JrDebugLogger

©2004

Note:
To change the product logo for your own print manual or PDF, click "Tools > Manual Designer" and modify the print manual template.

Title page 1

Use this page to introduce the product

by

This is "Title Page 1" - you may use this page to introduce your product, show title, author, copyright, company logos, etc.

This page intentionally starts on an odd page, so that it is on the right half of an open book from the readers point of view. This is the reason why the previous page was blank (the previous page is the back side of the cover)

JrDebugLogger

©2004

All rights reserved. No parts of this work may be reproduced in any form or by any means - graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems - without the written permission of the publisher.

Products that are referred to in this document may be either trademarks and/or registered trademarks of the respective owners. The publisher and the author make no claim to these trademarks.

While every precaution has been taken in the preparation of this document, the publisher and the author assume no responsibility for errors or omissions, or for damages resulting from the use of information contained in this document or from the use of programs and source code that may accompany it. In no event shall the publisher and the author be liable for any loss of profit or any other commercial damage caused or alleged to have been caused directly or indirectly by this document.

Printed: June 2014 in (whereever you are located)

Publisher

...enter name...

Managing Editor

...enter name...

Technical Editors

...enter name...
...enter name...

Cover Designer

...enter name...

Team Coordinator

...enter name...

Production

...enter name...

Special thanks to:

All the people who contributed to this document, to mum and dad and grandpa, to my sisters and brothers and mothers in law, to our secretary Kathrin, to the graphic artist who created this great product logo on the cover page (sorry, don't remember your name at the moment but you did a great work), to the pizza service down the street (your daily Capricciosas saved our lives), to the copy shop where this document will be duplicated, and and and...

Last not least, we want to thank EC Software who wrote this great help tool called HELP & MANUAL which printed this document.

Table of Contents

Foreword 6

Part I Introduction 7

1 JrDebugLogger ... 7
2 Features and Principles ... 8
3 From README ... 8

Part II Some Examples 15

1 A First Example ... 15
2 Simple Commandline Enabled Example ... 15
3 Configuring without Commandline Options ... 16
4 A More Involved Example ... 17
5 Different Kinds of Messages ... 19
6 Using a Local Callback Monitor ... 20
7 Outputing Custom Columns ... 21
8 Using DebugBlocks ... 22
9 Using JrDebugEnclose ... 23
10 Unit Testing with JrDebugLogger ... 24
11 More Samples ... 26

Part III Basic Usage 27

1 Suggested Installation ... 27
2 Adding the JrDebugLogger files to Your Project ... 27
3 Adding Basic Logging Statements ... 27
4 Viewing Debug Messages With the GUI Tool ... 28
5 Turning Off Compilation of DebugLogging Code ... 29

Part IV Advanced Usage 31

1 Logging Functions ... 31
  Printf Style ... 31
  Stream Style ... 32
2 Logging Other Events ... 33
  Alarms ... 33
  Activities ... 33
  DebugBlocks ... 33
  Assertions ... 34
  Exceptions ... 35
  Timing Activities ... 35
3 Configuration of Debug Logging Options ... 35
  Logging Destinations and Filters ... 35

Text Formatting Mode 35
4 Helper Functions 35
Commandline Parsing and Enabling 35
Startup and Announcing 36
Optional Callback Object 37
5 Managing Compilation and Release 37
Application Switch Files and Compile Flags 37
Excluding User Code from Compiling 38

Part V Unit Testing Support 39

1 Introduction 39
2 Unit Testing - Registration and Invocation 39
3 Unit Testing - Test Evaluation Functions 40
4 Unit Testing - Invoking Tests and Commandline 40

Part VI Troubleshooting 42

1 Troubleshooting Compilation Errors 42
2 No Output in Debug Viewer ^&@%# 42
3 __func__ Error or No FunctionName in Output 42

Part VII The GUI Debug Monitor Tool 44

1 Introduction to the GUI Tool 44
2 The User Interface 44
3 Viewer Options 45
4 Working with Files 45
5 Custom Columns 46
6 Using in MS Visual Studio IDE 46

Part VIII Extra Information 47

1 Version History and ToDo List 47
2 Fun Tricks with your C++ Compiler and Preprocessor 50
3 Related Tools 52

Index 0

©2004

Foreword

This is just another title page
placed between table of contents
and topics

1 Introduction

1.1 JrDebugLogger

An Open Source C++ Toolkit for Debug-Logging and Unit Testing

What is Debug-Logging?

Debug-Logging (sometimes known as printf-debugging) is when you add print statements to your code so that you can see the values of certains variables and where the program is at certain times. It also supports unit testing support functions.

This is the right tool for you if..

- You want to be able to disable the support code completely in release builds for **zero-impact on size and cpu** usage.
- You want a **cross-platform** solution that works in any C++ compiler.
- You want **flexibile output** (console based text, html, xml, custom GUI).

- You want to use standard syntax for debug logging, in **printf or stream io** formats.
- You want a **completely documented** system which is very easy to start using.
- You want support for lightweight **unit testing** support functions.

Jump directly to:
- See a Simple Example..
- Version History and ToDo List...
- Related Tools...
- Troubleshooting...

1.2 Features and Principles

What principles guided the design of JrDebugLogger?
- **Cross-platform** support, compile in **any C++ compiler**.
- There should be absolutely **no impact at all on the performance and size of your code if you disable the compilation** of the debug-logging feature.
- If you choose to disable but leave the debugging code compiled into your application in order to collect debug logs from users, the impact of the debugging code should be absolutely **minimal if not engaged**; should be easy to turn on and off.
- The debug output should be easily configured to **output via standard console, to a file, or through the standardized MS Windows system call DebugOutputString**
- Output should be possible in a variety of formats including **plaintext, xml, html, csv, etc.**
- The debugging library should provide **built-in support to parse commandline options** that configure debugging.

What are some key features of the toolkit?
- The syntax for debug-logging statements is unobtrusive and natural. You can **use normal printf and c++ stream io style formats,** or you can specify additional information with events, including custom fields.
- **Requires no special startup/shutdown code**; super easy to use.
- The JrDebug toolkit contains an **optional stand-alone Windows gui tool**, similar to the Debugview application, but cutomized for capturing/viewing/manipulating debug-logging events.
- Special functions for logging **assertions and exceptions; setting alarms and filters**.
- Very lighweight and easy to use **Unit Testing** support functions.
- Complete documentation.

What does JrDebugLogger include?
- A set of (cross-platform) c++ header files that you can add to your applications in order to support advanced debug-logging.
- A Microsoft Windows GUI application for capturing debug messages in real time and allowing you to view/sort/search them in an efficient manner.
- A visual component for Borland C++ Builder that can be used to capture DebugOutputString messages in your own BCB applications.

