rprintf API: User Guide and Reference Version 1

CML00002-01

Code Magus Limited (England reg. no. 4024745)
Number 6, 69 Woodstock Road
Oxford, OX2 6EY, United Kingdom
www.codemagus.com

August 16, 2016

Contents

1 Introduction

1.1 Overview of *rprintf*

The *rprintf* library is used for writing program output, usually for diagnostic purposes. It consists of a number of functions to open, write program output, close, set options and retrieve error messages. There are two distinct sets of functions:

1.1.1 Non Directed Output

The first set (non directed output) has no open function and the context of the output destination is kept internally in the library and never exposed to the caller. These routines are effectively a replacement in a program for calls to fprintf with a hard coded destination of ”stderr” where the caller is not concerned with opening or closing the stream (although with rprintf it is always a good idea to close the output stream as this releases resources obtained that are needed to process the output). There may only be one output destination per process at any one time with this set of functions. This output destination may be changed by closing it and re-opening it after changing an environment variable, but the key point is that ALL output written via this set of *rprintf* functions goes to this destination.

All the function names in this set of functions start with `rprintf_`.

1.1.2 Directed Output

The second set (directed output) uses an open function to instantiate an output destination context. This context (handle) must be passed back to any other function that operates on this output destination. These routines can be used as a replacement in a program for calls to fprintf where the destination is not fixed (i.e. possibly supplied by parameter, configuration or the user). The rprintf open takes a *recio* stream open string specification. This approach allows a caller to have multiple output destinations managed through as many contexts.

All the function names in this set of functions start with `frprintf_`.

1.1.3 *rprintf* Output Format

Irrespective of which set of functions is used the output is disposed of using a record based mode whereby output to a destination is only performed when a record boundary is encountered in the data. The record boundaries are the newline (\n) and any record that exceeds the maximum line length (the default of which is defined by RPC_OUTPUT_LINE_DEFAULTLENGTH). Any data that does not make up a complete record is held over and is either potentially disposed of when the next program output is appended to it or completely disposed of when a call to the library to close the stream is performed. The default output destination is stderr unless the environment variable CODEMAGUS_OUTPUT_SPEC is set or for a directed output context the parameter recio_open_string is set, in which case its value is used as a *recio* open string specification and the output is routed record by record to *recio*.

When any *rprintf* options are set such that a prefix is output before each line of program output the output includes an indicator character printed after the prefix and before the program output. This character indicates when a line has been wrapped because its length exceeds the limit imposed for the maximum line length. The indicator for the initial line is ':' and for any wrapped lines '+'. If no prefix options are set then this indicator is not output.

1.2 Handling errors from *rprintf*

Most of the routines return -1 on an error. In this instance one of the functions:

- rprintf_error() for failing functions using non directed output.
- frprintf_error() for failing functions using directed output.

may be called in order to retrieve the error message associated with the failure.

1.3 Implementing *rprintf* in current programs

For non directed output (the first set of functions); as there is no initialise or open function, conversion of current programs that use printf() or fprintf() to write program output to using rprintf() is a trivial task. At least, it is just a change to the function name for programs that only use printf and, at most, removal of the first parameter for programs using fprintf and stderr as the first parameter. As long as all program output ends in a newline a final close call to the rprintf library is not required. It is though, preferable to add a close call for freeing any resources (and flushing any non terminated records) or if the caller wishes to change the output destination and continue writing output.

For directed output the same holds true for converting fprintf code to rprintf. The main difference is that the caller can have any number of directed output destinations active

(through calling `fprintf_open()` multiple times to receive multiple contexts).

Options (verbosity, message prefix and maximum output line length) can all be set through either a call to the library before the first `rprintf()` call or through the associated environment variables. For directed output destinations, because the context must be established first, these can not be set before the `frprintf_open()` call, but may be set with the open call or afterwards.

1.4 Cautionary notes

1.4.1 Using the defined macros

It is always preferable to access the routines in this library through the macros (where they are defined), thus allowing the maintenance of the macros and routines to not impact on the user of the library. To enable this the routines are named with an underscore as the first character and the macros without (for example `_rprintf` and `rprintf` respectively). Routine names that do not have associated macros are named without the underscore.

1.4.2 Compiler compatibility

Note that the macros *frprintf* and *rprintf* are variadic macros and therefore have some restrictions in their use:

1. Compile errors will occur when using compilers that do not support the `ISO C standard of 1999`.

   This is the case (for example with) `MSVC V6` on `Windows`. This platform is catered for in the header file by not defining the macro as a variadic macro; for example `rprintf` is a synonym for the routine `_rprintfz` and the first 3 parameters of `_rprintf` are not required. An example of how to call these routines on such a platform can be found in the test program `testrp.c`. If other platforms exhibit the same behaviour the header file must be changed to reflect that platform. As most compilers are now (at least) c99 compliant this is a problem that is quickly disappearing.

2. The `ISO C standard of 1999` defines that when using variadic macros at least 1 argument must be provided. This causes irritating side affects when converting fprintf() to rprintf() when the call only specifies the format parameter and no variable parameters (for substitution) as follows:

   ```
   fprintf(stderr, "This message only uses a format specifier\n");
   ```

   which fails compilation when converted to:

   ```
   rprintf("This message only uses a format specifier\n");
   ```

rather it should be converted as follows:

```
rprintf("%s\n", "This message only uses a format specifier");
```

If no newline is required then it can be left out of the format specification parameter.

