Using NRF

Version 2.2

www.radisys.com
World Headquarters
5445 NE Dawson Creek Drive • Hillsboro, OR
97124 USA
Phone: 503-615-1100 • Fax: 503-615-1121
Toll-Free: 800-950-0044

International Headquarters
Gebouw Flevopoort • Televisieweg 1A
NL-1322 AC • Almere, The Netherlands
Phone: 31 36 5365595 • Fax: 31 36 5365620

RadiSys Microware Communications Software Division, Inc.
1500 N.W. 118th Street
Des Moines, Iowa 50325
515-223-8000

Revision C
November 1999

Copyright and publication information

This manual reflects version 2.2 of Non-Volatile RAM File Manager (NRF).

Disclaimer

Reproduction notice

November 1999

Table of Contents

Chapter 1: Overview

This chapter is an overview of NRF (Non-volatile RAM File Manager). NRF manages a random access file system stored on battery-backed system memory.

The following sections are included in this chapter:

Introduction

NRF (Non-volatile RAM File manager) manages a random access file system stored on non-volatile RAM (NVRAM). NVRAM is a Random Access Memory device that can hold its data without AC power because it is stored battery-backed. It is available as a Dual Ported Input/Output (DPIO) subsystem for OS-9.

NVRAM is a limited and precious resource for DAVID systems because they usually only have about 4-8k of NVRAM. NRF is specifically customized to use as little of this memory as possible for file system overhead by providing a flat file system using a link list of files and by using smaller and more compact file descriptors than what is found in the Random Block File Manager (RBF). Fragmentation in an NRF file system is zero.

An important feature of NRF is its robustness. If a system crash occurs during a file update, only the file being updated is lost. The rest of the file system remains intact.

The Application Programming Interface (API) for NRF is RBF compatible, allowing for easy portability between RBF disk applications and NRF. The following RBF utilities work with NRF files: `dir`, `list`, `copy`, `attr`, `del`, `nfree`, and `dump`.

NRF, like all OS-9 I/O systems, has four primary software blocks:

- Application Interface Library
- File manager
- Device driver
- Device descriptor

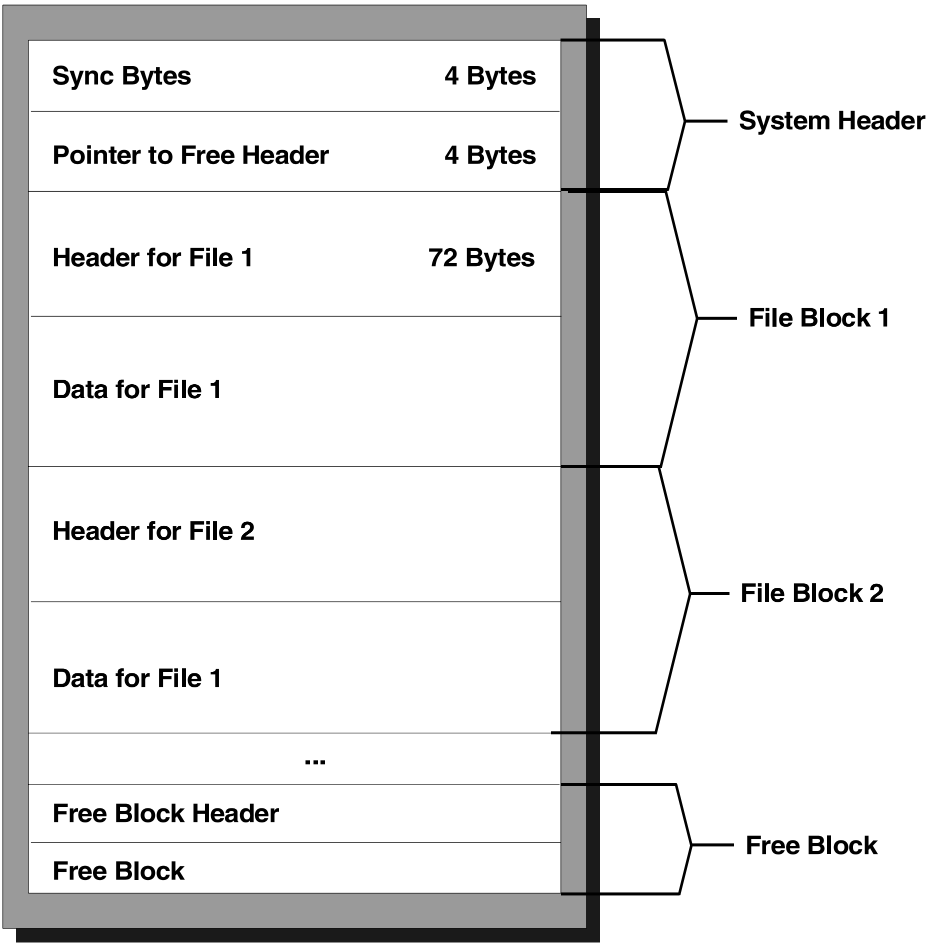
Applications make calls through the interface library, which are passed by the kernel down to the file manager. The file manager handles the call and if necessary, calls the driver. Both the file manager and the driver use the device descriptor during system initialization to determine the specific configuration details of the hardware.

