OSE systems
An Enea Company

Embedded File System

OSE
Embedded File System

User's Guide

OSE Systems

Copyright

Copyright © 2001 by OSE Systems. All rights reserved. No part of this publication may be reproduced, transmitted, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, optical, chemical or otherwise, without the prior written permission of OSE Systems. The software described in this document is furnished under a licence agreement or a non-disclosure agreement. The software may be used or copied only in accordance with terms of agreement.

Disclaimer

OSE Systems makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, OSE Systems reserves the right to revise this publication and to make changes from time to time in the contents hereof without obligation to OSE Systems to notify any person of such revision or changes.

Trademarks

OSE is a registered trademark of OSE Systems.

Document No: 420e/OSE52-2 R3.1

Contents

Embedded File System

1 Introduction

The **OSE Embedded File System, EFS**, is a file system for the **OSE Real Time Kernel**. EFS will enable application processes to use file system functionality in several different ways by using the interfaces described in this reference manual. The manual describes the concepts and design ideas used when implementing the EFS and contains information how to use the EFS in an OSE system.

1.1 About his manual

This **EFS User's Guide** contains a description of how to use and configure the EFS as well as a description of the shell and login support delivered with EFS. The guide is divided into the following chapters:

This user's guide describes:

- "EFS - System Description" on page 9 describes the Embedded File System
- "EFS Shell Commands" on page 27 describes the included shell and login processes with interactive tools which can be used for the basic administration of the file system.
- "Configuration of EFS" on page 35 explains how to configure an OSE system with EFS

See the **EFS Reference Manual** for all details regarding the interfaces to EFS. In the reference manual also an extensive glossary, with terms and definitions, is included.

1.2 New for EFS User's Guide

- In "EFS Shell Commands Reference" on page 28 under the command "ls" on page 30 the -a flag has been added and the syntax and description have been changed.

Embedded File System

1.3 Requirements

Those who need some kind of storage capacity and is therefore involved in writing applications, file system format managers or lower level device drivers for the EFS should read this manual. Since the EFS replaces other file systems such as those in UNIX or DOS, any application running on the OSE platform in need of file and device I/O should use the EFS. Some of the expected users are:

- Platform developers - writing format managers or device drivers and configuring systems.
- Application programmers - writing applications that use a file system.

The reader is expected to have knowledge of the C programming language since the EFS interface is expressed in C and examples will be given in C. Some knowledge of the OSE operating system is necessary since EFS runs on this platform and explanations sometimes depend on knowledge of issues such as OSE inter-process communication.

Version R3.0.0 of the EFS is a complete update, not backward compatible with the earlier version.

Embedded File System

1.4 Overview of the OSE Embedded File System

The OSE Embedded File System, EFS, is designed to handle volumes, files and directories and their attributes in an OSE environment. It also includes support for handling character device drivers through a file metaphor, similar to Unix.

Shell and Login Process - Shell

A shell process and a login process are delivered with the OSE Embedded File System, providing an interactive environment with which typically file system administration can be performed. See Shell User's Guide and Reference Manual.

Function Library Interface - FLIB

The Function Library (FLIB) function calls are used by applications. These are functions compatible with official standards such as ANSI-C and POSIX. Applications using FLIB do not need to use any of the signal interfaces below.

File System Server Signal Interface - FSS

The File System Server (FSS) provides access to all file system resources. All resources must register themselves with the FSS.

Format Manager and Volume Manager Signal Interfaces - FM (VM)

The Format Managers (FM) and Volume Managers (VM) take care of the file structure on the data stream on an underlying device. Applications using the signalling interface, and the optional BIOS trap interface, access the Embedded File System through the FM/VM interfaces.

Blocked Device Driver Signal Interface - DDB

The Blocked Device Driver (DDB) process provides a standardized access interface to EFS low level blocked device drivers.

Character Device Driver Signal Interface - DDC

The Character Device Driver (DDC) process provides a standardized access interface to EFS low level character device drivers, typically serial channels.

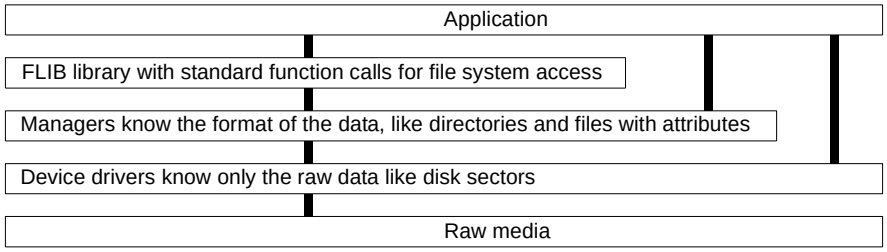
2 EFS - System Description

A **OSE Embedded File System, EFS**, provides a structured means of storing and retrieving data in the OSE distributed real time operating system.

From the application's point of view, data is stored in files within hierarchical directories and mounted volumes. Volumes are mounted using format managers, knowing the format of a specific file structure. The volume managers use device drivers to access the raw physical media.

