Package 'BuildSys'

October 12, 2022

Type Package

Title System for Building and Debugging C/C++ Dynamic Libraries

Version 1.1.2

Date 2021-06-11

Maintainer Paavo Jumppanen <paavo.jumppanen@csiro.au>

Author Paavo Jumppanen [aut, cre]

Description A build system based on 'GNU make' that creates and maintains (simply) make files in an R session and provides GUI debugging support through 'Microsoft Visual Code'.

License GPL-2

URL https://github.com/pjumppanen/BuildSys

BugReports https://github.com/pjumppanen/BuildSys/issues

Imports methods, digest

SystemRequirements 'Microsoft Visual Code', 'Rtools' on 'Windows', 'gdb' or 'lldb' depending on which OS

NeedsCompilation no

Repository CRAN

Date/Publication 2021-06-16 22:40:02 UTC

R topics documented:

BuildSys-package 2
BSysProject-class 9
BSysSourceFile-class 11
buildMakefile 12
clean 13
includePath 13
initProjectFromFolder 14
installIncludePath 17

installLibraryPath 18
libraryPath 19
loadLibrary 19
make 20
objPath 21
sourcePath 22
unloadLibrary 23
vcDebug 24

Index **26**

`BuildSys-package` *System for Building and Debugging C/C++ Dynamic Libraries*

Description

A build system based on 'GNU make' that creates and maintains (simply) make files in an R session and provides GUI debugging support through 'Microsoft Visual Code'.

Details

In the standard *R* approach dynamic libraries are typically built using the *R CMD* interface. That interface in turn uses the *GNU make* system to build the library through a generic makefile bundled with *R*. This is presumably done to simplify the process of building dynamic libraries for the less experience. The downside however, is adding complexity to the process when doing more esoteric programming that makes use of static or dynamic libraries outside of those bundled with *R*. In those cases it is likely that we may need to specify compiler include options to be able to successfully compile the code but *R CMD SHLIB* provides no direct method of acomplishing this task. Instead we have to create a *Makevars* file that defines the variables `PKG_CFLAGS`, `PKG_CXXFLAGS` or `PKG_FFLAGS` to achieve the desired outcome.

It is also common to have build issues with libraries not linking because of missing dependencies or link order issues (libraries must be linked in the correct order with *gcc* to ensure error free linking). Resolving these link related issues is made more complex through the *R CMD* interface because it is no longer possible to simply look at the makefile definition and make modifications to it in an attempt to resolve the build problem. By having a system approach whereby makefiles are the centre of the build process and not obfuscated away in an R installation folder, the process of debugging build issues is made simpler by virtue of the fact that all necessary changes to fix the build are in one file only, and a simple one at that.

Lastly, a major part of developing library code is debugging and standard *R* provides little support for that. Developers typically rely on *gdb / lldb* command line driven debugger support, which is daunting for an inexperienced user, so much so that many will simply resort to print statement based debugging approaches, which whilst capable of resolving bugs, is a cumbersome and slow approach. What is required is an easy to use and full featured GUI based debugger and to that end, *Microsoft Visual Code* coupled with the machine interface (/MI) support in *gdb / lldb* provide the perfect solution.

BuildSys provides a simple but complete and flexible way to create project makefiles, build libraries and launch GUI based debug sessions, all from within *R*. The makefiles it creates are permanent rather than transitory so in the event of build troubles it is easy to resort to tinkering with the makefile to resolve the cause of the build problem and then reflect the necessary fixes in the *R* based project definition.

Project Creation: BuildSys encapsulates a C/C++ dynamic library project definition into an S4 class BSysProject. To create a new project we simply create a new instance of that class and BuildSys will construct the appropriate project definition. For example,

```
Project <- new("BSysProject", "./MyFolder")
```

will search the directory ./MyFolder for source files (.c, .cpp etc) and add them to the project definition. In searching for source files it will also scan the found source files for include statements and any found includes will be saved as dependencies for the give source file. No attempt is made to properly parse the source so if the code uses conditional preprocessing statements the situation may arrise that an include file is marked as a dependency when for the given build configuration it may not be. In any case, this will not affect the ability to correctly build the library but will just mean the dependencies are marked wider than true.

Another feature of the dependency check is that any include files not found are considered externalities and if they belong to a known set, will result in the automatic addition of library and include path dependencies for the project. Currently BuildSys knows about TMB.hpp, Rcpp.hpp and RcppEigen.hpp, so if any of these are included into the source file then the necessary include file and define dependencies will be added. This added feature means *TMB* and *Rcpp* users have minimalistic usage requirements to construct libraries with BuildSys and need not provide any additional information other than the working folder and possibly the source files being compiled.

A typical project will also need to be given a name but if there is only one source file then the name can be omitted and is inferred from the filename of the source file. For instance, we can explicitly name the project with,

```
Project <- new("BSysProject", WorkingFolder="./MyFolder", Name="MyProject")
```

A WorkingFolder must be provided which names the folder where the source code resides and can be either an absolute path or one relative to the *R* working directory.

