
}
```

`read.AmigaBasic` *Read Amiga Basic files*

Description

Read an `AmigaBasic()` script from its binary format.

Usage

```
read.AmigaBasic(file, ...)
```

Arguments

`file`	A `character` string of the filename of the Amiga Basic file to be read.
`...`	Currently ignored

Details

Normally Amiga Basic code is stored encoded in a binary format (`rawToAmigaBasic()`). This function reads the binary data from a file (which can be stored on a virtual disk (`adf_file_con()`)) and converts in into an `AmigaBasic()` class objec.

Value

Returns an `AmigaBasic()` class object read from the `file`.

Author(s)

Pepijn de Vries

See Also

Other AmigaBasic.operations: AmigaBasic.reserved(), AmigaBasicBMAP, AmigaBasic, [.AmigaBasic(), as.AmigaBasicBMAP(), as.AmigaBasic(), as.character(), check.names.AmigaBasic(), names.AmigaBasic(), rawToAmigaBasicBMAP(), rawToAmigaBasic(), read.AmigaBasicBMAP(), write.AmigaBasic()

Other io.operations: read.AmigaBasicBMAP(), read.AmigaBasicShape(), read.AmigaBitmapFontSet(), read.AmigaBitmapFont(), read.AmigaIcon(), read.SysConfig(), read.iff(), write.AmigaBasicShape(), write.AmigaBasic(), write.AmigaBitmapFont(), write.AmigaIcon(), write.SysConfig(), write.iff()

Examples

```
## First create an AmigaBasic file
write.AmigaBasic(as.AmigaBasic("PRINT \"Hello world\""),
                 file.path(tempdir(), "helloworld.bas"))

## Now let's read the same file:
bas <- read.AmigaBasic(file.path(tempdir(), "helloworld.bas"))

## There's also a demo file included with the package
demo.bas <- read.AmigaBasic(system.file("demo.bas", package = "AmigaFFH"))
demo.bas
```

---

read.AmigaBasicBMAP *Read and write Amiga Basic BMAP files*

Description

Read and write AmigaBasicBMAP() binary file format.

Usage

```
read.AmigaBasicBMAP(file)

write.AmigaBasicBMAP(x, file)
```

Arguments

file	A character string of the filename of the Amiga Basic BMAP file to be read or written.
x	A AmigaBasicBMAP() class object that needs to be stored.

Details

An Amiga Basic BMAP file maps the offset of routines in Amiga libraries and can be read as a comprehensive list and written back to a binary file using these functions.

Value

Returns an AmigaBasicBMAP() class object read from the file in case of read.AmigaBasicBMAP. Otherwise, invisibly returns the result of the call of close to the file connection.

Author(s)

Pepijn de Vries

See Also

Other AmigaBasic.operations: AmigaBasic.reserved(), AmigaBasicBMAP, AmigaBasic, [.AmigaBasic(), as.AmigaBasicBMAP(), as.AmigaBasic(), as.character(), check.names.AmigaBasic(), names.AmigaBasic(), rawToAmigaBasicBMAP(), rawToAmigaBasic(), read.AmigaBasic(), write.AmigaBasic()

Other io.operations: read.AmigaBasicShape(), read.AmigaBasic(), read.AmigaBitmapFontSet(), read.AmigaBitmapFont(), read.AmigaIcon(), read.SysConfig(), read.iff(), write.AmigaBasicShape(), write.AmigaBasic(), write.AmigaBitmapFont(), write.AmigaIcon(), write.SysConfig(), write.iff()

Examples

```
## A small fragment of the dos.library BMAP would look like this:
dos.bmap <- as.AmigaBasicBMAP(list(
  xOpen = list(
    libraryVectorOffset = -30,
    registers = as.raw(2:3)
  ),
  xClose = list(
    libraryVectorOffset = -36,
    registers = as.raw(2)
  ),
  xRead = list(
    libraryVectorOffset = -42,
    registers = as.raw(2:4)
  )
))

## This will write the BMAP to a file:
write.AmigaBasicBMAP(dos.bmap, file.path(tempdir(), "dos.bmap"))

## This will read the same file:
dos.bmap.copy <- read.AmigaBasicBMAP(file.path(tempdir(), "dos.bmap"))
```

---

read.AmigaBasicShape *Read Amiga Basic Shape files*

Description

Read Amiga Basic Shape files

Usage

```
read.AmigaBasicShape(file, ...)
```

Arguments

file	A character string of the filename of the Amiga Basic Shape file to be read.
...	Arguments passed to rawToAmigaBasicShape().

Details

AmigaBasic used the term 'shapes' for graphics (sprites and blitter objects) which it could display. These graphics were stored in a specific binary format, which can be read with this function. See AmigaBasicShape() for more details. The file can also be read from a virtual Amiga disk (adf_file_con()).

Value

Returns an AmigaBasicShape() class object read from the file.

Author(s)

Pepijn de Vries

See Also

Other AmigaBasicShape.operations: AmigaBasicShape, rasterToAmigaBasicShape(), write.AmigaBasicShape()

Other io.operations: read.AmigaBasicBMAP(), read.AmigaBasic(), read.AmigaBitmapFontSet(), read.AmigaBitmapFont(), read.AmigaIcon(), read.SysConfig(), read.iff(), write.AmigaBasicShape(), write.AmigaBasic(), write.AmigaBitmapFont(), write.AmigaIcon(), write.SysConfig(), write.iff()

Examples

