

rlog Reference Manual

1.3

Generated by Doxygen 1.4.4

Sat Oct 14 22:04:06 2006

Contents

1	RLog - a C++ logging library	1
1.1	Introduction	1
1.2	Using RLog	2
1.3	Requirements	2
1.4	Downloads	3
2	rlog Module Index	5
2.1	rlog Modules	5
3	rlog Directory Hierarchy	7
3.1	rlog Directories	7
4	rlog Class Index	9
4.1	rlog Class List	9
5	rlog Page Index	11
5.1	rlog Related Pages	11
6	rlog Module Documentation	13
6.1	RLogMacros	13
7	rlog Directory Documentation	21
7.1	_dargs/ Directory Reference	21
7.2	_dargs/current/ Directory Reference	22
7.3	_dargs/current/rlog/ Directory Reference	23

7.4	rlog-1.3.7/rlog/ Directory Reference	24
7.5	rlog/ Directory Reference	25
7.6	rlog-1.3.7/ Directory Reference	26
8	rlog Class Documentation	27
9	rlog Page Documentation	29
9.1	RLog Channels	29
9.2	RLog Components	33

Chapter 1

RLog - a C++ logging library

Copyright ©2002-2004 Valient Gough <vgough@pobox.com>

Distributed under the LGPL license, see COPYING for details.

1.1 Introduction

RLog provides a flexible message logging facility for C++ programs and libraries. It is meant to be fast enough to leave in production code.

RLog provides macros which are similar to Qt's debug macros, which are similar to simple printf() statements:

```
void func(int foo)
{
    rDebug("foo = %i", foo);
    int ans = 6 * 9;
    if(ans != 42)
        rWarning("ans = %i, expecting 42", ans);
    rError("I'm sorry %s, I can't do that (error code %i)", name, errno);
}
```

The difference to Qt's macros is that the log messages are considered *publishers* and there can be any number of *subscribers* to log messages. Subscribers may choose which messages they want to receive in a number of different ways:

- subscribe to messages to a particular *channel*. Channels are hierarchical and can be easily created. See [RLog Channels](#).
- subscribe to anything from a particular *component*. See [RLog Components](#).
- subscribe to messages from a particular file name within a component.

If there are no subscribers to a particular logging statement, that statement can be said to be *dormant*. RLog is optimized to minimize overhead of dormant logging statements, with the goal of allowing logging to be left in release versions of software. This way if problems show up in production code, it is possible to activate logging statements in real time to aid debugging.

As an indication of just how cheap a dormant logging statement is, on a Pentium-4 class CPU with g++ 3.3.1, a dormant log in a tight loop adds on the order of 2-6 (two to six) clock cycles of overhead (1). By comparison a simple logging function such as Qt's `qDebug()` adds about 1000 (a thousand) clock cycles of overhead - even when messages are being thrown away.

In addition, logging statements in RLog can be individually activated at run-time without affecting any other statements, allowing targeted log reporting.

(1) The first time a logging statement is encountered, it must be registered in order to determine if there are any subscribers. So there is additional overhead the first time a statement is encountered.

1.2 Using RLog

In order to begin using RLog in your code, you should do the following:

- define `RLOG_COMPONENT` in your build environment. Eg: `librlog` is built with `-DRLOG_COMPONENT="rlog"`. You should use a unique name for your program or library (do not use "rlog"). If your program is made up of separate components, then you can define `RLOG_COMPONENT` as a different name for each component.
- (optional) add a call to `RLogInit()` in your main program startup code. This is not a requirement, however not including it may reduce functionality of external `rlog` modules.
- link with `librlog`
- add subscribers (`rlog::StdioNode` , `rlog::SyslogNode` , or your own) to catch any messages you are interested in.

1.3 Requirements

RLog has been tested on the following systems (all releases may not have been tested on all systems):

Platform	Operating System	Compiler	Notes
ix86	SuSE 9.2	GNU G++ 3.3.4	binary RPM available
	SuSE 9.0	Intel ICC 8.0	last test was prior to RLog 1.3.4
	RedHat 7.3	GNU G++ 2.96	binary RPM available
	OpenBSD 3.4	GNU G++ 2.95.3	Tested with 1.3.5
	FreeBSD 4.10-beta	GNU G++ 2.95.4	Support added in 1.3.6 release
sparc	Solaris 5.9	GNU G++ 3.3.2	
PowerPC	Darwin 5.5	gcc-932.1	Support added in 1.3.6 release

To build development versions, you will also need the GNU autoconf tools (with automake and libtool). Documentation is built using Doxygen.

