



RAM Disk Controller (RDC) Description

Copyright Tevero AS

This document may be freely distributed as long as it is not changed without the consent of Tevero AS. The mechanisms described within the document shall not be used in any form of device or application without the consent of Tevero AS.

Table of Contents

1 Introduction.....	3
1.1 Scope.....	3
1.2 Overview.....	3
1.3 Features.....	4
1.4 Definitons.....	4
2 Partition Layout.....	6
2.1 Partition Information.....	6
2.2 Block Status Table.....	6
3 Application Programming Interfaces (APIs).....	7
3.1 The RAM Driver Application Programming Interface (RD-API).....	7
3.1.1 nnRead.....	7
3.1.2 nnWrite.....	8
3.2 The RAM Protection Application Programming Interface (RP-API).....	9
3.2.1 Volatile Mode.....	9
3.2.2 Non-volatile Mode.....	9
3.2.3 sig_semRead.....	10
3.2.4 sig_semWrite.....	10
3.2.5 wai_semRead.....	10
3.2.6 wai_semWrite.....	10
3.3 The RDC User Application Programming Interface (RU-API).....	11
3.3.1 rdcClose.....	11
3.3.2 rdcDelete.....	11
3.3.3 rdcDiagMode.....	12
3.3.4 rdcFormat.....	13
3.3.5 rdcGetBlockSize.....	16
3.3.6 rdcGetInfoString.....	16
3.3.7 rdcGetMode.....	16
3.3.8 rdcGetNumberOfBlocks.....	17
3.3.9 rdcInit.....	17
3.3.10 rdcIsUsed.....	17
3.3.11 rdcOpen.....	18
3.3.12 rdcRead.....	20
3.3.13 rdcWrite.....	21
3.3.14 rdcWriteTransparent.....	22
3.3.15 Diagnostic Functions.....	22
Appendix A: RDC Integration Guide.....	23

1 Introduction

1.1 Scope

The scope of this document is to describe the basic principles and usage of the RAM disk controller (RDC). The document does not contain detailed design information, please refer to the source files.

1.2 Overview

The RAM disk controller (RDC) is a disk controller used to make a contiguous memory area accessible as a disk. It is specially made for use with TFFS, in systems where both a flash disk and RAM disk is present. RDC adds very little to the system footprint. (Using RAM media with the “Transaction *Flash* FileSystem” is not as strange as it may sound...)

RDC is intended as an optimal version of FDC, avoiding wasting partition space for housekeeping issues required when using flash, and RAM used for logical to physical mapping. RDC uses almost no partition overhead for housekeeping, and only a few bytes of data are used to hold info on every partition opened.

RDC can be configured run-time to operate in two modes:

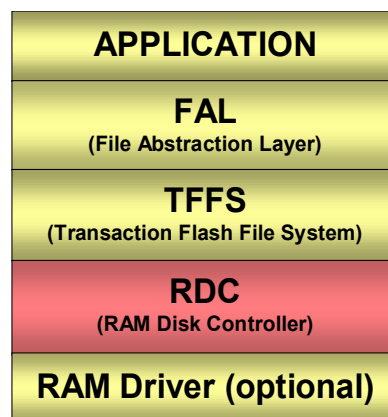
- a **non-volatile** mode for RAM disks in for instance battery backed up RAM. The non-volatile mode ensures that a block is never partially written, e.g. if the system restarts when a block is being written, then either all or nothing of the updated block is written. This is done in order to comply with TFFS requirements, where it is assumed a block write is finished or undone.

This may also be useful in some systems, typically for holding configuration data. For instance, if configuration data is stored in many small files each fitting into one block, then one can ensure that configuration data in a file is consistent when changing. (TFFS always caches one block, and multiple changes on the file will thus only be done on the cached block. Only when the file is closed or explicitly flushed, will the block be written to the RAM disk.)

The only penalty for using this mode is somewhat reduced write speed (appr. ½) compared to volatile mode.

- a **volatile** mode used for temporary RAM disks. A temporary disk will typically be empty at restart. This mode does not have to ensure that block writes are not aborted and thus is somewhat faster when writing.

RDC is a part of a modular implementation, and can be considered used as the figure below shows:



Notice that is hardly useful to use RDC standalone, since it is specially adapted to TFFS usage.

The RAM driver is optional. RDC can be configured to use an internal "RAM driver", i.e. direct memory access on write and/or read accesses.