Since NRF is RBF compatible at the application layer, the Application Interface Library for NRF is `os_lib.l`, a standard Ultra C library.

File System

The file system is maintained as a two-way linked list of files. The file system consists of a system header block, a link list of file blocks (with each file having its own header), and a free block. The free block is always kept at the bottom of the file system as one contiguous chunk. If a file has to be expanded/deleted, the other files are moved one-by-one to fill the gap. The following figure displays the file system.

Figure 1-1 NRF File System



Chapter 2: NRF Application Programming Interface (API)

This chapter explains the specific calls and entry points supported by NRF. Random Block File (RBF) manager compatibility and system recovery are also discussed.

The following sections are included in this chapter:

NRF API

Non-volatile RAM File Manager (NRF) provides a Random Block File Manager (RBF) compatible API for accessing random access files stored on non-volatile RAM. The RBF compatibility ensures that most OS-9 utilities (such as `dir`, `list`, and `attr`) work with NRF files. The table below lists the specific calls supported by NRF. NRF calls are located in `os_lib.l`, which is fully documented in the ***Ultra C Library Reference***. Your application may also use any standard ANSI C calls such as `fopen` and `fclose` for reading and writing file.

Table 2-1 NRF API Calls

Entry Point	System Call	Description
`attach`	`_os_attach()`	Initialize the NVRAM device
`detach`	`_os_detach()`	Deinitialize the NVRAM device
`create`	`_os_create()`	Create a file
`open`	`_os_open()`	Open a file
`close`	`_os_close()`	Close a file
`delete`	`_os_delete()`	Delete a file
`seek`	`_os_seek()`	Seek to a random file position
`read`	`_os_read()`	Read from current file position
`readln`	`_os_readln()`	Read a line from current file position
`write`	`_os_write()`	Write to the current file position

Table 2-1 NRF API Calls (continued)

Entry Point	System Call	Description
`writeln`	`_os_writeln()`	Write a line to the current file position
`getstat - SS_OPT`	`_os_gs_popt()`	Read path options
`getstat - SS_SIZE`	`_os_gs_size()`	Get the size of a file
`getstat - SS_EOF`	`_os_gs_eof()`	Check for end of file
`getstat - SS_FD`	`_os_gs_fd()`	Read the file descriptor
`getstat - SS_POS`	`_os_gs_pos()`	Read the file position
`getstat - SS_FDINF`	`_os_gs_fdinf()`	Read a specific file descriptor sector
`getstat - SS_FDAddr`	`_os_gs_fdaddr()`	Get the file descriptor block address **(OS-9 only)**
`getstat - SS_FREE`	`_os_gs_free()`	Get the amount of free space on the device
`setstat - SS_OPT`	`_os_ss_popt()`	Set the path options

Table 2-1 NRF API Calls (continued)

Entry Point	System Call	Description
setstat - SS_SIZE	_os_ss_size()	Set the file size
setstat - SS_ATTR	_os_ss_attr()	Set the file attributes

The entry points not supported include ChgDir (Change Directory) and MakDir (Make Directory) which are not required because NRF only provides a flat file system.

The file manager builds, maintains, and interprets the device file structure. The file manager depends on the device driver to initialize the hardware and read/write bytes to and from the physical device.