Finally, if our project is hierarchical in its structure (eg. having source in one folder, header files in another folder, and binary output in yet another) BuildSys can accomodate that too. In such case we set Flat=FALSE with,

```
Project <- new("BSysProject", WorkingFolder="./MyFolder", Name="MyProject", Flat=FALSE)
```

For this case the relevent folders are,

```
MyFolder/src
MyFolder/include
MyFolder/obj
```

These sub folder names are the default behaviour. If these sub folder names are not as required they can be renamed using the SourceName, IncludeName and ObjName arguments in the BSysProject initilizer. That is,

```
Project <- new("BSysProject", WorkingFolder="./MyFolder", Name="MyProject", Flat=FALSE,
SourceName="source", IncludeName="header")
```

Given an existing project, we can also re-initialize it using the initProjectFromFolder method which has the same argument list as the initialize method (it is actually called from the initialize method). For instance, as with the above,

```
Project <- initProjectFromFolder(Project, WorkingFolder="./MyFolder", Name="MyProject",
Flat=FALSE, SourceName="source", IncludeName="header")
```

Project Compilation: Once we have constructed a `Project` object compilation is simply a matter of calling the `make` method. As with traditional GNU makefiles, calling `make` with no arguments compiles the dynamic library. Calling make with a `"clean"` argument erases all object files and the built dynamic library. For example,

```
make(Project) # Compiles the source files and links the dynamic library

make(Project, "clean") # deletes all .o object files and the dynamic library
```

Since the makefile is permanent a subsequent call to `make` for an already built project will return immediately as the library is already present and up to date. If a source file is altered then only the files with stale dependencies will be recompiled and linked into a new dynamic library build.

Internally `make` will call the `buildMakefile` method to first construct an up to date makefile representing the project. Each makefile has an *md5 digest* in it which will be updated should you alter the project at all and this stamp is then used to determine if the makefile needs re-construction. If the makefile is re-constructed during a call to `make` then a make clean operation will be carried out to ensure the entire project is re-built. This should ensure that the state of the project, makefile and the build remains in sync.

Finally, if your project needs to install the library and/or include files to a specific location, that behaviour is also catered for in `BuildSys`. Simply specify the install locations with the `InstallLibraryName` and `InstallIncludeName` arguments in the `BSysProject` initializer. For instance,

```
Project <- new("BSysProject", WorkingFolder="./MyFolder", Flat=F, InstallLibraryName="inst/lib",
InstallIncludeName="inst/include")
```

Loading and Unloading the Library: Once the project is made using the `make` method the dynamic library can be loaded into *R* memory. We do so using the `loadLibrary` method and we can similarly unload the library using the `unloadLibrary` method. The library name is based upon the project name and the path to the library can be obtained with the `libraryPath` method. For example,

```
loadLibrary(Project) # load the library built by Project

unloadLibrary(Project) # unload the library built by Project

libraryPath(Project) # return the path to the library built by Project
```

Debugging the Library: To debug a dynamic library a debug build must be made to include the symbolic information needed with the dynamic library. By default, projects constructed using `BuildSys` will be created as debug builds. This is controlled by the `Debug` argument supplied in the call to the `BSysProject` initilizer. For instance,

```
Project <- new("BSysProject", Debug=TRUE)
```

creates a debug enabled project whereas,

```
Project <- new("BSysProject", Debug=FALSE)
```

create a release / optimised project. For an existing project we can make a debug build by supplying the `Debug` argument to the call to the `make` method as with,

```
Project <- make(Project, Debug=TRUE)
```

Recall that for *S4 class methods* to change the object state we need to assign the returned modified object to the original one, which is why the above example assigns the `make` call to `Project`.

Debugging is handled through *Microsoft Visual Code* and *gdb / lldb* and it is assumed that these components are installed and functional. For information regarding the installation of these components consult the next section.

To debug an existing dynamic library (built from a `BSysProject`) simply call the `vcDebug` method as follows.

```
vcDebug(Project) # Debug the library created by project
```

Calling this method will open an instance of *Microsoft Visual Code* that is correctly initialised to debug your library code. Within *Visual Code* you can open the source file/s of your library, set breakpoints and run a debug session. To run a debug session select the *run/Start Debugging* menu item. Doing so will result in a new instance of *R* being launched which contains the same environment (including loaded packages and loaded dynamic libraries) as the parent *R* session where `vcDebug` was first called. This *R* session is your *sandbox* to safely debug your library in and leaves the parent *R* session safe from loss should your code crash *R* completely.

Typically when debugging a new library the library requires *R* setup code to initialise as a foundation for library use and/or testing. With `BuildSys` you should perform this initial setup of *R* in the parent *R* session before calling `vcDebug` so that you never have to initialise *R* in your debug session. Its correct state will have been initialised from the parent *R* session.

Software Installation Requirements for Debugging: In order to debug code with `BuildSys` *Microsoft Visual Code* must be installed. You can download and install this software from here:

Download Visual Studio Code

Choose the appropriate installer for your operating system. On Windows and Linux the installer should include *code* in the `PATH` environment variable. Verify this by opening a new shell/command prompt and typing,

```
code
```

and enter. If *Visual Code* starts then the `PATH` enviroment variable is correctly set. If it fails to start then add it to the system `PATH` environment variable.

On MacOS the application name is `Visual Studio Code.app` and is an application bundle. For it to be usable by *BuildSys* drag the *Visual Studio Code* application bundle to the *Applications* folder using *Finder*.