Also serialized data channels are managed using the same file structured access methods.

Apart from the data itself, a file usually contains some file system specific attributes, such as file name, creation and modification time, access flags etc. The attributes for each file and the hierachical directory structure constitutes the format of a file system. For a serialized data channel the format includes terminal attributes, like control characters to be decoded.



Instead of using the traditional OSE signals for communicating with the file system managers, an application can use the FLIB function library, with standard ANSI-C and POSIX function calls for file system access.

Embedded File System

2.1 Embedded File System

The OSE EFS contains several different components.

2.1.1 EFS Components

- Function Library - FLIB - provides clients with a standardized API.
- The File System Server - FSS - manages the EFS resources.
- Format Managers - FM - imposes a format on the media and mounts volumes.
- Volume Managers - VM - created by format managers to handle each volume.
- Blocked Device Drivers - DDB - handles raw block type media, e.g. disks.
- Character Device Drivers - DDC - handles raw serialized media, e.g. terminal connections.

2.1.2 Additional Tools

In addition the following tools are delivered with EFS:

- Command Shell with daemon - command line interpreter with several standard commands.
- Login Process - handles the authentication of users.

2.1.3 Interactions with EFS and Registration of Resources

Applications usually (and preferably) only interact with the EFS through the function library. This library implements standard functions calls, as specified in POSIX and ANSI. Alternatively it uses the signal interfaces of the FSS and FM (VM uses the FM interface). The DDB and DDC device drivers are usually tightly integrated with the FM/VM processes and are seldom accesses directly.

All file system resources register with the FSS file system server, which acts as a resource broker to allow an application to transparently find a file system in whatever machine it is implemented. An FSS is required in each machine for access to the distributed EFS.

2.1.4 Interface Support Requests

Each component in the OSE Embedded File System should support a signal to be used to request a list of supported signals and features. This simplifies future updates and allows the same client to adapt to different types of components. The EFS signal interface is common to many different types of I/O systems and some signals might not be of much use to certain devices. A general rule however is to accept and respond to as many EFS signals as possible, even if there is no real operation to perform. Using this method a general client could communicate in the same way with many different types of I/O systems.

2.1.5 Implementation Dependent Parameters

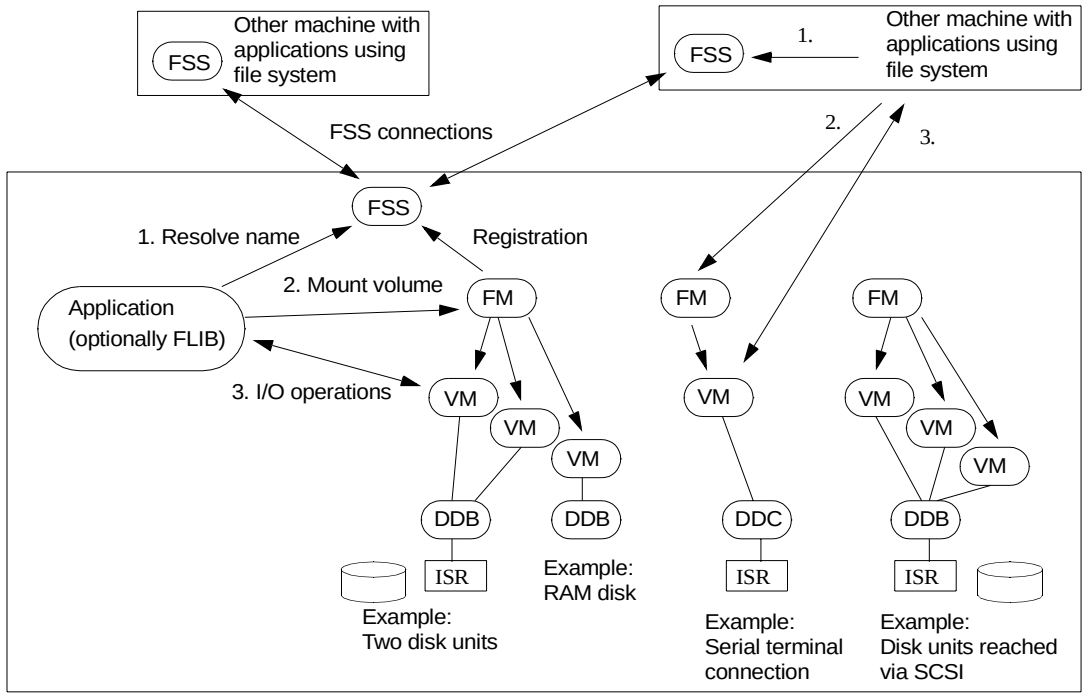
In several OSE signals with operations to the format or volume managers and the device drivers, optional parameters are sent in a text string argument, allowing implementation dependent parameters to be used within the current signal interface.

It is, therefore, essential that the implementer of different components clearly specifies and documents the names and values of these optional parameters.