Block alignment in a RDC partition can be chosen (minimum 4) in order to avoid unaligned accesses that are normally slower, or possibly not allowed on some architectures. Also, selecting alignment on cache line size, may increase speed if using cached RDC partitions.

External drivers **MUST** be provided for instance for the following reasons:

- if memory has some HW write protection that must be disabled before writing
- if using cached RAM to hold non-volatile RAM partition. In this case, data written to the partition must be explicitly flushed in the RAM driver. If not, the partition may not be recoverable if the system should restart during a RAM disk write access.
- using DMA to transfer data for increased speed

1.3 Features

The RAM Disk Controller (RDC) is a system for making it possible to use a contiguous RAM area as a disk with TFFS. RDC provides the following functionality:

- Usable in a preemptive context (e.g. reentrant). This allows for multiple simultaneous read accesses, but only one write access. (Disk can be read while writing, although inconsistency can arise on a higher level if doing simultaneous read/write to the same block.)
- Handling an unlimited number of RAM disk partitions.
- Supports removable RAM disks.
- The minimum write unit (block) is user defined per partition (minimum 32 bytes).
- Each partition may theoretically support up to $2^{32} - 1$ blocks
- Supports optionally (non-volatile mode) the recovery of any aborted write operation (power failure etc.)
- Optional support for 'errno' updating
- Some diagnostics functions supported
- Written in ANSI C.
- Host debugging and development support.

The following is required when using RDC and is not included in RDC:

- Optionally provide read and/or write functions if RAM can not be accessed directly without some special target dependent mechanisms.
- If used in a preemptive context, protection functions for the selected environment must be implemented if no higher level protection exists. (Since RDC is typically used with TFFS, the protection handling should rather be provided with TFFS.)
- Provide the following C library glue functions: *free*, *malloc*, *memcmp*, *memcpy*, *memset*, *strcmp*, *strcpy*, *strlen*, and *printf* (*printf* only if debugging or diagnostics enabled). The functions are defined in *shrdconf.h/rdcconf.h* and they **must** all be reentrant if RDC is used in a preemptive environment.

1.4 Definitions

Block By a block is meant a contiguous memory area holding data typically used by a file system. A file system will normally operate on blocks, binding larger data amounts

RAM Disk Controller (RDC) Description

into a file, which may be scattered around in the RAM disk partition. A block can be rewritten as often as required, however in order to be usable with TFFS, each block has a marker indicating if it is in use or not.

2 Partition Layout

When a RAM partition is formatted with RDC, the partition will look as follows:

Partition Information
Block Status Table (non-volatile mode only)
Alignment
Recovery block (non-volatile mode only)
Block 1
:
Block M

2.1 Partition Information

The partition information holds all information about the RAM partition. It is used by *rdcOpen()* to recognize how the partition is formatted and how to use the partition.

The partition information layout is described in the header of the *rdc.c* source file.

2.2 Block Status Table

The block status table (only used in non-volatile mode) consists of one bit for every block in the partition. The bit is used to mark if a block is used by TFFS or not.

3 Application Programming Interfaces (APIs)

3.1 The RAM Driver Application Programming Interface (RD-API)

Notice that the RD-API is only required if special functionality is required in order to access the RDC partition. The internal “drivers” may be used for plain memory access.

The RAM driver API required by RDC is very simple. It is however extremely important that it is implemented as specified, especially with respect to return values.

The RD-API functions are defined per RDC partition in the *rdcFormat()* and *rdcOpen()* functions.

If ‘errno’ support is enabled, this is handled by RDC and higher layers, so the driver should normally not change the ‘errno’ variable.

3.1.1 nnRead

int nnRead(void *userPointer, void *to, unsigned int length, U32 offset)

Description: Read bytes from the RDC partition. Notice that the driver must be able to handle the requests of any number of bytes. Single byte reads are quite frequently used by RDC.

Reentrancy: This function must be reentrant since RDC allows multiple reads to occur.