1.4 Downloads

RLog is available in source code and RPM packaged binaries for some systems.

RLog Version 1.3.7 - Oct 5, 2005 release.

- Tarball: [rlog-1.3.7.tgz](#) + [tarball GPG signature](#)
- Source RPM: [rlog-1.3.7-1.src.rpm](#)

Binary packages:

- SuSE 9.2 i586 RPM: [rlog-1.3.7-1suse92.i586.rpm](#)
- RedHat 7.3 i386 RPM: [rlog-1.3.7-1rh73.i386.rpm](#)

To check the signature, you can download my public key from a public key server, or from the link at the top of [my homepage](#).

If you wish to be notified automatically of new releases, you can subscribe to new release notifications on the [Freshmeat page](#).

Chapter 2

rlog Module Index

2.1 rlog Modules

Here is a list of all modules:

RLogMacros	13
----------------------	----

Chapter 3

rlog Directory Hierarchy

3.1 rlog Directories

This directory hierarchy is sorted roughly, but not completely, alphabetically:

_darcs	21
current	22
rlog	23
rlog	25
rlog-1.3.7	26
rlog	24

Chapter 4

rlog Class Index

4.1 rlog Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

rlog::Error (Error Used as exception thrown from <code>rAssert()</code> on failure)	??
rlog::ErrorData (Internal RLog structure - holds internal data for rlog::Error)	??
rlog::FileNode (Provides filename based logging nodes)	??
rlog::Lock	??
rlog::Mutex (Class encapsulating a critical section under Win32 or a mutex under Unix)	??
rlog::PublishLoc (Internal RLog structure - static struct for each rLog() state- ment)	??
rlog::RLogChannel (Implements a hierarchical logging channel)	??
rlog::RLogData (Data published through RLogNode)	??
rlog::RLogModule (Allows registration of external modules to rlog)	??
rlog::RLogNode (Core of publication system, forms activation network) . . .	??
rlog::RLogPublisher (RLog publisher used by rLog macros)	??
rlog::StdioNode (Logs subscribed messages to a file descriptor)	??
rlog::SyslogNode (Logs subscribed messages using syslog)	??

Chapter 5

rlog Page Index

5.1 rlog Related Pages

Here is a list of all related documentation pages:

RLog Channels	29
RLog Components	33

Chapter 6

rlog Module Documentation

6.1 RLogMacros

Defines

- `#define rDebug() _rMessage(LOGID, rlog::_RLDebugChannel, ##__VA_ARGS__)`
Log a message to the "debug" channel. Takes printf style arguments.
- `#define rInfo() _rMessage(LOGID, rlog::_RLInfoChannel, ##__VA_ARGS__)`
Log a message to the "debug" channel. Takes printf style arguments.
- `#define rWarning() _rMessage(LOGID, rlog::_RLWarningChannel, ##__VA_ARGS__)`
Log a message to the "warning" channel. Takes printf style arguments.
- `#define rError() _rMessage(LOGID, rlog::_RLErrorChannel, ##__VA_ARGS__)`
Log a message to the "error" channel. Takes printf style arguments.
- `#define rLog(channel,) _rMessage(LOGID, channel, ##__VA_ARGS__)`
Log a message to a user defined channel. Takes a channel and printf style arguments.
- `#define rDebug() _rMessage(LOGID, rlog::_RLDebugChannel, ##__VA_ARGS__)`
Log a message to the "debug" channel. Takes printf style arguments.