Embedded File System

2.2 Flow of Signals Through EFS Components

The following figure illustrates how the file system components interact, using OSE signals within a machine as well as across remote links within OSE.



All these operations are hidden within the FLIB library if the application uses it.

The File System Server serves as a name repository, where all other modules register themselves.

1. All format managers, FM, register themselves with there local file system server, FSS.
2. Any initiating routine can ask file system servers in different machines to connect to each other to share the registered resources.
3. An application asks FSS for a suitable format manager using a well known name. Example: "extfat" which handles an extended DOS FAT file system. A process id is returned, which could point to a local or remote FM process.
4. The application asks this FM to mount a volume on a given device and gives the volume name to be used for the volume. FM creates a volume manager VM to handle this volume. The volume manager will register itself with FSS, allowing others to find the now mounted volume.

5. A volume manager connects with the device driver and handles all operations between the application and the device driver. A device driver may include an ISR, interrupt service routine, to handle the actual hardware. A device like a RAM disk does not need any hardware access.
6. Alternatively, the application could directly ask FSS for an already mounted volume.
7. I/O operations for access to directories and files are given by the application through the volume manager VM.
8. If the volume handles a serial channel, special terminal connection I/O operations are available for handling user interactions. The login and shell processes uses a serial channel.
9. Finally a volume can be unmounted, unless it should not stay accessible.

Embedded File System

2.3 Path Names and Current Directory in EFS

2.3.1 Path Names, Volume Names and Labels

The EFS file system root can not contain files or directories, only volume names. A full absolute path name consists of the following components, separated with slashes (/).

1. **Volume name**. This is the name the volume is given when it is mounted. Example: "/ram"
2. Optional **directory names** as a hierarchical path.

NOTE the leading slash, which always must be a part of a registered volume name!

3. **File name**.

Example of an absolute path of a file:

```
"/ram/applications/database/index.dat"
```

A different example, where the volume name alone is used to access a serial channel:

```
"/tty"
```

A **label** is a name string stored permanently on some types of devices or on a part of a device that will be mounted as a volume. A usual case is that applications try to use this same name when mounting disk volumes, but there is no enforced connection between a label and a volume name.

2.3.2 Current Directory and Other Status Information

The application process is responsible for keeping status information about process specific things like current directory, current position in files, any per process file buffers etc. This is automatically done for the application if the FLIB function library is used.

Format/volume managers and device drivers only keep status information about the objects, e.g. things like file permission attributes, access locks, cache buffers in the volume manager or device driver and a count how many open file handles there are on each object.

2.3.3 Relative or Absolute Path Names

If the FLIB function library is used, a current directory for a process is remembered by FLIB and the application can use relative path names. A relative path is without a leading slash and assumed to be relative to the current directory.

In all other cases, absolute path names must be used.

Embedded File System

2.4 Function Library - FLIB

Applications can use standard ANSI-C or POSIX function calls to access EFS file systems, if they are linked with the EFS FLIB library. When an application uses FLIB, there is no need to use any of the other signal interfaces described here, except if File System Server (FSS) resources shall be shared with other machines, in which case an FSS signal need to be used. For best portability, it is recommended to use FLIB functions instead of the signal interfaces.

The function calls in FLIB is defined in the standard header files in POSIX and ANSI-C and in addition in the "efs.h". For the shell and login tools, see "shell.h" and "passwd.h".

2.4.1 Functions

A total of about 90 standard functions are available, including open(), fopen(), opendir(), tcgetattr() and many others. In addition, a few EFS-specific function calls are included. These are:

- efs_clone() to export the file system status, e.g. open files etc., from a process to another.
- efs_format() to format a mounted volume, e.g. a disk partition.
- efs_mount() to mount a volume, giving it a name.
- efs_relabel() to write a label string to a volume.
- efs_shared() to share file descriptors within a process segment.
- efs_sync() to force cache buffers to the physical storage.
- efs_unmount() to unmount a volume.
- eprintf() to simplify printing formatted data to standard error, compare printf().
- validate_user() is a declaration for a user-supplied function for LOGIN authentication.

2.4.2 Process Status

In addition to the function call interface, the FLIB also keeps track of process specific file system status information like the file descriptors, FILE structures and there contents with current positions, open modes etc. as well as the concept of a current directory for the process.

2.4.3 Linking Modules

The function library is linked with the application. The library is subdivided into many small modules which allow only the parts that really are used to be included in the application executable when it is linked with the library.

Embedded File System

2.4.4 EFS processes for function calls

The function calls in FLIB are directly translated into OSE signals that are sent to the different process components in the Embedded File System.

Signals are sent to:

- FSS to find resources.
- FM processes in order to mount volumes.
- VM processes in order to interact with volumes.

There is no direct communication between applications using FLIB and device drivers (DDB, DDC).

Embedded File System

2.5 File System Server - FSS

The file system server, FSS, acts as a file system resource broker. It should "know" all file system resources, both local resources and remote resources needed by any process in the system. When new resources are created, the resources must register themselves with the FSS to allow others to find them. FSS is no longer involved when a client process has found a resource and starts using it.