```
filename <- system.file("ball.shp", package = "AmigaFFH")
ball     <- read.AmigaBasicShape(filename)
## This is a sprite:
ball$flags[["fVSprite"]]

filename <- system.file("r_logo.shp", package = "AmigaFFH")
## The palette is not stored with an Amiga Basic Shape, so let's provide one:
r_logo   <- read.AmigaBasicShape(filename,
                                 palette = c("#FFFFFF", "#2266BB", "#3366BB", "#4477AA",
                                 "#778899", "#999999", "#AAAAAA", "#BBBBBB"))
## This is a blitter object:
r_logo$flags[["fVSprite"]]

## Just for fun, plot it:
plot(r_logo)
```

`read.AmigaBitmapFont` *Read an AmigaBitmapFont class object from a file*

Description

Amiga Font Bitmaps of distinctive font heights are stored in separate files, which in combination form a font collection or set. This function can be used to read a specific bitmap from a set and returns it as an `AmigaBitmapFont()` class object.

Usage

```
read.AmigaBitmapFont(file, ...)
```

Arguments

`file`	The file name of a font subset is usually simply a numeric number indicating the font height in pixels. Use `file` as a `character` string representing that file location.
`...`	Arguments passed on to `rawToAmigaBitmapFont()`.

Details

Individual font bitmaps are stored in a font's subdirectory where the file name is usually equal to the font height in pixels. This function will read such a font bitmap file and return it as an `AmigaBitmapFont()` class object. It can also read such files from virtual disks (`adf_file_con()`) objects, but that requires the adfExplorer package to be installed.

Value

Returns an `AmigaBitmapFont()` object read from the specified file.

Author(s)

Pepijn de Vries

See Also

Other AmigaBitmapFont.operations: `AmigaBitmapFont`, `availableFontSizes()`, `c()`, `fontName()`, `font_example`, `getAmigaBitmapFont()`, `rasterToAmigaBitmapFont()`, `rawToAmigaBitmapFontSet()`, `rawToAmigaBitmapFont()`, `read.AmigaBitmapFontSet()`, `write.AmigaBitmapFont()`

Other io.operations: `read.AmigaBasicBMAP()`, `read.AmigaBasicShape()`, `read.AmigaBasic()`, `read.AmigaBitmapFontSet()`, `read.AmigaIcon()`, `read.SysConfig()`, `read.iff()`, `write.AmigaBasicShape()`, `write.AmigaBasic()`, `write.AmigaBitmapFont()`, `write.AmigaIcon()`, `write.SysConfig()`, `write.iff()`

Examples

```
data(font_example)

## Let's store the example font first:
write.AmigaBitmapFontSet(font_example, tempdir())

## Now read a specific subset from the font files:
font.sub <- read.AmigaBitmapFont(file.path(tempdir(), "AmigaFFH", "9"))

## The same can be done with a virtual Amiga disk. The following
## examples require the 'adfExplorer' package.
if (requireNamespace("adfExplorer")) {
  library("adfExplorer")
  virtual_disk_file <- tempfile(fileext = ".adf") |>
    create_adf_device(write_protected = FALSE) |>
    prepare_adf_device("font_disk") |>
    make_adf_dir("FONTS")

  dest <- virtual_path(virtual_disk_file, "DF0:FONTS")
  write.AmigaBitmapFontSet(font_example, dest)
  font.read <- read.AmigaBitmapFont(
    virtual_path(virtual_disk_file, "DF0:FONTS/AmigaFFH/9")
  )
  close(virtual_disk_file)
}
```

`read.AmigaBitmapFontSet` *Read AmigaBitmapFontSet from *.font file*

Description

Reads `AmigaBitmapFontSet()` from *.font file including all nested bitmap images for all font heights.

Usage

```
read.AmigaBitmapFontSet(file, ...)
```

Arguments

`file`	A `character` string of the filename of the *.font file to be read.
`...`	Currently ignored.

Details

The *.font file only holds meta-information. The bitmap images for each font height are stored in separate files, which are listed in the *.font file. The function reads the *.font file, including all nested bitmap files and returns it as a AmigaBitmapFontSet().

It can also read *.font files from virtual disks ((adf_file_con())) objects, but that requires the adfExplorer package to be installed.

Value

Returns an AmigaBitmapFontSet() object read from the specified file.

Author(s)

Pepijn de Vries

See Also

Other AmigaBitmapFont.operations: AmigaBitmapFont, availableFontSizes(), c(), fontName(), font_example, getAmigaBitmapFont(), rasterToAmigaBitmapFont(), rawToAmigaBitmapFontSet(), rawToAmigaBitmapFont(), read.AmigaBitmapFont(), write.AmigaBitmapFont()

Other io.operations: read.AmigaBasicBMAP(), read.AmigaBasicShape(), read.AmigaBasic(), read.AmigaBitmapFont(), read.AmigaIcon(), read.SysConfig(), read.iff(), write.AmigaBasicShape(), write.AmigaBasic(), write.AmigaBitmapFont(), write.AmigaIcon(), write.SysConfig(), write.iff()

Examples