Input:

<i>userPointer</i>	A user pointer, which is provided transparently in case the <i>nnRead()</i> function needs to identify the circumstances for doing the read. The <i>userPointer</i> is provided with the <i>rdcFormat()</i> and the <i>rdcOpen()</i> functions.
--------------------	---

<i>to</i>	The pointer to the location to place the bytes read. The buffer must be large enough to hold the specified number of bytes to read.
-----------	---

<i>length</i>	The number of bytes to read.
---------------	------------------------------

<i>offset</i>	The offset to the first byte in the RDC partition to read.
---------------	--

Output:

<i>int</i>	The exact number of bytes read must be returned. <0 indicates that the read could not be done. Notice that RDC treats all return values different from <i>length</i> as a fault.
------------	---

3.1.2 nnWrite

int nnWrite(void *userPointer, const void *buffer, unsigned int length, U32 offset)

Description: Write bytes to the RDC partition. Notice that the driver must be able to handle the requests of any number of bytes. Single byte writes are quite frequently used by RDC.

Reentrancy: This function does not have to be reentrant since the user should provide RDC with protection functions if required.

Input:

<i>userPointer</i>	A user pointer, which is, provided transparently in case the <i>nnWrite()</i> function needs to identify the circumstances for doing the write. The <i>userPointer</i> is provided with the <i>rdcFormat()</i> and the <i>rdcOpen()</i> functions.
--------------------	--

<i>buffer</i>	The pointer to the location which holds the bytes to write to the RDC partition.
---------------	--

<i>length</i>	The number of bytes to write.
---------------	-------------------------------

<i>offset</i>	The offset to the first byte in the RDC partition to write.
---------------	---

Output:

<i>int</i>	The <u>exact</u> number of bytes written must be returned. <0 indicates that the write could not be completed successfully. Notice that RDC treats all return values different from <i>length</i> as a fault.
------------	---

3.2 The RAM Protection Application Programming Interface (RP-API)

This chapter discusses protection mechanisms used to be able to use RDC in a reentrant system. Notice that RDC is generally used with TFFS, and the protection mechanisms required for TFFS should be followed, please refer to TFFS documentation.

3.2.1 Volatile Mode

In volatile mode, no protection is strictly necessary from a RDC perspective. Of course writing and reading to the same block simultaneously on a higher level may cause application inconsistencies. (I.e. one user is writing to a file while another user is reading from it.)

3.2.2 Non-volatile Mode

In non-volatile mode, RDC only allows one user to write at a time. Multiple users can read simultaneously, also when another user is writing.

An implementation suggestion is indicated below with pseudo-code; be careful to use the defined sequence of operations:

```
int wai_semRead()
{
    /* Always OK to read, just return */
    return(0);
}

void sig_semRead()
{
    /* Nothing to do here */
}

int wai_semWrite()
{
    /* Get write access to partition, inhibiting new write accesses. */
    Obtain semaphore

    /* If timeout is supported */
    if (timeout)
        return(-1);

    return(0);
}

void sig_semWrite()
{
    /* Let others request write access */
    Release semaphore
}
```

3.2.3 sig_semRead

void sig_semRead(void *userPointer)

Description: End read access to the RDC partition.

n:

Reentrancy: Reentrant

y:

Input: *userPointer* Pointer provided by user and passed from RDC. Can be used to identify driver type to be used, partition etc.

3.2.4 sig_semWrite

void sig_semWrite(void *userPointer)

Description: End write access to the RDC partition.

Reentrancy: Reentrant

Input: *userPointer* Pointer provided by user and passed from RDC. Can be used to identify driver type to be used, partition etc.

3.2.5 wai_semRead

int wai_semRead(void *userPointer)

Description: Wait for read access to the RDC partition. Multiple users may normally access the RDC partition.

Reentrancy: Reentrant

Input: *userPointer* Pointer provided by user and passed from RDC. Can be used to identify driver type to be used, partition etc.

Output: *int* 0 - access granted
<0 – unable to grant access (RDC will then cancel the operation)

3.2.6 wai_semWrite

int wai_semWrite(void *userPointer)

Description: Wait for write access to the RDC partition. Only one may write to the RDC partition at a time in non-volatile mode.

Reentrancy: Reentrant

Input: *userPointer* Pointer provided by user and passed from RDC. Can be used to identify driver type to be used, partition etc.

Output: *int* 0 - access granted
<0 – unable to grant access (RDC will then cancel the operation)

3.3 The RDC User Application Programming Interface (RU-API)

The user functions are the functions available for user of RDC. A typical user would be a file system, although one may in theory include usage of RDC directly in the application.

3.3.1 rdcClose

int rdcClose(int id)

Description: Close a RAM partition. All associated memory is freed.