- `#define rInfo() _rMessage(LOGID, rlog::_RLInfoChannel, ##__VA_ARGS__)`
Log a message to the "debug" channel. Takes printf style arguments.
- `#define rWarning() _rMessage(LOGID, rlog::_RLWarningChannel, ##__VA_ARGS__)`
Log a message to the "warning" channel. Takes printf style arguments.
- `#define rError() _rMessage(LOGID, rlog::_RLErrorChannel, ##__VA_ARGS__ - _)`
Log a message to the "error" channel. Takes printf style arguments.
- `#define rLog(channel,) _rMessage(LOGID, channel, ##__VA_ARGS__)`
Log a message to a user defined channel. Takes a channel and printf style arguments.
- `#define rDebug() _rMessage(LOGID, rlog::_RLDebugChannel, ##__VA_ARGS__)`
Log a message to the "debug" channel. Takes printf style arguments.
- `#define rInfo() _rMessage(LOGID, rlog::_RLInfoChannel, ##__VA_ARGS__)`
Log a message to the "debug" channel. Takes printf style arguments.
- `#define rWarning() _rMessage(LOGID, rlog::_RLWarningChannel, ##__VA_ARGS__)`
Log a message to the "warning" channel. Takes printf style arguments.
- `#define rError() _rMessage(LOGID, rlog::_RLErrorChannel, ##__VA_ARGS__ - _)`
Log a message to the "error" channel. Takes printf style arguments.
- `#define rLog(channel,) _rMessage(LOGID, channel, ##__VA_ARGS__)`
Log a message to a user defined channel. Takes a channel and printf style arguments.

6.1.1 Detailed Description

These macros are the primary interface for logging messages:

- `rDebug(format, ...)`
- `rInfo(format, ...)`
- `rWarning(format, ...)`

- rError(format, ...)
- rLog(channel, format, ...)
- rAssert(condition)

These macros are the primary interface for logging messages:

- rDebug(format, ...)
- rInfo(format, ...)
- rWarning(format, ...)
- rError(format, ...)
- rLog(channel, format, ...)
- rAssert(condition)

These macros are the primary interface for logging messages:

- rDebug(format, ...)
- rInfo(format, ...)
- rWarning(format, ...)
- rError(format, ...)
- rLog(channel, format, ...)
- rAssert(condition)

6.1.2 Define Documentation

6.1.2.1 `#define rDebug() _rMessage(LOGID, rlog::_RLDebugChannel, ##__VA_ARGS__)`

Log a message to the "debug" channel. Takes printf style arguments.

Format is ala printf, eg:

```
rDebug("I'm sorry %s, I can't do %s", name, request);
```

When using a recent GNU compiler, it should automatically detect format string / argument mismatch just like it would with printf.

Note that unless there are subscribers to this message, it will do nothing.

6.1.2.2 #define rDebug() _rMessage(LOGID, rlog::_RLDebugChannel, ##__VA_ARGS__)

Log a message to the "debug" channel. Takes printf style arguments.

Format is ala printf, eg:

```
rDebug("I'm sorry %s, I can't do %s", name, request);
```

When using a recent GNU compiler, it should automatically detect format string / argument mismatch just like it would with printf.

Note that unless there are subscribers to this message, it will do nothing.

6.1.2.3 #define rDebug() _rMessage(LOGID, rlog::_RLDebugChannel, ##__VA_ARGS__)

Log a message to the "debug" channel. Takes printf style arguments.

Format is ala printf, eg:

```
rDebug("I'm sorry %s, I can't do %s", name, request);
```

When using a recent GNU compiler, it should automatically detect format string / argument mismatch just like it would with printf.

Note that unless there are subscribers to this message, it will do nothing.

6.1.2.4 #define rError() _rMessage(LOGID, rlog::_RLErrorChannel, ##__VA_ARGS__)

Log a message to the "error" channel. Takes printf style arguments.

An error indicates that something has definately gone wrong.

Format is ala printf, eg:

```
rError("bad input %s, aborting request", input);
```

When using a recent GNU compiler, it should automatically detect format string / argument mismatch just like it would with printf.

Note that unless there are subscribers to this message, it will do nothing.

6.1.2.5 #define rError() _rMessage(LOGID, rlog::_RLErrorChannel, ##__VA_ARGS__)

Log a message to the "error" channel. Takes printf style arguments.

An error indicates that something has definately gone wrong.

Format is ala printf, eg:

```
rError("bad input %s, aborting request", input);
```

When using a recent GNU compiler, it should automatically detect format string / argument mismatch just like it would with printf.

Note that unless there are subscribers to this message, it will do nothing.

6.1.2.6 #define rError() _rMessage(LOGID, rlog::_RLErrorChannel, ##__VA_ARGS__)

Log a message to the "error" channel. Takes printf style arguments.

An error indicates that something has definately gone wrong.

