# Parsytec File System
## User's Guide

Version 2.1
August 20, 2013

Authors:

| | |
|---|---|
| Adruni Ishan | basic Version (design and implementation) |
| Matthias Clauss | basic Version (design) |
| Hans-Joachim Ermen | checker |
| Christian Fleischer | MSC device driver, `ptar` utility |
| Günter Lauven | multivolume extensions and checker integration |
| Oliver Imbusch | v2.1 bug fixes and minor extensions |

This documentation was written by Oliver Imbusch,
contributions by A Ishan (Version 1.1 Handbook), H-J Ermen (Checker Documentation), N Garnett (Technical Report No 20 of Perihelion Software, "Device Configuration"), G Lauven ("Short description of PFS v2.0"), and C Fleischer ("MSC Device Driver Documentation", "The `ptar` utility").

# Contents

# List of Figures

# Chapter 1

# Introduction

The Parsytec File System (PFS) is a powerful tool to handle mass storage devices in multi-transputer networks. Various applications which need highest data transfer rates can be realized in combination with SCSI devices which can be accessed by Parsytec's "Mass Storage Controller" (MSC) board. Typical fields for applications which can make use of this product are for example image processing and distributed databases.

## 1.1 Main features

- The Parsytec File System (PFS) is based on the Berkeley Fast File System (FFS) up from version 4.2BSD. A lot of extensions have been made to achieve maximal performance and to guarantee an optimal use of the disc capacities.

- It is possible to connect up to seven SCSI devices to one MSC which then can be managed by one PFS. In addition to the standard devices, new SCSI drives can easily be added by editing the driver's `scsiinfo.src` configuration file.

- The PFS offers a concept of multivolume/multipartition. It is for example possible to declare two harddiscs to appear as one volume to the user (multipartition) or one harddisc may be split into several volumes (multivolume). It all depends on the volume and partition description of the user in `devinfo.src`. Devices with changeable media and raw devices are not allowed to be multipartitioned and have to be single volumes.

- The file system checker for structured volumes checks the consistency of data and data structures of a file system. A lot of possible errors are recovered; there are very few situations the checker cannot be successfull. When the checker succeeds, the file system is guaranted to be in a consistent state and the file server can work with it. There are three checking modes: no checks, basic checks and full checks. The checker is noninteractive and is called when a structured volume is loaded (mounted).

- Several mechanisms have been added to handle fatal error situations. Bad blocks of structured devices are reassigned automatically. When buffer cache's checksum is enabled, data are only written on disc if no inconsistence did occur. On error the data is left unchanged on disc. Termination or unloading (unmounting) of volumes is done with respect to active processes working on that volume.

- The basic block size is set to 4 KBytes (whereas the maximum file size is 4 GBytes). In dependancy on the client's request, the server is able to build packets of various sizes in its buffer cache to fit the request in an optimal way. The packet handling is done in a transparent manner to the client tasks.

- Directories and files are placed on the disc or partition by using different strategies to get a good compromise between spreading and clustering of information (minimal fragmentation). To implement such balancing procedures, the file server divides the disc into a number of equal sized "cylinder groups" which are controlled separately.

- In contrast to the BSD Fast File System the inode handling was implemented in a more straight-forward way: inodes and entry names build "Directory Entries" which are kept in special directory blocks. You are able to create as many directories as you want, in one directory up to 448 entries (files or sub-directories) can be combined.

- To achieve maximal flexibility for the user, the server as well as the driver is full parametrizable (via the `devinfo` file, see section 3.3, and the `scsiinfo` file, see section 4.3). With the desired hardware and application environment in mind, it is possible (and necessary) to manipulate the parameters of the devices and the buffer cache in a very wide range.

- Various utilities are added to the software package to ease handling of the server. Especially protection mechanisms based on capabilities are enabled.

- The `ptar` utility helps you to archive lots of MBytes on your streamer tape.

- The PFS is delivered with a configurable device driver. This driver performs data transfer directly from and to the user's buffer, thereby eliminating intermediate copy operations. Requests regarding the same disc device are sorted to reduce head movement. By using the transputer event line, reaction times could be minimised.

## 1.2 The PFS data structures on disc

This section gives you a brief description of important data structures as managed by the Parsytec File System. It will make easier to understand the basic principles of the file system checker (described below) and to follow the different phases of the checking process.