1.3 From README

```
//---------------------------------------------------------------------------
JrDebug Readme
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// Version History:
//  see help file
```

```
//------------------------------------------------------------------------



//------------------------------------------------------------------------
// Tested with the following compilers:
//  1. MS Visual C++ .Net2003 (7.1)          1.01.17* (main testing platform)
//  2. MS Visual C++ 6                       1.01.12
//  3. Intel C++                             1.01.12
//  4. Borland C++                           1.01.12
//  5. Borland C++ Builder 6                 1.02.01
//  6. Dev-C++ (mingw32)                     1.01.12
//  7. Digital Mars                          1.01.12
//  8. Cygwin Gcc                            1.01.12
//  9. Linux Gcc                             1.01.12
//  10.Greenhills C++                        1.01.12
//  11.Metroworks CodeWarrior                1.01.12
//  12.Comeau (using bcc)                    1.01.12
//------------------------------------------------------------------------

//------------------------------------------------------------------------
// ToDo:
//  make it thread safe
//  add support for unicode text
//  write new gui mode (hierarchical db)
//  write tester gui
//  write piped output code for *nix
//  benchmark memory and cpu
//  update documentation
//  add some compiler-specific hints
//  find better way to detect snprintf vs. _snprintf
//  add stdout summaries of each test, for picking up from tester gui
//  should exceptions not get treated as errors?
//  add support for specifying tests to run by name as well as keyword (fix keyword matching)
//  improve version announcement (include compiler info, date, exename, jrdebug version)
//  remove ATTN: extra comments that are out of date
//  add file Ahref links to html output (and smart linking via gui tester)
//  update gui tool icons
//  add template wrappers for registering test functions with return values and parameters
//  fix strange bug in gui monitor tool where incoming messages get stored wrong
//  make callback objects self-register
//  retest new stuff on different compilers
//  flush output after each message (files not updating till exit)?
//  fix subsystem:windows console output to send to current console if one is opened rather than open
new
//  windows console opened for output is too small, buffer history gets lost (can we fix and is there other
console flush settings we want)
//   should we implement a custom pop up log viewer for windows? (ie richedit) (ie -dbo ;winform)?
//  test with -wall and -ansi and other strict warnings
//  add more shortcut macros for different levels of output to make it easier to filter out annoyances
//   add enums for common levels (ie HighLevel,LowLevel,MedLevel,ErrorLevel,WarnLevel,etc.)
//   ie debugoutlow (dbprintflow) , debugouthigh (debugprintfhigh) for low and high priority
//   and add -dbf to set filter level (low, norm, high) (0,50,100)
```

```
//  add file size limits for -dbo output
//  add a column for reporting os+process specific info like thread id
//  add ability to specify for each instantiated formatter object to be told which fields to display/hids
//  add object / stl hex-dumping
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// OTHER ACCESSORY TOOLS INCLUDED WITH JrDebug:
//  1. A custom GUI monitoring/viewing utility that knows how to parse the JrDebug
//     generated debug messages exists for MS Windows and provides a convenient
//     way to view, sort, and filter debug messages (C++ Builder).
//  2. A component for catching debug messages has (C++ Builder).
//  3. A custom GUI tool for running tests (C++ Builder).
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// Customization:
//  See the file "jrdebug_switch.h" for switches you may want to play with,
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// Basic Usage Instructions
// ------------------------
//  Include this header file in all of your source code files that you want
//   to use debug output in.  Then include the file jrdebug_main.cpp
//   *once* and only once (usually in your main.cpp)
//
//  Then just make calls to JrDebugPrintf (or the short version dbprintf) as
//   you  would a normal printf statement.  If debugging is enabled, this line
//   will be sent via Windows OutputDebugString debugging protocol, or
//   to stderr if compiled under non-windows OS.  Extra info about location
//   in code will be added automatically.
//
//  ex:
//   dbprintf("This is a debugging message.");
//  OR use the stream-style output ex:
//   debugout << "This is a debugging message.";
//
//  resulting debug output:
//   "|*RunId*| 1091904190 |*TimeStamp*| 1091904193 |*Index*| 1 |*Type*| Message |*Level*| 50 |*File*| ".
\src\learningmodule.cpp" (Fri Aug  6 16:50:18 2004) |*Line*| 1118 |*Func*|
LearningModule::ReceiveSpecialSignal |*Message*| This is a debugging message."
//
```

```
// The extra info added includes filename, file line, and some other info, in
//   a format designed to be both human+machine readable.
// You can use any format modifiers and arguments that you would use with
//   printf (if you want to output a c++ string you can use dprints instead).
//
// NOTE: Defining the JrDebugDIRECTIVE_DontCompileDebugging macro in your code
//   will eliminate all debugging code, and result in no wasted overhead in
//   executing any debug statements.  This code was also designed to have a
//   very minimal impact on performance if it is compiled in but disabled.
//---------------------------------------------------------------------------



//---------------------------------------------------------------------------
// Intermediate Usage Instructions
// --------------------------------
// In addition to the standard dbprintf(..) format, you are encouraged to
//   use the more informative extended format, which adds fields for messagetype,
//   submessagetype, activitystring, and severity level.
//
// void dbprintf(const JrdType inmtype const int inseverity=50, const char* format, ... )
// ex:
//   dbprintf(JrdWarning,50,"This is a debugging message.");
//   dbprintf(JrdWarning,"This is a debugging message.");
// resulting debug output:
//   "|*RunId*| 1091904190 |*TimeStamp*| 1091904193 |*Index*| 1 |*Type*| Warning |*Level*| 50 |*File*| ".
\src\learningmodule.cpp" (Fri Aug  6 16:50:18 2004) |*Line*| 1118 |*Func*|
LearningModule::ReceiveSpecialSignal |*Message*| This is a debugging message."
//
// or
//
// void dbprintf(const JrdType inmtype , const char *dmsubtypestr, const char *inactivity, const int
inseverity, const char* format, ... )
// ex:
//   dbprintf(JrdWarning,"minor","testing",10,"This is a debugging message.");
// resulting debug output:
//   "|*RunId*| 1091904190 |*TimeStamp*| 1091904193 |*Index*| 1 |*Type*| Warning.minor |*Level*| 10 |
*Activity*| testing |*File*| ".\src\learningmodule.cpp" (Fri Aug  6 16:50:18 2004) |*Line*| 1118 |*Func*|
LearningModule::ReceiveSpecialSignal |*Message*| This is a debugging message."
//
// JrdType can be from: { JrdMessage, JrdWarning, JrdError, JrdAlarm, JrdNote , JrdTest , JrdCustom}
//---------------------------------------------------------------------------
//
// Stream Mode Usage
// ------------------
// You can use more modern stream style debug logging instead of the printf style if you prefer.
// Use the JrDebugStream (or debugout for short) stream as you would any other stream:
//    debugout << jrd_stdstringT("This is a loop iteration number ") << count;
// In addition, you can specify some optional parameters by passing them as follows:
//    debugout(JrdWarning) << jrd_stdstringT("This is a loop iteration number ") << count;
// Parameters available are the same as in the printf version:
//    (const JrdType inmtype const int inseverity=50, const char* format, ... )
//    or (const JrdType inmtype , const char *dmsubtypestr, const char *inactivity, const int inseverity,
```

```
const char* format, ... )
//---------------------------------------------------------------------------



//---------------------------------------------------------------------------
// Advanced Usage Instructions
// -------------------------------
//  There are a number of global/static properties you can manipulate.
//
//  Activities:
//    The debugger can maintain a stack of current activities and annotate all
//    output with the current activity hierarchy. This can be useful in
//    tracking recursive operations or nested function calls.
//          JrDebug::SetActivity(const char *inactivity);
//          JrDebug::PushActivity(const char *inactivity);
//          JrDebug::PopActivity(const char *inactivity);
//
//  Debug Block Activities
//     You can use automatically scoped debugblock helpers to time blocks
//     A block will announce start and end and elapsed time
//     A report can be used within a block to display current time offset
//  JrDebugBlock(const char *label);
//  JrDebugBlockReport(const char *label);
//
//  File Logging:
//    You can ask the debugger to log all messages to a disk file:
//      JrDebug::WriteToFilename(char *fname,bool appendmode);
//     use appendmode==true to append to any existing file, or false to overwrite.
//     by default logging occurs to both file and debug stream if this is enabled.
//
//  Filtering:
//    You can disable and limit logging at runtime:
//                      JrDebug::SetFilterLevelRange(int min,int max);
//                      JrDebug::SetEnabled(bool val);
//                      JrDebug::SetEnabledFile(bool val);
//                      JrDebug::SetEnabledStandard(bool val);
//                      JrDebug::SetEnabledStdout(bool val);
//
//  Assertions:
//    When JrDebugging is enabled, JrDebug takes over the normal assert()
//     and verify() macros, sending failures to the debug log before exiting
//     the application.  The assertion failure message is sent to both stdout
//     and stderr, and a windows MessageBox is shown if compiled on windows.
//    By default, asssertions that are not violated are not displayed, but
//     you can change this by calling:
//                      JrDebug::SetDisplayPassingAsserts(bool val);
//
//    If JrDebug compilation is disabled (see macro at top of file),
//     then the normal assert and verify macros are invoked.
//
//  Helper Functions:
//     To help you add optional debugging to your app via a commmanline parameter,
```

```
//      you can use the following function call from your main procedure (see demo):
//       bool JrDebug::GrabCommandlineDebugFile(int &argc,char **argv,bool disableifnofile=false, bool
dontchangeargcv=false)
//     That call will look for parameter(s) of the form '-dbf [filename]' on
//      the commandline (and strip them if found, modifying argc), and
//      then enable debugging and writes output to the file specified
//      (or jrdebugout.log by default).  Pass a 3rd parameter of true to this
//      function to disable logging if parameter not found.  In this way,
//      your program will run without debugging unless -dbf is passed on the
//      commandline.
//     You might also want to include a call to:
//       void JrDebug::Announce(char* appdescription,bool evenifdisabled=true,int argc=0, char
**argv=NULL)
//      which will generate a log line describing that JrDebug is running and its version #
//
// Alarms:
//  If you use the JrdAlarm type in any message, a messagebox will also be shown to user on windows
platforms.
//  You can also set a global alarm level via JrDebug::SetAlarmFilterLevel(int)
//   any message with severity level >= that level will trigger a messagebox.
//
// Exception Logging:
//  JrDebugThrow (dbthrow) is a replacement for the normal throw command (though note
//   that it uses a different calling convention:  JrDebugThrow(exception X, char * Y)
//  This will cause the normal exception, and will generate a logging message beforehand.
//  Exception logging is disabled by default, you can enable it with:
//   JrDebug::EnableExceptionLogging(int exceptionlevel=50);
//   exceptionlevel specifies the severity level assigned to exception messages (combine with Alarms to
get popup messageboxes)
//  You can redisable with:
//   JrDebug::DisableExceptionLogging();
//---------------------------------------------------------------------------


//---------------------------------------------------------------------------
// New unit testing functions
//  I haven't written proper documentation for the unit testing support functions yet;
//   and more output formats are needed to make these more pleasant to use, but
//   you can get a feel for how to do testing by looking at demo sample #9.
//---------------------------------------------------------------------------


//---------------------------------------------------------------------------
// Commandline Functionality
//
//  -dbol
//    list all available output styles
//
//  -dbo STYLE[:options][;FILENAME][;stdout][;sysout]
//   where STYLE is one of the outout styles (xml,html,comp,text,jrd)
//   and fields after the STYLE separated by ; represent output targets
//    FILENAME can contain some special flags (use "" to quote spaces):
//     Full Date: $DATE
```

```
//     Year: $YYYY, $YY
//     Month: $MMM, $MM, $M
//     Day: $DDD, $D
//     Unique Number: $#####, $####, $###, $##, $#
//     Executable Name: $EXE
//  if filename ends in -a then it will be opened in append mode (e.g. log.txt-a) otherwise existing contents
will be overwritten.
//
//
//
//  -dbd
//    disable debugging
//
//  -dbtl [KEYWORD[;KEYWORD]+]
//    list available tests (that match keyword)
//
//  -dbt [|KEYWORD[;KEYWORD]+]
//    run available tests (that match any keyword)
//
//  -dbf [filteroptions]
//    set filter based on options (ex. -dbf eb):
//       e = onlyshowerrors
//       b = show all blocks
//
//
// Examples:
//   myapp.exe -dbo xml;test.xml?stdout
//   myapp.exe -dbo html;results_$EXE_$MONTH_$###.html
//   myapp.exe -dbo jrd;sysout
//   myapp.exe -dbo comp:gcc;"C:\results.txt"
//---------------------------------------------------------------------------
```

2 Some Examples

2.1 A First Example

Here's a complete simple .cpp program that demonstates the basic logging commands.

```
//---------------------------------------------------------------------------
// Include the Debugging Library file (only our main should include this
//   file, additional .cpp files should include "jrdebug.h"
//  Alternatively, you could add this cpp to your project or makefile and
//   then just include "jrdebug.h" here, which is a more standard approach.
#include "jrdebug_main.cpp"
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
int main(int argc, char *argv[])
{
	// C++ iostreams style logging:
	debugout << "Main started with argc = "<<argc;
	// printf style logging:
	dbprintf("Main started with argc=%d",argc);
	// return success
	return 0;
}
//---------------------------------------------------------------------------
```

2.2 Simple Commandline Enabled Example

Here is a complete example that shows how easy it is to make an application that turns on debugging ONLY if the user passes the command line flag -dbc (for console output) or -dbf (for file output). Call your app with -dbc to see debug output on console, and/or -dbf to write debugging info to file. Add -dbb for briefmode output.

```
//---------------------------------------------------------------------------
// JrDebug simplest demo
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// System includes for std::cout usage below
#include <iostream>
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// Include the Debugging Library file (only our main should include this
//  file, additional .cpp files should include "jrdebug.h"
//  alternatively, you could add this cpp to your project or makefile.
#include "../../jrdebug_main.cpp"
//---------------------------------------------------------------------------


//---------------------------------------------------------------------------
int main(int argc, char *argv[])
{
	// automatically parse commandline options and exit if appropriate
	if (JrDebugParseCommandlineArgc("Demo1",&argc,argv,"-db"))
		return 0;
```

```
        // C++ iostreams style logging:
        debugout << "Main started with argc = "<<argc;
        // printf style logging:
        dbprintf("Main started with argc=%d",argc);

        // return success
        std::cout << "Program has finished; if you were running the debug monitor you
should have seen 2 messages."<<std::endl;
        return 0;
}
//-----------------------------------------------------------------------------
```

2.3 Configuring without Commandline Options

Here is a complete example that shows to set default initial options not using commandline.

```
//-----------------------------------------------------------------------------
// JrDebug simplest demo
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// System includes for std::cout usage below
#include <iostream>
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// Include the Debugging Library file (only our main should include this
//  file, additional .cpp files should include "jrdebug.h"
//  alternatively, you could add this cpp to your project or makefile.
#include "../../jrdebug_main.cpp"
//-----------------------------------------------------------------------------



//-----------------------------------------------------------------------------
int main(int argc, char *argv[])
{
        // set our own commandline options as one big string, not using actual comline
args
        JrDebug::ParseCommandlineArgString("-dbo text;debugout.txt",NULL,false);

        // C++ iostreams style logging:
        debugout << "Main started with argc = "<<argc;
        // printf style logging:
        dbprintf("Main started with argc=%d",argc);

        // return success
        std::cout << "Program has finished; if you were running the debug monitor you
should have seen 2 messages."<<std::endl;
        return 0;
}
//-----------------------------------------------------------------------------
```