RBF Compatibility

As previously mentioned, the API for NRF is RBF compatible. This is possible because of translations performed by the file manager between NRF file descriptors and RBF file descriptors and vice versa. Therefore, even though the NRF file descriptor is different (smaller) than the RBF descriptor, an API request for a file/directory entry results in a RBF compatible data structure returning. As a result, most RBF utilities except `free` work with NRF disks. To determine the amount of free space on a NVRAM disk, you use a special utility called `nfree`.

File Names

NRF accepts any legal RBF file name up to a maximum of 28 bytes (including the `null` terminator).

NRF Design Impact on Applications

The file system is maintained as a two-way link list of files. Since zero fragmentation is the goal for NRF disks, the free block is always kept at the bottom of the file system in one contiguous chunk. If a file has to be expanded/deleted, the other files are potentially moved to create and fill the gap. To minimize this disk reorganization, applications that know the minimum size of the data they are going to write to a file can declare it at file open time using the `I_SIZE` option.

Path/Logical Unit Options

The path and logical unit options structure are in:

```
$MWOS/SRC/DEFS/DAVID/nrf.h
```

Different Disk Types

nrf.h defines the following types of disks that can be managed by NRF.

```
#define NRF_NVRAM_DISK 0          /* regular nrf disk on nvram */
#define NRF_FAKERAM_DISK 1   /* fake nrf disk on system ram */
#define NRF_DATAMODULE_DISK 2     /* nrf disk in a data module */
```

The NRF_FAKERAM_DISK tests the NRF subsystem using system RAM as pseudo NVRAM. The memory is allocated by the driver.

The NRF_DATAMODULE_DISK is a NRF disk in a data module. The data module is called <descriptor name>_mod and is created by the driver. You can write to this disk and then burn the data module into ROM to get a read-only NRF disk available at run-time. This enables embedded, diskless applications to access a read-only NRF disk.

This is useful in diskless environments where a small amount of predefined data for fonts or network tables needs to be stored.

Crash Recovery

In the event of a system crash during file system updates, NRF attempts to limit the damage to the file system to just the files being updated. When the system comes up again, NRF runs a disk recovery algorithm. All files that are potentially damaged are renamed to "_ _<filename>" after recovery. Files that are completely unrecoverable due to loss of some key attributes are deleted. The remaining files should be completely intact.

Detection of a Valid Formatted Disk

When an unformatted NVRAM disk is first accessed, NRF writes the system header block and the free header block on the disk. The system header starts with a two-byte sync prefix (`NRF_SYNC_PREFIX`) and a two-byte sync suffix (`NRF_SYNC_SUFFIX`). In subsequent accesses to the disk, the first two sync bytes are never touched. During updates to the disk:

1. The sync suffix bytes in the system header are updated
2. The sync bytes in the file header are updated
3. The file itself is updated

When the device is initialized, NRF first checks the first two bytes (the sync prefix bytes) of the disk. If their value is not equal to `NRF_SYNC_PREFIX`, the disk is assumed to be an unformatted disk and is formatted by writing the system and the free header blocks. If the sync prefix is acceptable, the sync suffix is checked. If the sync suffix is not equal to `NRF_SYNC_SUFFIX`, the disk is assumed to be corrupted, and the recovery strategy is executed.

Chapter 3: Porting NRF Drivers

This chapter explains how to port NRF drivers. The following topics are included:

- **NRF Device Driver**
- **Making the Driver**
- **NRF Device Descriptor**

MICROWARE SOFTWARE

NRF Device Driver

The NRF device driver is designed to initialize the hardware and read/write bytes to and from the application. The implementation of the file system is performed by the file manager.

NRF Device Driver Assumptions

The generic driver included in this package assumes that the device is memory mapped. It can handle both contiguous and non-contiguous NVRAM (using only alternate bytes) by using a flag set in the descriptor.

Note

NRF assumes that device drivers never sleep in any of their entry points.