Format is ala printf, eg:

```
rError("bad input %s, aborting request", input);
```

When using a recent GNU compiler, it should automatically detect format string / argument mismatch just like it would with printf.

Note that unless there are subscribers to this message, it will do nothing.

6.1.2.7 #define rInfo() _rMessage(LOGID, rlog::_RLInfoChannel, ##__VA_ARGS__)

Log a message to the "debug" channel. Takes printf style arguments.

Format is ala printf, eg:

```
rInfo("I'm sorry %s, I can't do %s", name, request);
```

When using a recent GNU compiler, it should automatically detect format string / argument mismatch just like it would with printf.

Note that unless there are subscribers to this message, it will do nothing.

6.1.2.8 #define rInfo() _rMessage(LOGID, rlog::_RLInfoChannel, ##__VA_ARGS__)

Log a message to the "debug" channel. Takes printf style arguments.

Format is ala printf, eg:

```
rInfo("I'm sorry %s, I can't do %s", name, request);
```

When using a recent GNU compiler, it should automatically detect format string / argument mismatch just like it would with printf.

Note that unless there are subscribers to this message, it will do nothing.

6.1.2.9 **#define rInfo() _rMessage(LOGID, rlog::_RLInfoChannel, ##__VA_ARGS__)**

Log a message to the "debug" channel. Takes printf style arguments.

Format is ala printf, eg:

```
rInfo("I'm sorry %s, I can't do %s", name, request);
```

When using a recent GNU compiler, it should automatically detect format string / argument mismatch just like it would with printf.

Note that unless there are subscribers to this message, it will do nothing.

6.1.2.10 **#define rLog(channel) _rMessage(LOGID, channel, ##__VA_ARGS__)**

Log a message to a user defined channel. Takes a channel and printf style arguments.

An error indicates that something has definately gone wrong.

Format is ala printf, eg:

```
static RLogChannel * MyChannel = RLOG_CHANNEL( "debug/mine" );
rLog(MyChannel, "happy happy, joy joy");
```

When using a recent GNU compiler, it should automatically detect format string / argument mismatch just like it would with printf.

Note that unless there are subscribers to this message, it will do nothing.

6.1.2.11 **#define rLog(channel) _rMessage(LOGID, channel, ##__VA_ARGS__)**

Log a message to a user defined channel. Takes a channel and printf style arguments.

An error indicates that something has definately gone wrong.

Format is ala printf, eg:

```
static RLogChannel * MyChannel = RLOG_CHANNEL( "debug/mine" );
rLog(MyChannel, "happy happy, joy joy");
```

When using a recent GNU compiler, it should automatically detect format string / argument mismatch just like it would with printf.

Note that unless there are subscribers to this message, it will do nothing.

6.1.2.12 **#define rLog(channel) _rMessage(LOGID, channel, ##__VA_ARGS__)**

Log a message to a user defined channel. Takes a channel and printf style arguments.

An error indicates that something has definately gone wrong.

Format is ala printf, eg:

```
static RLogChannel * MyChannel = RLOG_CHANNEL( "debug/mine" );
rLog(MyChannel, "happy happy, joy joy");
```

When using a recent GNU compiler, it should automatically detect format string / argument mismatch just like it would with printf.

Note that unless there are subscribers to this message, it will do nothing.

6.1.2.13 **#define rWarning() _rMessage(LOGID, rlog::_RLWarningChannel, ##__VA_ARGS__)**

Log a message to the "warning" channel. Takes printf style arguments.

Output a warning message - meant to indicate that something doesn't seem right.

Format is ala printf, eg:

```
rWarning("passed %i, expected %i, continuing", foo, bar);
```

When using a recent GNU compiler, it should automatically detect format string / argument mismatch just like it would with printf.

Note that unless there are subscribers to this message, it will do nothing.

6.1.2.14 **#define rWarning() _rMessage(LOGID, rlog::_RLWarningChannel, ##__VA_ARGS__)**

Log a message to the "warning" channel. Takes printf style arguments.

Output a warning message - meant to indicate that something doesn't seem right.

Format is ala printf, eg:

```
rWarning("passed %i, expected %i, continuing", foo, bar);
```

When using a recent GNU compiler, it should automatically detect format string / argument mismatch just like it would with printf.

Note that unless there are subscribers to this message, it will do nothing.