```
data(font_example)

## in order to read, we first need to write a file"
write.AmigaBitmapFontSet(font_example, tempdir())

## The font is written as 'AmigaFFH.font' as that name
## is embedded in the AmigaBitmapFontSet object 'font_example'.
## We can read it as follows:
font.read <- read.AmigaBitmapFontSet(file.path(tempdir(), "AmigaFFH.font"))

## similarly, the file can also be written and read from and to
## a virtual amiga disk. The following codes requires the 'adfExplorer'
## package:
if (requireNamespace("adfExplorer")) {
  library("adfExplorer")
  virtual_disk_file <- tempfile(fileext = ".adf") |>
    create_adf_device(write_protected = FALSE) |>
    prepare_adf_device("font_disk") |>
    make_adf_dir("FONTS")

  dest <- virtual_path(virtual_disk_file, "DF0:FONTS")
  write.AmigaBitmapFontSet(font_example, dest)
  font.read <- read.AmigaBitmapFontSet(
```

```
    virtual_path(virtual_disk_file, "DF0:FONTS/AmigaFFH.font")
  )
  close(virtual_disk_file)
}
```

`read.AmigaIcon` *Read an Amiga Workbench icon (info) file*

Description

Graphical representation of files and directories (icons) are stored as separate files (with the .info extension) on the Amiga. This function reads such files and imports them as `AmigaIcon()` class objects.

Usage

```
read.AmigaIcon(file, ...)
```

Arguments

`file`	A `character` string representing the file name from which the icon data should be read.
`...`	Arguments passed on to `rawToAmigaIcon()`.

Details

The `AmigaIcon()` S3 object provides a comprehensive format for Amiga icons, which are used as a graphical representation of files and directories on the Amiga. The `AmigaIcon()` is a named list containing all information of an icon. Use this function to read an Amiga icon (with the .info extension) from a file and convert it into an `AmigaIcon()` object.

Value

Returns an `AmigaIcon()` class object as read from the `file`.

Author(s)

Pepijn de Vries

See Also

Other AmigaIcon.operations: `AmigaIcon`, `rawToAmigaIcon()`, `simpleAmigaIcon()`, `write.AmigaIcon()`

Other io.operations: `read.AmigaBasicBMAP()`, `read.AmigaBasicShape()`, `read.AmigaBasic()`, `read.AmigaBitmapFontSet()`, `read.AmigaBitmapFont()`, `read.SysConfig()`, `read.iff()`, `write.AmigaBasicShape()`, `write.AmigaBasic()`, `write.AmigaBitmapFont()`, `write.AmigaIcon()`, `write.SysConfig()`, `write.iff()`

Examples

```
## create a simple AmigaIcon:
icon <- simpleAmigaIcon()

## write the icon to the temp dir:
write.AmigaIcon(icon, file.path(tempdir(), "icon.info"))

## read the same file:
icon2 <- read.AmigaIcon(file.path(tempdir(), "icon.info"))
```

---

read.iff *Read Interchange File Format (IFF)*

Description

Read the Interchange File Format (IFF) as an IFFChunk() object.

Usage

```
read.iff(file)
```

Arguments

file	A filename of an IFF file to be read, or a connection from which binary data can be read.

Details

Information is stored as 'chunks' in IFF files (see IFFChunk()). Each chunk should at least contain a label of the type of chunk and the data for that chunk. This function reads all chunks from a valid IFF file, including all nested chunks and stores them in an IFFChunk() object. IFF files can hold any kind of data (e.g. images or audio), this read function does not interpret the file. Use interpretIFFChunk() for that purpose.

Value

Returns a IFFChunk() object read from the specified file.

Author(s)

Pepijn de Vries

See Also

Other io.operations: read.AmigaBasicBMAP(), read.AmigaBasicShape(), read.AmigaBasic(), read.AmigaBitmapFontSet(), read.AmigaBitmapFont(), read.AmigaIcon(), read.SysConfig(), write.AmigaBasicShape(), write.AmigaBasic(), write.AmigaBitmapFont(), write.AmigaIcon(), write.SysConfig(), write.iff()

Other iff.operations: IFFChunk-class, WaveToIFF(), as.raster.AmigaBasicShape(), getIFFChunk(), interpretIFFChunk(), rasterToIFF(), rawToIFFChunk(), write.iff()

Examples