On *Windows* you will have to install *Rtools* which you can find here:

Using Rtools40 on Windows

You will need to add *R* and *Rtools* to the system *Path*. For instance add,

```
C:\rtools40\mingw64\bin\
C:\rtools40\usr\bin\
C:\Program Files\R\R-4.0.1\bin\x64\
```

but be aware that the locations may differ in your case depending on which version of *R* and *Rtools* and where these products were installed. To set environment variables in *Windows 10*,

1. Open the Start Search, type in "env", and choose "Edit the system environment variables":
2. Click the "Environment Variables..." button.
3. Set the environment variables as needed. The New button adds an additional variable.
4. Dismiss all of the dialogs by choosing "OK". Your changes are saved!

In Windows after installing *Rtools* we also need to install *gdb*. To install *gdb*, open an *MSYS2* shell (In explorer migrate to the Rtools40 folder and double click on *msys2.exe*) and enter the following commands,

```
pacman -Sy
pacman -S mingw-w64-{i686,x86_64}-gdb
```

After installing verify that *gdb* is installed by opening a new *DOS box* and typing,

```
gdb -v
```

If all is correct it should respond with something like,

```
GNU gdb (GDB) 8.3.1
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

On *MacOS / OS X* you will need to install *Xcode* to have access to both *clang* and *lldb*. Go to *Mac App Store* and download and install *Xcode*.

On *Linux* you will need to have *gcc* installed. Installation of *gcc* on *Linux* is omitted because of the many and varied ways to do so depending on the distribution being used. Please consult *Google* on this topic for more information on installing development tools on *Linux*.

On *Windows* and *Linux*, you will also need to install the *Microsoft C/C++ IntelliSense, debugging, and code browsing* extension in *Microsoft Visual Code*. To install, press *Ctrl+Shift+X* and search for *C/C++* and then click on *Install* beside the *Microsoft C/C++ IntelliSense, debugging, and code browsing* entry in the list. On *MacOS / OS X* you will need to install the *Microsoft C/C++ IntelliSense, debugging, and code browsing* and *CodeLLDB* extensions. To install press *CMD+Shift+X* and search for *C/C++* and then click on *Install* beside the *Microsoft C/C++ IntelliSense, debugging, and code browsing* entry in the list, then also search for *CodeLLDB* and click on the *Install* next to *Native debugger based on LLDB* in the list.

On *MacOS (OS X 10.11 and later) Apple* introduced *System Integrity Protection* which restricts the ability to debug your code from within an R session. To debug code in R using *BuildSys* and *CodeLLDB* will require that the debugging restrictions are disabled. To do so shutdown *MacOS*, then press and hold down *CMD+R* keys while pressing the power button and keep them depressed until you see a language selection dialog. After selecting the language you should see the *Utilities Window*. In the menu bar at the top of the screen select the *utilities/Terminal* menu and in the terminal session enter the follow command:

```
csrutil enable --without debug
```

You will be prompted for your password. Enter your password to carry out the command. If successful you will be presented with a warning about it being a non-standard configuration. Now re-boot your Mac by selecting the re-boot option from the Apple menu at the top of the screen. You should now be able to debug your R dynamic libraries with *BuildSys*.

Note

If the absolute path to source and dynamic library files contains spaces then debugging in *Visual Code* fails to function correctly. To ensure code is debuggable users will need to avoid putting code in folders with whitespace in the folder names.

Author(s)

Paavo Jumppanen [aut, cre]

Maintainer: Paavo Jumppanen <paavo.jumppanen@csiro.au>

See Also

Debug C++ in Visual Studio Code Disabling and Enabling System Integrity Protection Enabling parts of System Integrity Protection while disabling specific parts? make buildMakefile vcDebug loadLibrary unloadLibrary libraryPath sourcePath includePath objPath installLibraryPath installIncludePath clean

Examples

```
ProjectFolder <- tempdir()

# Create source file for finite convolution example in "Writing R Extensions"
lines <- c(
"#include <R.h>",
"#include <Rinternals.h>",
"",
"SEXP convolve2(SEXP a, SEXP b)",
"{",
"    int na, nb, nab;",
"    double *xa, *xb, *xab;",
"    SEXP ab;",
"",
"    a = PROTECT(coerceVector(a, REALSXP));",
"    b = PROTECT(coerceVector(b, REALSXP));",
"    na = length(a); nb = length(b); nab = na + nb - 1;",
"    ab = PROTECT(allocVector(REALSXP, nab));",
"    xa = REAL(a); xb = REAL(b); xab = REAL(ab);",
"    for(int i = 0; i < nab; i++) xab[i] = 0.0;",
"    for(int i = 0; i < na; i++)",
"        for(int j = 0; j < nb; j++) xab[i + j] += xa[i] * xb[j];",
"    UNPROTECT(3);",
"    return ab;",
"}")

SourceFilePath <- paste0(ProjectFolder, "/convolve.c")
writeLines(lines, SourceFilePath)

# digest need not be loaded but the digest package needs to be installed
# as it is used to create a digest of the project to track the need for
# makefile re-creation.
require(BuildSys)

# create project to build shared library, a flat project with source in current working directory.
Project <- new("BSysProject", ProjectFolder)

# re-initialise project from current working directory, new("BSysProject") calls this internally
Project <- initProjectFromFolder(Project, ProjectFolder)
```

```
# build the shared library
make(Project)