6.1.2.15 `#define rWarning() _rMessage(LOGID, rlog::_RLWarningChannel, ##__VA_ARGS__)`

Log a message to the "warning" channel. Takes printf style arguments.

Output a warning message - meant to indicate that something doesn't seem right.

Format is ala printf, eg:

```
rWarning("passed %i, expected %i, continuing", foo, bar);
```

When using a recent GNU compiler, it should automatically detect format string / argument mismatch just like it would with printf.

Note that unless there are subscribers to this message, it will do nothing.

Chapter 7

rlog Directory Documentation

7.1 _darcs/ Directory Reference

Directories

- directory [current](#)

7.2 _darcs/current/ Directory Reference

Directories

- directory [rlog](#)

7.3 _darcs/current/rlog/ Directory Reference

Files

- file _darcs/current/rlog/Error.cpp
- file _darcs/current/rlog/Error.h
- file _darcs/current/rlog/Lock.h
- file _darcs/current/rlog/Mutex.h
- file _darcs/current/rlog/rlog-c99.h
- file _darcs/current/rlog/rlog-novariadic.h
- file _darcs/current/rlog/rlog-prec99.h
- file _darcs/current/rlog/rlog.cpp
- file _darcs/current/rlog/rlog.h
- file _darcs/current/rlog/RLogChannel.cpp
- file _darcs/current/rlog/RLogChannel.h
- file _darcs/current/rlog/rloginit.cpp
- file _darcs/current/rlog/rloginit.h
- file _darcs/current/rlog/rloglocation.cpp
- file _darcs/current/rlog/rloglocation.h
- file _darcs/current/rlog/RLogNode.cpp
- file _darcs/current/rlog/RLogNode.h
- file _darcs/current/rlog/RLogPublisher.cpp
- file _darcs/current/rlog/RLogPublisher.h
- file _darcs/current/rlog/RLogTime.h
- file _darcs/current/rlog/StdioNode.cpp
- file _darcs/current/rlog/StdioNode.h
- file _darcs/current/rlog/SyslogNode.cpp
- file _darcs/current/rlog/SyslogNode.h
- file _darcs/current/rlog/test.cpp
- file _darcs/current/rlog/testlog.cpp

7.4 rlog-1.3.7/rlog/ Directory Reference

Files

- file 1.3.7/rlog/common.h
- file rlog-1.3.7/rlog/Error.cpp
- file rlog-1.3.7/rlog/Error.h
- file rlog-1.3.7/rlog/Lock.h
- file rlog-1.3.7/rlog/Mutex.h
- file rlog-1.3.7/rlog/rlog-c99.h
- file rlog-1.3.7/rlog/rlog-novariadic.h
- file rlog-1.3.7/rlog/rlog-prec99.h
- file rlog-1.3.7/rlog/rlog.cpp
- file rlog-1.3.7/rlog/rlog.h
- file rlog-1.3.7/rlog/RLogChannel.cpp
- file rlog-1.3.7/rlog/RLogChannel.h
- file rlog-1.3.7/rlog/rloginit.cpp
- file rlog-1.3.7/rlog/rloginit.h
- file rlog-1.3.7/rlog/rloglocation.cpp
- file rlog-1.3.7/rlog/rloglocation.h
- file rlog-1.3.7/rlog/RLogNode.cpp
- file rlog-1.3.7/rlog/RLogNode.h
- file rlog-1.3.7/rlog/RLogPublisher.cpp
- file rlog-1.3.7/rlog/RLogPublisher.h
- file rlog-1.3.7/rlog/RLogTime.h
- file rlog-1.3.7/rlog/StdioNode.cpp
- file rlog-1.3.7/rlog/StdioNode.h
- file rlog-1.3.7/rlog/SyslogNode.cpp
- file rlog-1.3.7/rlog/SyslogNode.h
- file rlog-1.3.7/rlog/test.cpp
- file rlog-1.3.7/rlog/testlog.cpp