1.5 Process flow overview

1.5.1 Non Directed Output

Examples of the flow of routine calls are:

1. **Example 1**

```
rprintf(...);
rprintf(...);
rprintf(...);
...
```

2. **Example 2**

```
rprintf_setopts(...);
rprintf_setwidth(...);
rprintf(...);
rprintf(...);
...
rprintf_close(...);
```

3. **Example 3**

```
rprintf(...);
...
rprintf_setopts(...);
rprintf_setwidth(...);
rprintf(...);
...
rprintf_close(...);
```

1.5.2 Directed Output

Examples of the flow of routine calls are:

1. **Example 1**

```
rpc = frprintf_open(...);
frprintf(rpc, ...);
frprintf(rpc, ...);
frprintf(rpc, ...);
...
frprintf_close(rpc, ...);
```

2. **Example 2**

```
rpc = frprintf_open(...); /* opts can also be set here */
frprintf_setopts(rpc, ...);
frprintf_setwidth(rpc, ...);
frprintf(rpc, ...);
frprintf(rpc, ...);
frprintf(rpc, ...);
...
frprintf_close(rpc, ...);
```

3. **Example 3**

```
rpc = frprintf_open(rpc, ...);
frprintf(rpc, ...);
frprintf(rpc, ...);
...
...receive an error on frprintf()
msgptr = frprintf_error(rpc, ...);
frprintf_close(rpc, ...);
```

2 *rprintf* Non Directed Output

2.1 Macro `rprintf()`

2.1.1 Format

```
int rprintf(const char *format, ...);
```

2.1.2 Description

Function `rprintf()` will accept a format string and a variable number of arguments in the style of `printf()` and print them to a specified destination based on a record mode approach. This record mode approach means that the destination is record based rather than streams based; In other words every newline encountered in the output separates the output into a new record. Further to this there is a defined maximum record length, whose default is defined by `RPC_OUTPUT_LINE_DEFAULTLENGTH`, which if exceeded will also cause a record boundary break. The default destination is stderr, but may be overridden by the environment variable `CODEMAGUS_OUTPUT_SPEC` whose value will be used as a *recio* open string specification to perform a *recio* open and all subsequent output will be passed to *recio*. A simple example of such a *recio* open string is:

```
export CODEMAGUS_OUTPUT_SPEC="text(myoutput.txt,mode=a)"
```

this specification names the output file `myoutput.txt` (in the current working directory) as the destination and `mode=a` means that it will be appended to. This routine will also handle any resources or state associated with the destination context, for example opening (and keeping open) a *recio* stream destination before the first write. It is however the caller's responsibility to call `rprintf_close()` to close and free all associated resources.

2.1.3 Parameters

format (required). A `printf()` style format string.

... (one required). Any number (greater than zero) of substitution parameters as per `printf()`. See note 2 in sub section 1.4.2 on page 3 for the reason why one or more substitution parameters are required.

2.1.4 Return Value

Success zero is returned;

Failure The value -1 is returned and a message can be retrieved by calling `rprintf_error()`.

2.2 Macro `rprintf_close`

2.2.1 Format

```
int rprintf_close(void);
```

2.2.2 Description

Function `rprintf_close()` should always be called when a program is finished writing output. All resources associated with the print destination are freed and any remaining data that has not yet been written will be flushed. This routine can also be used to effectively change the destination of the output in that once closed the environment variable can be reset or unset and further output will be directed to the new destination.

2.2.3 Parameters

none

2.2.4 Return Value

Success zero is returned;

Failure The value -1 is returned and a message can be retrieved by calling `rprintf_error()`. However, if this routine fails it still attempts to free all resources and close any streams related to the output destination. In other words the output destination is, as far as the caller is concerned, closed and unusable.

2.3 Function `rprintf_error()`

2.3.1 Format

```
char *rprintf_error(void);
```

2.3.2 Description

Function `rprintf_error()` will return the last error associated with the output destination currently active. This routine can be safely used even after a failed first write in which case the underlying library will return the reason for the (open or write) failure.

2.3.3 Parameters

none

2.3.4 Return Value

Success A pointer to a NULL terminated error message is returned.

Failure None; This routine has no fail return code.

2.4 Function `rprintf_setopts()`

2.4.1 Format

```
int rprintf_setopts(unsigned int opts);
```

2.4.2 Description

Function `rprintf_setopts()` allows the caller to set various options pertaining to writing output. These options may be supplied in one call by using the binary operator 'or' or individually on as many calls as are necessary.

2.4.3 Parameters

opts (required). unsigned integer holding the bit value of 1 or more of the following flags.

`RP_NULLFLAGS` This is a NULL value used when no flag values need to be specified.

`RP_VERBOSE` Causes verbose output (direct to stderr) of rprintf and underlying libraries (e.g. recio). This may also be set by setting the environment variable

`CODEMAGUS_OUTPUT_VERBOSE=1`

`RP_PREFIX` Turns on outputting the prefix before a message. This may also be set by setting the environment variable

`CODEMAGUS_OUTPUT_PREFIX=1`

`RP_PREFIX_TS` Prefixes the message with a timestamp. Setting this on and not setting `RP_PREFIX` results in a prefix of only the timestamp. This may also be set by setting the environment variable

`CODEMAGUS_OUTPUT_PREFIX_TS=1`

2.4.4 Return Value

Success zero is returned;

Failure The value -1 is returned and a message can be retrieved by calling `rprintf_error()`. This call may fail for many reasons the most common being that if it is the first call to any *rprintf* function, resources need to be acquired and that action may fail (especially if the output destination is via recio).

2.5 Function `rprintf_unsetopts()`

2.5.1 Format

```
int rprintf_unsetopts(unsigned int opts);
```

2.5.2 Description

Function `rprintf_unsetopts()` allows the caller to unset various options pertaining to writing output. These options may be supplied in one call by using the binary operator 'or' or individually on as many calls as are necessary.