2.4 A More Involved Example

Here's a more involved .cpp program that demonstates the logging commands. You will find this demo project in the DebugLibrary source code directory, along with project files for ms .net2003,vc6, and dev-c ++.

```
//-----------------------------------------------------------------------------
// JrDebug demo that demonstrates various features
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// System includes
#include <iostream>
#include <stdlib.h>
#include <assert.h>
#include <exception>
#include <stdexcept>
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// Include the Debugging Library files
//  Note that only our main .cpp needs to include the jrdebug_main.cpp file
//  alternatively, you could add this cpp to your project or makefile.
#include "../../jrdebug_main.cpp"
//-----------------------------------------------------------------------------



//-----------------------------------------------------------------------------
int main(int argc, char *argv[])
{
	// main function

	// automatically parse commandline options and exit if appropriate
	if (JrDebugParseCommandlineArgc("Demo3",&argc,argv,"-db"))
		return 0;

	// you want very brief human readable output? (this does not display well in
gui tool!)
	// JrDebug::SetBriefOutput(true);

	// if you like you can manually turn on file writing of debug statements
	// JrDebug::WriteToFilename("demologoutput.log",false);

	// for fun we let the tool announce itself
	JrDebugAnnounce("Test Application",true,argc,argv);

	// simple loop
	for (int count=0;count<10;++count)
		{
		dbprintf("Performing iteration #%d",count);
		// an assert that passes and won't be displayed
		assert(25<100);
		}

	// turn on display of even asserts that pass and do a passing assert; pass more
than 3 args to cause assert failure
```

```
        JrDebugSetDisplayPassingAsserts(true);
        assert(argc<4);

        // here are two more complicated debug statements (first one just specifies
type and severity level; second specifies also a subtype and the current 'activity')
        dbprintf(JrdWarning,25,"This is a debug warning statement.");
        dbprintf(JrdNote,"useless","testing",10,"This is another debug statement.");

        // here is another loop but with activity push/pop - this will report timings
of activities
        JrDebugActivityPush("Looping");
        int count3,count4,count5;
        for (int count2=0;count2<10;++count2)
                {
                JrDebugActivityPush("iterate");
                // take some time
                for (count3=0;count3<3000;++count3)
                        {
                        // now here is a manual push pop, and further more, we use a * in
front of description to avoid display (only used if other messages displayed inside)
                        JrDebugActivityPush("*innerloop");
                        for (count4=0;count4<3000;++count4)
                                {
                                // just eat up some cpu cycles
                                count5=count3-count2; count5++; count5--; count5=count3-
count2;
                                }
                        JrDebugActivityPop("*innerloop");
                        }
                JrDebugActivityPop("iterate");
                }
        JrDebugActivityPop("Looping");

        // the JrdAlarm type always triggers a windows messagebox if compiled on
windows
        // dbprintf(JrdAlarm,1,"This is an alarm.");

        // want to try some stream style output?
        debugout << "here is stream debug mesg (argc="<<argc<<").";
        debugout(JrdTest,"stream","testing",100) << "here is another more specific
stream debug message";


        dbprintf("Program has finished, now testing an exception.");

        // how about a debuglogged exception
        // JrDebug::EnableExceptionLogging();
        // // // for fun we can also turn on an alarm filter to generate an additional
pop up messagebox for severity levels >=10
        // // // JrDebug::SetAlarmFilterLevel(5);
        // // the debuglogged repalcement for:  throw exception();
        try
                {
                JrDebugThrow(std::exception(),"testing an exception throw");
```

```
                }
        catch (...)
                {
                }


        // new test to verify we dont get double printf-style arge replacements
        std::string troublestring ="C:\\Mydrive%d%s.";
        dbprintf("Here is my troublesomestring '%s'.",troublestring.c_str());

        // exit
        std::cout << "Hit 'Enter' key to exit."<<std::endl;
        std::cin.get();
        return 0;
}
//---------------------------------------------------------------------------
```

2.5 Different Kinds of Messages

Here's a simple example showing different kinds of messages. You will find this demo project in the DebugLibrary source code directory, along with project files for ms .net2003,vc6, and dev-c++.

```
//---------------------------------------------------------------------------
// System includes
#include <iostream>
#include <exception>
#include <stdexcept>
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// Include the Debugging Library file (only our main should include this
//  file, additional .cpp files should include "jrdebug.h"
//  alternatively, you could add this cpp to your project or makefile.
#include "../../jrdebug_main.cpp"
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
int main(int argc, char *argv[])
{
        int count;
        int max=20;
        debugout << "Starting strees test of messages.";

        // loop
        for (count=1;count<=max;++count)
                dbprintf(JrdMessage,"test #%d of %d",count,max);
        for (count=1;count<=max;++count)
                dbprintf(JrdWarning,"test #%d of %d",count,max);
        for (count=1;count<=max;++count)
                dbprintf(JrdError,"test #%d of %d",count,max);
        for (count=1;count<=max;++count)
                dbprintf(JrdNote,"test #%d of %d",count,max);
        for (count=1;count<=max;++count)
                dbprintf(JrdTest,"test #%d of %d",count,max);
```

©2004

```
	for (count=1;count<=max;++count)
		dbprintf(JrdCustom,"test #%d of %d",count,max);

	// enable some stuff
	JrDebug::EnableExceptionLogging();
	JrDebug::SetDisplayPassingAsserts(true);

	// only one alarm
	dbprintf(JrdAlarm,"alarm single test");
	// an assert
	dbassert(count<1000);
	// an exception
	try
		{
		dbthrow(std::exception("jr exception"),"");
		}
	catch (...)
		{
		}


	// done
	debugout << "Ending stress test of messages.";

	// return success
	std::cout << "Program has finished; if you were running the debug monitor you
should have seen 2 messages."<<std::endl;
	return 0;
}
//----------------------------------------------------------------------------
```

2.6 Using a Local Callback Monitor

This example shows how to implement a callback in your code so you get notified of all debug messages that your application sends. You will find this demo project in the DebugLibrary source code directory, along with project files for ms .net2003,vc6, and dev-c++.

```
//----------------------------------------------------------------------------
// System includes for std::cout usage below
#include <iostream>
//----------------------------------------------------------------------------

//----------------------------------------------------------------------------
// Include the Debugging Library file (only our main should include this
//  file, additional .cpp files should include "jrdebug.h"
//  alternatively, you could add this cpp to your project or makefile.
#include "../../jrdebug_main.cpp"
//----------------------------------------------------------------------------

//----------------------------------------------------------------------------
// Helper callback object - can be used if user code want to intercept events
//  To use, derive your own version of this class, create an instance of it, and pass
it to the static
```

```
//  function JrDebug::SetJrDebugCallbackPointer(class JrDebugCallback *p);
// Callback should return true to process as normal, or false to not send output as
normal
class MyDebugCatcher : public JrDebugCallback {
public:
	virtual bool EventNotify(const char *fulltext) { std::cout << "Local debug
catcher got: "<<fulltext;return true;};
};
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
int main(int argc, char *argv[])
{
	// set up local catcher
	JrDebug::SetBriefOutput(true);
	MyDebugCatcher mycatcher;
	JrDebug::SetCallbackPointer(&mycatcher);

	// C++ iostreams style logging:
	debugout << "Main started with argc = "<<argc;
	// printf style logging:
	dbprintf("Main started with argc=%d",argc);

	// return success
	std::cout << "Program has finished; if you were running the debug monitor you
should have seen 2 messages."<<std::endl;
	return 0;
}
//---------------------------------------------------------------------------
```

2.7 Outputing Custom Columns

This example shows how to output custom columns for the MS Windows Debug Viewer tool, which will automatically create custom columns for you. You will find this demo project in the DebugLibrary source code directory, along with project files for ms .net2003,vc6, and dev-c++.

```
//---------------------------------------------------------------------------
// System includes for std::cout usage below
#include <iostream>
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// Include the Debugging Library file (only our main should include this
//  file, additional .cpp files should include "jrdebug.h"
//  alternatively, you could add this cpp to your project or makefile.
#include "../../jrdebug_main.cpp"
//---------------------------------------------------------------------------



//---------------------------------------------------------------------------
int main(int argc, char *argv[])
{
	// start
	int count;
	int max=20;

	// loop - note the custom column headers |*COLUMNNAME*| value
```

```
	for (count=1;count<=max;++count)
		dbprintf(JrdMessage,"test #%d of %d |*CustomTestNum*| 1",count,max);
	for (count=1;count<=max;++count)
		dbprintf(JrdWarning,"test #%d of %d |*CustomTestNum*| 2",count,max);
	for (count=1;count<=max;++count)
		dbprintf(JrdError,"test #%d of %d |*CustomTestNum*| 3",count,max);
	for (count=1;count<=max;++count)
		dbprintf(JrdNote,"test #%d of %d |*CustomTestNum*| 4",count,max);
	for (count=1;count<=max;++count)
		dbprintf(JrdTest,"test #%d of %d |*CustomTestNum*| 5",count,max);
	for (count=1;count<=max;++count)
		dbprintf(JrdCustom,"test #%d of %d |*CustomTestNum*| 6",count,max);

	// return success
	std::cout << "Program has finished."<<std::endl;
	return 0;
}
//---------------------------------------------------------------------------
```

2.8 Using DebugBlocks

This example shows how to use the debugblock construct; it will automatically report the start and end of scoped blocks, and the time taken; you can also use the debugblockreport() function to report current offset time within the current block. You will find this demo project in the DebugLibrary source code directory, along with project files for ms .net2003,vc6, and dev-c++.

```
//---------------------------------------------------------------------------
// System includes
#include <iostream>
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// Include the Debugging Library files
//  Note that only our main .cpp needs to include the JRutils_debugout_main.h file
//  alternatively, you could add this cpp to your project or makefile.
#include "../../jrdebug_main.cpp"
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
int main(int argc, char *argv[])
{
	// disable or enable?
	// JrDebug::SetEnabled(false);

	// main function report
	debugblock("mymain");

	// here is another loop but with activity push/pop - this will report timings
of activities
	// note we enclose it in a {} to properly scope the debugblock
	{
	debugblock("all looping");
	int count3,count4,count5;
	for (int count2=0;count2<10;++count2)
		{
```

```
            // report this loop block
            debugblock("inner loop");
            // take some time
            for (count3=0;count3<30000;++count3)
                  {
                  for (count4=0;count4<300;++count4)
                        {
                        // just eat up some cpu cycles
                        count5=count3-count2; count5++; count5--; count5=count3-
count2;
                        }
                  }
            }
      }

      // we can also just report current offset time of nearest enclosing debugblock
(main)
      debugblockreport("before ending");

      // anything here is outside of scope of the "all looping" debugblock
      dbprintf("Program has finished.");

      // all done
      return 0;
}
//---------------------------------------------------------------------------
```

2.9 Using JrDebugEnclose

This example shows how to use the the JrDebugEnclose macro with the debugblock construct; it will automatically report the start and end of scoped blocks, and the time taken; you can also use the debugblockreport() function to report current offset time within the current block. You will find this demo project in the DebugLibrary source code directory, along with project files for ms .net2003,vc6, and dev-c ++.

```
//---------------------------------------------------------------------------
// System includes
#include <iostream>
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// Include the Debugging Library files
//  Note that only our main .cpp needs to include the JRutils_debugout_main.h file
//  alternatively, you could add this cpp to your project or makefile.
#include "../../jrdebug_main.cpp"
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// Forward declaration
int WorkFunction();
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
int main(int argc, char *argv[])
{
```

```
        // call the work function and report it + time it
        int retv;
        JrDebugEnclose(WorkFunction());
        JrDebugEnclose(retv=WorkFunction());

        dbprintf("Program has finished: %d.",retv);

        // all done
        return 0;
}
//-----------------------------------------------------------------------------


//-----------------------------------------------------------------------------
int WorkFunction()
{
        int count3,count4,count5;
        for (int count2=0;count2<10;++count2)
                {
                // report this loop block
                debugblock("inner loop");
                // take some time
                for (count3=0;count3<30000;++count3)
                        {
                        for (count4=0;count4<300;++count4)
                                {
                                // just eat up some cpu cycles
                                count5=count3-count2; count5++; count5--; count5=count3-
count2;
                                }
                        }
                }
        // return dummy value
        return 999;
}
//-----------------------------------------------------------------------------
```

2.10 Unit Testing with JrDebugLogger

This example shows how to use the basic unit testing support functions. You will find this demo project in the DebugLibrary source code directory, along with project files for ms .net2003,vc6, and dev-c++.

```
//-----------------------------------------------------------------------------
// JrDebug demo that demonstrates various features
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// System includes
#include <iostream>
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// Include the Debugging Library files
//  Note that only our main .cpp needs to include the JRutils_debugout_main.h file
//  alternatively, you could add this cpp to your project or makefile.
```

```
#include "../../jrdebug_main.cpp"
//---------------------------------------------------------------------------



//---------------------------------------------------------------------------
// code for the unit tests
void MyUnitTest()
{
	dbprintf("my unit test!");
}
void MyUnitTest2()
{
	dbprintf("my unit test that should throw an exception.");
	// two tests, one that shouldnt fail and will abort further testing if it does
	dbtestex(10<100);
	// and another test that will fail and generate a message
	dbtest(100<10);
	// now throw an exception to show how it will be caught by the testing system
	throw "mystrexception";
}

class DummyClass {
	// this demonstrates that you can register static class functions as unit test
functions
	public:
	static void ClassStaticTest() {dbprintf("in static class member test");};
};

// register the unit tests - this is all you have to do to tell the system about your
test functions
JrDebugRegisterTestFunction("simpletst","Simple Unit Test",MyUnitTest,"simple");
// note how we can register a (static) member class function as a test function
JrDebugRegisterTestFunction("statictest","Static Class Member
Test",DummyClass::ClassStaticTest,"advanced");
//---------------------------------------------------------------------------





//---------------------------------------------------------------------------
int main(int argc, char *argv[])
{
	// announce
	printf("Demo program, to run unit tests call on commandline with -dbt");

	// if you really want to you can register test functions from within code
instead of globally
	// JrDebugRegisterTestFunction("simpletest","Simple Unit Test
#2",MyUnitTest2,"advanced");

	// normal commandline parsing, and will turn on tests if appropriates
	JrDebug::GrabCommandlineDebugFile(argc,argv,true);
	// exit right away if tests run - you might want to add this to your code if
you want -dbt to run tests and then exit
	if (JrDebug::RanTests())
```

```
            return 0;

        // force tests to run manually in this particular demo?
        if (!JrDebug::RanTests())
            {
            printf("Running tests manually since they weren't invoked on
commandline.");
            // let the debugger run all registered tests
            JrDebug::RunTests("all");
            // or we could manually invoke functions without even registering them
            // JrDebugRunTestFunction("Manual Test#2",MyUnitTest2);
            }

        // all done
        return 0;
}
//---------------------------------------------------------------------------
```

2.11 More Samples

The samples directory in the source code includes more samples than those shown in the help file.