# get project library path
libraryPath(Project)

# get project source path
sourcePath(Project)

# get project include path
includePath(Project)

# get project object path
objPath(Project)

# get project install library path
installLibraryPath(Project)

# get project install include path
installIncludePath(Project)

# load the library
loadLibrary(Project)

# R wrapper on .Call
conv <- function(a, b) .Call("convolve2", a, b)

# Test data
a <- rnorm(100)
b <- rep(1.0, times=10)

# call the shared library function
conv(a, b)

## Not run:
# open a debug session - assumes Visual Studio Code is installed as directed in the
# package documentation. This will open a Visual Studio Code session. In that session
# open the convolve.c source file and set breakpoints, then select the "run/start debugging"
# menu to debug. This will start a new R session with the same state as the parent
# R session where vcDebug() is first called. In the new debug session run the following
# command:
#
# conv(a, b)
vcDebug(Project)

## End(Not run)

# unload the shared library
unloadLibrary(Project)

# clean up example
make(Project, "clean")
```

```
clean(Project)
unlink(SourceFilePath)
```

BSysProject-class *Class* "BSysProject"

Description

BSysProject implements a build system based on GNU make that creates and maintains (simply) makefiles in an R session and provides GUI debugging support through Microsoft Visual Code.

Objects from the Class

Objects can be created by calls of the form:

```
new("BSysProject",
    WorkingFolder,
    Name,
    SourceFiles,
    SourceName,
    IncludeName,
    ObjName,
    InstallLibraryName,
    InstallIncludeName,
    Flat,
    Packages,
    Includes,
    Defines,
    Libraries,
    CFLAGS,
    CXXFLAGS,
    FFLAGS,
    LDFLAGS,
    LDLIBS,
    DEFINES,
    Debug)
```

For more information consult initProjectFromFolder.

Slots

WorkingFolder: Object of class "character" that is the absolute path to the root folder of the project.

ProjectName: Object of class "character" that is the name given to the project and will form the basename of the shared library.

SourceName: Object of class "character" that is the sub-path / sub-drectory of the source folder (containing .c, .cpp files) in the project.

`IncludeName`: Object of class `"character"` that is the sub-path / sub-drectory of the include folder (containing `.h`, `.hpp` files) in the project.

`ObjName`: Object of class `"character"` that is he sub-path / sub-drectory of the object folder (containing the compiled `.o` files) in the project.

`InstallLibraryName`: Object of class `"character"` that is the sub-path / sub-drectory of the folder to install the compiled shared library to.

`InstallIncludeName`: Object of class `"character"` that is the sub-path / sub-drectory of the folder to install the shared library header files to.

`Flat`: Object of class `"logical"` that determines if the project is assumed *flat* or *heirarchical*. If *flat* then all the source and include files are assumed to reside in the `WorkingFolder`, otherwise they are assumed to be in the sub-folders descibed by the `SourceName`, `IncludeName` and `ObjName` arguments.

`SourceFiles`: Object of class `"list"` is a `"BSysSourceFile"` `list` that names the source files of the project.

`Packages`: Object of class `"character"` is a `string` `list` that names the packages that this library is depenedent on (C/C++ include dependencies).

`Includes`: Object of class `"character"` is a `string` `list` that contains the include paths of external dependencies of the library code.

`Defines`: Object of class `"character"` is a `string` `list` that contains preprocessor (*#defines*) defines required to compile the code.

`Libraries`: Object of class `"character"` is a `string` `list` that names the additional library dependencies needed to link the library. The format is as required by *gcc*. For instance, `"-L/usr/local/lib -lexpat"` would constitute a single item in the list.

`CFLAGS`: Object of class `"character"` is a `string` `list` naming additional *gcc* flags to apply when compiling `.c` files.

`CXXFLAGS`: Object of class `"character"` is a `string` `list` naming additional *gcc* flags to apply when compiling `.cpp` files.

`FFLAGS`: Object of class `"character"` is a `string` `list` naming additional *gcc* flags to apply when compiling `.f` files.

`LDFLAGS`: Object of class `"character"` is a `string` `list` naming additional *gcc* flags to apply when linking `.o` files.

`LDLIBS`: Object of class `"character"` is a `string` `list` naming additional *gcc* linker flags to apply when linking `.o` files.

`DEFINES`: Object of class `"character"` is a `string` `list` that contains preprocessor (*#defines*) defines supplied when compiling the code.

`IsDebug`: Object of class `"logical"` is a `Boolean` that indicates whether to create a debug build.

`DebugState`: Object of class `"list"` used internally to manage debugging.

Methods

buildMakefile `signature(.Object = "BSysProject")`: see `buildMakefile`

includePath `signature(.Object = "BSysProject")`: see `includePath`

initProjectFromFolder `signature(.Object = "BSysProject")`: see `initProjectFromFolder`

installIncludePath signature(.Object = "BSysProject"): see installIncludePath

installLibraryPath signature(.Object = "BSysProject"): see installLibraryPath

libraryPath signature(.Object = "BSysProject"): see libraryPath

loadLibrary signature(.Object = "BSysProject"): see loadLibrary

make signature(.Object = "BSysProject"): see make

objPath signature(.Object = "BSysProject"): see objPath

sourcePath signature(.Object = "BSysProject"): sourcePath

unloadLibrary signature(.Object = "BSysProject"): unloadLibrary

vcDebug signature(.Object = "BSysProject"): vcDebug

clean signature(.Object = "BSysProject"): clean

Author(s)

Paavo Jumppanen [aut, cre]

Maintainer: Paavo Jumppanen <paavo.jumppanen@csiro.au>

BSysSourceFile-class *Class* "BSysSourceFile"

Description

This class is for internal use only and is used to track the source files and corresponding dependecies that make up a BSysProject.

Objects from the Class

Objects can be created by calls of the form new("BSysSourceFile", Filename, SrcFolder, IncludeFolder, Type). Users should not be using this class directly. Instances of this class are used to track the source files and corresponding dependecies that make up a BSysProject.

Slots

Filename: Object of class "character" is the file name and path of a file in a BSysProject object instance.

Type: Object of class "character" is a string naming the source file type ("c", "cpp" or "f").

Dependencies: Object of class "list" is a list of fully qualified include file dependencies for this source file.

Externals: Object of class "list" is a list of external include file dependencies (the file is not in the includePath.

Methods

No methods defined with class "BSysSourceFile" in the signature.

Author(s)

Paavo Jumppanen [aut, cre]

Maintainer: Paavo Jumppanen <paavo.jumppanen@csiro.au>

`buildMakefile` *Build a GNU makefile*

Description

Calling `buildMakefile` builds a GNU makefile based on the project specification in the S4 `BSysProject` object instance.

Usage