2.5.3 Parameters

opts (required). unsigned integer holding the bit value of 1 or more of the flags to unset. These flags are the same as in section 2.4.3 on page 8.

2.5.4 Return Value

Success zero is returned;

Failure The value -1 is returned and a message can be retrieved by calling `rprintf_error()`. This call may fail for many reasons the most common being that if it is the first call to any *rprintf* function, resources need to be acquired and that action may fail (especially if the output destination is via recio).

2.6 Function `rprintf_width()`

2.6.1 Format

```
int rprintf_setwidth(unsigned int width);
```

2.6.2 Description

Function `rprintf_setwidth()` allows the caller to set the width of an output destination. This does not include the prefix of the message. If the width is given as zero then the width is set to the default value. This is currently defined by `RPC_OUTPUT_LINE_DEFAULTLENGTH`.

2.6.3 Parameters

width (required). Integer holding the output maximum width in characters.

2.6.4 Return Value

Success zero is returned;

Failure The value -1 is returned and a message can be retrieved by calling `rprintf_error()`. This call may fail for many reasons; the most common being that if it is the first call to any *rprintf* function, resources need to be acquired and that action may fail.

3 *frprintf* Directed Output

3.1 Function `frprintf_open()`

3.1.1 Format

```
rp_context_t *frprintf_open(char *recio_open_string, int flags, int width);
```

3.1.2 Description

Function `frprintf_open()` will return a context to a directed output destination.

3.1.3 Parameters

recio_open_string (optional). A string that identifies a recio access module, object and options and is used to direct the output via *recio*. If not specified then the value of the environment variable `CODEMAGUS_OUTPUT_SPEC` is used as the recio open string specification. If neither are specified then the output is directed to standard error. A simple example of a *recio* open string is:

```
export CODEMAGUS_OUTPUT_SPEC="text(myoutput.txt,mode=a)"
```

this specification names the output file myoutput.txt (in the current working directory) as the destination and ”mode=a” means that it will be appended to.

flags (optional). Any combination of flags or the NULLFLAGS flag. See subsubsection 2.4.2 on page 8 for information on these flags.

width (optional). An integer that specifies the width of the output or the default specification. See subsubsection 2.4.3 on page 8 for information on the default and how to specify the default value.

3.1.4 Return Value

Success A context handle is returned as a pointer.

Failure The value NULL is returned and a message can be retrieved by calling `rprintf_error()`.

3.2 Macro `frprintf()`

3.2.1 Format

```
int frprintf(rp_context_t *rpc, const char *format, ...);
```

3.2.2 Description

Function `frprintf()` will accept a format string and a variable number of arguments in the style of `printf()` and print them to a specified destination based on a record mode approach. This record mode approach means that the destination is record based rather than streams based; In other words every newline encountered in the output separates the output into a new record. Further to this there is a defined maximum record length, whose default is defined by `RPC_OUTPUT_LINE_DEFAULTLENGTH`, which if exceeded will also cause a record boundary break.

The default destination is stderr, but if when calling `fprintf_open()` a *recio* open string is passed in or (as a default) the environment variable `CODEMAGUS_OUTPUT_SPEC` is set then that value will be used as a *recio* open string specification to perform a *recio* open and all subsequent output will be passed to *recio*. A simple example of such a *recio* open string is:

```
export CODEMAGUS_OUTPUT_SPEC="text(myoutput.txt,mode=a)"
```

this specification names the output file myoutput.txt (in the current working directory) as the destination and ”mode=a” means that it will be appended to.

3.2.3 Parameters

rprintf_context (required). The context acquired from calling `frprintf_open()`.

format (required). A `printf()` style format string.

... (one required). Any number (greater than zero) of substitution parameters as per `printf()`. See note 2 in sub section 1.4.2 on page 3 for the reason why one or more substitution parameters are required.

3.2.4 Return Value

Success zero is returned;

Failure The value -1 is returned and a message can be retrieved by calling `rprintf_error()`.

3.3 Macro `frprintf_close()`

3.3.1 Format

```
int frprintf_close(rp_context_t *rpc);
```

3.3.2 Description

Function `frprintf_close()` should always be called when a program is finished writing output. All resources associated with the print destination are freed and any remaining data that has not yet been written will be flushed.

3.3.3 Parameters

rpc (required). A valid context returned from `frprintf_open()`.

3.3.4 Return Value

Success zero is returned;

Failure The value -1 is returned and a message can be retrieved by calling `rprintf_error()`. However, if this routine fails it still attempts to free all resources and close any streams related to the output destination. In other words the output destination is, as far as the caller is concerned, closed and unusable.

3.4 Function `frprintf_error()`

3.4.1 Format

```
char *frprintf_error(rp_context_t *rpc);
```

3.4.2 Description

Function `frprintf_error()` will return the last error associated with the specified output destination context. This routine can be safely used even after a failed open (and a NULL context is returned). By calling this routine with a NULL context the last open error message will be returned.

3.4.3 Parameters

rpc (required). A context returned from `frprintf_open()` or NULL.

3.4.4 Return Value

Success A pointer to a NULL terminated error message is returned.

Failure None; This routine has no fail return code.

3.5 Function `frprintf_setopts()`

3.5.1 Format

```
int frprintf_setopts(rp_context_t *rpc, unsigned int opts);
```

3.5.2 Description

Function `frprintf_setopts()` allows the caller to set various options pertaining to writing output for the specified context. These options may be supplied in one call by using the binary operator 'or' or individually on as many calls as are necessary.