3 Basic Usage

3.1 Suggested Installation

Here is how we recommend you install JrDebugLogger so that you can most easily use it in your applications:

1. Copy the JrDebugLogger directory somewhere permanent where you can easily refer to it from your other project (for example, **/libraries/jrdebug** or **C:\Libraries\JrDebug**).
2. For projects that want to use JrDebug support, add the directory with the JrDebugLogger header files to your project include directory path.
3. *Optional:* If you want to customize the JrDebug compilation options for different projects, copy the jrdebug_switch.h file into your program's directory (this will let you customize whether to enable/ disable logging for your application without effecting others).

3.2 Adding the JrDebugLogger files to Your Project

One **and only one** of your source code files, usually the one containing your main() procedure, should #include "jrdebug_main.cpp" (it defines some static variables and you will get a link error if you forget to include it).

Alternatively, you could add this cpp to your project or makefile, and just use the #include "jrdebug.h" here too.

```
//---------------------------------------------------------------------------
// One (and only one) .cpp in your application should do:
#include "jrdebug_main.cpp"
//---------------------------------------------------------------------------
```

All other files in your application that need to utilize some debugging functions should #include the "jrdebug.h" file.

```
//---------------------------------------------------------------------------
// Other files in your application that want to use debug logging:
#include "jrdebug.h"
//---------------------------------------------------------------------------
```

The other header files in the JrDebug directory (and jrdebug_switch.h) are referenced automatically by the jrdebug.h file and you do not need to reference them from your code.

3.3 Adding Basic Logging Statements

JrDebugLogger was designed to let you add debug logging statements to your code in a syntax identical to normal output statements You may use both printf-style debug statements and stream io style debug statements. Some aliases are also defined to let you use the identifiers you prefer (short lowercase names or longer uppercase names);

Basic printf-style debug logging statements:

```
dbprintf("This is a debugging message.");
dbprintf("There are %d items.",count);
```

The above statements could also be written as:

```
JrPrintf("This is a debugging message.");
```

```
JrPrintf("There are %d items.",count);
```

Basic stream io style debug logging statements:

```
debugout << "This is a debugging message.";
debugout << "There are "<<count<<" items.";
```

Or alternatively:

```
JrDebugStream << "This is a debugging message.";
JrDebugStream << "There are "<<count<<" items.";
```

You can mix and match printf style and stream style output; sometimes printf style can be easier for unusual formatting requirements, but stream output is safer and is easier to use with arbitrary objects.

NOTE: The examples above show only the most basic use of debug logging statements; see the advanced section for more sophisticated ways to add information to your statements.

3.4 Viewing Debug Messages With the GUI Tool

There are two main parts that comprise the JrDebugLogger toolkit, a set of cross-platform header files for adding debug logging support to your code, and a powerful windows program for capturing and displaying logging messages.

To get the maximum use out of JrDebugLogger, you will use a gui tool that knows how to capture and properly display messages that are generated from JrDebugLogger.

The JrDebugLogger files include code that determine if the application is compiled on MS Windows; if so, debug messages are send to the operating system using the **DebugOutputString** function. DebugOutputString is a function call that can be used in windows applications to broadcast a text message to any debugger applications that are listening for it.

The MS Windows Debug Monitor Viewing gui tool that comes with the JrDebugLogger toolkit lets you capture and view DebugOutputString messages, and can parse the output into columns to make it ideally suited for viewing, searching, sorting, and organizing your logging output.

JrDebugMonitor - v1.00.10

File.. | Capturing Events | Options.. | Erase Current Data | Auto Size Columns | Help..

RunId

Index	Icon	Type	Message	TimeStamp	Level	File	Line	Func
RunId : 8/15/2004 4:40:34 AM (124)								
RunId : 8/15/2004 4:40:40 AM (17)								
1		Message	Starting strees test of messages.	8/15/2004 4:40:40 AM	50	main.cpp	25	main
22		Warning	test #1 of 20	8/15/2004 4:40:41 AM	50	main.cpp	31	main
41		Warning	test #20 of 20	8/15/2004 4:40:42 AM	50	main.cpp	31	main
42		Error	test #1 of 20	8/15/2004 4:40:42 AM	50	main.cpp	33	main
58		Error	test #17 of 20	8/15/2004 4:40:42 AM	50	main.cpp	33	main
61		Error	test #20 of 20	8/15/2004 4:40:42 AM	50	main.cpp	33	main
62		Note	test #1 of 20	8/15/2004 4:40:42 AM	50	main.cpp	35	main
81		Note	test #20 of 20	8/15/2004 4:40:43 AM	50	main.cpp	35	main
98		Test	test #17 of 20	8/15/2004 4:40:44 AM	50	main.cpp	37	main
100		Test	test #19 of 20	8/15/2004 4:40:44 AM	50	main.cpp	37	main
101		Test	test #20 of 20	8/15/2004 4:40:44 AM	50	main.cpp	37	main
102		Custom	test #1 of 20	8/15/2004 4:40:44 AM	50	main.cpp	39	main
118		Custom	test #17 of 20	8/15/2004 4:40:44 AM	50	main.cpp	39	main
121		Custom	test #20 of 20	8/15/2004 4:40:44 AM	50	main.cpp	39	main
122		Alarm	alarm single test	8/15/2004 4:40:44 AM	50	main.cpp	46	main
123		AssertPass	count<1000	8/15/2004 4:40:44 AM	0	main.cpp	48	main
124		Message	Ending stress test of messages.	8/15/2004 4:40:44 AM	50	main.cpp	60	main
RunId : 8/15/2004 4:41:03 AM (124)								
RunId : 8/15/2004 4:41:10 AM (124)								

Records: 1133 / 1133

Start the Debug Monitor Viewer application and then run any program that has been compiled with JrDebugLogger supprt. Any debug messages sent by the program will be captures in the Viewer automatically - there is no need to do anything special. To get the best use out of the viewer, you will want to customize the columns displayed and the grouping of columns.

3.5 Turning Off Compilation of DebugLogging Code

A main design guideline for the JrDebugLogger code was that it be easy to tell your compiler to compile code that removed all debug log related code from the compiled application, so as to ensure no performance costs for code generated without debugging functions.

The options described here are configured through the **jrdebug_switch.h** file, or can be **overridden by passing -D define statements** in your project or makefile.

There are actually two ways to turn off debug output from your code:

- You can completely turn off the compilation of the debug code - the actual debug logging code is not compiled-in, and should have absolutely no impact on your code at all.
- You can leave the code compiled in, but *disable* it at startup (and re-enable it when you want).

The second option can be very useful if you want to allow your program to be run with debugging enabled, and JrDebugLogger even includes a simple commandline parsing helper function to automatically look for the -dbf commandline option and enable debug logging to file if it is found.

Even when debug logging code is compiled-in but disabled, JrDebugLogger was designed to have absolutely minimal impact on runtime performance. None of debug output function calls are evaluated when logging is disabled - all calls amount to just a simple compare and jump statement.

You can control the conditional compilation of debugging functions by defining a symbol described in the file **jrdebug_switch.h**. Just uncomment lines to build code that has all debugging functions removed or disabled.
In fact, the **jrdebug_switch.h** file serves no other purpose than to let you specify these compilation flags:

```
// Usually you want to automatically enable logging when you build
//  in Debug mode, and not compile-in logging when you build in release mode
//  (signified when symbol NDEBUG is defined); this also checks for the symbols
//  DEBUG and __DEBUG__ to indicate a debug compilation switch.
#define JrDebugDIRECTIVE_ObeyNDEBUG

// If you want to always disable all debugging and eliminate all runtime
//  overhead, unless overriding with -D define, just uncomment this line to
//  prevent debugging code from being compiled in:
// #define JrDebugDIRECTIVE_DontCompileDebugging

// If you want to compile in the ability to turn on debugging at runtime, but
//  turn off debugging at startup, you can define the following symbol (note
//  it has no effect if debugging is compiled out):
// #define JrDebugDIRECTIVE_StartupDisabled

// If this symbol is not set then a minimal extra function will be compiled-in
//  even when debugging is set to not compile-in, which will warn any
//  user if they try to invoke a debug commandline flag.
#define JrDebugDIRECTIVE_CompiledOutCommandlineParseOff
```

See also: The advanced discussion of compiling-in options for instructions on working with the jrdebug_switch.h file and compiler flag overrides.

4 Advanced Usage

4.1 Logging Functions

4.1.1 Printf Style

Basic printf-style debug logging statements look like this:

```
dbprintf("This is a debugging message.");
dbprintf("There are %d items.",count);
```

But you can also specify more information in your dbprintf (aka JrDebugPrintf) function calls, which add useful information to your debugging statements. To get the most out of JrDebugLogger you should learn to specify this additional information.

Some examples of providing additional information:

```
// add message type:
dbprintf(JrdWarning,"This is a debugging message.");
// add message type and severity level:
dbprintf(JrdWarning,100,"This is a debugging message.");
// add message type, subtypestring, current activity, and severity level:
dbprintf(JrdError,"minor","testing",100,"This is a debugging message.");
```

This extra information is mainly just to help you identify the messages - and shows up as different columns in the Debug Monitor Viewer. However some of these extra fields can be used to control logging behavior:

- You can set filters on severity levels to indicate that certain severity levels should be ignored; you can set alarms to trigger on messages with severity levels above some value.

Prototypes for dbprintf:

- `void operator()(const std::string str);`
- `void operator()(const char* format, ... );`
- `void operator()(const JrdType inmtype , const std::string str);`
- `void operator()(const JrdType inmtype , const char* format, ... );`
- `void operator()(const JrdType inmtype , const int inseverity, const std::string str);`
- `void operator()(const JrdType inmtype , const int inseverity, const char* format, ... );`
- `void operator()(const JrdType inmtype , const char *dmsubtypestr, const char *inactivity, const int inseverity, const std::string str);`
- `void operator()(const JrdType inmtype , const char *dmsubtypestr, const char *inactivity, const int inseverity, const char* format, ... );`

Message types are specified with the JrdType enum:

```
enum JrdType { JrdMessage, JrdWarning, JrdError, JrdAlarm, JrdNote , JrdTest ,
JrdCustom , JrdAssertPass, JrdAssertFail, JrdException };
```

Special handling of Message Types:

- The following message types are special and should not be specified manually: **JrdAssertPass, JrdAssertFail, and JrdException**.
- If the JrdCustom type is specified, the text from the dmsubtypestr is used alone for the Type field; otherwise the Type field will be messagetype.subtype (ie "Error.minor".
- Any message sent of type JrdAlarm will trigger a popup windows messagebox (or an output to stdout if compiled on *nix), to alert the user to the event even if a debug monitor is not engaged.