```
  ## S4 method for signature 'BSysProject'
buildMakefile(.Object, Force = FALSE)
```

Arguments

`.Object`	`.Object` is an object instance of class `BSysProject` that describes the code project.
`Force`	`Force` is a boolean that when `TRUE` forces re-construction of the makefile.

Details

`buildMakefile` constructs a makefile that represents the project specfication in the S4 `BSysProject` object instance. That instance is created with a call to `new("BSysProject", ...)` constructor (see `initProjectFromFolder`). If the makefile already exists and is in sync with `Object` then the makefile is left untouched, unless `Force=TRUE`, in which case it is written afresh. A makefile is determined to be in sync with the parent project by comparison of an *md5 digest* of the `BSysProject` object instance and the digest stored in the header comment line of the makefile.

Value

Returns `TRUE` if the makefile is created or re-created and `FALSE` otherwise.

Note

see `BuildSys-package` for examples of use.

Author(s)

Paavo Jumppanen [aut, cre]

Maintainer: Paavo Jumppanen <paavo.jumppanen@csiro.au>

See Also

`make` `initProjectFromFolder` `BuildSys-package`

clean *Remove Files Create by this Project*

Description

Calling clean will remove the files and folders created by the project.

Usage

```
  ## S4 method for signature 'BSysProject'
clean(.Object)
```

Arguments

.Object	.Object is an object instance of class BSysProject that describes the code project.

Value

this method has no return value.

Note

see BuildSys-package for examples of use.

Author(s)

Paavo Jumppanen [aut, cre]

Maintainer: Paavo Jumppanen <paavo.jumppanen@csiro.au>

See Also

BuildSys-package

includePath *Get the Path to the Project Include Files*

Description

Calling includePath will return the fully qualified path to the include file folder of the project specification in the S4 BSysProject object instance.

Usage

```
  ## S4 method for signature 'BSysProject'
includePath(.Object)
```

Arguments

.Object	.Object is an object instance of class BSysProject that describes the code project.

Value

returns the fully qualified path to the include file folder of the project.

Note

see BuildSys-package for examples of use.

Author(s)

Paavo Jumppanen [aut, cre]

Maintainer: Paavo Jumppanen <paavo.jumppanen@csiro.au>

See Also

libraryPath sourcePath objPath installLibraryPath installIncludePath

---

initProjectFromFolder *Initialise an S4* BSysProject *Instance based on a Project Folder*

Description

Calling initProjectFromFolder initialises a project specification in an S4 BSysProject object instance. That object instance can then be used to compile and debug the resulting shared library. A new BSysProject object instance can be constructed by calling new("BSysProject", ...) where the argument list (...) is the same as for initProjectFromFolder.

Usage

```
## S4 method for signature 'BSysProject'
initProjectFromFolder(.Object,
                      WorkingFolder = "NULL",
                      Name = "",
                      SourceFiles = "NULL",
                      SourceName = "src",
                      IncludeName = "include",
                      ObjName = "obj",
                      InstallLibraryName = as.character(NULL),
                      InstallIncludeName = as.character(NULL),
                      Flat = TRUE,
                      Packages = as.character(c()),
                      Includes = as.character(c()),
                      Defines = as.character(c()),
```