3.5.3 Parameters

rpc (required). A valid context returned from `frprintf_open()`.

flags (required). See subsubsection 2.4.3 on page 8 for a description of the flags that may be set.

3.5.4 Return Value

Success zero is returned;

Failure The value -1 is returned and a message can be retrieved by calling `rprintf_error()`.

3.6 Function `frprintf_unsetopts()`

3.6.1 Format

```
int frprintf_unsetopts(rp_context_t *rpc, unsigned int opts);
```

3.6.2 Description

Function `frprintf_unsetopts()` allows the caller to unset various options pertaining to writing output for the specified context. These options may be supplied in one call by using the binary operator 'or' or individually on as many calls as are necessary.

3.6.3 Parameters

rpc (required). A valid context returned from `frprintf_open()`.

flags (required). See subsubsection 2.4.3 on page 8 for a description of the flags that may be set.

3.6.4 Return Value

Success zero is returned;

Failure The value -1 is returned and a message can be retrieved by calling `rprintf_error()`.

3.7 Function `frprintf_width()`

3.7.1 Format

```
int frprintf_setwidth(rp_context_t *rpc, unsigned int width);
```

3.7.2 Description

Function `frprintf_setwidth()` allows the caller to set the width of a specific output destination. The width does not include the prefix of the message. If the width is given as zero then the width is set to the default value. This is currently defined by `RPC_OUTPUT_LINE_DEFAULTLENGTH`.

3.7.3 Parameters

rpc (required). A valid context returned from `frprintf_open()`.

width (required). Integer holding the output maximum width in characters.

3.7.4 Return Value

Success zero is returned;

Failure The value -1 is returned and a message can be retrieved by calling `rprintf_error()`.