### 1.2.1 Disc layout

The Parsytec File System divides a disc into independant (logical) parts of equal sizes[1], called "cylinder groups". This approach has several advantages: in combination with algorithms for block allocation and creation of new directory entries and files, it can be assured that the storage space on disc is used in the most efficient way: on the one hand, fragmentation of disc space is minimized, on the other hand, one gets optimal medium access times by spreading data in a regular way over the disc.

All information required to manage a cylinder group is collected within one data block, known as an "info block". Every cylinder group has it's own info block and it is placed with a fixed "relative rotational offset". The most important data structure, building the major part of each info block, is a "bitmap"—an array of bytes used to note all free and allocated blocks within the specific cylinder group. Some additional summary information give a short description of the actual usage of a cylinder group. They contain e g the number of free blocks and the number of sub-directories which are kept in a cyclinder group.

Another "special purpose" data block, called "summary block", collects the summary information from all cylinder group info blocks and some additional ones which describe the whole file system with all cyclinder groups as one unit. Using this summary information makes it especially easy and efficient for the file server to decide where to allocate new blocks on disc to expand ane existing file or to create a new sub-directory or file.

There should also be a place on disc where some fixed parameters of the established file system can be found. These are for example items like the number of cylinder groups, the size of each cylinder group and the "rotational offset" for a placement of info blocks within the different cylinder groups. The Parsytec File System puts this information and a lot of other important data in a structure called "superblock". To have enough redundacy, a copy of this superblock is held within each info block. Having formatted a disc once, the content of the superblock structure should never be changed under normal operating conditions.

See figure 1.1 for an overview.

### 1.2.2 Directory entries and files

All relevant information required to describe a sub-directory entry or a file is kept in a data structure called "inode". These are for example items like the creation date, the size (measured in bytes), and a type flag to determine whether the entry describes a sub-directory, a symbolic link, or an ordinary file. Additional information like an access matrix and the number of blocks allocated by this entry and others can also be found within in the corresponding inode.

To keep an inode as small as possible without loosing too much efficiency, only a restricted number of direct block references to data blocks are made ("direct blocks"). If a larger file has to be managed, a "single indirect block" is allocated, which exclusively contains

[1]They all contain the same number of blocks.

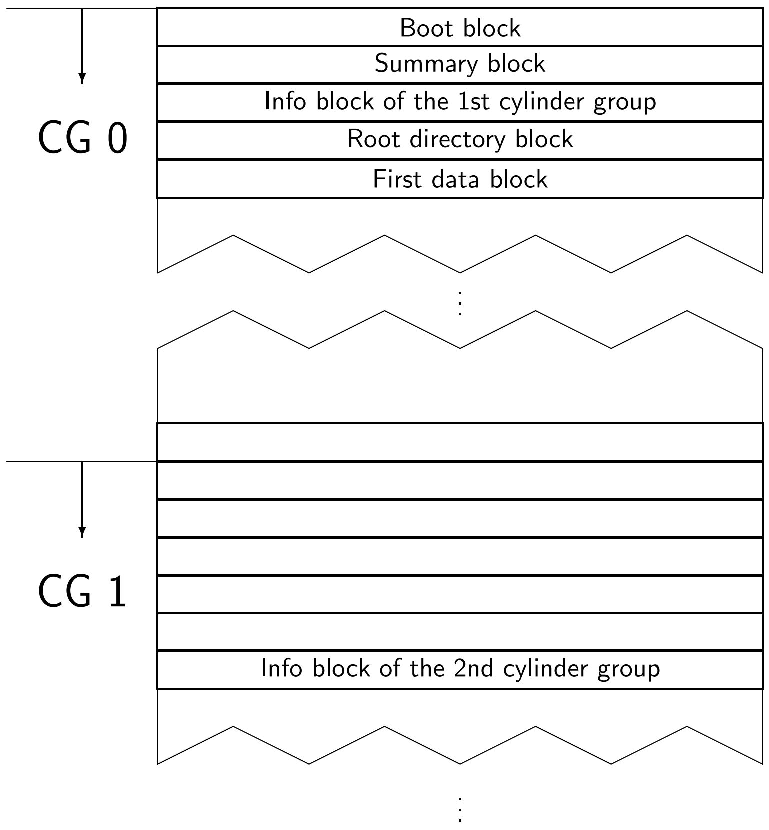
Boot block
Summary block
Info block of the 1st cylinder group
Root directory block
First data block
CG 0
CG 1
Info block of the 2nd cylinder group

Figure 1.1: Disc layout overview

references to data blocks. But one level of indirection is not the limit: very large files of hundreds of Megabytes are described by using a second level of indirection. In this case, a "double indirect block" is allocated and noted in the inode which keeps references to single indirect blocks, which again refer data blocks. This builds the basic mechanism to handle a wide range of file sizes (up to 4 GBytes) without wasting too much disc space for allocating blocks in advance.