Reentrancy: Reentrant (if protection functions provided with *rdcOpen()*)

Input: *id* Partition ID for partition to close. This id was obtained with the *rdcOpen()* function.

Output: *int* 0 - partition closed successfully
<0 - trying to close non-opened partition

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO or EBUSY

3.3.2 rdcDelete

int rdcDelete(int id, U32 block)

Description: Delete a block, e.g. mark it as not in use. TFFS requires that the disk controller knows if a block is in use or not, in order to handle recovery issues. This is only required in non-volatile mode.

Reentrancy: Reentrant (if protection functions provided with *rdcOpen()*)

Input: *id* Partition ID for partition to delete block

block Block number to delete (mark unused)

Output: *int* 0 - block deleted
<0 - some fault occurred

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL, EIO or EBUSY

3.3.3 rdcDiagMode

int rdcDiagMode(int id, U8 mode)

Description: Control diagnostic mode for a partition.

This function is only included if diagnostic is enabled in *rdcconf.h* (RDC_DIAG set to 1)

Reentrancy: Reentrant

Input:

<i>id</i>	Partition ID for partition to set diagnostic mode
<i>mode</i>	Diagnostic mode to set. Use one of: RDC_DIAG_DISABLE - Disable diagnostics logging RDC_DIAG_ERRORS - Show errors RDC_DIAG_ALL - Show all diagnostics actions and errors

Output: *int* Returns previous diagnostic mode
<0 - Invalid partition ID and no change done.

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL or EBUSY

3.3.4 rdcFormat

```
int rdcFormat(
    char *info,
    void *startAddress,
    U8 blockAlign,
    U32 partitionSize,
    U8 blockSize,
    RDC_MODE mode,
    void *userPointer,
    int (*ramRead)(void *userPointer, void *to, unsigned int length, U32 offset),
    int (*ramWrite)(void *userPointer, const void *buffer, unsigned int length, U32 offset),
    int (*wai_semWrite)(void *userPointer),
    void (*sig_semWrite)(void *userPointer),
    U8 diagMode)
```

Description: Format a RAM partition in order to be able to use with the RDC system. Some data in the defined partition will be lost. When completed, the partition may be opened.

The format is very fast, especially in volatile mode where it is constant and independent of partition size.

NOTICE:

This function allocates memory needed for various issues. The memory is kept only during formatting.

Reentrancy: Reentrant on different non-overlapping partitions.

Input:

<i>info</i>	Pointer to 0 terminated info string (user definable). Must not be longer than RDC_MAX_INFO characters. The string is included in the partition info and can be used for identification.
<i>startAddress</i>	Start address used to address first byte of partition. This pointer is only used if using the internal RAM device driver.
<i>blockAlign</i>	Block alignment, N in 2 ^N . The address that a block shall be aligned to start at. For instance, if using data cache on the partition being formatted, it may be an idea to align the block to the cache line size. 'blockAlign' can not be more than 'blockSize'. If less than 2, it is forced to 2 in order to ensure minimum alignment on address divisible by 4.
<i>partitionSize</i>	The size in bytes of the RAM area available for the partition.
<i>blockSize</i>	The block size is the operating unit that a higher layer system can read and write to. It is N in 2 ^N and can not be less than 5 (e.g. minimum blocksize is 32).

RAM Disk Controller (RDC) Description

<i>mode</i>	Mode to use when formatting: <u>eRdcVolatile</u> : Volatile mode, a block write will not be recovered if the system is restarted during a block write. <u>eRdcNonVolatile</u> : Non volatile mode, a block write will either be undone or completed if the system is restarted during a block write. Non-volatile can be used in for instance battery backed up RAM. Volatile mode should only be used on partitions that are formatted.
<i>userPointer</i>	Pointer that can be set by the user and possibly used by user provided functions that it is passed on to. Not used by RDC.
<i>ramRead</i>	Function pointer to device driver used to read from the RAM. If NULL, then RDC will use its own internal access method (in which case 'startAddress' must be provided).
<i>ramWrite</i>	Function pointer to device driver used to write to the RAM. If NULL, then RDC will use its own internal access method (in which case 'startAddress' must be provided).
<i>wai_semWrite</i>	Function pointer to function waiting for write access semaphore for the RAM device. Set to NULL if the RAM access shall be treated without preemption concerns.
<i>sig_semWrite</i>	Function pointer to function releasing the write access semaphore for the RAM device. Set to NULL if the RAM access shall be treated without preemption concerns.
<i>diagMode</i>	Diagnostic mode to use while formatting. Use RDC_DIAG_XXX defines. Only used if RDC_DIAG is enabled in <i>rdccconf.h</i>