```
## let's read a bitmap image stored in IFF as provided with this package:
filename <- system.file("ilbm8lores.iff", package = "AmigaFFH")
example.iff <- read.iff(filename)

## And plot it:
plot(example.iff)
```

`read.SysConfig` *Read an Amiga system-configuration file*

Description

Read a binary Amiga system-configuration file and return as SysConfig object.

Usage

```
read.SysConfig(file)
```

Arguments

`file`	The file name of a system-configuration file to be read. Can also be a connection that allows reading binary data.

Details

Amiga OS 1.x stored system preferences in a binary system-configuration file. This function returns the file in a comprehensive format (a SysConfig object).

Value

Returns an S3 SysConfig class object based on the file that is read.

Author(s)

Pepijn de Vries

See Also

Other SysConfig.operations: `SysConfig`, `rawToSysConfig()`, `simpleSysConfig()`, `write.SysConfig()`

Other io.operations: `read.AmigaBasicBMAP()`, `read.AmigaBasicShape()`, `read.AmigaBasic()`, `read.AmigaBitmapFontSet()`, `read.AmigaBitmapFont()`, `read.AmigaIcon()`, `read.iff()`, `write.AmigaBasicShape()`, `write.AmigaBasic()`, `write.AmigaBitmapFont()`, `write.AmigaIcon()`, `write.SysConfig()`, `write.iff()`

Examples

```
## Put a simple SysConfig object into the tempdir:
write.SysConfig(simpleSysConfig(), file.path(tempdir(), "system-configuration"))

## Now read the same file:
sc <- read.SysConfig(file.path(tempdir(), "system-configuration"))

## and plot it
plot(sc)
```

`simpleAmigaIcon` *Create simple AmigaIcon objects*

Description

Graphical representation of files and directories (icons) are stored as separate files (with the .info extension) on the Amiga. This function writes `AmigaIcon()` class objects to such files.

Usage

```
simpleAmigaIcon(
  version = c("OS1.x", "OS2.x"),
  type = c("WBDISK", "WBDRAWER", "WBTOOL", "WBPROJECT", "WBGARBAGE", "WBDEVICE",
    "WBKICK", "WBAPPICON"),
  two.images = TRUE,
  back.fill = FALSE,
  ...
)
```

Arguments

`version`	A `character` string indicating the Amiga OS version with which the icon should be compatible. "`OS2.x`" indicates \>=OS2.0 and "`OS1.x`" indicates <OS2.0.
`type`	A `character` string indicating the type of object (file, disk, directory, etc.) the icon should represent. See the 'Usage' section for all posible options.
`two.images`	A single `logical` value, indicating whether the selected icon is depicted as a second image (in which case the icon contains two images). The default value is `TRUE`.
`back.fill`	A single `logical` value, indicating whether the selected image of the icon should use the `back fill' mode (default). If set to` FALSE complement' mode is used. Note that back fill is not compatible when the icon holds two images. In the `complement' mode, the image colours are inverted when selected. In the` back fill' exterior first colour is not inverted.
`...`	Reserved for additional arguments. Currently ignored.

Details

This function creates basic AmigaIcon() objects which can be modified afterwards. It uses simple generic images to represent different types of files or directories.

Value

Returns a simple S3 object of class AmigaIcon().

Author(s)

Pepijn de Vries

See Also

Other AmigaIcon.operations: AmigaIcon, rawToAmigaIcon(), read.AmigaIcon(), write.AmigaIcon()

Other raw.operations: as.AmigaBasic(), as.raw.AmigaBasic(), colourToAmigaRaw(), packBitmap(), rawToAmigaBasicBMAP(), rawToAmigaBasicShape(), rawToAmigaBasic(), rawToAmigaBitmapFontSet(), rawToAmigaBitmapFont(), rawToAmigaIcon(), rawToHWSprite(), rawToIFFChunk(), rawToSysConfig()

Examples

```
## Create an AmigaIcon object using the default arguments:
icon <- simpleAmigaIcon()
```

---

simpleSysConfig *Function to generate a simple Amiga system-configuration representation*

Description

SysConfig objects are comprehensive representations of binary Amiga system-configuration files. Use this function to create a simple SysConfig object.

Usage

```
simpleSysConfig(options)
```

Arguments

options	A named list with elements of the target SysConfig() object that need to be modified.

Details

The Amiga used the system-configuration file to store certain system preferences in a binary file. In the AmigaFFH package such files can be represented by the more comprehensive SysConfig class object. Use this function to create such an object with basic settings (which can be modified).

Value

Returns a comprehensive representation of a system-configuration file in the for of a SysConfig class object.

Author(s)

Pepijn de Vries

See Also

Other SysConfig.operations: SysConfig, rawToSysConfig(), read.SysConfig(), write.SysConfig()

Examples

```
## Create a simple system-configuration (S3 SysConfigClass)
sc <- simpleSysConfig()