7.5 rlog/ Directory Reference

Files

- file **common.h**
- file **rlog/Error.cpp**
- file **rlog/Error.h**
- file **rlog/Lock.h**
- file **rlog/Mutex.h**
- file **rlog/rlog-c99.h**
- file **rlog/rlog-novariadic.h**
- file **rlog/rlog-prec99.h**
- file **rlog/rlog.cpp**
- file **rlog/rlog.h**
- file **rlog/RLogChannel.cpp**
- file **rlog/RLogChannel.h**
- file **rlog/rloginit.cpp**
- file **rlog/rloginit.h**
- file **rlog/rloglocation.cpp**
- file **rlog/rloglocation.h**
- file **rlog/RLogNode.cpp**
- file **rlog/RLogNode.h**
- file **rlog/RLogPublisher.cpp**
- file **rlog/RLogPublisher.h**
- file **rlog/RLogTime.h**
- file **rlog/StdioNode.cpp**
- file **rlog/StdioNode.h**
- file **rlog/SyslogNode.cpp**
- file **rlog/SyslogNode.h**

7.6 rlog-1.3.7/ Directory Reference

Directories

- directory [rlog](#)

Chapter 8

rlog Class Documentation

Chapter 9

rlog Page Documentation

9.1 RLog Channels

An RLog Channel is a naming method for logging messages.

All logs are associated with a single channel, however there a variety of ways of subscribing to a log message.

9.1.1 Channel Hierarchy

Channels are hierarchical. For example, if a log message is published on the "debug" channel:

```
rDebug("hi");  
// same as  
static RLogChannel *myChannel = DEF_CHANNEL("debug", Log_Debug);  
rLog(myChannel, "hi");
```

In the example above, all subscribers to the "debug" channel receive the messages, but *not* subscribers to "debug/foo" or other sub-channels.

If a log is published under "debug/foo/bar":

```
static RLogChannel *myChannel = DEF_CHANNEL("debug/foo/bar", Log_Debug);  
rLog(myChannel, "hi");
```

In that example, all subscribers to "debug/foo/bar", "debug/foo", and "debug" will receive the message.

All channels are considered to be derived from a root channel. It doesn't have a true name and is referenced as the empty string "". So, to capture *all* messages:

```
// capture all messages and log them to stderr
StdioNode stdLog( STDERR_FILENO );
stdLog.subscribeTo( GetGlobalChannel("") ); // empty string is root channel
```

9.1.2 Channel Components

Or in mathematical terms, the cross product of channels and components.

Channels are componentized. By default, all log messages using one of the rLog type macros is actually published on the component-specific version of the channel (the component being the value of RLOG_COMPONENT at compile time). So, instead of just saying a message was published on "debug" channel, we need to also say which component it was part of, which we could represent as a pair (< COMPONENT, CHANNEL >) – eg <"rlog", "debug">.

This means that two separate components, both using [rDebug\(\)](#) (for example) could be subscribed to separately, or together.

There is a way to subscribe to channels in the following ways:

- <COMPONENT, CHANNEL> : subscribe to a particular channel from a component
- <COMPONENT, *> : subscribe to all channels from a component
- <*, CHANNEL> : subscribe to a channel from *all* components
- <*, *> : subscribe to all channels from all components

```
{
    // this is published on the channel "debug", and the component
    // [RLOG_COMPONENT]
    rDebug("hi");

    StdioNode stdLog( STDERR_FILENO );

    // subscribe to a particular channel, from the current component
    // ([RLOG_COMPONENT])
    stdLog.subscribeTo( RLOG_CHANNEL("debug/foo") );

    // subscribe to all channels from the current component ([RLOG_COMPONENT])
    // (the root channel is the empty string "")
    stdLog.subscribeTo( RLOG_CHANNEL("") );

    // subscribe to a particular channel from all components
    stdLog.subscribeTo( GetGlobalChannel("debug/foo") );

    // subscribe to all channels from all components
    stdLog.subscribeTo( GetGlobalChannel("") );
}
```

As you can see from the pattern above, using the RLOG_CHANNEL() macro limits the selection to the current component. If you want to specify a component other than

the current component, use `GetComponentChannel()` which takes the component name as the first argument.

An RLog Channel is a naming method for logging messages.

All logs are associated with a single channel, however there a variety of ways of subscribing to a log message.

9.1.3 Channel Hierarchy

Channels are hierarchical. For example, if a log message is published on the "debug" channel:

```
rDebug("hi");  
// same as  
static RLogChannel *myChannel = DEF_CHANNEL("debug", Log_Debug);  
rLog(myChannel, "hi");
```

In the example above, all subscribers to the "debug" channel receive the messages, but *not* subscribers to "debug/foo" or other sub-channels.