4.1.2 Stream Style

Basic printf-style debug logging statements look like this:

```
debugout << "This is a debugging message.";
debugout << "There are " << count << " items.";
```

But you can also specify more information in your stream (aka JrDebugStream) invocations, which add useful information to your debugging statements. To get the most out of JrDebugLogger you should learn to specify this additional information.

Some examples of providing additional information:

```
// add message type:
debugout(JrdWarning) << "This is a debugging message.");
// add message type and severity level:
debugout(JrdWarning,100) << "This is a debugging message.";
// add message type, subtypestring, current activity, and severity level:
debugout(JrdError,"minor","testing",100) << "This is a debugging message.";
```

This extra information is mainly just to help you identify the messages - and shows up as different columns in the Debug Monitor Viewer. However some of these extra fields can be used to control logging behavior:

- You can set filters on severity levels to indicate that certain severity levels should be ignored; you can set alarms to trigger on messages with severity levels above some value.

Prototypes for stream configuration:

- `operator()(const JrdType inmtype , const int inseverity=JrDebugDefaultLevel);`
- `operator()(const JrdType inmtype , const char *indmsubtypestr, const char *inactivity="", const int inseverity=JrDebugDefaultLevel);`

Message types are specified with the JrdType enum:

```
enum JrdType { JrdMessage, JrdWarning, JrdError, JrdAlarm, JrdNote , JrdTest ,
JrdCustom , JrdAssertPass, JrdAssertFail, JrdException };
```

Special handling of Message Types:

- The following message types are special and should not be specified manually: **JrdAssertPass, JrdAssertFail, and JrdException**.
- If the JrdCustom type is specified, the text from the dmsubtypestr is used alone for the Type field; otherwise the Type field will be messagetype.subtype (ie "Error.minor".
- Any message sent of type JrdAlarm will trigger a popup windows messagebox (or an output to stdout if compiled on *nix), to alert the user to the event even if a debug monitor is not engaged.

4.2 Logging Other Events

4.2.1 Alarms

Any message sent with dbprintf or debugout of type JrdAlarm will trigger a popup windows messagebox (or an output to stdout if compiled on *nix), to alert the user to the event even if a debug monitor is not engaged.

You can also tell the JrDebugLogger system to flag any message above or equal to a certain severity level is treated as an alarm (and therefore to pop up a messagebox), by calling the global function:

```
JrDebugSetAlarmFilterLevel(int severitylevel);
```

4.2.2 Activities

JrDebugLogger uses the concept of "Activities" to help you indicate what your program is currently doing when it sends debug logging messsages.

You can specify a specific "current activity" along with any message you send, but you can push and pop activities to help you keep track of recursive operations and blocks of operations without manually specifying activities:

```
JrDebugSetActivity(const char *inactivity,bool announce);
JrDebugActivityPush(const char *inactivity,bool announce);
JrDebugActivityPop(const char *dummyp=NULL,bool announce);
```

An example:

```
JrDebugActivityPush("Looping");
for (int count=0;count<10;++count)
     {
     JrDebugActivityPush("InnerLooping");
     for (int count2=0;count2<10;++count2)
            dbprintf("Performing iteration %d,%d",count,count2);
     JrDebugActivityPop("InnerLooping");
     }
JrDebugActivityPop("Looping");
```

In this example, each iteration of the loop will have the activity "Looping.InnerLooping".

When *announce* is true, the change of activity will be reported in the log.

Activities can also be used for timing activities; any time an activity finishes, ie a PopActivity is logged (announce==true), the log will describe how long the activity took.

4.2.3 DebugBlocks

In addition to Activities, you can use DebugBlocks to help you very easily report the start and end of a block of statements, including timing information. While Activities must be pushed and popped, DebugBlocks take advantage of the scoping mechanism in c++ to automatically report the start and end of blocks.

Inside of a function or code block {} you create a DebugBlock using:

```
JrDebugBlock("label string");
```
or
```
debugblock("label string");
```

The block will be announced at start, and automatically When the block finishes.
You can also report the current time offset within a block:

```
JrDebugBlockReport("label string");
```
or
```
debugblockreport("label string");
```

For example:

```
void myfunction()
{
debugblock("myfunction");
debugblockreport("here");
}
```

Will result in 3 messages:
"starting myfunction"
"at myfunction.here (xxx seconds)"
"finishing myfunction (xxx seconds)"

4.2.4 Assertions

Assert and Verify statements can be added to your code in order to check for conditions that you expect to be true, and which signify a major error if not.
The code to perform normal Assert tests is automatically not compiled in when building release versions, to minimize overhead costs for tests that are not expected to be violated in relase code. Verify tests are compiled in even in release code.

JrDebugLogger provides replacement functions for assert and verify which can be used to log violations (or passing tests) before triggering normal assert/verify behavior:

Instead of:
```
assert(condition)
verify(condition)
```
use:
```
dbassert(condition)
dbverify(condition)
```

You can enable and disable the logging of passing assertions (disabled by default):
```
JrDebugSetDisplayPassingAsserts(bool val);
```

JrDebugLog will trigger a messagebox or stderr output (when compiled on *nix) when assert/verify tests fail, and then exit the program immediately.

If debug log compilation is disabled, these calls map to the normal assert/verify macros. **BUT** if debug logging code is compiled in, assert tests will always be checked, even when building relase mode builds.

There is also a setting in the **jrdebug_switch.h** file that can be set to have JrDebug compilation redefine the normal ASSERT, VERIFY, and TRACE macros to map to the JrDebug equivelants, which is disabled by default. This might be useful for working with legacy code you dont want to modify:

```
#define JrDebug_TakeoverAssertTrace
```

4.2.5 Exceptions

JrDebugLog can be used to easily log any thrown exceptions.

To log exceptions, change (you can also use JrDebugThrow if you prefer the longer version):

```
throw Exception
```

to

```
dbthrow(Exception,char *reason)
```

When debugging is compiled out, these statements are reduced to the normal throw statements; when debugging is compiled in, the exceptions are thrown as normal after a debug logging event is triggered.

Because some applications make heavy use of exceptions for normal processing, you may want to disable the logging of exceptions by default:

```
JrDebugDisableExceptionLogging();
```

You can enable it (and specify the severity level to assign to exceptions):

```
JrDebugEnableExceptionLogging(int exceptionlevel=50);
```

You can combine this with the setting of alarm levels if you want to have a popup notify you when exceptions are triggered.

4.2.6 Timing Activities

You can use JrDebugLogger to easily time activities; see the help chapters on Activities and DebugBlocks for more details.

4.3 Configuration of Debug Logging Options

4.3.1 Logging Destinations and Filters

@torevise -- see source code

4.3.2 Text Formatting Mode

@torevise -- see source code

4.4 Helper Functions

4.4.1 Commandline Parsing and Enabling

SEE SAMPLE SOURCE CODE FOR BETTER EXPLANATION

In order to make it easier for you to enable commandline configuration of logging functions, JrDebugLogger provides a helper commandline parser function. When invoked it will scan the

©2004

commandline for specific arguments and. if found, use them to enable or disable debugging; any arguments found will be removed from your argc and argv variables so that your program can parse the rest of the commandline arguments as per normal.

Invoke the commandline parser helper function from your main() function as follows:

```
JrDebug::GrabCommandlineDebugFile(int &argc,char **argv,bool disableifnofile=false,
bool donthangeargcv=false)
```

The function returns true if debugging was enabled through the commandline.

```
int main(int argc, char *argv[])
{
     // automatically enable if -dbc or -dbf parameters on commandline; disable if
not
     JrDebug::GrabCommandlineDebugFile(argc,argv,true);
}
```

Invoke like:

```
demo1.exe -dbo text;debugout.txt
```

Options:

```
-db                  enable debug logging
-dbc                 enable debug logging to stdout console
-dbo STYLE[:options][;FILENAME][;stdout][;sysout][;info]    send debug output to
file (and/or stdout,sysout), optionally with info about where the debug file is
being written
-dbb                 enable brief mode debug logging
-dbd                 disable debug logging
```

IMPORTANT NOTE:

You can also call a simpler commandline initialization function that doesn't use argc and argv:

```
     JrDebug::ParseCommandlineArgString("commandline args as one long
string",fullpathofexe|NULL,bool &foundcommandlineargs);
```

ex:

```
     JrDebug::ParseCommandlineArgString("-dbo text;debugout.txt",NULL,false);
```

4.4.2 Startup and Announcing

It may be useful to ask the JrDebugLog functions to generate an initial output message announcing that it is running and its current version on startup. That way you can see that debug messages are getting through:

```
void JrDebug::Announce(char* appdescription,bool evenifdisabled, int argc, char
**argv)
```

By calling it with evenifdisabled=true, the announcement will be made even if debugging is disabled (though no output will be seen if debugging is 'compiled-out'). The announcement will look like:

```
JrDebug version 1.00.09 - Test Application - (enabled=1 asserts=0 exceptions=0) -
```

```
ComlineArgs(3) [demo,myarg1,myarg2]
```

If you want to make sure that you don't release an application with the debug code accidentally compiling-in or compiling-out, you can use the following calls, which will result in **compilation** errors if they are not met:

```
JrDebugLog::ErrorIfNotCompiledIn();
JrDebugLog::ErrorIfCompiledIn();
```

4.4.3 Optional Callback Object

Sometimes you may want to capture your own debug logging events.

The easiest way to do this is using the JrDebugCallback class. Make your own class, derived from:

```
class JrDebugCallback {
public:
     virtual bool EventNotify(const char *fulltext) {return true;};
};
```

Your derived class may invoke a global function or a class member function inside its EventNotify code. Create an instance of your derived class, and register it with the global JrDebug class:

```
JrDebug::SetCallbackPointer(class JrDebugCallback *inp);
```

Your EventNotify will be called before the event is written out; return false to prevent it from being written to any output; return true to process as normal.

4.5 Managing Compilation and Release

4.5.1 Application Switch Files and Compile Flags

There are a few different ways to control whether your application is built with debug logging compiled in and enabled.

For basic introduction on setting the flags or commandline options, see the introductory topic, which explains how to modify the values in the **jrdebug_switch.h** file to configure compilation.

Different options for different applications:

The reason that we use a separate file (jrdebug_switch.h) to hold the flags to enable compilation is to accomodate users who want to use a **single** copy of the JrDebugLog header files, but want to be able to configure multiple applications differently. Because of compilation order issues, you cannot simply add the compilation #define macros to your own application code.

The way to accomplish this is to make a ***copy*** of the **jrdebug_switch.h** file inside your application directory. Then ***REMOVE*** this file from the main DebugLibrary directory containing jrdebug.h (you may have to adjust your project include directory paths to make sure it will look in your application directory). In this way, when your applications are compiled, and jrdebug.h tries to #include jrdebug_switch.h, it will find the copy in your application directory, allowing you to use different versions of jrdebug_switch.h for each application.

Overiding jrdebug_switch.h:

You may also overide the settings in the **jrdebug_switch.h** file by passing options to your c++ compiler. Use the **-DJrDebugNoSwitchfile** to tell the compiler not to parse the switch file. You can then *add* **-DJrDebugDIRECTIVE_StartupDisabled** or **-**

DJrDebugDIRECTIVE_DontCompileDebugging to set the mode explicitly.

Disabling Windows-specific Extras when Building on Windows:
It may be useful to test compiling your code with debugging and avoiding ANY windows-related functions and includes from the JrDebug files; these are normally used to add some extra functionality like determining a safe path to store relative debug files if they can't be saved in application folder under Vista/Win7. To do this, add the **-DJrDebugDIRECTIVE_PretendNonWindows** directive.

See also: Turning Off Compilation of DebugLogging Code

4.5.2 Excluding User Code from Compiling

If you have code that is only called by debugging routines or unit testing routines, you can enclose it in #ifdef blocks conditioned on debug compilation.