## And modify it as you wish.
## in this case change the setting for the printer
## from the parallel port to the serial port:
sc$PrinterPort <- factor("SERIAL_PRINTER", levels(sc$PrinterPort))

## It is also to provide modifications to the configuration
## via the 'options' argument:
sc <- simpleSysConfig(options = list(FontHeight = 9))
```

SysConfig *The S3 SysConfig class*

Description

A comprehensive representation of an Amiga system-configuration file.

Details

The system-configuration file is a binary file stored in the 'devs' folder of the root of a bootable Amiga DOS device, containing system preferences. It was used in Amiga OS 1.x. Although it could be used in later versions, it was gradually phased out and some settings may not be usable in the later versions. See references below for more details.

Definitions of the system-configuration have file been revised at some points. Revisions are minor and usually targeted at backward compatibility. Here revision V38.2 (released on 16 September 1992) is implemented, which is the latest documented version.

The sytem-configuration file contains settings for the serial and parallel port and the printer. It also contains some settings for the 'workbench' which was the Amiga equivalent of what is now mostly known as the computers desktop. Colours for the workbench and the shape of the mouse pointer are also stored in this file. Settings for the mouse and basic screen resolution are also part of the file.

The SysConfig object is a comprehensive representation of the binary system-configuration file. It is a a `list` where the elements have identical names as listed in the documents provided the references. The names are usually self-explanatory, but the referred documents can also be consulted to obtain more detailed information with respect to each of these elements. The mouse pointer is included as a `hardwareSprite()` object in the list. The pointer image can be replaced by a different `hardwareSprite()`, but make sure it has an height of 16 pixels.

It is possible to change the values of the list, but not all values may be valid. Note that they will not be fully checked for validity. Invalid values may result in errors when writing to a binary file using `write.SysConfig()`, or may simply not work properly on an Amiga or in an emulator.

Use `simpleSysConfig()` for creating a simple SysConfig object which can be modified. Use `read.SysConfig()` to read, and `write.SysConfig()` to write system-configuration files. With `rawToSysConfig()` and `as.raw()` SysConfig can be coerced back and forth from and to its raw (binary) form.

Author(s)

Pepijn de Vries

References

https://wiki.amigaos.net/wiki/Preferences#Preferences_in_1.3_and_Older_Versions_of_the_OS http://amigadev.elowar.com/read/ADCD_2.1/Includes_and_Autodocs_2._guide/node00D5.html http://amigadev.elowar.com/read/ADCD_2.1/Includes_and_Autodocs_3._guide/node063B.html

See Also

Other SysConfig.operations: `rawToSysConfig()`, `read.SysConfig()`, `simpleSysConfig()`, `write.SysConfig()`

---

timeval *Get an Amiga timeval struct value from raw data*

Description

Some Amiga applications use a timeval struct (see references) to represent a time span in seconds. This function coerces raw data to such a numeric time span.

Usage

```
timeval(x)
```

Arguments

x	a `vector` of `raw` data that need to be converted into Amiga timeval structs.

Details

Timeval is a structure (struct) as specified in device/timer.h on the Amiga (see references). It represents a timespan in seconds. This function retrieves the numeric value from `raw` data. Amongst others, the timeval struct was used in the system-configuration file (see SysConfig) to specify key repeat speed, key repeat delay and mouse double click speed. Use `as.raw` for the inverse of this function and get the original raw data.

Value

Returns a `numeric vector` of a timespan in seconds. It is represented as an S3 AmigaTimeVal class.

Author(s)

Pepijn de Vries

References

http://amigadev.elowar.com/read/ADCD_2.1/Includes_and_Autodocs_2._guide/node0053.html

Examples

```
## First four raw values represent seconds, the latter four microseconds:
temp <- timeval(as.raw(c(0, 0, 0, 1, 0, 0, 0, 1)))
print(temp)

## You can use 'as.raw' to get the original raw data again:
as.raw(temp)
```

`WaveToIFF` *Convert WaveMC objects into an Interchange File Format object*

Description

Convert `tuneR::WaveMC()` objects (or objects that can be coerced to `WaveMC` objects) into an `IFFChunk-class()` object which can be stored as a valid Iterchange File Format (`write.iff()`).

Usage