One FSS process, a static process named "ose_fss", is required in each machine where there are file system resources or where there are applications wanting to use file system resources. The signal interface to FSS is defined in the "fss.sig" header file.

2.5.1 FSS can Share their Global Resources with Other Machines

In a distributed system, a suitable initiating routine can create connections between the local FSS and other remote FSSes in other machines to share the own resources with the remote machines. As long as this remote connection is open, any changes in the resources will be updated on the other machines. FSS will attach to the connected remote FSS processes and re-establish the connection

The FSS connections are unidirected, i.e. "I share my resources with you". Thus other connections must be opened from the other remote FSSes to my FSS to enable me to reach their resources.

A resource can be registered as "local" in FSS. In this case it is not shared with any remote FSS. A resource which is not local is called a global resource.

2.5.2 All Resources Must Register with FSS

File system resources are typically different Format Managers, Device Drivers and dynamically created Volume Managers. When created, these must be registered with the local FSS. Usually when these resource processes are started, they hunt for ose_fss and register themselves with FSS before they start waiting for signals to perform requested actions. The FSS will unregister them when the resource processes are killed. A resource is registered as:

- Resource **type**, e.g. Format Manager, Device Driver, Volume Manager, Remote FSS.
- Resource **location**, e.g. if it shall be local or globally shared with other FSSes.
- Resource **name**, e.g. the name string with which a resource is found.

If the local FSS process is not reachable when a resource is started, it shall wait and avoid performing any actions until it has successfully registered with FSS.

Embedded File System

2.5.3 Resource Names and Resource Types

Resources are registered in FSS with a name string, which must be unique in FSS, depending on the type of resource. Typical names of format managers can be "fat", "extfat", "confm". Volume managers which are dynamically created (see path and volume names in “Path Names and Current Directory in EFS” on page 14) usually have names similar to disk volume labels or a serial ports and ALWAYS begins with a slash (/) as they are the "root" of the volumes. Typical device driver names are "serdd", "dosfm".

FSS recognized a range of file system resource type names, listed in fss.sig. An additional wildcard resource type, FSS_TYPE_ANY, is defined to be optionally used when finding resources.

2.5.4 Resource Resolving with Timeout

To find a resource without using FLIB, an application or other client process hunts for the FSS (ose_fss) and sends a signal to resolve a resource name. If the resource is not immediately available, FSS will save the request and when the resource finally registers, the reply is sent to the application. The reply signal is sent with the resource as "sender", allowing the application to continue to interact directly with the resource.

A timeout is specified in the request, allowing the client to define what to if no resource appears during a long time.

Embedded File System

2.6 File Managers - FM

A format manager, FM, is a process which imposes a format (structure) on a raw device. The FM knows the structure, which typically can be files in directories stored on the raw blocks on a disk unit or it can be the structure with lines ended with new-line and control characters input or output on a serial channel. The format manager uses device drivers to access the raw devices, see DDB and DDC.

When created, each format manager must register itself with the FSS.

The same common OSE signal interface is defined for FM and VM. The interface is defined in the "fm.sig" header file. See “Common Signal Interface for FM and VM” on page 22.

2.6.1 FM will Mount Volume Managers and Share Signal Interface

In OSE EFS, the main task of the format manager process is to respond to mount requests and create different Volume Manager processes for each volume which is mounted. The application or other client finds the registered name of a suitable FM from the FSS and sends a mount request to it. Further communication is performed directly with the new Volume Manager.

Alternatively a client can directly find an already mounted Volume Manager in the FSS and communicate with it without using the FM at all.

Embedded File System

2.6.2 Different Types of Format/Volume Managers

Many different format managers might exist in a system, supporting different types of data. The EFS is delivered with the following format managers:

- DOS/FAT format manager, handling a 12 and 16 bit FAT file structure on a disk, limited to the usual 8+3 file name size and only the modification time. This format manager (dosfm) registers itself as "fat" in the FSS.
- Extended FAT format manager, handling long file names, up to 48 characters, and additional attributes for read/write/execute(search) permissions as well as creation and modifiation time with one second resolution. This format manager (extfm) registers itself as "extfat" in the FSS.
- Console terminal format manager, handling a serial terminal connections through character devices (DDC), like serial devices or telnet connection. POSIX standard termios type control character handling is included. This format manager (confm) registers itself as "confm" in the FSS.

The telnet device driver is not a part of EFS but a component in the Internet Utilities product.

2.6.3 Location

A volume manager is usually dynamically spawned by its format manager and often share code with the FM. In addition a volume manager may need efficient communication with the device driver. Therefore the most usual situation is having the format and volume manager as well as the device driver within the same machine. This is not necessary if all communication is through signals, but if the optional BIOS trap interface for fast data read/write shall be used between the VM and the device drivers, they must be in the same machine.