Entry Points

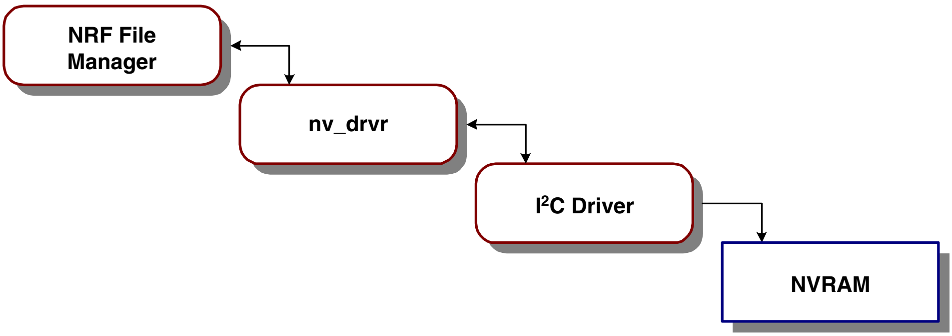
`v_init`	Initialization of hardware. The sample driver uses system RAM as pseudo NVRAM. This entry point allocates system RAM.
`v_terminate`	De-initialization of hardware. Deallocates pseudo NVRAM, if required.
`v_read`	Read specified number of bytes from a specified location on the disk. The location is specified relative to the start of the disk.
`v_write`	Write specified number of bytes to a specified location on the disk. The location is specified relative to the start of the disk.

`v_move`	Move bytes on the disk. Since the file manager frequently reorganizes the disk, a `v_move` entry point has been added to move data on the disk. This should speed up disk moves on devices where the NVRAM is memory mapped and a direct memory to memory copy is possible for moving data. If this is not the case for a particular driver implementation, the driver can choose to return `E_UNKSVC`. The file manager performs the move operation using reads and writes.
`v_getstat`	If the file manager receives a `getstat` it cannot handle, it calls this entry in the driver and returns `E_UNKSVC`.
`v_setstat`	If the file manager receives a `setstat` it cannot handle, it calls this entry in the driver and returns `E_UNKSVC`.

The Hellcat NRF Driver

On the Hellcat, the NVRAM hardware is accessed over the I^2C bus. The I^2C bus is managed by the I^2C device driver, which supports an inter-driver communication API. The NRF driver uses this inter-driver API to pass requests to the I^2C driver to access the bus.

Figure 3-1 Inter-driver Communication on the Hellcat



For More Information

Refer to ***Using NullFM*** for more information about inter-driver communication.

Making the Driver

The driver source is in:

```
$MWOS/SRC/DPIO/NRF/DRVR/NVDRV
```

The makefile location is dependent on the type of system and is called `drv.mak`. The makefile for the generic port is in:

```
$MWOS/OS9000/PPC/PORTS/EXAMPLES/NRF
```

The driver source for the Hellcat port is in the directory:

```
$MWOS/SRC/DPIO/NRF/DRVR/I2CDRV
```

The makefile for the Hellcat port is in:

```
$MWOS/OS9000/821/PORTS/HELLCAT/NRF
```

NRF Device Descriptor

The makefile for the generic PowerPC port is called `desc.mak` and is located in:

```
$MWOS/OS9000/PPC/PORTS/EXAMPLES/NRF
```

In the same directory, there is a file called `desc.h` that must be customized for your NRF subsystem. The following variables can be set:

`PORTADDR`	Base address of NVR hardware

`NRF_DISK_TYPE`	Defines one of three disk types:
	`NRF_NVRAM_DISK`
	`NRF_FAKERAM_DISK`
	`NRF_DATAMODULE_DISK`

`NV_RAM_SIZE`	NV RAM size
`CONTIGUOUS_RAM_FLAG`	Set to 1 if NVRAM is contiguous; otherwise, set to 0.
`VECTOR`	Interrupt vector
`IRQLEVEL`	Board IRQ level
`PRIORITY`	IRQ polling priority
`LUN`	Logical Unit Number
`IOMODE`	Permissions
`DRIVERNAME "name"`	NRF driver name

Index

Symbols

A

C

I

M

N

O

P

R

S

U

W

Product Discrepancy Report

To: Microware Customer Support

FAX: 515-224-1352

From:____

Company:_______________________________________

Phone:____

Fax:_____________________Email:____________________

Product Name:

Description of Problem:

____

____

____

____

____

____

____

____

____

____

____

____

Host Platform_____________________________________

Target Platform____________________________________

RadiSys.

MICROWARE SOFTWARE