If JrDebugLog is compiled into your code the following symbols are defined:

```
#define JrDebuggingFlag true
#define JrDebugging
```

If JrDebugLog is not compiled into your code, the following symbols are defined:

```
#define JrDebuggingFlag false
```

So if you have some test unit code for example you might write it as follows:

```
#ifdef JrDebugging
        // code for the unit tests
        void MyUnitTest()
                {
                dbprintf("my unit test!");
                }
        // register the unit tests - this is all you have to do to tell the
system about your test functions
        JrDebugRegisterTestFunction("Simple Unit Test",MyUnitTest,"simple");
#endif
```

In this way, the testing code will not be compiled in if the program is built without JrDebugLogging code.

5 Unit Testing Support

5.1 Introduction

Unit Testing has been around for a long time under different names, but was codified by the recent eXtreme Programing practices. It refers to building support functions for explicitly testing your code to ensure it is working properly. The idea is to write special functions/classes whose only purpose is to test the rest of the code to make sure it does what it is supposed to, won't crash when fed bad data, etc.

The use of such testing functions can help you to be sure that when you make changes, everything still works as expected. There is nothing magical about unit testing - it simply represents a more formal process for ensuring that your code functions the way it should and is well behaved.

Unit Testing support libraries are designed to help the developer write and perform unit testing, and there are quite a few open source and commercial unit testing support libraries for c++. Most of these require you to create special test classes and go through some hoops in order to interface with the testing system.

JrDebugLogger takes a minimalistic approach. It provides a bare minimum of functions that can be used to specify which functions should be considered testing functions, and then can automatically invoke these functions and report their behavior using the standard JrDebugLogger output functions. Essentially you write the test functions like normal functions and just register them with the system - you don't have to derive test classes from special classes, etc.

It was my view that this *minimalistic* approach is appropriate for all but the largest code bases, and that the focus should be on making it as easy as possible to add testing functions to your code. Testing functions can be placed in any file, and simply require a single line registering that function, placed in that same file. No need to add code to your main procedure to inform the engine about new test cases, they are automatically discovered when compiling files.

There is no separate executable for test cases - code is part of your main executable (you can of course make your own separate compilation flag for building test version).

The normal JrDebugLogger commandline parser will automatically detect commandline arguments to perform testing of all or keyword-matching tests. Results are generated as normal, and can therefore include any information you might want to produce during your tests (ie not just errors).

An example of the unit testing support can be found in an example program.
For a list of alternative testing libraries and tools, see the Related Tools topic.

Right now the output features of JrDebugLogger are geared towards the gui viewer, and are not ideal for unit testing reports; better unit test reports are next on the todo list.

5.2 Unit Testing - Registration and Invocation

As shown in the Unit Test Example, JrDebugLogger does not require a special testing class hierarchy.

You write your testing functions as normal global functions (or static class member functions), and then register them with the unit testing system using the following command/macro:

```
JrDebugRegisterTestFunction(string TestNameString,string
DescriptionString,FunctionName,string Keywords);
ex.
  JrDebugRegisterTestFunction("simpletest","Simple Unit Test",MyUnitTest,"simple");
```

Test functions must be declared to take no arguments and return void. You can assign a string of (space separated) keywords for a test; these keywords can be used to run only matching test functions.

The above 'function' is actually a macro that can be placed in global scope, near the test function code. This allows you to register your unit testing functions from the files that define them, without having to add registration code to your main() procedure or anywhere else. So the following would be in global scope in the file (see Example):

```
void MyUnitTest() {dbprintf("my unit test!");}
JrDebugRegisterTestFunction("simpletest","Simple Unit Test",MyUnitTest,"simple");
```

You may want to enclose all your testing functions in #define blocks that will not be compiled if the code is built without JrDebugLog support.

5.3 Unit Testing - Test Evaluation Functions

For testing conditions inside your test functions you can use the standard dbprint,etc. or you can use some special helper functions:

```
dbtest(bool condition)                          - generates test failure if condition
is not true, and continues
dbtestex(bool condition)                        - generates test failure message  if
condition is not true, and throws an exception
dbtestinfo(char* infostring,bool condition)    - generates test failure including
infostring if condition is not true, and continues
dbtestexinfo(char *infostring,bool condition) - generates test failure message
including infostring if condition is not true, and throws an exception
```

5.4 Unit Testing - Invoking Tests and Commandline

The easiest way to control the invocation of unit testing in your codebase is to let the JrDebugLogger commandline argument parser scan your commandline and invoke it auotmatically:

Unit Testing commandline arguments:

```
      -dbt                  runs all unit tests
      -dbt "keywords"       runs all unit tests matching any of the keywords specified
      -dbtl                 lists all registered unit tests along with their keywords
and descriptions
```

For more information about using the commandline parser see the CommandlLine Parsing or CommandLine Parsing Example Topics.

You can also run tests manually from your code. You can also use this approach to design tests that run other subtests:

```
JrDebug::RunTests(string Keywords);
ex.
JrDebug::RunTests("all");
```

Or you can run a function that has never been registered as a test (again these functions must take no arguments and return type void):

```
JrDebugRunTestFunction(string TestName,FunctionName);
ex.
JrDebugRunTestFunction("Manual Test#2",MyUnitTest2);
```

You can call '`bool JrDebug::RanTests()`' to see if tests were invoked from the commandline, which can be useful if, for example, you want to have your program exit after tests are run.

6 Troubleshooting

6.1 Troubleshooting Compilation Errors

Compiler errors complain of no such functions snprintf and/or vsnprintf?
JrDebug tries to se a version of printf (sprintf) that checks for and protects against printinf lines that are too long (bigger than 4096 bytes). But some compilers may not provide this function. Uncomment the following line in **jrdebug.h** to force it to compile with normal sprintf and vsprintf:.

```
//#define JrDebugDIRECTIVE_DontTryToUseSnprintf
```

You might alternatively try to force JrDebugLogger to try different versions of _snprintf if you suspect it is failing to properly guess whether your compiler uses _snprintf or snprintf, by uncommenting one of the following:

```
//#define JrDebugDIRECTIVE_DontUseSnprintfUnderscores
//#define JrDebugDIRECTIVE_UseSnprintfUnderscores
```

Linker errors about missing external static variables from the JrDebug class?
You probably forget to #include "jrdebug_main.cpp" from your main .cpp file (or alternatively to add it to your project or makefile). More Information...

Linker errors about multiple definitions for static variables from the JrDebug class?
You probably #included "jrdebug_main.cpp" from more than one of your .cpp files (or both added the #include and added the .cpp to your project file). More information...

6.2 No Output in Debug Viewer ^&@%#

Things that can cause OutputDebugString messages not to get to the Monitor Viewer GUI tool:

- Are you running it under administration priveledges? If you are not an administrator the tool may not receive messages; right click the shortcut to the tool and configure it to run with administration priveledges.
- Is the MS visual studio ide or another debugger running? Debuggers (including the MS visual studio ide) can capture DebugOutputStrings and prevent the tool from receiving them.
- Is the viewer set to capture messages - there is a drop down combo box on the toolbar, make sure it is set to capture.
- If you're still not having any luck, add the following to your main code to verify that debug logging is actually compiled in and working:

  ```
  JrDebug::WriteToFilename("demologoutput.log",false);
  JrDebug::Announce("Test Application",true);
  ```

  That will cause debug messages to be written to the file demologoutout.log, AND force the debug logger to generate an announcement log message even if debugging is disabled. If you dont see this it means debuglogging was not compiled into your code.
- If you are still get no output and suspect that JrDebugLogger is not being compiled in, you can insure it is by calling the following function which will trigger a compilation error if JrDebugLogging in not compiled into your code:

  ```
  JrDebug::ErrorIfNotCompiledIn();
  ```

6.3 __func__ Error or No FunctionName in Output

- When debugging messages are logged, information about the source code file and line number are included. Information about the calling function is also added, but some compilers may have trouble

determining the current function name, depending on your c++ preprocessor. JrDebugLogger tries to guess what macro name to use but if it fails on your compiler, you can set the proper macro explicitly, in **jrdebug.h**. You may also need to uncomment this to set __func__ to "" if your compiler complains that **__func__** is not defined.

```
// If your compiler does not support the __func__ macro, you need to
//  define a replacement or uncomment this to make it blank.
// #define __func__ ""
```

7 The GUI Debug Monitor Tool

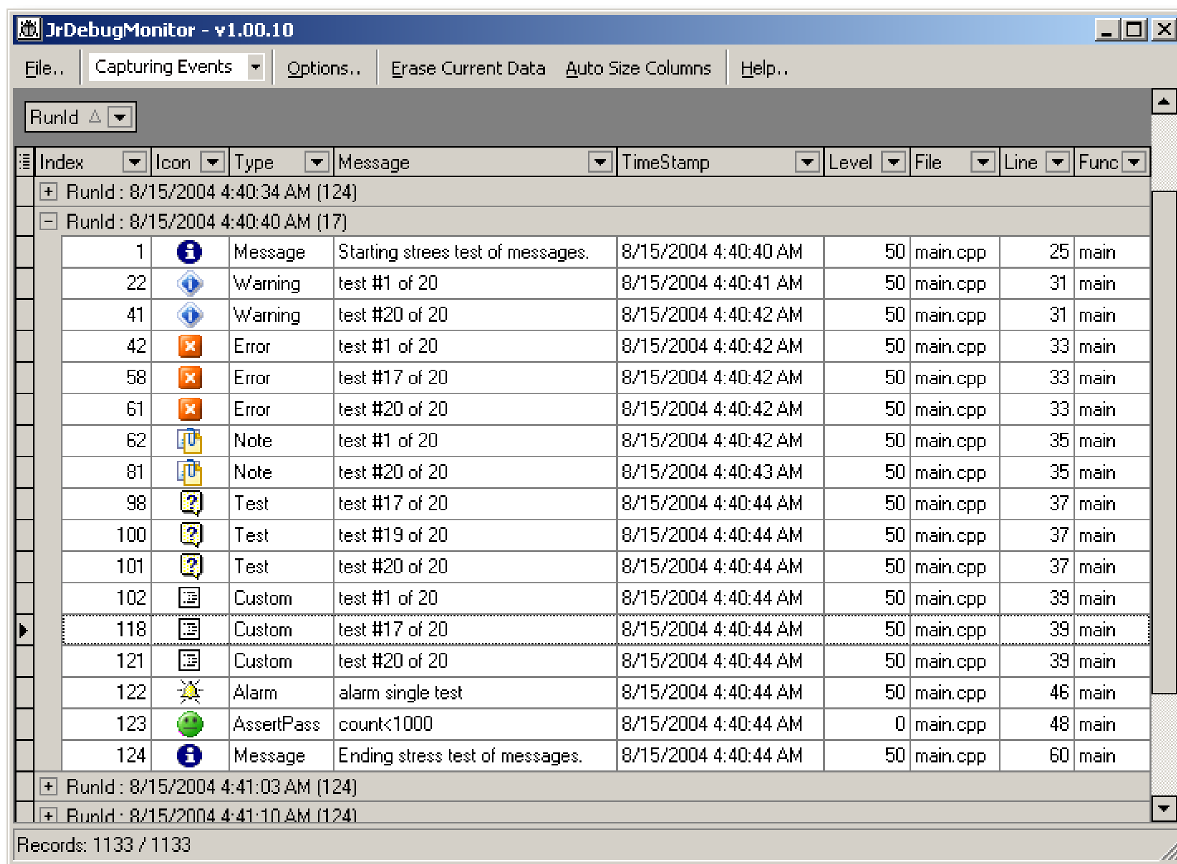
7.1 Introduction to the GUI Tool

There are two main parts that comprise the JrDebugLogger toolkit, a set of cross-platform header files for adding debug logging support to your code, and a powerful windows program for capturing and displaying logging messages.

NOTE: If you don't use Microsoft Windows, you can still JrDebugLog to add debug logging features to your code, but viewing the debugging output is more difficult. One of the things planned for future versions of JrDebugLogger is an option more standard output (such as xml) that would make it easier to view logs with generic gui tools, and support for using unix interprocess message passing. For now, the main benefits of JrDebugLogger are for the MS Windows developer.