Output: *int* 0 - format completed successfully
 <0 - error during formatting

If 'errno' usage is enabled, 'errno' is updated on error to:
ENOMEM, EINVAL, EIO or EBUSY

3.3.5 rdcGetBlockSize

int rdcGetBlockSize(int id, U8 *size)

Description: Get block size, N in 2^N

Reentrancy: Reentrant

Input: *id* Partition ID for partition to get block size

number Location to store fetched block size

Output: *int* 0 - Block size fetched
<0 - Fault, number is not fetched.

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO

3.3.6 rdcGetInfoString

const char *rdcGetInfoString(int id)

Description: Get pointer to RAM disk partition info string. The partition info string is defined at format time, and can be used to identify the partition.

Reentrancy: Reentrant

Input: *id* Partition ID for partition to get info string pointer.

Output: *const char ** Ptr to partition info string. Notice that the string can be 0 length. NULL is returned if invalid ID is provided. The string is statically located and must not be modified.

3.3.7 rdcGetMode

int rdcGetMode(int id, RDC_MODE *mode)

Description: Get partition mode, volatile or non-volatile.

Reentrancy: Reentrant

Input: *id* Partition ID for partition to get mode for.

mode Pointer to location to place mode

Output: *int* 0 - Mode fetched
<0 - Fault, mode is not fetched

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO

3.3.8 rdcGetNumberOfBlocks

int rdcGetNumberOfBlocks(int id, U32 *number)

Description: Get number of blocks in partition.

Reentrancy: Reentrant

Input: *id* Partition ID for partition to get number of blocks

number Pointer to location to place number of blocks

Output: *int* 0 - Number of blocks in partition fetched
<0 - Fault, number is not fetched

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO

3.3.9 rdclnit

void rdclnit(U32 max)

Description: Initialize RDC system. Should be invoked once before using RDC.

Reentrancy: Reentrant

Input: *max* Max number of RDC partitions that can be defined. A pointer is allocated for every partition that can be defined. The pointer table is later used for fast lookup.

3.3.10 rdclsUsed

int rdclsUsed(int id, U32 block)

Description: Check if a block is used (e.g. written to) or not. This functionality is required by TFFS in order to handle recovery issues.

NOTICE: This function can only be used for non-volatile partitions.

Reentrancy: Reentrant

Input: *id* Partition ID for partition to check if used block

block Block to check if used

Output: *int* 1 - Block is used (e.g. contains data)
0 - Block is not used (and does not contain valid data).
<0 - Illegal usage (wrong parameters)

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL, EIO or EBUSY

3.3.11 rdcOpen

```
int rdcOpen(
    void *startAddress,
    void *userPointer,
    int (*ramRead)(void *userPointer, void *to, unsigned int length, U32 offset),
    int (*ramWrite)(void *userPointer, const void *buffer, unsigned int length, U32 offset),
    int (*wai_semRead)(void *userPointer),
    void (*sig_semRead)(void *userPointer),
    int (*wai_semWrite)(void *userPointer),
    void (*sig_semWrite)(void *userPointer),
    U8 diagMode)
```

Description: Open a RAM disk partition. The partition is scanned and the following are done as part of the scanning:

- recover any abnormal previous termination:
- build partition info status

NOTICE:

This function allocates memory needed to hold partition info. The memory is kept until closing the partition.

Reentrancy: Reentrant on different partitions.

Input:

<i>startAddress</i>	Start address used to address first byte of partition. This pointer is only used if using the internal RAM device driver.
<i>userPointer</i>	Pointer that can be set by the user and possibly used by user provided functions that it is passed on to. Not used by RDC.
<i>ramRead</i>	Function pointer to device driver used to read from the RAM. If NULL, then RDC will use its own internal access method (in which case 'startAddress' must be provided).
<i>ramWrite</i>	Function pointer to device driver used to write to the RAM. If NULL, then RDC will use its own internal access method (in which case 'startAddress' must be provided).
<i>wai_semRead</i>	Function pointer to function waiting for read access semaphore for the RAM device. Set to NULL if the RAM access shall be treated without preemption concerns.
<i>sig_semRead</i>	Function pointer to function releasing the read access semaphore for the RAM device. Set to NULL if the RAM access shall be treated without preemption concerns.
<i>wai_semWrite</i>	Function pointer to function waiting for write access semaphore for the RAM device. Set to NULL if the RAM access shall be treated without preemption concerns.
<i>sig_semWrite</i>	Function pointer to function releasing the write access semaphore for the RAM device. Set to NULL if the RAM access shall be treated without preemption concerns.
<i>diagMode</i>	Diagnostic mode to use for partition. Use RDC_DIAG_XXX defines. Only used if RDC_DIAG is enabled.