```
WaveToIFF(
  x,
  loop.start = NA,
  octaves = 1,
  compress = c("sCmpNone", "sCmpFibDelta"),
  ...
)
```

Arguments

x	A tuneR::WaveMC() object that needs to be converted into an IFFChunk() object. x can also be any other class object that can be coerced into a tuneR::WaveMC() object. tuneR::Wave() and PTSample() objects are therefore also allowed.
loop.start	If the sample should be looped from a specific position to the end of the sample, this argument specifies the starting position in samples (with a base of 0) for looping. loop.start therefore should be a whole non-negative number. When set to NA or negative values, the sample will not be looped.
octaves	A whole positive numeric value indicating the number of octaves that should be stored in the resulting IFF chunk. The original wave will be resampled for each value larger than 1. Each subsequent octave will contain precisely twice as many samples as the previous octave.
compress	A character string indicating whether compression should be applied to the waveform. "sCmpNone" (default) applies no compression, "sCmpFibDelta" applies the lossy deltaFibonacciCompress()ion.
...	Currently ignored.

Details

tuneR::WaveMC() objects can be read from contemporary file containers with tuneR::readWave() or tuneR::readMP3(). With this function such objects can be converted into an IFFChunk-class() object which can be stored conform the Interchange File Format (write.iff()).

When x is not a pcm formatted 8-bit sample, x will first be normalised and scaled to a pcm-formatted 8-bit sample using tuneR::normalize(). If you don't like the result you need to convert the sample to 8-bit pcm yourself before calling this function.

Value

Returns an IFFChunk-class() object with a FORM container that contains an 8SVX waveform based on x.

Author(s)

Pepijn de Vries

References

https://en.wikipedia.org/wiki/8SVX

See Also

Other iff.operations: IFFChunk-class, as.raster.AmigaBasicShape(), getIFFChunk(), interpretIFFChunk(), rasterToIFF(), rawToIFFChunk(), read.iff(), write.iff()

Examples

```
## First get an audio sample from the ProTrackR package
snare.samp <- ProTrackR::PTSample(ProTrackR::mod.intro, 2)

## The sample can easily be converted into an IFFChunk:
snare.iff <- WaveToIFF(snare.samp)

## You could also first convert the sample into a Wave object:
snare.wav <- as(snare.samp, "Wave")

## And then convert into an IFFChunk. The result is the same:
snare.iff <- WaveToIFF(snare.wav)

## You could also use a sine wave as input (although you will get some warnings).
## This will work because the vector of numeric data can be coerced to
## a WaveMC object
# sine.iff <- WaveToIFF(sin((0:2000)/20))
```

---

write.AmigaBasic *Write an AmigaBasic object to a file*

Description

Write an AmigaBasic() class object to a file in its binary format.

Usage

```
write.AmigaBasic(x, file)
```

Arguments

x	The AmigaBasic() class object that needs to be stored.
file	A character string specifying the file location to which x (an AmigaBasic() object) needs to be written.

Details

This function encodes the Amiga Basic code in its binary format (using as.raw()) and writes it to a file. The file can also be stored onto a virtual Amiga disk (adf_file_con()).

Value

Invisibly returns the result of the call of close to the file connection.

Author(s)

Pepijn de Vries

See Also

Other AmigaBasic.operations: AmigaBasic.reserved(), AmigaBasicBMAP, AmigaBasic, [.AmigaBasic(), as.AmigaBasicBMAP(), as.AmigaBasic(), as.character(), check.names.AmigaBasic(), names.AmigaBasic(), rawToAmigaBasicBMAP(), rawToAmigaBasic(), read.AmigaBasicBMAP(), read.AmigaBasic()

Other io.operations: read.AmigaBasicBMAP(), read.AmigaBasicShape(), read.AmigaBasic(), read.AmigaBitmapFontSet(), read.AmigaBitmapFont(), read.AmigaIcon(), read.SysConfig(), read.iff(), write.AmigaBasicShape(), write.AmigaBitmapFont(), write.AmigaIcon(), write.SysConfig(), write.iff()

Examples

```
## First create an AmigaBasic object:
bas <- as.AmigaBasic("PRINT \"hello world!\"")

## write to tempdir:
write.AmigaBasic(bas, file.path(tempdir(), "helloworld.bas"))
```

---

write.AmigaBasicShape *Write an AmigaBasicShape object to a file*

Description

Write an AmigaBasicShape() class object to a file in its binary format.

Usage

```
write.AmigaBasicShape(x, file)
```

Arguments

x	The AmigaBasicShape() class object that needs to be stored.
file	A character string specifying the file location to which x (an AmigaBasicShape() object) needs to be written.

Details

This function coerces the Amiga Basic Shape into its binary format (using as.raw()) and writes it to a file. The file can also be stored onto a virtual Amiga disk (adf_file_con()).

Value

Invisibly returns the result of the call of close to the file connection.

Author(s)

Pepijn de Vries

See Also

Other AmigaBasicShape.operations: AmigaBasicShape, rasterToAmigaBasicShape(), read.AmigaBasicShape()

Other io.operations: read.AmigaBasicBMAP(), read.AmigaBasicShape(), read.AmigaBasic(), read.AmigaBitmapFontSet(), read.AmigaBitmapFont(), read.AmigaIcon(), read.SysConfig(), read.iff(), write.AmigaBasic(), write.AmigaBitmapFont(), write.AmigaIcon(), write.SysConfig(), write.iff()

Examples

```
filename <- system.file("ball.shp", package = "AmigaFFH")
ball     <- read.AmigaBasicShape(filename)
write.AmigaBasicShape(ball, file.path(tempdir(), "ball.shp"))
```

---

write.AmigaBitmapFont *Write an AmigaBitmapFont(set) file*

Description

Functions to write AmigaBitmapFont() and AmigaBitmapFontSet() class objects to files.

Usage