A C source listings

A.1 *rprintf.h*

```
#ifndef RPRINTF_H
#define RPRINTF_H
  /* File: rprintf.h
   *
   * This file contains the definitions required for writing program output,
   * usually for diagnostic purposes.
   * The library consists of a number of functions to write, close, set options
   * and retrieve error messages. There are two distinct sets of functions.
   *
   * The first set (non directed output) has no open function and the context of
   * the output destination is kept internally in the library and never exposed
   * to the caller. These routines are effectively a replacement in a program
   * for calls to fprintf with a hard coded destination of "stderr" where the
```

```
* caller is not concerned with opening or closing the stream (although with
* rprintf it is always a good idea to close the output stream).
* The default output destination is stderr unless the environment variable
* CODEMAGUS_OUTPUT_SPEC is set, in which case its value is used as a recio
* open string specification and the output is routed record by record to
* recio.
* All the function names in this set of functions start with rprintf_ .
*
* The second set (directed output) uses an open function to instantiate an
* output destination context. This context (handle) must be passed back to
* any other function that operates on this output destination. These routines
* can be used as a replacement in a program for calls to fprintf where the
* destination is not fixed (ie posssibly supplied by parameter, config or the
* user). The rprintf_open takes a recio stream open string specification and
* if it is NULL the default output destination is set to stderr.
* All the function names in this set of functions start with frprintf_ .
*
* Irrespective of which set of functions is used the output is disposed of
* using a record based mode whereby output to a destination is only performed
* when a record boundary is encountered in the data. The current record
* boundaries are the newline (\n) and any record that exceeds the maximum
* line length (the default of which is defined by
* RPC_OUTPUT_LINE_DEFAULTLENGTH). Any data that does not make up a
* complete record is held over and is either appended to with the next
* program output or a call to the library to close the stream.
* The program name and line number of where the message was printed can also
* be optionally prefixed to the output. A timestamp and PID may also be
* optionally prefixed to the message.
*
* Most of the routines return -1 on an error. If this happens then the caller
* can call [f]rprintf_error() to retrieve the error message associated
* with the failure.
*
* For non directed output (the first set of functions); as there is no
* initialise or open function, conversion of current programs that use
* [f]printf() to write program output to using rprintf() is a trivial task.
* At least, it is just a change to the function name for programs that only
* use printf and removal of the first parameter for programs using fprintf to
* stderr. As long as all program output ends in a newline a final close call
* to the rprintf library is not required. It is preferable to add a close
* call for freeing any resources (and flushing any non terminated records) or
* if the caller wishes to change the output destination and continue writing
* output.
* For directed output the same holds true for converting fprintf code to
* rprintf. The main difference is that the caller can have any number of
* directed output destinations active (through calling fprintf_open multiple
* times and to receive multiple contexts).
*
* Options (verbosity, message prefix and maximum output line length) can all
* be set through either a call to the library before the first rprintf() call
* or associated environment variables. For directed output destinations these
* may also be set on the open call.
```

```
*
* It is always preferable to access the routines in this library through the
* macros (where they are defined), thus allowing the maintainance of the
* macros and routines to not impact on the user of the library. To enable
* this the routines are named with an underscore as the first character and
* the macros without (for example _rprintf and rprintf respectively). Routine
* names that do not have associated macros are named without the underscore.
*
* Also note that the macros [f]rprintf are variadic macros and therefore have
* some restrictions in their use:
* 1. Compile errors will occur when using compilers that do not support the
*    ISO C standard of 1999. This is the case for MSVC V6 on Windows. For
*    these platforms an entry must be made below (see where
*    HAS_VARIADIC_MACRO_SUPPORT is defined and/or undefined) to specify that
*    variadic macros are not supported. This has already been done for MSVC6.
*    Once this is set then rprintf can be used normally by programs compiled
*    on these (antiquated) compilers. The values printed as the file name and
*    line number of the code generating the message are incorrect as they are
*    generated by the rprintf library, so will always be the same.
*
* 2. The ISO C standard of 1999 defines that when using variadic macros at
*    least 1 argument must be provided. This causes irritating side affects
*    when converting fprintf() to rprintf() with the syntax that only
*    specifies the format parameter and no variable parameters (for
*    substitution) as follows:
*       fprintf(stderr, "This message only uses a format specifier\n");
*    fails when converted to:
*       rprintf("This message only uses a format specifier\n");
*    rather it should be converted as follows:
*       rprintf("%s\n", "This message only uses a format specifier");
*
*    If no newline is required then it can be left out of the format
*    specification parameter.
*
* Author: Patrick Hayward.
*
* Copyright (c) 2008 Code Magus Limited. All rights reserved.
*
* $Author: release $
* $Date: 2015/05/18 08:56:11 $
* $Id: rprintf.h,v 1.24 2015/05/18 08:56:11 release Exp $
* $Name:  $
* $Revision: 1.24 $
* $State: Exp $
*
* $Log: rprintf.h,v $
* Revision 1.24  2015/05/18 08:56:11  release
* Changes for port to CYGWIN
*
* Revision 1.23  2013/05/28 14:12:36  hayward
* Add functions [f]rprintf_unsetopts() to
* allow callers to unset any of the options
```

```
* that can be set via [f]rprintf_setopts().
*
* Revision 1.22  2010/10/22 14:16:39  hayward
* Allow GCC to check all rprintf functions
* that use "char *format, ..." for consistency.
* This may help stop some abends in error
* messages where the wrong type has been used.
*
* Revision 1.21  2010/04/06 18:08:54  hayward
* z/OS 1.10 C89 does not allow variadic
* macros so disable them for this platform.
*
* Revision 1.20  2010/04/06 11:57:11  hayward
* Add system header includes for SunOS.
*
* Revision 1.19  2009/09/29 09:57:48  hayward
* Improve the ability to trace what is going
* on with a recio write when it fails.
*
* Revision 1.18  2008/10/02 08:18:23  hayward
* Add include for stdarg.h for programs that do not
* specifically include it themselves.
*
* Revision 1.17  2008/09/29 15:13:08  hayward
* Expose function _vfrprintf(). This allows frdumpbuff to call it
* directly.
*
* Revision 1.16  2008/09/26 10:27:17  hayward
* Add in missing _vrprintf prototype
*
* Revision 1.15  2008/09/25 22:32:47  hayward
* Add vrprintf(). This function is the same as rprintf() except instead
* of neing a variadic function it takes a va_list parameter.
*
* Revision 1.14  2008/08/27 10:02:32  hayward
* Increase the default line length to 4096 to stop
* irritating wrapping occurring.
*
* Revision 1.13  2008/07/08 09:25:44  hayward
* Print recio open error if VERBOSE is set. If this is not
* done then no error message is printed and the caller of
* rprintf seldom checks return codes.
* Also add rprintf_info() call to get build information,
* this is useful in GDB.
* Add to testrp unit tests to test the above error message.
*
* Revision 1.12  2008/06/30 10:16:11  hayward
* Improve documentation - especially WRT variadic Macro support.
*
* Revision 1.11  2008/06/03 18:59:31  hayward
* Changes to make it easier to use rprintf across platforms
* that do and do not support variadic macros.
```

```
   * Remove [f]rprintf_strerror(); callers must only use
   * [f]rprintf_error(). This also removes multiple defined entry
   * points in libraries with multiple object files as these routines
   * where wholly defined in the header file.
   *
   * Revision 1.10  2008/05/26 13:16:12  justin
   * Temporary fixes to allow compile on VS6
   *
   * Revision 1.9  2008/05/21 20:05:32  hayward
   * Add memdebug directives to Makefile.
   * Change [f]rprintf_strerror to [f]rprintf_error.
   * This is in line with other CML code. For now both functions
   * are supported but [f]rprintf_strerror will be removed soon.
   *
   * Revision 1.8  2008/05/20 19:01:52  hayward
   * Temporary change for flush of output after each write.
   *
   * Revision 1.7  2008/05/16 13:25:26  hayward
   * Implement directed output mode. This requires an open to
   * obtain a output destination context and then that context
   * is required as a parameter to the following calls for output.
   * The set of functions starting with frprintf achieve this.
   *
   * Revision 1.6  2008/04/28 11:28:56  hayward