Embedded File System

2.7 Volume Managers - VM

Applications send all file related I/O operations to volume managers, which handle any formatting of data or operations regarding the structure of the file systems and reads or writes data using operations towards device drivers. See DDB and DDC.

Volume managers are dynamically created by the format manager when a volume is mounted and must register itself with the FSS, allowing other applications to find and use it, in addition to the application performing the mount operation. Also remote applications can find the volume through FSS and communicate with the volume manager to access the volume.

When unmounted, a volume manager must terminate, allowing FSS to unregister it automatically.

The same common OSE signal interface is defined for FM and VM, and the volume managers often share code with the format manager. The interface is defined in the "fm.sig" header file.

2.7.1 Volume Manager Status and Control

As it is the volume manager that knows how the raw data shall be interpreted, it has the responsibility to keep the status of objects in the data and control access to it.

Embedded File System

2.7.2 Common Signal Interface for FM and VM

The same common OSE signal interface is defined for FM and VM. The interface is defined in the "fm.sig" header file.

A format/volume manager should respond to all defined signals, but can select to reply with an error reply informing the sender about not supported features.

Operations are available for the following groups of operations:

- Mounting volumes. This is the only operations sent to FM and creates a volume manager.
- Unmounting volumes. An unmount operation is sent to the VM (not FM).
- Formatting a volume, validating the structure, examine a volume, label a volume or force cache buffers to be written to a volume (sync).
- Operations on object (typically directories or files) directly using path names. Create, remove, rename, examine, set size, attributes, times or owner.
- Operations on open objects using a handle. Open, close, examine, set size or attributes, read or write data with buffers in the signal or using pointers (only within the same machine), force cache data for object to be written to the volume, wait for events, cancel requests, perform lock operations.
- Operations special for terminal connections, using a handle. Examine and set terminal configurations or flush the I/O buffers.

Reading from a volume can be performed by activating asynchronous read. The volume manager will then continue to send data until the read is deactivated or an error occurs. This is typically of use when reading from an asynchronous serial line.

Embedded File System

2.8 Block Device Drivers - DDB

Storage devices, like disks, are accessed from the Volume Managers through a device driver, which can handle block structured raw data. Device drivers are usually not accessed directly from applications.

Device drivers in the OSE EFS are processes, which must register with the FSS to allow volume managers to find them. Interrupt Service Routines ISR are often used to handle hardware.

The DDB signal interface is a standard interface between the volume manager and the device driver and shall be implemented by all device drivers handling block structured devices. The interface is defined in the "ddb.sig" header file.

A device driver usually controls one I/O controller, like a SCSI controller or an IDE controller. One controller can have multiple disk units. Unit numbers 0 - 31 are supported. In addition there is support for mounting only a part of a disk unit, see below.

2.8.1 Operations

Operations are included to support low level formatting and for the use of cache buffers in the device driver. Device attributes are supported for things like removable, read-only, random access. The examine request can return various hardware attributes for a disk as well as for the device itself. A shutdown operation should force a device driver to terminate after proper actions.

The block device drivers knows nothing about the structure of the data, but sees the device as one big stream of blocks, where each block has a given fixed size. The volume manager must remember things like the current position for read and write etc.

2.8.2 Mounting Units or Partitions

Mount/unmount operations are used to reserve either an entire physical disk unit or a part of it for one volume manager. The area is given as a range of blocks within a unit. The volume managers are however responsible for staying within the reserved area as the blocks in read/write operations are always given as absolute block numbers on the unit.

Optionally a device driver could enforce locking of a mounted area. There is support in the signal interface for this.

Embedded File System

2.8.3 Types of DDB Device Drivers

Many different device drivers might exist in a system, supporting different types of devices. The EFS is delivered with the following block device driver:

- RAM disk device driver. This is delivered in source code with EFS as an example device driver suitable as a starting point for custom made device drivers. The RAM disk driver registers itself as "ramdisk" with the FSS and uses RAM allocated from the OSE pool as the media. It is suitable for use together with one of the FAT format managers. and also contains a BIOS trap handler to be used by the VM for efficient data transfer.

2.8.4 BIOS Trap Signal Operations

A BIOS trap handler can be returned at an interface request on a device driver. This handle can be used between the volume manager and the device driver for efficient data transfer without context switches. If sig is a signal with a read or write operation, a call like the following will cause the driver to "receive" the signal and perform the operation, after which the client can inspect the result in sig->status as usual and after a read, use the data.

```
sig = allocate_request_signal();
ret = biosCall(handle, fmPid, &sig); /* The signal operation is performed
*/
if (sig->status == EFS_SUCCESS) use_reply_signal ...;
free_reply_signal(&sig);
```

Embedded File System

2.9 Character Device Drivers - DDC

Terminal connections through serial ports or telnet connections contain data as a stream of characters which can not be randomly addressed and where some characters are control characters needing special treatment. These type of devices are handled by Character Device Drivers and also need volume managers of which know about how the control characters are treated. Character device drivers are accessed from Volume Managers and usually not accessed directly from applications.