There is one type of data block, called "directory block" which is noted by an inode of type "directory" (a sub-directory entry) and represents a special variant of an ordinary data block: a directory block only contains inode data structures and allows the Parsytec File System by this way to build hierarchical directory structures.

## 1.3 Checker

The Parsytec File System Checker is a maintenance tool to guarantee consistency of block based file systems. Under various conditions, a file system kept on a harddisc may come into an inconsistent state. Events like a power loss, a hardware malfunction or other disastrous errors like a head crash may result in damaged data structures on disc. Under certain conditions, it may be even impossible to get initial access to the file system's root directory afterwards.

To recover from such an error and to repair corrupted parts of a file system the Parsytec File System Checker was written. It builds an integral part of the Parsytec File System and is executed every time when a volume is loaded. By this way, it can be assured in general that a file system is definitely in a consistent state after the checker has made all essential tests and corrections.

### 1.3.1 How to use it ...

The Parsytec File System Checker is started directly after the successful boot of a volume. All checks and corrections—if required—are done automatically and there is no need for user interaction. In general, the file system checker works in a "silent" mode, only reporting the different phases of the checking processes which are entered. If an error occurs that needs corrections on disc, the checker reports it by writing a message to the server window.

After finishing the checking process, the start-up sequence of the volume is completed and the file system is accessable by the user in the normal manner[2].

The system administrator has the opportunity to select from two different checking modes: at first, it is possible to perform a **BASIC-CHECK** when a volume is loaded. In this case, all data structures on disc which are used to manage block allocation are tested. If the File System Checker is called with **FULL-CHECK** mode enabled, the complete directory tree is scanned and each entry is tested, regardless whether it is a file, a symbolic link or a sub-directory entry. This is the default checking mode. If some of the BASIC-CHECKs fail, the FULL-CHECK mode is called automatically to detect other possible inconsistencies and to correct them afterwards.

The Parsytec File System can be booted with various options. The checker relevant ones are

| | |
|---|---|
| -f: | perform FULL-CHECKs (default) |
| -b: | perform only BASIC-CHECKs |
| -n: | bypass the checker completely |

There may be one fatal error condition which makes it impossible to run the checker and to make corrections properly: a damaged superblock within the info block of the first cylinder group. In this case, differences between the data structures describing a physical disc and data kept within `devinfo` may be recognized. Run the `mksuper` command to make the file system bootable again[3]. All required information is taken from the actual `devinfo` file and a new superblock structure is filled up. Afterwards, the file server can be booted in the normal manner.

[2]The checking process usually finishes successfully. Only under fatal error conditions like checking a non-formatted disc or working with a damaged disc controller, the File System Checker may fail. In this case, the file system cannot be booted, neither checked in the usual way, until the hardware malfunction or other reasons for the failure have been removed.

[3]An empty file system is created with `makefs`, that destroys all data when executed on a file system that already contains files.

### 1.3.2 The different phases of the checking process

Consistency checking and correction operations done by the Parsytec File System Checker are split into four main phases:

#### 1.3.2.1 Phase I

contains all tests which are executed running in BASIC-CHECK mode:

At first, the validity of the superblock data structure as found in the info block of the first cylinder group is prooved. Afterwards, this superblock is compared with the copies found in the info blocks of the other cylinder groups. The next step is to compare the content of the different cylinder group bitmaps with the summary information kept in the info blocks and in the summary block.

Finally, the root directory inode is inspected to make sure that it is possible to get initial access to the file system's root directory. (The root directory inode is part of the summary block.)

If the Parsytec File System Checker is called in BASIC-CHECK mode and no error was detected, the checking process is finished at this moment. Otherwise, the FULL-CHECK mode is automatically entered and the following three phases are also performed.

#### 1.3.2.2 Phase II

is used to traverse the complete directory tree and to make all essential checks for single directory entries. These test operations build the main part of the checking process when running in FULL-CHECK mode: beginning with the inode of the root directory, the whole directory tree is worked out and each individual entry is tested under various conditions. If certain error limits are exceeded, an entry may be detected as "non-usable" and is deleted afterwards.