   * Add comment about variadic macros and number of arguments.
   *
   * Revision 1.5  2008/04/24 14:54:38  hayward
   * Improve functionality by adding setopts, setwidth and environment variable
   * overrides. Also improved header file documentation.
   *
   * Revision 1.4  2008/04/23 11:37:00  hayward
   * Change to use spilPath to get just program name.
   * Changes to accommodate both compilers that handle or do not
   * handle variadic macros.
   *
   * Revision 1.3  2008/04/23 10:32:05  hayward
   * Port to windows and specifically to a platform that
   * does not support variadic macros.
   *
   * Revision 1.2  2008/04/22 22:13:00  hayward
   * Fix typo in rprintf macro.
   *
   * Revision 1.1.1.1  2008/04/22 22:11:15  hayward
   * Added rprintf new sources to CVS.
   */

static char *cvs_rprintf_h =
   "$Id: rprintf.h,v 1.24 2015/05/18 08:56:11 release Exp $";

  /*
   * Header files required by this header.
   * For Variadic macros.
```

```
   * For getpid() calls.
   */
#include <stdarg.h>
#ifndef WIN32
#include <sys/types.h>
#include <unistd.h>
#endif

  /*
   * Types and structures:
   */
typedef struct rp_context rp_context_t;

  /*
   * Macros and functions.
   */
#define RPC_OUTPUT_LINE_DEFAULTLENGTH  4096
  /* Determine which platforms can or can not handle variadic macros
   */
#define HAS_VARIADIC_MACRO_SUPPORT
#if defined WIN32 && _MSC_VER <= 1200
#undef HAS_VARIADIC_MACRO_SUPPORT
#endif
#if defined  __HOS_MVS__
#undef HAS_VARIADIC_MACRO_SUPPORT
#endif


  /* Function rprintf_info() will return a pointer to a string that holds build
   * information about the rprintf library.
   */
char *rprintf_info(void);

  /*
   * ************************************************************************
   * Non Directed output routine definitions. These routines (and macros)
   * operate on a single context which is held internally in the library and not
   * exposed to the caller. (For directed output functions see the relevant
   * heading below these functions.).
   * ************************************************************************
   */

  /* Function rprintf() will accept a format string and a variable number of
   * arguments in the style of printf() and print them to a specified
   * destination based on a record mode approach. This record mode approach
   * means that the destination is record based rather than streams based; In
   * other words every newline encountered in the output separates the output
   * into a new record. Further to this there is a defined maximum record
   * length, whose default is defined by RPC_OUTPUT_LINE_DEFAULTLENGTH, which if
   * exceeded will also cause a record boundary break.
   *
   * The default destination is stderr, but if the environment variable
```

```
   * CODEMAGUS_OUTPUT_SPEC is set then its value will be used as a recio open
   * string specification to perform a recio open and all subsequent output will
   * be passed to recio. A simple example of such a recio open string is:
   *   export CODEMAGUS_OUTPUT_SPEC="text(myoutput.txt,mode=a)"
   * this specification names the output file myoutput.txt (in the current
   * working directory) as the destination and "mode=a" means that it will be
   * appended to.
   *
   * This routine will also handle any resources or state associated with the
   * destination, for example opening (and keeping open) a recio stream
   * destination before the first write. It is however the caller's
   * responsibility to call rprintf_close to close and free all associated
   * resources.
   *
   * Return values:
   * If writing the output to the destination is successful then 0 is returned,
   * but when not successful -1 is returned and a message can be retrieved by
   * calling rprintf_error().
   *
   */
#ifdef HAS_VARIADIC_MACRO_SUPPORT
   #define rprintf(format, ...)\
      _rprintf(__FILE__, __LINE__, getpid(), format, __VA_ARGS__)
#else
   #define rprintf _rprintfz
#endif

  /* Function vrprintf() will accept a format string and a va_arg parameter in
   * the style of vprintf. In other aspects it delivers exactly the same
   * functionality as rprintf().
   */
#define vrprintf(format, ap)\
   _vrprintf(__FILE__, __LINE__, getpid(), format, ap)

  /* Function rprintf_close() should always be called when a program is finished
   * writing output. All resources associated with the print destination are
   * freed and any remaining data that has not yet been disposed of will be
   * flushed to the output destination. This routine can also be used to
   * effectively change the destination of the output in that once closed the
   * environment variable can be reset or unset and further output will be
   * directed to the new destination.
   *
   * Return values:
   * On successful completion (which may include a write to flush residual data)
   * 0 is returned. If the close fails for any reason -1 is returned and the
   * last error can be accessed by calling rprintf_error(). However, if this
   * routine fails it still attempts to free all resources and close any streams
   * related to the output destination. In other words the output destination
   * is, as far as the caller is concerned, closed and unuseable.
   */
#define rprintf_close() _rprintf_close(__FILE__, __LINE__, getpid())
```

```
  /* Function rprintf_error() will return the last error associated with the
   * output destination currently active. This routine can be safely used even
   * after a failed first write in which the underlying library will return the
   * reason for the (open or write) failure.
   *
   * Return values:
   * This routine always returns a pointer to a string and will not fail.
   */
char *rprintf_error(void);

  /* Function rprintf_setopts() allows the caller to set various options
   * pertaining to writing output. The following options can be set
   * RP_VERBOSE            - Causes verbose output (to stderr) of rprintf and
   *                         underlying libraries (e.g. recio)
   *                         This may also be set by setting the environment
   *                         variable CODEMAGUS_OUTPUT_VERBOSE=1
   * RP_PREFIX             - Turns on outputting the prefix before a message.
   *                         This may also be set by setting the environment
   *                         variable CODEMAGUS_OUTPUT_PREFIX=1
   * RP_PREFIX_TS          - Prefixes the message with a timestamp. Setting this
   *                         on and not setting RP_PREFIX results in a prefix
   *                         of only the timestamp.
   *                         This may also be set by setting the environment
   *                         variable CODEMAGUS_OUTPUT_PREFIX_TS=1
   * RP_FLUSH_ALL          - (Temporary fix) Flushes the output stream after
   *                          each write. This will be removed once a solution
   *                          to multiple processes writing to the same file is
   *                          solved in recio. In the mean time, though this may
   *                          slow the process down it minimises the possibilty
   *                          of losing outptut messages when more than one
   *                          process is writing to the same file.
   *
   * These options may be or'ed together and supplied in one call or they may be
   * supplied individually on as many calls as are necessary.
   *
   * Return values:
   * This routine will return 0 on successful completion. If it completes
   * unsuccessfully then -1 is returned and the error message can be retrieved
   * by rprintf_error(). This call may fail for many reasons the most common
   * being that if it is the first call to any rprint* function, resources need
   * to be aquired and that action may fail (especially if the output
   * destination is via recio).
   */

int rprintf_setopts(unsigned int opts);
#define RP_NULLFLAGS          0           /* No flags to be change/set  */
#define RP_VERBOSE            0x00000001  /* Be verbose (for diagnosis) */
#define RP_PREFIX             0x00000002  /* Output a prefix */
#define RP_PREFIX_TS          0x00000004  /* Include a timestamp in the pfx */
#define RP_FLUSH_ALL          0x00000008  /* Flush output after each write. */
#define RP_TRACE              0x00000010  /* Be more verbose (for diagnosis) */
```

```
  /* Function rprintf_unsetopts() allows the caller to un set various options
   * pertaining to writing output. The options that can be unset are the same as
   * those that can be set by rprintf_setopts().
   *
   * These options may be or'ed together and supplied in one call or they may be
   * supplied individually on as many calls as are necessary.
   *
   * Return values:
   * This routine will return 0 on successful completion. If it completes
   * unsuccessfully then -1 is returned and the error message can be retrieved
   * by rprintf_error(). This call may fail for many reasons the most common
   * being that if it is the first call to any rprint* function, resources need
   * to be aquired and that action may fail (especially if the output
   * destination is via recio).
   */
int rprintf_unsetopts(unsigned int opts);

  /* Function: rprintf_setwidth() allows the caller to set the width of an
   * output destination. This does not include the prefix of the message.
   *