Device drivers in the OSE EFS are processes, which must register with the FSS to allow volume managers to find them. Interrupt Service Routines ISR are often used to handle hardware.

The DDC signal interface is a standard interface between the volume manager and the device driver and shall be implemented by all device drivers handling serial streams of characters. The interface is defined in the "ddc.sig" header file.

A device driver usually controls one I/O controller, like a controller for serial ports. One controller can have multiple channels, e.g. communication port 1 and 2.

2.9.1 Operations

The main operation of the DDC device driver is to transmit and receive data on a channel and control the hardware for this purpose. Additionally it can cache data in the device driver.

Operations are included to support the features in the POSIX termios.h interface, which is to a large extent implemented in the volume manager and in FLIB. Device attributes supported are specially designed for the lower levels of the POSIX interface needs with control of the transmit and receive streams and the serial hardware parameters.

A shutdown operation should force a device driver to terminate after proper actions.

Mounting and unmounting can be used to power on/off hardware in addition to selecting which physical channel to access.

Reading from a device is always performed by activating asynchronous read. The device driver will then continue to send data, as it arrives, until the read is deactivated or an error occurs.

Embedded File System

2.9.2 Types of DDC Device Drivers

Many different device drivers might exist in a system, supporting different types of devices. The EFS is delivered with the following characteristic device driver:

- Serial device driver "serdd" for controlling serial communication ports. It uses a Board Support Package (BSP) serial device driver to actually manipulate the hardware. It is suitable for use together with the confm terminal format managers. and also contains a BIOS trap handler used by the VM for efficient data transfer.

With the serdd driver and the confm format manager running on the target system, an RS232 terminal or terminal emulator can be connected to the RS232 port of the target and the terminal will be available for input and output in the file system. If the login process and SHELL tools are used, a user might login to the system and perform shell commands.

Embedded File System

3 EFS Shell Commands

3.1 Command Shell

EFS has a couple of optional commands which can be added to the shell through late start hooks. The term added means to register the commands in the group with the shell daemon so shell knows about them. See the Shell manual for details. The EFS optional commands are:

Group initEfsCmd

Embedded File System

3.2 EFS Shell Commands Reference

The following syntax is used when describing the command syntax:

[xxx] xxx is optional.

<yyy>Replace yyy with the actual value.

zzz zzz is a literal to be types exactly as shown.

zz/wwThe / means that either zz or ww can be used.

... The preceding item may be given multiple times.

Error messages

Generally errors from the commands are reported to standard error.

cat

Syntax cat [<filename> ...]

Description The cat command concatenates files. It reads each <filename> in sequence and outputs the file's data on the standard output. Without filename, it reads from standard input. Exit the standard input read mode by pressing Ctrl-x.

Start Hook Enabled with the late start handler START_OSE_HOOK2(initEfsCmds).

cd

Syntax cd [<dirname>]

Description The cd command changes the working directory of the shell process. If no argument is specified and the OSE environment variable HOME has a value, cd tries to change directory to the value of HOME.

Start Hook Enabled with the late start handler START_OSE_HOOK2(initEfsCmds).

Embedded File System

chmod

Syntax chmod <mode> <path> ...

Description The chmod command changes the access permission mode (read/write/execute) for the specified files or directories. The mode is given as one three-digit octal number or as a sequence of the ‘r’, ‘w’ and ‘x’ characters. With letters, only one combination can be given which is set for all classes. With a three-digit octal number the three digits are for the classes owner, group and others.

Any support of classes depends on the format manager.

Note that volumes are never "owned" by anyone.

Example

chmod 444 file	allow only read for all classes
chmod r file	same as above
chmod rwx file	allow read, write, and execute (x=search for directories)
chmod 000 file	remove all permissions for all classes

Restrictions The DOS file system only handles write permission. For both the DOS FAT and the Extended DOS FAT file systems, the owner, group and other attributes are super positioned, if a permission is set for one class it is set for all three classes.

Start Hook Enabled with the late start handler START_OSE_HOOK2(initEfsCmds).

cp

Syntax cp <source> ... <destination>

Description The cp command copies one or more files to a destination directory or copies one file to a new name.

Restrictions There is no -p option, which would copy a file and keep the modification date. The handling of the modification date depends on the volume manager.

Start Hook Enabled with the late start handler START_OSE_HOOK2(initEfsCmds).

Embedded File System

format

Syntax format <volume> [<parameters>]

Description The format command formats the specified volume. The volume must exist, i.e. be the result of a device mounted on a format manager. No files may be open on the volume. See the command “mount” on page 31.

Device specific parameters can be specified in the optional <parameter> in the form "name1=value1;name2=value2 .... ". Also boolean parameters can be given, i.e. a name without any "=value".

Restrictions If the format fails (for instance if the parameters are invalid) the specified device may be left in a inconsistent state. Make sure that the format command always has valid parameters.