RAM Disk Controller (RDC) Description

Output: *int* >=0 - The ID of the partition opened
 <0 - unable to open partition

If 'errno' usage is enabled, 'errno' is updated on error to: EEXIST, EINVAL, ENOMEM, EIO or EBUSY

3.3.12 rdcRead

int rdcRead(int id, U32 block, U8 *to, U32 length, U32 offset)

Description: Read data from a block in a RAM disk partition. This provides the user access point to RDC.

Reentrancy: Reentrant (protection functions must be provided as needed)

Input: *id* Partition ID for partition to read block from
 block Block to read data from
 to Pointer to location to place read data into
 length Number of bytes to read from block. Should not be larger than the size of a block.
 offset Offset from start of block to read bytes from.

Output: *int* 0 if requested data read. In non-volatile mode, if the block is not used, this is considered a fault by RDC.
 <0 if some fault occurred.

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL, EIO or EBUSY

3.3.13 rdcWrite

int rdcWrite(int id, U32 block, const U8 *from, U32 length)		
Description:	Write data to a block on a RAM disk. This provides the user access point to RDC.	
Reentrancy:	Reentrant (protection functions must be provided as needed).	
Input:	<i>id</i>	Partition ID for partition to do write
	<i>block</i>	Block number to write data to.
	<i>from</i>	Pointer to location to place write data from
	<i>length</i>	Number of bytes to write to block. Must not be larger than the available number of bytes in a block.
	<i>category</i>	Not used, for API compliance only.
Output:	<i>int</i>	0 if write successful. <0 if some fault occurred
	If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL, EIO or EBUSY	

3.3.14 rdcWriteTransparent

int rdcWriteTransparent(int id, U32 block, const U8 *from, U32 length, U32 offset)

Description: Write data to a block on a RAM disk that may already have been written to. This write operates in transparent mode, not modifying the status bits for the block (except for the first time writing to an unused block). Thus, the write is not recoverable except the first time writing to a block.

Only non-volatile partition has any perception of a used block. First time writing to a non-used block, offset in block must be 0, however, this check can not be done in volatile mode.

WARNING: Use this function with care!

Reentrancy: Reentrant (protection functions must be provided as needed) Only one may be writing at a time, and no read operation is allowed during this.

Input:

<i>id</i>	Partition ID for partition to do write
<i>block</i>	Block number to write data to.
<i>from</i>	Pointer to location to place write data from
<i>length</i>	Number of bytes to write to block. Must not be larger than the available number of bytes in a block.
<i>offset</i>	Offset in block to start writing to. This value must be 0 the first time writing to a block (e.g. writing to an unused block).
<i>category</i>	Not used, for API compliance only.

Output: *int* 0 if write successful.
<0 if some fault occurred

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL, EIO or EBUSY

3.3.15 Diagnostic Functions

Some diagnostic functions are provided, and more may be provided in the future. Please refer to the source code for available functions. (Diagnostics functions are located in *rdcd*.c* files.)

The diagnostics functions use the *rdcPrintf* macro defined in *rdcconf.h*, which normally in turn refers to *shrdPrintf* macro defined in *shrdconf.h*. This function must be provided in order to make any use of the diagnostics functions.

Appendix A: RDC Integration Guide

This guide assumes that you have a basic knowledge of your compiler and build environment. It provides a step-by-step instruction in order to start using RDC on your target.