7.2 The User Interface

The main interface of the tool:



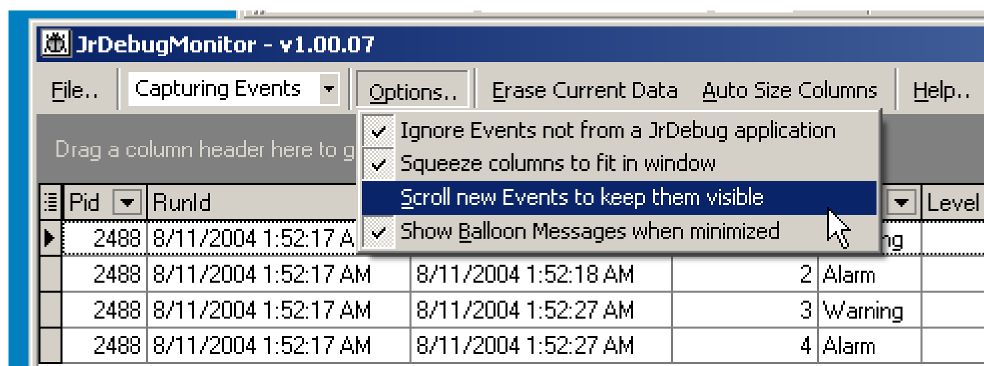
The key to using the Debug Monitor Viewing tool efficiently is to customize your display to help you focus on the information that is relevant to you.

©2004

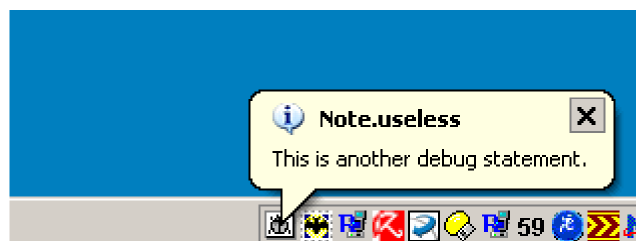
- The first thing you will want to do is disable certain columns that are not useful. You can right click on a column header to select option, including to remove the column (you can always bring it back later). You can also click on the tiny little button to the far left of the columns to quickly enable and disable columns.
- You can drag columns to rearrange the order or change the size.
- You can drag columns to the upper bar to group records - **this can be extremely useful in order to group records by the RunId**. Group headers show the number of records within the group.
- You can sort by columns by clicking on a blank area of the column.
- Click on the black upsidedown triangle to restrict the view to records with only certain values; a filter will appear at the bottom which you can then customize.

7.3 Viewer Options

There are only a view options in the DebugMonitor Viewer tool:



- Ignore Events not from a JrDebug application - if not checked, the tool will display all events sent via OutputDebugString, no matter the source; if checked, only those strings that have a |*Type*| field (all JrDebug application) will be displayed.
- Squeeze columns to fit in window - Resizes column widths so that all columns fit on screen at same time.
- Scroll new Events to keep them visible - tries to scroll the window to keep new events visible if you are at the bottom of the window (doesn't work vey well and can make the window flicker).
- Show Baloon Messages when minimized - when the program is minimized it can display new incoming events in a little baloon in the system tray:

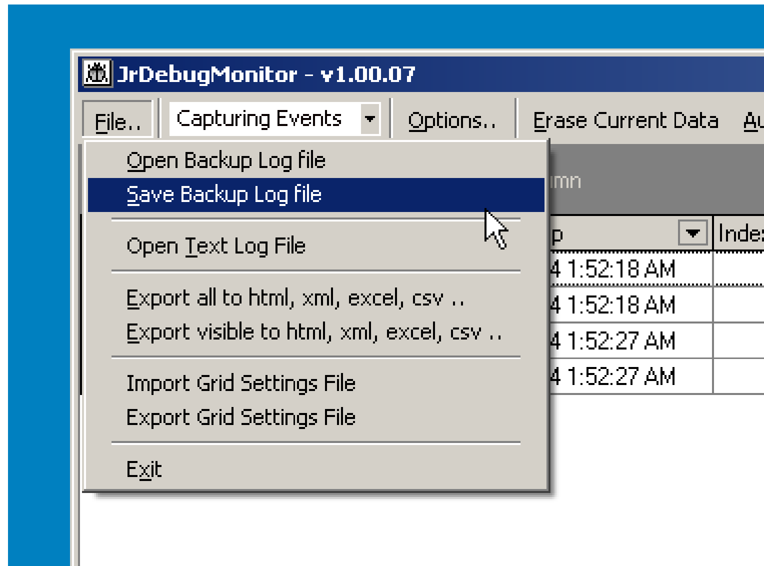


7.4 Working with Files

The DebugMonitor Viewer tool uses a databse engine to store events as they are captured. It always saves the current records into a set of files DebugEventsTable.*. You may at any time save the current records to a single backup database file (*.bpk), and load the records back from this file. If you want to share debug events file with someone else, do it by exchanging backup files. You can delete records by selecting them and deleting them.

You can also import .log or .txt log files generated by the JrDebugLogger file writing functions, and export debug records to a variety of formats.

You may also want to save and load Grid Settings files, which determine the visible columns, their sizes, and any grouping and filtering.



7.5 Custom Columns

The viewer dynamically creates and adjusts the parsed columns it displays as it receives messages; this was done specifically so that it could adapt to new custom columns.

To have your code output specific information into a new column, simply add "|*COLUMNNAME*| VALUE" to the text to you send to debugging output. The VALUE data will be shown in a column named COLUMNNAME.

If the data is numeric, then for proper sorting you should label COLUMNNAME so that it ends with "Num" if it is an integer or "#" if it is a floating point value.

7.6 Using in MS Visual Studio IDE

By default the Microsoft Visual Studio C++ Ide will capture system debug messages if programs are run through the debugger.
(so debugview and the JRDebugViewer GUI program won't see them.

Here's a tip for runing a program from the IDE that will pass through the messages:

If you "debug->run" it (F5), the messages will not be intercepted by the IDE
while if you just run (CTRL+F5) then they will be.

8 Extra Information

8.1 Version History and ToDo List

Version History:

v1.12.01 - May 7, 2013
- Now properly saves time/date stamps by default in output.

v1.11.01 - Nov 13, 2010
- By default if a relative filename is specified for debug output (-dbo), it will be created in the application directory. HOWEVER with windows vista/win7 this has become illegal/problematic. JrDebug will now attempt to use the AppData directory (e.g. Users\Documents and Settings \AppData\JrDebug\Exename) by default on vista/win7 in these cases. You can specify ;info along with the -dbo to have the directory used output;
- Added debug directive to disable all windows-specific code, see Application Switch Files and Compile Flags.

v1.10.01 - Apr 7, 2010
- [bugfix] due to an ambiguous matching of function definitions, dbprintf could result in wrong passing of arguments (only happened with 2 args passed to it, one int and one char*), which could cause assertion fault message.

v1.09.01 - Mar 27, 2010
- [minor bugfix] debug messages sent to system were sometimes being split onto 2 lines (thanks ewemoa)
- [minor feature] added option that can be specified with plain text outputters (including system output) to not show the "MessageType:" prefix on messages.

v1.08.01 - Feb 26, 2009
- [minor bugfix] Fixed some deprecated compiler warnings generated by the latest gcc. (thanks Gothic).
- [minor bugfix] added string.h and stdlib.h to compiled-in headers needed to compile on latest gcc (thanks Gothic).
- [feature] added some documentation about using under *nix build system (thanks Gothic).

v1.07.01 - July 26, 2008
- [minor feature] added code to add an AnsiString streaming helper for C++ Builder users (nothing extra for users to do, you can now stream AnsiString objects; define JrDebugDIRECTIVE_DontAddAnsiStringStreaming to disable its creation).

11.06.01 - July 5, 2008
- [minor feature] put -a at the end of a log output filename to signify it should be opened in append mode (otherwise it will be overwritten).

11.05.01 - June 9, 2007
- bugfix -- jrdebug_notcompiled_macros.h had a missing #define
- added fflush() after each file line output -- ensures file output is always up to date
- re-ordered this changelog to put more recent stuff at top.

1.04.01 - February 7, 2007
- [minor bugfix] quoted commandline arguments were not being properly terminated in some cases [thanks Rob/walkerr]

©2004

1.03.01 - first public release
- improved help

1.02.01 - December 12, 2005
- minor changes to help file
- [minor bugfix] in jrdebug_commandline.h there was a harmless but improper string comparison that led to compiler warning.
- [minor bugfix] uninitialized variable warnings in jrdebug_commandline.h

v1.01.18 - July 7, 2005
- added bugfix that was preventing stdout of errors with c++ builder compilation
- added separate Readme_CommandlineArgs.txt to show how to call apps using commandline args.
- bugfix;added #ifdef WIN32 blocks around some windows-specific console redirection code added in last version

v1.01.17
- important fix to nested if..else statements calling dbprintf or debugstreams

v1.01.15
- added msvc compiler switch to disable annoying pointer truncation warning in the jrdebug_output.h console creation code

v1.01.14
- modified commandline parse command to take &argc instead of (ref)argc, and changed to ParseCommandlineArgc / ParseCommandlineStr macros

v1.01.13
- painfully added 3 dif __func__, __FUNC__, __FUNCTION__ macros to properly pick up function name on all compilers

v1.01.12
- added enable and disable macros, esp. for unit tests which want to fail

v1.01.10
- added ifdebug macro

v1.01.09
- added default file extensions, and ability to use strftime formats in filenames, and use $### for unique numbered files

v1.01.08
- added flag to enable a commandline warning function for use when jrdebug is compiled out

v1.01.07
- added dberror func and debugerr stream for easoer error reporting w/o specifying JrdType

v1.01.06
- added code to 'touch' unused arguments and other tweaks to prevent compiler warnings

v1.01.05
- major changes in output, commandline, block handling...

v1.00.28
- split jrdebug.h into a set of files for better clarity

v1.00.27
- completely revamping debug blocks / activities

v1.00.26
- activities no longer take a separate parameter true/false for whether to announce, just use * to mean DONT announce

v1.00.25
- documentation fixes

1.00.24 - 12/22/04
- adding #defines for user code blocks and help on when to use
- tweaked some compiled-out tests of code size, more to test

1.00.23 - 11/21/04
- improved dbtest exception throwing
- improved help file

- added info option to debug functions

11.00.22 - 11/21/04
- added dbtest and dbtestex macro similar to dbverify and dbassert

1.00.21 - 12/21/04
- fixed test function registration bug (only first and last funcs were registered)
- test functions can now be registered from within code

1.00.20 - 12/21/04
- fixed a bug in unit testing code that was causing reseting of message indices
- better reporting of unit test announcement locations

1.00.19 - 12/19/04
- a significant bug was fixed that caused dbprintf strings to be multiply evaluated for printf-style arg replacement, which could could access violations and bad behavior (thanks, TU).

1.00.18 - 12/18/04
- adding testing functionality

1.00.15 - 09/15/04
- added debugblocks and added it to tricks section.
- added a macro to enclose a function with debugblock notices around it, ie JrDebugEnclose(CallMyFunc())

1.00.14 - 09/10/04
- changed jrdebug_main.h file to jrdebug_main.cpp and changed help to indicate this better reflection of the fact that this file contains global declarations that should only be performed once.
- added more demos in distribution and in help file.

1.00.13 - 09/04/04
- added support to automatically report the time taken to perform hierarchical activities.
- fixed bug in reporting hierarchical activity reporting.

1.00.12 - 08/19/04
- fixed bug in documentaiton and sample of dbthrow exception handling.

1.00.11 - 08/18/04
- fixed option in jrdebug_switch.h for automatically disabling in release mode.
- added better instructions to the demo files.
- added more options to commandline grabber.
- added icon column to debug monitor viewer.
- added optional callback class to JrDebugLogger.
- added more links to Related Tools.
- expanded maximum size of messages stored in viewer tool database (3000 bytes long).
- added help page describing how to create custom columns.