#### 1.3.2.3 Phase III

takes the results of the Phases I and II and corrects all block allocation errors which were previously detected:

There are two serious error conditions which are handled during phase III. At first, there may be a block which is referred by more than one entry. The file system checker has to decide which entry may be the legal owner (most recent modifier) of such a block and has to do the required corrections. Secondly, there may be a block which is noted in a bit-map as allocated but not referred by any of the entries touched during phase II. Such a "lost" block may become an entry in the `/lost+found`-directory if it contains directory information (inodes). By this way, it is possible to re-integrate sub-directory structures which were cut-off due to an error condition.

#### 1.3.2.4 Phase IV

performs some general tidyup-operations and adjusts the file system's summary information which are kept in the summary block.

## 1.4 Related manuals

The following manuals may be useful as references:

- MSC Mass Storage Controller Technical Documentation
- Perihelion Technical Report No. 20 Device Configuration
- The Helios Parallel Operating System
- ANSI X2.131-1986 Small Computer System Interface
- Manuals to your SCSI devices

In addition to that, the file `readme.pfs` contains most actual changes not considered in this handbook.

# Chapter 2

# Upgrade

This chapter covers the hardware changes and software upgrade to use PFS v2.1. It has only to be read by former users of an old MSC board, the Helios File System v1.x, and the Parsytec File System v2.0.x. You will find a step by step guideline for a proper update to the PFS v2.1.

## 2.1 Backups

If you follow the instructions in this chapter, you should find a perfectly working PFS afterwards. But:

> If anything can go wrong, it will.

With this in mind, it's urgently recommended to backup your data to your host system before installing the new PFS.

## 2.2 Hardware update

This section has to be read by all users performing an update.

There are several requirements for proper operation of the MSC Device Driver (Jumper and EPROM placement is shown in figure 2.4):

- Xilinx EPROM
  The EPROM for Xilinx initialisation has to be updated to Revision 6 (at least). Only these revisions support event generation as necessary for the driver. The EPROM has to be placed as shown in figure 2.1. Please check which version is installed on your MSC.
- Jumper settings
  Some jumpers have to be adjusted to ensure suitable event generation:

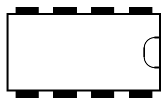

Figure 2.1: Xilinx EPROM

- The ERROR PAL (U52) activates the ERRORIN input of the transputer when the data transfer controller or the extension board requests an event. J0 enables or disables the activation of the ERRORIN input in by the error PAL cause of an error condition: program error, address error or parity error. It has to be placed in 1-2 position as shown in figure 2.2.

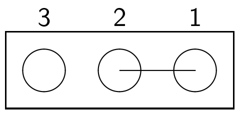

Figure 2.2: Jumper J0

- J9 makes connection between the ERRORIN pin and the EVENTREQ pin of the transputer. So if this jumper is inserted as shown in figure 2.3 an activation of ERRORIN of the ERROR PAL U52 also generates an event.

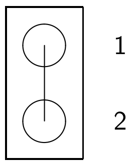

Figure 2.3: Jumper J9

Figure 2.4 shows the exact location of the EPROM and the two jumpers.

If these requirements are not met, the device driver will not start up. Instead it will send a message to the server window, complaining about missing events from the hardware.

It's not too difficult to carry out the necessary changes on your own, but if you are thumb-fingered or want to prevent loss of warranty of your MSC board, please send the board to Parsytec. We will perform the update (including tests) for you.

## 2.3 Software upgrade

The general installation procedure is described in chapter 3. Refer to that and install the PFS before you proceed. The next sections go into detail about changes of your `devinfo.src`. In addition to that, the declaration of your MSC in your resource map has changed. For more details see section 3.4.

Figure 2.4: MSC board

### 2.3.1 `devinfo` file, coming from PFS v2.0.x

This section is only for users who have worked with PFS v2.0.x up to now. It is assumed that you have worked through the device configuration example in the PFS v2.0 manual. Therefore, only the differences to the original `bible.src` file are outlined here.

For each drive, the *type* and *id* specifications have to be changed. The *type* entry now refers to the position of the device declaration in the `scsiinfo.src` file (which is the configuration file for the MSC device driver, see chapter 4), and *id* values are now given as hexadecimal numbers to allow the specification of different Logical Unit Numbers within a single device (0x31 refers to SCSI address 3, LUN 1).