Example format "/ram" "fatsize=16,clustersize=4,fats=1,rootsectors=64"

Start Hook Enabled with the late start handler START_OSE_HOOK2(initEfsCmds).

ln

Syntax ln <filename> [<linkname>]

Description The ln command creates a link with the given name (linkname) to an existing file (filename). Observe the nonintuitive order of the arguments. If the linkname is omitted, the link gets the same name as the last component of the filename and is created in the current directory.

Restrictions **Currently not implemented**!
Only links to files on the same volume are supported. Not all file systems (e.g. DOS FAT and Extended DOS FAT) support links.

Start Hook Enabled with the late start handler START_OSE_HOOK2(initEfsCmds).

ls

Syntax ls [<options>][<dirname> ...]

Description The ls command lists the specified files or the files and subdirectories in that directory where the command is executed. Two option flags are possible.

Options -l Long listing with information such as access modes, ownership, size and modification date.

-a Includes files starting with a ’.’ in the listing.

Combining -l and -a should be done by doing ls -la or ls -al.

Start Hook Enabled with the late start handler START_OSE_HOOK2(initEfsCmds).

Embedded File System

mkdir

Syntax mkdir <directory> ...

Description The mkdir command creates the specified directories. The access permission attributes are set to allow read, write, execute for everybody.

Example mkdir "/ram/dir" "/ram/dir/subdir" "/ram/dir/subdir/subsub"

Restrictions Deep directories must be created one path component at a time.

Start Hook Enabled with the late start handler START_OSE_HOOK2(initEfsCmds).

mount

Syntax mount <volume> <format> <device> [<parameters>]

Description The mount command mounts a device on the format manager creating a volume with the given name. A volume name shall always have a leading /. See the efs_mount() FLIB function for details of the parameters.

Device specific parameters can be specified in the optional <parameter> in the form "name1=value1;name2=value2 .... ". Also boolean parameters can be given, i.e. a name without any "=value".

Example mount "/ram" "extfat" "ramdisk" "unit=0,lo=0,hi=199,fatbuffers=9,dirbuffers=20"

Start Hook Enabled with the late start handler START_OSE_HOOK2(initEfsCmds).

mv

Syntax mv <source> ... <destination>

Description The mv command moves or renames files or directories. The command moves the specified source files to the destination directory. If only one source file is specified and the destination is not an existing directory, the source file is renamed to the destination name. The destination will be removed if it exists, but must not be an open file. A file can be renamed to another path, i.e. moved, as long as it is moved within the same volume, only if the file manager supports this.

When moving/renaming a directory, the destination must not exist (or at least be empty as it will be removed).

Restrictions Files cannot be moved between volumes. Actually some file managers even have the restriction that files cannot be moved between different directories.

Start Hook Enabled with the late start handler START_OSE_HOOK2(initEfsCmds).

pwd

Syntax pwd

Description The pwd command prints the full path of the working directory to standard output. This includes the volume name.

Start Hook Enabled with the late start handler START_OSE_HOOK2(initEfsCmds).

rm

Syntax rm <filename> ...

Description The rm command removes (deletes) the specified files.

Restrictions The file(s) must not be open.

Start Hook Enabled with the late start handler START_OSE_HOOK2(initEfsCmds).

rmdir

Syntax rmdir <directory> ...

Description The rmdir command removes (deletes) the specified directories.

Restrictions The directories to remove must be empty.

Start Hook Enabled with the late start handler START_OSE_HOOK2(initEfsCmds).

sync

Syntax sync <volume>

Description The sync command synchronizes the volume with the physical disk. When the command returns, any cache buffers have been written by the volume manager to the physical disk.

Restrictions Any output buffers within processes, in open streams or file descriptors, are NOT forced out by this command, only the volume manager and device driver buffers.

Start Hook Enabled with the late start handler START_OSE_HOOK2(initEfsCmds).

Embedded File System

unmount

Syntax unmount [-f] <volume>

Description The unmount command unmounts the specified volume. It must be mounted and no files must be open on the volume, unless -f is given.

Options -f Forces unmounting also if files are open on the volume.

Start Hook Enabled with the late start handler START_OSE_HOOK2(initEfsCmds).

vols

Syntax vols [<volume> ...]

Description The vols command reports volume statistics about the specified volumes or all volumes if none specified.

Start Hook Enabled with the late start handler START_OSE_HOOK2(initEfsCmds).

Embedded File System

4 Configuration of EFS

The **OSE Embedded File System** consists of several components. Many of these components are separate processes that should be declared static in **osemain.con** or created dynamically. The function library is linked to the applications and is thus no separate process.

Any shell process is dynamically created by the login process after a successful login. The login process needs open standard file descriptors and thus need to be dynamically started, if it shall be used. Common volumes might be mounted by an initiating process if needed.

osemain.con should declare the process **start_efs**. A standard version of this process in implemented in **startefs.c** in the OSE installation. Here any common initiations can be performed if needed.

Please read the file **efs.txt** in the OSE installation for detailed configuration information.

If you want to use the Shell with the EFS please read the Shell manual for Configuration details.