   * If the width is given as zero then the width is set to the default value.
   * This is currently RPC_OUTPUT_LINE_DEFAULTLENGTH.
   *
   * Return values:
   * This routine will return 0 on successful completion. If it completes
   * unsuccessfully then -1 is returned and the error message can be retrieved
   * by rprintf_error(). This call may fail for many reasons; the most common
   * being that if it is the first call to any rprint* function, resources need
   * to be aquired and that action may fail.
   */
int rprintf_setwidth(unsigned int width);

  /*
   * Other Exported functions:
   */
  /* Functions: _rprintf() and _rprintf_close() are all functions that although
   * exposed should not be used. The macros with the same name (without the
   * leading underscore) should be used instead. See the documentation relevant
   * to each of the macros for information about these functions.
   */
int _rprintf(const char *id, const int ref, const int instance,
      const char *format, ...)
#if defined(__linux__) || defined(__CYGWIN__)
__attribute__ ((format (printf, 4, 5)))
#endif
;
int _rprintf_close(const char *id, const int ref, const int instance);
int _vrprintf(const char *id, const int ref, const int instance,
      const char *format, va_list ap)
#if defined(__linux__) || defined(__CYGWIN__)
__attribute__ ((format (printf, 4, 0)))
#endif
```

```
;

#ifndef HAS_VARIADIC_MACRO_SUPPORT
int _rprintfz(const char *format, ...)
#if defined(__linux__) || defined(__CYGWIN__)
__attribute__ ((format (printf, 1, 2)))
#endif
;
#endif

  /*
   * ************************************************************************
   * Directed output routine definitions. These routines (and macros)
   * operate on a given context which is held by the caller and obtained from a
   * call to frprintf_open(). (for non directed output see the routines above).
   * ************************************************************************
   */

  /* Function frprintf_open() will return a context to a directed output
   * destination.
   * The three parameters are all optional and if NULL or zero is supplied they
   * receive a default value.
   * recio_open_string identifies a recio access module, object and options and
   * are used to call recio_open to direct the output via recio. If not
   * specified then the environment variable as detailed CODEMAGUS_OUTPUT_SPEC
   * is used to supply the recio open string specification. If that is also not
   * specified the the output is directed to stderr.
   * flags are the flags as defined in rprintf_setopts().
   * width is the width as defined in rprintf_setwidth().
   *
   * Return values:
   * This routine will return a context handle to the output destination if
   * successful. If not successful NULL is returned and the associated error
   * message can be retrieved with a call to frprintf_error().
   */
rp_context_t *frprintf_open(char *recio_open_string, unsigned int flags,
      unsigned int width);

  /* Function frprintf() will accept a format string and a variable number of
   * arguments in the style of printf() and print them to a specified
   * destination based on a record mode approach. This record mode approach
   * means that the destination is record based rather than streams based; In
   * other words every newline encountered in the output separates the output
   * into a new record. Further to this there is a defined maximum record
   * length, whose default is defined by RPC_OUTPUT_LINE_DEFAULTLENGTH, which if
   * exceeded will also cause a record boundary break.
   *
   * The default destination is stderr, but if when calling fprintf_open() a
   * recio open string is passed in or (as a default) the environment variable
   * CODEMAGUS_OUTPUT_SPEC is set then that value will be used as a recio open
   * string specification to perform a recio open and all subsequent output will
   * be passed to recio. A simple example of such a recio open string is:
```

```
   *   export CODEMAGUS_OUTPUT_SPEC="text(myoutput.txt,mode=a)"
   * this specification names the output file myoutput.txt (in the current
   * working directory) as the destination and "mode=a" means that it will be
   * appended to.
   *
   * Return values:
   * If writing the output to the destination is successful then 0 is returned,
   * but when not successful -1 is returned and a message can be retrieved by
   * calling frprintf_error().
   *
   * For callers with compilers that do not support variadic macros frprintf is
   * simply defined as _frprintf and they should supply the first 3 parameters.
   * For an example see testrp.c.
   */
#ifdef HAS_VARIADIC_MACRO_SUPPORT
   #define frprintf(rpc, format, ...)\
      _frprintf(__FILE__, __LINE__, getpid(), rpc, format, __VA_ARGS__)
#else
   #define frprintf _frprintfz
#endif

  /* Function vfrprintf() will accept a rp context, a format string and a va_arg
   * parameter in the style of vfprintf. In other aspects it delivers exactly
   * the same functionality as rprintf().
   */
#define vfrprintf(rpc, format, ap)\
   _vfrprintf(__FILE__, __LINE__, getpid(), rpc, format, ap)

  /* Function frprintf_close() should always be called when a program is
   * finished writing output. All resources associated with the print
   * destination are freed and any remaining data that has not yet been written
   * will be flushed.
   *
   * Return values:
   * On successful completion (which may include a write to flush residual data)
   * 0 is returned. If the close fails for any reason -1 is returned and the
   * last error can be accessed by calling frprintf_error(). However, if this
   * routine fails it still attempts to free all resources and close any streams
   * related to the output destination. In other words the output destination
   * is, as far as the caller is concerned, closed and unuseable.
   */
#define frprintf_close(rpc) \
      _frprintf_close(__FILE__, __LINE__, getpid(), rpc)

  /* Function frprintf_error() will return the last error associated with the
   * output destination currently active. This routine can be safely used even
   * after a failed open in which the underlying library will return the
   * reason for the open failure.
   *
   * Return values:
   * This routine always returns a pointer to a string and will not fail.
   */
```

```
char *frprintf_error(rp_context_t *rpc);

  /* Function frprintf_setopts() allows the caller to set various options
   * pertaining to writing output. See rprintf_setopts() for a description of
   * the options and the #defines of the values that can be set.
   *
   * These options may be or'ed together and supplied in one call or they may be
   * supplied individually on as many calls as are necessary.
   *
   * Return values:
   * This routine will return 0 on successful completion. If it completes
   * unsuccessfully then -1 is returned and the error message can be retrieved
   * by frprintf_error().
   */
int frprintf_setopts(rp_context_t *rpc, unsigned int opts);

  /* Function frprintf_unsetopts() allows the caller to set various options
   * pertaining to writing output. See rprintf_setopts() for a description of
   * the options and the #defines of the values that can be unset.
   *
   * These options may be or'ed together and supplied in one call or they may be
   * supplied individually on as many calls as are necessary.
   *
   * Return values:
   * This routine will return 0 on successful completion. If it completes
   * unsuccessfully then -1 is returned and the error message can be retrieved
   * by frprintf_error().
   */
int frprintf_unsetopts(rp_context_t *rpc, unsigned int opts);

  /* Function: frprintf_setwidth() allows the caller to set the width of an
   * output destination. This does not include the prefix of the message.
   *
   * If the width is given as zero then the width is set to the default value.
   * See rprintf_setwidth() for the definition of this value.
   *
   * Return values:
   * This routine will return 0 on successful completion. If it completes
   * unsuccessfully then -1 is returned and the error message can be retrieved
   * by frprintf_error().
   */
int frprintf_setwidth(rp_context_t *rpc, unsigned int width);

  /*
   * Other Exported functions:
   */

  /* Functions: _frprintf() and _frprintf_close() are all functions that
   * although exposed should not be used. The macros with the same name (without
   * the leading underscore) should be used instead. See the documentation
   * relevant to each of the macros for information about these functions.
   */
```

```
int _frprintf(const char *id, const int ref, const int instance,
      rp_context_t *rpc, const char *format, ...)
#if defined(__linux__) || defined(__CYGWIN__)
__attribute__ ((format (printf, 5, 6)))
#endif
;
int _frprintf_close(const char *id, const int ref, const int instance,
      rp_context_t *rpc);
int _vfrprintf(const char *id, const int ref, const int instance,
      rp_context_t *rpc, const char *format, va_list ap)
#if defined(__linux__) || defined(__CYGWIN__)
__attribute__ ((format (printf, 5, 0)))
#endif
;

#ifndef HAS_VARIADIC_MACRO_SUPPORT
int _frprintfz(rp_context_t *rpc, const char *format, ...)
#if defined(__linux__) || defined(__CYGWIN__)
__attribute__ ((format (printf, 2, 3)))
#endif
;
#endif

#undef INCLUDED_FROM_RPRINTF_C
#endif /* RPRINTF_H */
```