```
write.AmigaBitmapFont(x, file)

write.AmigaBitmapFontSet(x, path = getwd())
```

Arguments

x	Respectively an AmigaBitmapFont() or a AmigaBitmapFontSet() object depending on which of the write-functions is called. This is the object that will be written to the specified file.
file	A character string specifying the file location to which x (an AmigaBitmapFont() object) needs to be written. It is common practice on the Amiga to use the font height in pixels as file name.
path	A character string specifying the path where x (an AmigaBitmapFontSet() object) needs to be stored. The filename for the font set will be extracted from x using fontName() followed by the *.font extension. A subdirectory will be created with the same name (without the extension) if it doesn't already exists. In this subdirectory all the nested AmigaBitmapFont() objects are stored.

Details

AmigaBitmapFontSet() class objects are written to a *.font file. The filename used for this purpose is obtained from the object itself using fontName(). In addition, a subdirectory is created automatically (when it doesn't already exist) to which al the separate bitmap images for each font height are written to individual files.

AmigaBitmapFont() class objects can also be written to a file. In order to use it on a Commodore Amiga or emulator, it is better to embed the font bitmap in a font set (using c()) and write the set to corresponding files.

Value

Invisibly returns the result of the call of close to the file connection.

Author(s)

Pepijn de Vries

See Also

Other AmigaBitmapFont.operations: AmigaBitmapFont, availableFontSizes(), c(), fontName(), font_example, getAmigaBitmapFont(), rasterToAmigaBitmapFont(), rawToAmigaBitmapFontSet(), rawToAmigaBitmapFont(), read.AmigaBitmapFontSet(), read.AmigaBitmapFont()

Other io.operations: read.AmigaBasicBMAP(), read.AmigaBasicShape(), read.AmigaBasic(), read.AmigaBitmapFontSet(), read.AmigaBitmapFont(), read.AmigaIcon(), read.SysConfig(), read.iff(), write.AmigaBasicShape(), write.AmigaBasic(), write.AmigaIcon(), write.SysConfig(), write.iff()

Examples

```
## obtain a bitmap font set:
data(font_example)

## write the font set to their files. The file name
## is extracted from the font object, so you only have
## to provide the path:
write.AmigaBitmapFontSet(font_example, tempdir())

## extract a font bitmap:
font <- getAmigaBitmapFont(font_example, 9)

## and write it to the temp dir:
write.AmigaBitmapFont(font, file.path(tempdir(), "9"))

## The following examples require the 'adfExplorer' package:
if (requireNamespace("adfExplorer")) {
  library("adfExplorer")
  virtual_disk_file <- tempfile(fileext = ".adf") |>
    create_adf_device(write_protected = FALSE) |>
    prepare_adf_device("font_disk") |>
    make_adf_dir("FONTS")

  dest <- virtual_path(virtual_disk_file, "DF0:FONTS")
  write.AmigaBitmapFontSet(font_example, dest)
  close(virtual_disk_file)
}
```

write.AmigaIcon *Write an Amiga Workbench icon (info) file*

Description

Graphical representation of files and directories (icons) are stored as separate files (with the .info extension) on the Amiga. This function writes AmigaIcon() class objects to such files.

Usage

```
write.AmigaIcon(x, file)
```

Arguments

x	An AmigaIcon() class object.
file	A character string representing the file name to which the icon data should be written.

Details

The AmigaIcon() S3 object provides a comprehensive format for Amiga icons, which are used as a graphical representation of files and directories on the Amiga. The AmigaIcon() is a named list containing all information of an icon. Use this function to write this object to a file which can be used on the Commodore Amiga or emulator.

Value

Returns NULL or an integer status passed on by the close() function, that is used to close the file connection. It is returned invisibly.

Author(s)

Pepijn de Vries

See Also

Other AmigaIcon.operations: AmigaIcon, rawToAmigaIcon(), read.AmigaIcon(), simpleAmigaIcon()

Other io.operations: read.AmigaBasicBMAP(), read.AmigaBasicShape(), read.AmigaBasic(), read.AmigaBitmapFontSet(), read.AmigaBitmapFont(), read.AmigaIcon(), read.SysConfig(), read.iff(), write.AmigaBasicShape(), write.AmigaBasic(), write.AmigaBitmapFont(), write.SysConfig(), write.iff()

Examples

```
## create a simple AmigaIcon:
icon <- simpleAmigaIcon()

## write the icon to the temp dir:
write.AmigaIcon(icon, file.path(tempdir(), "icon.info"))
```

`write.iff` *Write Interchange File Format (IFF)*

Description

Write an `IFFChunk()` object conform the Interchange File Format (IFF).

Usage