4.1 Process Types And Priorities

Since file system operations require that a connection to the FSS file system server is established to find EFS resource, the FSS should have as least the same priority as the highest prioritized client.

Processes having great need for fast real-time response (such as disk device drivers and serial device drivers) should have high priorities, maybe in the range of 2-10.

Processes having no such need for fast real-time response (such as format/volume managers and the RAM disk device driver) should have lower priorities, maybe in the range of 15-25.

The login and shell daemon processes are processes that might consume CPU for long periods of time. These processes do not need real-time responses in the millisecond range and it is preferable to make them background processes. The login process should be created dynamically at system start.

Embedded File System

4.2 Static configuration

4.2.1 EFS File System Block

The processes implementing the Embedded File System must execute in supervisor mode and thus must be placed or created in a supervisor block. The **DEFAULT** block in osemain.con is such a block but a special EFS block can also be created.

```
DEF_BLOCK( EFS, 0, DEFAULT, SUPERVISOR_MODE, poolsize, 180,
568, 1568, 8300, 500, 1000, 2000, 4000 )
```

Replace **poolsize** with the amount of pool that is needed by the EFS block. 180, 568, 1568 and 8300 are buffer sizes suitable for use with the extended DOS FAT format manager. 500, 1000, 2000, 4000 are the stack sizes that should match the stack sizes used for the processes in this block.

4.2.2 File System Server - FSS

Since the file system server defines the existence of the file system it is proper to define this process as a static process in **osemain.con**. An FSS must exist in each machine which either implements a file system or shall be able to access it.

```
PRI_PROC( ose_fss, ose_fss, 2000, priority, block, 0, NULL )
```

Replace **priority** with actual priorities and replace **block** with the name of a supervisor block such as **EFS** or **DEFAULT**.

4.2.3 Format Managers - FM

Select one of the two disk format managers, **dosfm** for DOS/FAT compatibility of **extfm** for higher functionality. If a terminal connection (RS232, TELNET) with the system is wanted, also specify the console format manager **confm**. The dosfm will register with the name "fat" in FSS, the extfm uses "extfat" and the confm simply registers as "confm". Declare these processes as static in **osemain.con**:

```
PRI_PROC( ose_dosfm, ose_dosfm, 2000, priority, block, 0, NULL )
PRI_PROC( ose_extfm, ose_extfm, 2000, priority, block, 0, NULL )
PRI_PROC( ose_confm, ose_confm, 2000, priority, block, 0, NULL )
```

Replace **priority** with actual priorities and replace **block** with the name of a block such as **EFS** or **DEFAULT**. To enhance performance, the dosfm and extfm format managers read the client's memory space, i.e. FLIB will use the signals with pointers towards dosfm and extfm where possible. In a system with memory protection (using the MMS) this is only possible if the CPU is executing in supervisor mode. To accomplish this, the dosfm and extfm processes must be placed in a supervisor block (such as DEFAULT).

4.2.4 Block Device Drivers - DDB

If any of the disk format managers (dosfm or extfm) should use the RAM disk then configure the RAM disk device driver. The RAM disk example delivered with EFS will register with the name "ramdisk". Declare this a static process in **osemain.con**:

Embedded File System

```
PRI_PROC(ose_ramdisk, ose_ramdisk, 2000, priority, block, 0, NULL)
```

Replace **priority** with actual priorities and replace **block** with the name of the block the corresponding dosfm or extfm process is placed in.

4.2.5 Shell Commands

To enable the built-in shell commands that manipulates the File System, Processes, they must be initialized in the shell daemon by running a start hook in **osemain.con**. Please read the file **shellcmds.txt** in the OSE installation and Shell manual for detailed configuration information.

File system commands (**cd**, **chmod**, **ls**, **cat** etc.) are enabled by this hook:

```
START_OSE_HOOK2( initEfsCmds )
```

Embedded File System

4.3 Dynamic Configuration

Some EFS configurations must be performed dynamically by a custom-written process. This process should initialize processes, mount devices and start login services. The startefs.c source is provided with EFS as an example, which could be used as a base for a customized process. See the Shell manual for more information regarding Shell configuration.

4.3.1 Mount Devices

4.3.1.1 Mount and Format RAM Disk

Use a format manager to mount a volume manager on the RAM disk. Then format the RAM disk with 8K clusters (16 sectors with 512 bytes each). Below the current directory is set to the volume created and will later be inherited by the login process, and thus any started shell processes.

```
#define DISKNAME "/ram"
if (efs_mount(DISKNAME, "extfat", "ramdisk", "unit=0") != 0)
error(0xDEAD0000 + __LINE__);
if (efs_format(DISKNAME, "clustersize=16", False) != 0)
error(0xDEAD0000 + __LINE__);
if (chdir(DISKNAME) != 0)
error(0xDEAD0000 + __LINE__);
```

4.3.2 Stop Configuration Process

Close our own copies of the configuration process' stdio file descriptors and hibernate forever.

```
stop(current_process());
```

Embedded File System

Index

Embedded File System

V

Embedded File System