B Program examples

C Detailed test program scenarios

For the convenience of developers maintaining rprintf there is a test program that must be maintained. It should be enhanced to include new tests if and when new functionality is added or bugs are fixed.

C.1 Using `testrp.c`

The program `testrp.c` provides the standard regression tests for the rprintf `API`. Note that when a group of tests includes a prefix (of any sort) then a continuation indicator is always printed before the second and subsequent lines of a continued message; see section 1.1.3 on page 1 for more details. The line length is defined as 101 bytes so that line wrapping tests can easily be performed. This number was chosen as a suitably large (in terms of readability width) prime number.

C.2 Test cases

The tests are divided into groups; they are:

Group 1 **Non-Directed, Normal prefix including Timestamp.** This group performs the standard tests, but sets the *rprintf* options to prepend a prefix and a timestamp to the message.

Group 2 **Non-Directed, Timestamp prefix only** This group performs the standard non-directed tests, but sets the *rprintf* options for a timestamp prefix only.

Group 3 **Non-Directed, Normal prefix only** This group performs the standard non-directed tests, but sets the *rprintf* options for a normal prefix only.

Group 4 **Non-Directed, No prefix** This group performs the standard non-directed tests, but sets the *rprintf* options for a no prefix.

Group 5 **Non-Directed, Normal prefix including Timestamp** This group performs the standard tests, but sets the *rprintf* options to prepend a prefix and a timestamp to the message. The options are set via an environment variable rather than through a call to `rprintf_setopts()`.

Group 6 **Directed, Normal prefix including Timestamp** This group runs further tests using a `recio` open string to direct the output via `recio`. These tests should only test explicit functionality to do with directed output.