1.00.10 - 08/12/04
- added #include <cstddef> for size_t, some compilers needed it
- fixed use of sprintf vs. _snprintf
- added project files for relo ide

1.00.09 - 08/11/04
- added -dbc commandline for console output and -dbb for brief mode output
- added announce parameter to SetActivity,PushActivity,PopActivity
- trims trailing \n from debug lines
- limited length of activities to prevent recursive explosions.
- added argv list to announcement
- added fix for vc6 lack of __func__
- fixed ostringstream error on vc6
- added namespace jrdebug (but added a using jrdebug; to end of jdebug.h by default)

1.00.08 - 08/11/04
- added support for stream style debug logging
- added ctrl+c copy of selected records to debug log monitor gui tool
- added brief eventstring property to bcb component

* added GenerateTestData to bcb component

1.00.07

* fixed bug that caused crash on compilers which use unsigned size_t
* when specifying a subtype with maintype of JrdCustom will use *only* subtype (rather than append it)
* added better explanations to demo source code
* added replacement of \r and \n to avoid line splitting
* added JrDebugThrow macro and helper functions for debugging exceptions
* added JrdAlarm message type and alarm filter levels which trigger windows messageboxes
* added a function JrDebug::Announce() to send a log entry announcing jrdebug
* added fix for __func__ under gcc, and compatibility macro
* added JRutils_debugout_switch.h file for better project management

1.00.06

* added grouping summaries
* added summary record count on statusbar
* added export functions
* locks focus when adding new items
* added commandline stripper helper
* see demo main.cpp for call to 'GrabCommandlineDebugFile(int &argc,char **argv,bool disableifnofile=false, bool dontchangeargcv=false)'
* modified the way the files are to be linked in to user projects, now two .h instead of .cpp and .h

1.00.05 - first public release

* dramatically improved speed/responsiveness of Grid

8.2 Fun Tricks with your C++ Compiler and Preprocessor

Some interesting tricks are used in the JrDebugLogger files to make the debugging syntax look right, but still make it possible to easily compile-out all of the code for minimal performance impact.

* **Hidden C++ class constructors in order to support __FILE__ information**
  The idea to use C++ class constructors to emulate calling a global function, in order to grab information about the current source code being parsed was suggested by Paul Mclachlan, http://www.codeproject.com/debug/location_trace.asp#xx386001xx. Basically the code expands what looks like a global function call into an object constructor (which takes as arguments the preprocessor macros), followed by an operator() call, which actually does the work. The macros passed as arguments are stored in the constructed class for subsequent use:
  ```
  #define JrDebugInternalPrintf (JrDebug( __FILE__ , __LINE__ , __TIMESTAMP__ ,
  __FUNCTION__ ))
  inline void JrDebug::operator()(const char* format, ... ) {;};
  ```

* **Indirect operations through a conditionally defined if block**
  In order to ensure that the JrDebugLogging code can be removed from compilation without any overhead, while not requireing the user to comment out the debug logging statements, all debug printing statements are redirected through an **if** statement that conditionally compiled to **if(0)** when the debug code is not compiled in. This will cause the c++ compiler to remove that code during any optimization, since it knows it will never be executed. When compiled in, the if statement used is **if (enabled)**, which allows you to compile-in the debug logging code with a trivial impact on overhead

when it is disabled (simple cmp and jz).

- **Trick to avoid problems with streams and temporary objects**
JrDebugLogger provides both printf style debug logging statements and stream io style statements. In order to get the stream style io mechanism to work properly, and interesting kludge was required. The mechanism described above for invoking a class constructor in order to save state and then using the operator() does not quite work in this case. The problem is that this mechanism generates a **temporary** object and then uses that object for the operation, deleting it immediately after use. However, as described by Doug Harrison, http://groups.google.com/groups?selm=qrvkjvk67b6539rpngrjc6cs20v162cup8%404ax.com, this can cause a very subtle problem. Some important stream io operators (for example those that work with std::string), are global non-member operators, which will refuse to be invoked on temporary objects (since the global operators expect non-const objects). So in order to provide full stream io support, we use a kludge to overcome this issue:

```
#define JrDebugInternalStream (JrDebugs( __FILE__ , __LINE__ ,  __TIMESTAMP__ ,
__FUNCTION__ )())
inline JrDebugs& JrDebugs::operator()() {return *this;};
```

The last () in the define, which invokes the operator() is the key because it causes the temporary object to return a non-const stream object, which then can be used with all standard non-member stream operators.

- **Using automatic scoping of local objects to report block timing**
In order to make it easy to report the timing of blocks of code, a trick is used to create an (anonymous) local object on the stack, which announces its creation, and which will be automatically deleted when it goes out of scope, reporting the end of the time interval when it is destructed. This falls under the general heading of RAII (Resource Allocation is Initialization). We employ a few macro tricks in order to anonymously name the local object so user doesnt have to make up a dummy name. We use the line number of the file to name the object:

```
     #define JrDebugStringify2args(x,y) x ## y
     #define JrDebugInternalDebugBlock2If(Nstr, F,L,T,U) JrDebugBlockObject
JrDebugStringify2args(jrdscopedobj_, L);   if (JrDebugEnabled)
JrDebugStringify2args(jrdscopedobj_, L).init(Nstr,F,L,T,U);
     #define JrDebugInternalDebugBlock(Nstr) JrDebugInternalDebugBlock2If(Nstr,
__FILE__ , __LINE__ ,  __TIMESTAMP__ , __FUNCTION__)
```

We also use a very lightweight wrapper object for the object that keeps timing information, so that if debugging code is compiled in but disabled, only the lightweight wrapper is constructed on the stack, not the full heavier object, which is only created above via init() if debugging is enabled.

- **Self-registering objects and calling 'functions' from global scope**
In order to let a user register test functions from within each file that contains them, without requiring that they modify their main procedure, some tricks are necesary. When the user calls (in global score) this macro:

```
JrDebugRegisterTestFunction("Simple Unit Test",MyUnitTest,"simple");
```

A global object is created, whose constructor code actually creates a new test object (this insures that this function macro can be called from any scope and wont die on exit of scope), and registers it with the global JrDebug class. There is a problem however, which is that c++ doesn't guarunte the order of creation of global variables, so we aren't sure if the global JrDebug object has been created yet that we need to register with. To handle this, before registering the test function, we create a helper class whose constructor checks if the global singleton JrDebug has been created and initialized yet; if not creates it before returning.

```
class JrDebugSingletonInsurer {
 bool globalkill;
public:
```

```
 JrDebugSingletonInsurer(bool doinit=true) {if (GlobalJrDebugp==NULL)
{GlobalJrDebugp = new JrDebug(doinit);}; globalkill=doinit;};
 ~JrDebugSingletonInsurer() {if (GlobalJrDebugp!=NULL && globalkill==true)
delete GlobalJrDebugp; GlobalJrDebugp=NULL; globalkill=false;};
};
```

In this way, we insure that the the global singleton JrDebug object is always ready to receive other globally instantiated object, regardless of c++ order of construction. The helper object (we call it an insurer) has at least one global instantiation (invoked with doinit=true) but may be instantiated dynamically anywhere else in the code where we need to insure that the global JrDebug singleton has been created (with doinit=false). The globalkill/doinit flag makes sure that on program exit, when the global insurance object is destroyed, it will take responsibility for destroying the global JrDebug object, even if it wasn't the creator of it.

- **Trouble with macros inside nested if statements**

```
#define DEBUGSTREAMMACRO if (debugenabled) DebugStreamObject
```

If the user does:

```
if (something) DEBUGSTREAMACRO << "hello"; else foobar();
```

The expansion is:

```
if (something) if (debugenabled) DebugStreamObject << "hello"; else foobar();
```

The result is that the last else becomes attached to the ***macro's*** if clause, instead of the original (something) clause, ie:

```
if (something) { if (debugenabled) DebugStreamObject << "hello"; else foobar(); }
```

The solution is to modify the macro:

```
#define DEBUGSTREAMMACRO if (!debugenabled) ; else DebugStreamObject
```

Which expands to:

```
if (something) if (!debugenabled) ; else DebugStreamObject << "hello"; else
foobar();
```

Which is equivelent to:

```
if (something) {if (!debugenabled) ; else DebugStreamObject << "hello";} else
foobar();
```

8.3 Related Tools

There are many tools which operate similarly to JrDebugLogger.

Commercial Logging Tools:

- iTech Logging - http://www.itech-software.com - looks like a powerful tool that is actively being developed; demo is available; expensive ($400 and up).
- MMD Logger - http://www.mmdfactory.com - similar code + gui approach as JrDebugLogger ($10).

Tool for Viewing Windows DebugOutputStrings:

- DebugView - http://www.sysinternals.com/ntw2k/freeware/debugview.shtml - very nice, powerful, free (but no source) tool for capturing and viewing DebugOutputStrings sent by applications on windows. Similar to the gui monitor viewer application in JrDebugLog, but does not support the special parsing of columns for JrDebugLogger messages.

Miscelaneous Source Code to Aid Logging:

- http://libcwd.sourceforge.net - powerful, advanced, complete, open source library; actively developed; may be overkill for casual use.
- http://www.codeproject.com/debug/qafdebug.asp - very comprehensive debugging assert macros, log file output, and some cppunit (unit testing) support functions. similar in spirit to jrdebug. supports some windows-specific error handling and multithreading.

- http://www.codeproject.com/macro/ss_log.asp - nice gui
- http://www.cuj.com/documents/s=8464/cujcexp0308alexandr - enhanced asserts.
- http://log4cplus.sourceforge.net
- http://ryan.ript.net/sir
- http://www.dogma.net/markn/articles/ConStream/constream.htm
- http://www.blong.com/Conferences/BorCon2002/Debugging/3188.htm
- http://www.unixwiz.net/techtips/outputdebugstring.html
- http://www.codeproject.com/debug/statuslog.asp
- http://www.codeguru.com/Cpp/V-S/debug/tracing/article.php/c4429/
- http://www.codeguru.com/Cpp/V-S/debug/logging/article.php/c4425/
- http://www.codeguru.com/Cpp/V-S/debug/logging/article.php/c7231/
- http://www.codeguru.com/Cpp/V-S/debug/article.php/c1263/
- http://www.codeguru.com/Cpp/V-S/debug/article.php/c1251/
- http://www.codeguru.com/Cpp/V-S/debug/article.php/c1277/
- http://www.codeguru.com/Cpp/V-S/debug/article.php/c1265/
- http://www.codeguru.com/Cpp/V-S/debug/article.php/c4405/
- http://www.codeguru.com/Cpp/V-S/debug/article.php/c4419/
- http://www.codeguru.com/Cpp/V-S/debug/article.php/c4431/
- http://www.codeproject.com/debug/location_trace.asp
- http://www.codeproject.com/debug/debugout.asp
- http://www.codeproject.com/debug/debug_macros.asp
- http://www.codeproject.com/debug/logtrace.asp
- http://www.codeproject.com/tips/drawtechniques.asp
- http://www.codeproject.com/debug/mfcconsole.asp
- http://www.codeproject.com/debug/debugtoolkit.asp
- http://www.codeproject.com/csharp/debugconsole.asp
- http://www.codeproject.com/debug/tracer.asp
- http://www.codeproject.com/debug/log.asp
- http://www.codeproject.com/debug/alx_log.asp
- http://www.codeproject.com/debug/debugout.asp
- http://www.codeproject.com/csharp/assert.asp
- http://www.codeproject.com/debug/LogDispatch.asp
- http://www.steveswebshed.com/snip_002.html
- http://www.devx.com/vb2themax/Article/19868/1954?pf=true

C++ Unit Testing:

- http://www.boost.org/libs/test/doc/index.html
- http://cpptest.sourceforge.net/
- http://codesink.org/cutee_unit_testing.html
- http://cppunit.sourceforge.net/cgi-bin/moin.cgi
- http://tut-framework.sourceforge.net/
- http://www.uwyn.com/projects/qtunit/
- http://www.parasoft.com/jsp/products/home.jsp?product=CppTest&itemId=46

Endnotes 2... (after index)

©2004

Back Cover