```
                    Libraries = as.character(c()),
                    CFLAGS = as.character(c()),
                    CXXFLAGS = as.character(c()),
                    FFLAGS = as.character(c()),
                    LDFLAGS = as.character(c()),
                    LDLIBS = as.character(c()),
                    DEFINES = as.character(c()),
                    Debug = TRUE)
```

Arguments

`.Object`	`.Object` is an S4 object instance of class `BSysProject` that the code project will be based off.
`WorkingFolder`	The path to the root folder of the project, either relative or absolute and must be supplied.
`Name`	The name given to the project and will form the basename of the shared library.
`SourceFiles`	The a list of file names in WorkingFolder that should be compiled as part of the project or `NULL`. If `NULL` then all compilable source files in WorkingFolder are included.
`SourceName`	The sub-path / sub-drectory of the source folder (containing `.c`, `.cpp` files) in the project.
`IncludeName`	The sub-path / sub-drectory of the include folder (containing `.h`, `.hpp` files) in the project.
`ObjName`	The sub-path / sub-drectory of the object folder (containing the compiled `.o` files) in the project.
`InstallLibraryName`	The sub-path / sub-drectory of the folder to install the compiled shared library to.
`InstallIncludeName`	The sub-path / sub-drectory of the folder to install the shared library header files to.
`Flat`	A `Boolean` determining if the project is assumed *flat* or *heirarchical*. If *flat* then all the source and include files are assumed to reside in the `WorkingFolder`, otherwise they are assumed to be in the sub-folders descibed by the `SourceName`, `IncludeName` and `ObjName` arguments.
`Packages`	A `string list` that names the packages that this library is depenedent on (C/C++ include dependencies)
`Includes`	A `string list` that contains the include paths of external dependencies of the library code
`Defines`	A `string list` that contains preprocessor (*#defines*) defines required to compile the code
`Libraries`	A `string list` that names the additional library dependencies needed to link the library. The format is as required by *gcc*. For instance, `"-L/usr/local/lib -lexpat"` would constitute a single item in the list.
`CFLAGS`	A `string list` naming additional *gcc* flags to apply when compiling `.c` files

CXXFLAGS	A string list naming additional *gcc* flags to apply when compiling .cpp files
FFLAGS	A string list naming additional *gcc* flags to apply when compiling .f files
LDFLAGS	A string list naming additional *gcc* flags to apply when linking .o files
LDLIBS	A string list naming additional *gcc* linker flags to apply when linking .o files
DEFINES	A string list that contains preprocessor (*#defines*) defines supplied when compiling the code
Debug	A Boolean that indicates whether to create a debug build

Details

initProjectFromFolder constructs a project specification by scanning the WorkingFolder and relevant sourceName sub path for source files to add to the project. If the SourceFiles is included the the source file scan is omitted and only the named source files included. The named files are assumed relative to the Working Folder so if a hierarchical project is being created then the relative path to the source folder will need to be included in the SourceFiles list. Any .c, .cpp and .f files it finds are added to the project. In the case of Fortran files the following extensions are scanned; .f, .for, .f90, .f95 and .f77. Any added files are scanned for include directives and those includes are added as project dependencies. If the dependency is not found in the path described by the includeName it is regarded as an external dependency.

If the external dependency is known (TMB.hpp for example) then the appropriate Includes, Defines and Libraries are added to the project definition. BuildSys is currently aware of TMB, Rcpp, RcppEigen and the BLAS library support in R. External dependencies not know to BuildSys need to be handle by adding the appropriate Includes, Defines and Libraries to the Project manually with the arguments provided above.

initProjectFromFolder has useful defaults meaning in most cases very few arguments need be provided. At most, the WorkingFolder and Name arguments will be required but if the project has only a single source file then even Name can be omitted as the name will be inferred from the source file name.

Value

returns an S4 object instance of class BSysProject that describes the code project.

Note

see BuildSys-package for examples of use.

Author(s)

Paavo Jumppanen [aut, cre]

Maintainer: Paavo Jumppanen <paavo.jumppanen@csiro.au>

See Also

make buildMakefile vcDebug loadLibrary unloadLibrary libraryPath sourcePath includePath objPath installLibraryPath installIncludePath BuildSys-package

installIncludePath *Get the Install Path for the Project Include Files*

Description

Calling `installIncludePath` will return the fully qualified path to the install include file folder of the project specification in the S4 `BSysProject` object instance. This is the folder where the shared library include files will be copied to when calling `make(Project, "install")`.

Usage

```
  ## S4 method for signature 'BSysProject'
installIncludePath(.Object)
```

Arguments

`.Object`	`.Object` is an object instance of class `BSysProject` that describes the code project.

Value

returns the fully qualified path to the install include file folder of the project specification.

Note

see `BuildSys-package` for examples of use.

Author(s)

Paavo Jumppanen [aut, cre]

Maintainer: Paavo Jumppanen <paavo.jumppanen@csiro.au>

See Also

`libraryPath` `sourcePath` `includePath` `objPath` `installLibraryPath`

`installLibraryPath` *Get the Install Path for the Project Shared Library File*

Description

Calling `installLibraryPath` will return the fully qualified path to the install library file folder of the project specification in the S4 `BSysProject` object instance. This is the folder where the shared library will be copied to when calling `make(Project, "install")`.

Usage

```
  ## S4 method for signature 'BSysProject'
installLibraryPath(.Object)
```

Arguments

`.Object`	`.Object` is an object instance of class `BSysProject` that describes the code project.

Value

returns the fully qualified path to the install library file folder of the project specification.

Note

see `BuildSys-package` for examples of use.

Author(s)

Paavo Jumppanen [aut, cre]

Maintainer: Paavo Jumppanen <paavo.jumppanen@csiro.au>

See Also

`libraryPath` `sourcePath` `includePath` `objPath` `installIncludePath`

libraryPath *Get the Path to the Built Shared Library*

Description

Calling `libraryPath` will return the fully qualified path to the built shared library of the project specification in the S4 `BSysProject` object instance.

Usage