If a log is published under "debug/foo/bar":

```
static RLogChannel *myChannel = DEF_CHANNEL("debug/foo/bar", Log_Debug);  
rLog(myChannel, "hi");
```

In that example, all subscribers to "debug/foo/bar", "debug/foo", and "debug" will receive the message.

All channels are considered to be derived from a root channel. It doesn't have a true name and is referenced as the empty string "". So, to capture *all* messages:

```
// capture all messages and log them to stderr  
StdioNode stdLog( STDERR_FILENO );  
stdLog.subscribeTo( GetGlobalChannel("") ); // empty string is root channel
```

9.1.4 Channel Components

Or in mathematical terms, the cross product of channels and components.

Channels are componentized. By default, all log messages using one of the rLog type macros is actually published on the component-specific version of the channel (the component being the value of `RLOG_COMPONENT` at compile time). So, instead of just saying a message was published on "debug" channel, we need to also say which component it was part of, which we could represent as a pair (`< COMPONENT, CHANNEL >`) – eg `<"rlog", "debug">`.

This means that two separate components, both using `rDebug()` (for example) could be subscribed to separately, or together.

There is a way to subscribe to channels in the following ways:

- `<COMPONENT, CHANNEL>` : subscribe to a particular channel from a component
- `<COMPONENT, *>` : subscribe to all channels from a component
- `<*, CHANNEL>` : subscribe to a channel from *all* components
- `<*, *>` : subscribe to all channels from all components

```
{
    // this is published on the channel "debug", and the component
    // [RLOG_COMPONENT]
    rDebug("hi");

    StdioNode stdLog( STDERR_FILENO );

    // subscribe to a particular channel, from the current component
    // ([RLOG_COMPONENT])
    stdLog.subscribeTo( RLOG_CHANNEL("debug/foo") );

    // subscribe to all channels from the current component ([RLOG_COMPONENT])
    // (the root channel is the empty string "")
    stdLog.subscribeTo( RLOG_CHANNEL("") );

    // subscribe to a particular channel from all components
    stdLog.subscribeTo( GetGlobalChannel("debug/foo") );

    // subscribe to all channels from all components
    stdLog.subscribeTo( GetGlobalChannel("") );
}
```

As you can see from the pattern above, using the `RLOG_CHANNEL()` macro limits the selection to the current component. If you want to specify a component other than the current component, use `GetComponentChannel()` which takes the component name as the first argument.

9.2 RLog Components

An RLog Component is typically a group of files with some shared purpose.

When programs are built with RLog, the value of `RLOG_COMPONENT` is used as the component name. If `RLOG_COMPONENT` is not set at compile time, you may receive a compiler warning, and the component will be set to "[unknown]".

For example when `rlog` is built, it specifies **`-DRLOG_COMPONENT="rlog"`**.

For more detail on how to use components in subscriptions, see [RLog Channels](#).

An RLog Component is typically a group of files with some shared purpose.

When programs are built with RLog, the value of `RLOG_COMPONENT` is used as the component name. If `RLOG_COMPONENT` is not set at compile time, you may receive a compiler warning, and the component will be set to "[unknown]".

For example when `rlog` is built, it specifies **`-DRLOG_COMPONENT="rlog"`**.

For more detail on how to use components in subscriptions, see [RLog Channels](#).

Index

- [_dargs/ Directory Reference, 21](#)
- [_dargs/current/ Directory Reference, 22](#)
- [_dargs/current/rlog/ Directory Reference, 23](#)
- rDebug
 - [RLogMacros, 15, 16](#)
- rError
 - [RLogMacros, 16, 17](#)
- rInfo
 - [RLogMacros, 17, 18](#)
- rLog
 - [RLogMacros, 18, 19](#)
- [rlog-1.3.7/ Directory Reference, 26](#)
- [rlog-1.3.7/rlog/ Directory Reference, 24](#)
- [rlog/ Directory Reference, 25](#)
- [RLogMacros, 13](#)
- RLogMacros
 - [rDebug, 15, 16](#)
 - [rError, 16, 17](#)
 - [rInfo, 17, 18](#)
 - [rLog, 18, 19](#)
 - [rWarning, 19, 20](#)
- rWarning
 - [RLogMacros, 19, 20](#)