1. **Configuration:** Do the entire configuration required by modifying the *src/rdc/rdcconf.h* and *src/shared/shrdconf.h* files. It is recommended that debugging is enabled at first, if some support for *printf()* is provided.
2. **Ensure that RDC compiles in your environment:** First include the RDC source files (*src/rdc/*.c* and *src/shared/*.c*) in your make or build environment. You should ensure that all the include files (*src/rdc/*.h* and *src/shared/*.h*) are available when compiling the source files.
3. **RAM driver:** Design and **test** the RAM device driver to be used for the target if not using internal RAM device driver. Please refer to “The RAM Driver Application Programming Interface (RD-API)” chapter for details.
4. **Protection:** If RDC is to be used without higher level reentrancy protection in a multi-user environment, design the protection functions as described in the “The RAM Protection Application Programming Interface (RP-API)” chapter. (Most likely RDC will be used with TFFS, and reentrancy protection is defined for TFFS partition instead.)
5. **Opening and formatting in Non-Volatile mode:**

Below is a code example on how to open a 1MB RAM partition in non-volatile mode.

```
int applRdcOpenNonVolatile(void *start, U32 size)
{
    int ret;

    /* Must be done once before any use of RDC */
    rdcInit(1);

    /* Try to open RDC partition, format if not able to open it */
    ret = rdcOpen(start,          /* Address to start of partition */
                 NULL,
                 NULL,           /* Use internal RAM read driver */
                 NULL,           /* Use internal RAM write driver */
                 NULL,           /* No reentrancy protection provided */
                 NULL,
                 NULL,
                 RDC_DIAG_ALL);

    if (ret < 0)
    {
        ret = rdcFormat("EXAMPLE",
                       start,    /* Address to RAM to use as NV partition */
                       2,        /* Align blocks on address divisible by 4 */
                       size,     /* Size of RAM partition */
                       9,        /* Block size 512 (2^9) bytes */
                       eRdcNonVolatile,
                       NULL,     /* Use internal RAM read driver */
                       NULL,     /* Use internal RAM write driver */
                       NULL,     /* No reentrancy protection provided */
                       NULL,
                       RDC_DIAG_ALL);

        if (ret < 0)
        {
            printf("RDC format failed\n");
            return(ret);
        }

        /* Same open as above */
        ret = rdcOpen(start,          /* Address start of partition */
```

```
        NULL,
        NULL,          /* Use internal RAM read driver */
        NULL,          /* Use internal RAM write driver */
        NULL,          /* No reentrancy protection provided */
        NULL,
        NULL,
        NULL,
        RDC_DIAG_ALL);
    if (ret < 0)
    {
        printf("RDC open failed\n");
        return(ret);
    }
}

return(ret);
}
```

And the application would look something like below:

```
int rdcHandle;  /* Keep this for later use */

rdcHandle = applRdcOpenNonVolatile((void *)0x20000000, 0x100000);
if (rdcHandle < 0)
{
    /* Can't use RDC for some reason.... */
}
```

6. Opening and formatting in Volatile mode:

Below is a code example on how to open a 1MB RAM partition in volatile mode.

```
int applRdcOpenVolatile(void *start, U32 size)
{
    int ret;

    /* Must be done once before any use of RDC */
    rdcInit(1);

    /* Should normally always just format volatile partition first */
    ret = rdcFormat("EXAMPLE",
        start,          /* Address to RAM to use as volatile partition */
        2,              /* Align blocks on address divisible by 4 */
        size,           /* Size of RAM partition */
        9,              /* Block size 512 (2^9) bytes */
        eRdcVolatile,
        NULL,           /* Use internal RAM read driver */
        NULL,           /* Use internal RAM write driver */
        NULL,           /* No reentrancy protection provided */
        NULL,
        RDC_DIAG_ALL);

    if (ret < 0)
    {
        printf("RDC format failed\n");
        return(ret);
    }

    /* Same open as above */
    ret = rdcOpen(start,          /* Address to start of partition */
        NULL,
        NULL,          /* Use internal RAM read driver */
        NULL,          /* Use internal RAM write driver */
        NULL,          /* No reentrancy protection provided */
        NULL,
        NULL,
        NULL,
        RDC_DIAG_ALL);

    if (ret < 0)
```

RAM Disk Controller (RDC) Description

```
    {
        printf("RDC open failed\n");
        return(ret);
    }

    return(ret);
}
```

And the application would look something like below:

```
int rdcHandle; /* Keep this for later use */

rdcHandle = applRdcOpenVolatile((void *)0x20000000, 0x100000);
if (rdcHandle < 0)
{
    /* Can't use RDC for some reason.... */
}
```

If all this has succeeded, the next step is to start using RDC by writing and reading blocks. RDC will probably be used with TFFS, since standalone use is of little use.