```
  ## S4 method for signature 'BSysProject'
libraryPath(.Object)
```

Arguments

`.Object`	`.Object` is an object instance of class `BSysProject` that describes the code project.

Value

returns the fully qualified path to the built shared library of the project specification.

Note

see `BuildSys-package` for examples of use.

Author(s)

Paavo Jumppanen [aut, cre]

Maintainer: Paavo Jumppanen <paavo.jumppanen@csiro.au>

See Also

`sourcePath` `includePath` `objPath` `installLibraryPath` `installIncludePath`

loadLibrary *Load the Built Shared Library*

Description

Calling `loadLibrary` will load the built library of the project specification in the S4 `BSysProject` object instance into the R session.

Usage

```
  ## S4 method for signature 'BSysProject'
loadLibrary(.Object)
```

Arguments

.Object	.Object is an object instance of class BSysProject that describes the code project.

Details

Internally this method delagates to dyn.load. Refer to the doucmentation of dyn.load for further details.

Value

returns an object of class DLLInfo.

Note

see BuildSys-package for examples of use.

Author(s)

Paavo Jumppanen [aut, cre]

Maintainer: Paavo Jumppanen <paavo.jumppanen@csiro.au>

See Also

unloadLibrary dyn.load

---

make *Make the Shared Library*

Description

Calling make will *build*, *clean* or *install* your C/C++ library.

Usage

```
  ## S4 method for signature 'BSysProject'
make(.Object, Operation = "", Debug = NULL)
```

Arguments

.Object	S4 BSysProject object instance defining the library to be made. See BuildSys-package for more information.
Operation	the make operation to be carried out. Can be one of, • "" build the library by compiling and linking all source files • "clean" clean the library by deleting all object files (.o files) and the library itself (.dll or .so files)

- "install" copy the built library file and/or header files to the install locations (see initProjectFromFolder and BuildSys-package)

Debug — A Boolean indicating whether to make a debug build. If NULL the IsDebug attribute in the BSysProject determines if a debug build is being made. If non-null the IsDebug attribute is updated using the Debug argument.

Details

Calling make results in the project described by the Object instance being transformed into a *GNU makefile* and then GNU make being called with the appropriate operation as determined by the Operation argument. The makefile will be written to the sub-folder specified by the ObjName attribute of the Object instance. If the makefile already exists it will only be re-created if the existing one in not in sync with the project definition. This is determined by an *md5 digest* of the project definition which is stored as the header comment line in the makefile. If the makefile is re-written then a *make clean* operation will be automatically carried out to ensure the built library remains in sync with the makefile.

Value

This method returns an updated S4 BSysProject object instance. If the any change to Debug state is to be preserved then the returned result should be assigned to the calling BSysProject object instance passed in the Object argument.

Note

see BuildSys-package for examples of use.

Author(s)

Paavo Jumppanen [aut, cre]

Maintainer: Paavo Jumppanen <paavo.jumppanen@csiro.au>

See Also

buildMakefile initProjectFromFolder BuildSys-package

---

objPath — *Get the Path to the Project Object Files*

Description

Calling objPath will return the fully qualified path to the object file folder of the project specification in the S4 BSysProject object instance. The object file folder will contain the makefile and compiled (object) .o files as well as the linked shared library.

Usage

```
  ## S4 method for signature 'BSysProject'
objPath(.Object)
```

Arguments

.Object	.Object is an object instance of class BSysProject that describes the code project.

Value

returns the fully qualified path to the object file folder of the project specification.

Note

see BuildSys-package for examples of use.

Author(s)

Paavo Jumppanen [aut, cre]

Maintainer: Paavo Jumppanen <paavo.jumppanen@csiro.au>

See Also

libraryPath sourcePath includePath installLibraryPath installIncludePath

---

sourcePath *Get the Path to the Project Source Files*

Description

Calling sourcePath will return the fully qualified path to the source file folder of the project specification in the S4 BSysProject object instance.

Usage

```
  ## S4 method for signature 'BSysProject'
sourcePath(.Object)
```

Arguments

.Object	.Object is an object instance of class BSysProject that describes the code project.

Value

returns the fully qualified path to the source file folder of the project specification.

Note

see BuildSys-package for examples of use.

Author(s)

Paavo Jumppanen [aut, cre]

Maintainer: Paavo Jumppanen <paavo.jumppanen@csiro.au>

See Also

libraryPath includePath objPath installLibraryPath installIncludePath

---

unloadLibrary *Unload the Built Shared Library*

Description

Calling unloadLibrary will unload the built library of the project specification in the S4 BSysProject object instance from the R session.

Usage

```
  ## S4 method for signature 'BSysProject'
unloadLibrary(.Object)
```

Arguments

.Object	.Object is an object instance of class BSysProject that describes the code project.

Details

Internally this method delagates to dyn.unload. Refer to the doucmentation of dyn.unload for further details.

Value

this method has no return value.

Note

see BuildSys-package for examples of use.

Author(s)

Paavo Jumppanen [aut, cre]

Maintainer: Paavo Jumppanen <paavo.jumppanen@csiro.au>

See Also

loadLibrary dyn.unload

---

vcDebug *Debug the Shared Library*

Description

Calling vcDebug will initiate a debug enabled *Microsoft Visual Code* instance that can be use as a GUI debugger of your C/C++ code.

Usage