```
write.iff(x, file)
```

Arguments

`x`	An `IFFChunk()` object that needs to be written to a file.
`file`	A filename for the IFF file to which the `IFFChunk()` needs to be saved, or a connection to which the data should be written.

Details

Writes an `IFFChunk()` object (including all nested chunks) to the specified file. Only the structure of the object needs to be valid, however, a correctly structured file does not necessarily result in an interpretable file (see examples).

Value

Returns either `NULL` or an `integer` status invisibly as passed by the `close()` statement used to close the file connection.

Author(s)

Pepijn de Vries

References

https://en.wikipedia.org/wiki/Interchange_File_Format

See Also

Other io.operations: read.AmigaBasicBMAP(), read.AmigaBasicShape(), read.AmigaBasic(), read.AmigaBitmapFontSet(), read.AmigaBitmapFont(), read.AmigaIcon(), read.SysConfig(), read.iff(), write.AmigaBasicShape(), write.AmigaBasic(), write.AmigaBitmapFont(), write.AmigaIcon(), write.SysConfig()

Other iff.operations: IFFChunk-class, WaveToIFF(), as.raster.AmigaBasicShape(), getIFFChunk(), interpretIFFChunk(), rasterToIFF(), rawToIFFChunk(), read.iff()

Examples

```
## read an IFF file as an IFFChunk object:
example.iff <- read.iff(system.file("ilbm8lores.iff", package = "AmigaFFH"))

## This will write the IFF file (in this case a bitmap image)
## to the temp directory:
write.iff(example.iff, file.path(tempdir(), "image.iff"))
```

---

write.SysConfig *Write an Amiga system-configuration file*

Description

Write a SysConfig class object to an Amiga binary system-configuration file.

Usage

```
write.SysConfig(x, file)
```

Arguments

x	An S3 SysConfig class object.
file	A file name to which the binary file should be written.

Details

Amiga OS 1.x stored system preferences in a binary system-configuration file. This function writes a SysConfig class object as such a binary file. This file can be used on an Amiga or in an emulator.

Value

Returns NULL or an integer status passed on by the close() function, that is used to close the file connection. It is returned invisibly.

Author(s)

Pepijn de Vries

See Also

Other SysConfig.operations: SysConfig, rawToSysConfig(), read.SysConfig(), simpleSysConfig()

Other io.operations: read.AmigaBasicBMAP(), read.AmigaBasicShape(), read.AmigaBasic(), read.AmigaBitmapFontSet(), read.AmigaBitmapFont(), read.AmigaIcon(), read.SysConfig(), read.iff(), write.AmigaBasicShape(), write.AmigaBasic(), write.AmigaBitmapFont(), write.AmigaIcon(), write.iff()

Examples

```
## First generate a simple SysConfig object to write to a file:
sc <- simpleSysConfig()

## And write to the tempdir:
write.SysConfig(sc, file.path(tempdir(), "system-configuration"))
```

[.AmigaBasic *Extract or replace lines of Amiga Basic code*

Description

Extract or replace lines of Amiga Basic code

Usage

```
## S3 method for class 'AmigaBasic'
x[i]

## S3 replacement method for class 'AmigaBasic'
x[i] <- value

## S3 method for class 'AmigaBasic'
x[[i]]

## S3 replacement method for class 'AmigaBasic'
x[[i]] <- value
```

Arguments

x	An AmigaBasic class object from which specific lines need to be extracted or replaced.
i	In case of [[', an integer index, representing the line-number of basic code to be selected a vector of numeric indices. This index is used to select specific lines. Negative values will deselect lines.
value	A vector of character strings or an AmigaBasic() class object that is used to replace the selected indices i. value should represent the same number of lines of code as the selected number of lines.

Details

Extract or replace specific lines in an AmigaBasic()-class object.

Value

The extraction method returns an AmigaBasic() object based in the lines selected with i. The replacement method returns an AmigaBasic() object with the selected lines replaced with value.

Author(s)

Pepijn de Vries

See Also

Other AmigaBasic.operations: AmigaBasic.reserved(), AmigaBasicBMAP, AmigaBasic, as.AmigaBasicBMAP(), as.AmigaBasic(), as.character(), check.names.AmigaBasic(), names.AmigaBasic(), rawToAmigaBasicBMAP(), rawToAmigaBasic(), read.AmigaBasicBMAP(), read.AmigaBasic(), write.AmigaBasic()

Examples

```
## First generate a few lines of Basic code:
bas <- as.AmigaBasic(c(
  "LET a = 1",
  "a = a + 1",
  "PRINT \"a now equals\";a",
  "INPUT \"clear screen (y/n)? \", b$",
  "IF UCASE$(b$) = \"Y\" THEN CLS"
))

## Select only lines 4 and 5:
bas[4:5]

## use negative indices to deselect specific lines.
## deselect line 2:
bas[-2]

## replace line 2
bas[2] <- "a = a + 2"

## You can also use AmigaBasic class object as replacement
bas[2] <- as.AmigaBasic("a = a + 3")

## single lines can also be selected with '[['
bas[[2]]
```

Index