```
  ## S4 method for signature 'BSysProject'
vcDebug(.Object, LaunchEditor=TRUE)
```

Arguments

.Object	BSysProject object instance defining the library to be debugged. See BuildSys-package for more information.
LaunchEditor	LaunchEditor is a reserved argument that should not be used by the user. A call to it with LaunchEditor=FALSE is used in the *.Rprofile* file for the debug R session to do the necessary tasks to initialise the R session state.

Details

vcDebug creates the necessary c_cpp_properties.json and launch.json files needed for *Microsoft Visual Code* to effectively debug the project dynamic library and launches an instance of *Visual Code*. It is assumed that the dynamic library has already been compiled with make and Debug support enabled. R needs to be configured to be able to build C/C++ libraries (RTools installed on Windows) and it is assumed that *Microsoft Visual Code* is installed. These requirements are discussed at length in BuildSys-package.

Within *Visual Code* you can open the source file/s of your library, set breakpoints and run a debug session. To run a debug session select the *run/Start Debugging* menu item. Doing so will result in a new instance of *R* being launched which contains the same environment (including loaded packages and loaded dynamic libraries) as the parent *R* session where vcDebug was first called. This *R* session is your *sandbox* to safely debug your library in and leaves the parent *R* session safe from loss should your code crash *R* completely.

Debugging a new library will typically require some *R* setup code to be executed before using and/or testing it. With BuildSys you should perform this initial setup of *R* in the parent *R* session before calling vcDebug so that you never have to initialise *R* in your debug session. The debug *R* session will inherit the state of the parent *R* session at the time that vcDebug was called.

Value

this method returns no value.

Note

see BuildSys-package for examples of use.

If the absolute path to source and dynamic library files contains spaces then debugging in *Visual Code* fails to function correctly. To ensure code is debuggable users will need to avoid putting code in folders with whitespace in the folder names.

Author(s)

Paavo Jumppanen [aut, cre]

Maintainer: Paavo Jumppanen <paavo.jumppanen@csiro.au>

See Also

initProjectFromFolder make BuildSys-package

Index

∗ **classes**
BSysProject-class, 9
BSysSourceFile-class, 11
∗ **debugging**
buildMakefile, 12
BuildSys-package, 2
clean, 13
includePath, 13
initProjectFromFolder, 14
installIncludePath, 17
installLibraryPath, 18
libraryPath, 19
loadLibrary, 19
make, 20
objPath, 21
sourcePath, 22
unloadLibrary, 23
vcDebug, 24
∗ **package**
BuildSys-package, 2
∗ **programming**
buildMakefile, 12
BuildSys-package, 2
clean, 13
includePath, 13
initProjectFromFolder, 14
installIncludePath, 17
installLibraryPath, 18
libraryPath, 19
loadLibrary, 19
make, 20
objPath, 21
sourcePath, 22
unloadLibrary, 23
vcDebug, 24
∗ **utilities**
buildMakefile, 12
BuildSys-package, 2
clean, 13
includePath, 13
initProjectFromFolder, 14
installIncludePath, 17
installLibraryPath, 18
libraryPath, 19
loadLibrary, 19
make, 20
objPath, 21
sourcePath, 22
unloadLibrary, 23
vcDebug, 24

BSysProject-class, 9
BSysSourceFile-class, 11
buildMakefile, *7*, *10*, 12, *16*, *21*
buildMakefile,BSysProject-method
*(*BSysProject-class*)*, 9
BuildSys *(*BuildSys-package*)*, 2
BuildSys-package, 2

clean, *7*, *11*, 13
clean,BSysProject-method
*(*BSysProject-class*)*, 9

dyn.load, *20*
dyn.unload, *23*, *24*

includePath, *7*, *10*, *11*, 13, *16–19*, *22*, *23*
includePath,BSysProject-method
*(*BSysProject-class*)*, 9
initProjectFromFolder, *9*, *10*, *12*, 14, *21*, *25*
initProjectFromFolder,BSysProject-method
*(*BSysProject-class*)*, 9
installIncludePath, *7*, *11*, *14*, *16*, 17, *18*, *19*, *22*, *23*
installIncludePath,BSysProject-method
*(*BSysProject-class*)*, 9
installLibraryPath, *7*, *11*, *14*, *16*, *17*, 18, *19*, *22*, *23*
installLibraryPath,BSysProject-method
*(*BSysProject-class*)*, 9

libraryPath, *7*, *11*, *14*, *16–18*, 19, *22, 23*
libraryPath,BSysProject-method
(BSysProject-class), 9
loadLibrary, *7*, *11*, *16*, 19, *24*
loadLibrary,BSysProject-method
(BSysProject-class), 9

make, *7*, *11, 12*, *16*, 20, *24, 25*
make,BSysProject-method
(BSysProject-class), 9

objPath, *7*, *11*, *14*, *16–19*, 21, *23*
objPath,BSysProject-method
(BSysProject-class), 9

sourcePath, *7*, *11*, *14*, *16–19*, *22*, 22
sourcePath,BSysProject-method
(BSysProject-class), 9

unloadLibrary, *7*, *11*, *16*, *20*, 23
unloadLibrary,BSysProject-method
(BSysProject-class), 9

vcDebug, *7*, *11*, *16*, 24
vcDebug,BSysProject-method
(BSysProject-class), 9