The entry for the Wren VI drive will now look as follows:

```
drive        # Wren VI
{
  id   0x30  # SCSI address 3, LUN 0
  type 0     # first entry in scsiinfo
}
```

The description of partitions has not been changed, but the *controller* specification of the *discdevice* block. It's now given in hexadecimal notation, too. Thus, your *discdevice* entry for the MSC should be:

```
discdevice msc21          # MSC driver for PFS v2.1
{
  name        msc21.dev   # refers to /helios/lib/msc21.dev
  controller  0x70        # uses Address 7, LUN 0
  addressing  1024        # use 1 KByte blocks

  partition               # number 0
  {
    :
  }

  :
  :

  drive                   # number 0
  {
    :
  }
}                         # end of discdevice
```

After modifying the `devinfo.src` file, compile it to binary form with

```
% gdi devinfo.src /helios/etc/devinfo
```

### 2.3.2 `devinfo` file, coming from HFS v1.x

This section is only for users who have worked with HFS v1.x up to now.

**CAUTION:**
This is a mini example for those users who work with only one MSC connected to one harddisc.

1. Create a new file `upgrade.src` in the directory `/helios/etc` (based on `/helios/info.src/hfs2pfs.src`):

```
fileserver msc21
{
  device       msc21
  blocksize    4096

  smallpkt     1
  mediumpkt    4
  hugepkt      16

  smallcount   20
  mediumcount  8
  hugecount    28

  volume
  {
    name      <Old>  # The file server doesn't care about your
                     # volume's name and you could change it any
                     # time, but with respect to consistency of
                     # shell scripts, makefiles, and user programs
                     # it's recommended to use the old name
    partition 0
    cgsize    <Old>  # This value MUST be taken from the old
                     # devinfo.src, as well as
    ncg       <Old>  # this one.
  }
}

discdevice msc21
{
  name        msc21.dev
  controller  0x70
  addressing  1024

  partition
  {
    drive 0
  }

  drive
  {
    id    0x00  # Usually, only one harddisc is connected to the
                # MSC and occupies SCSI address 0 (1st nibble),
                # Logical Unit Number 0 (second nibble).
    type  0     # This refers to a device description number in
                # the scsiinfo.src file. Probably you have a WREN
                # harddisc, it's description is at position number 0.
  }
}
```

2. Translation:

```
% gdi /helios/etc/upgrade.src /helios/etc/upgrade.di
```

3. Starting the file server, loading the only volume (prepared for `mksuper`):

```
% remote -d MSC fs upgrade.di msc21
% load -m /<VolumeName>
```

4. Now, the file server tries to find out the correct disc parameters, following this description (decreasing priority):

   (a) Reading the superblock from the harddisc
   The HFS has stored the values for *cgsize* and *ncg* in this block.

   (b) Evaluating the configuration file,
   which is `/helios/etc/upgrade.di` in this case. If the devinfo-parameter was not given at file server's startup time, `/helios/etc/devinfo` is used by default.

   **CAUTION:**
   If the superblock is damaged, the priority is given to the values in the configuration file. For that reason, *cgsize* and *ncg* in `upgrade.src` **must** be given the same values as in your old `devinfo.src`.

5. Write parameters to disc
Now, the correct PFS parameters have to be written to the superblock by executing:

```
% mksuper /<VolumeName>
```

In some cases there may be problems: you have discovered the values for *cgsize* and *ncg* and have written them down to `devinfo`, but the `mksuper` command couldn't be executed successfully. This behaviour occurs if the product of *ncg* and *cgsize* exceeds the size of your harddisc. This error was not checked by the HFS and the wrong parameters were written to disc. To solve this problem, decrease *ncg* in `upgrade.src`, compile it, and restart the file server until `mksuper` is performed succesfully. **By doing this, some data may get lost, of course.** So **do** backup your file system now at the latest (use your old `devinfo`) to rescue as many data as possible.

# Chapter 3

# Installation

## 3.1 Requirements

### 3.1.1 Hardware

The file server makes use of Parsytec's MSC-board which offers at least 4 MBytes RAM and a T805 Transputer for processing. The PFS will run on this processor and builds up its buffer cache structures by making use of the MSC's memory. A suitable SCSI-device with a capacity of up to 2 GBytes can be added. If you have used HFS v1.x of PFS v2.0.x up to now, see section 2.2 for necessary hardware updates.

**NOTE:**
If you have used an old version of the MSC board up to now, please refer section 2.2 for an update of your hardware.

Figure 3.1 demonstrates how the PFS is integrated into a system which makes use of the MSC and a SCSI-device.

### 3.1.2 Software

To work with the PFS properly, the "Helios distributed operating system" is required. The software is able to run with Helios version 1.2 and upper.

## 3.2 Copying files

The Parsytec File System is shipped on one 5,25" (HD) disc in IBM-PC compatible format respectively on a quarter inch streamer tape. Your disc/tape is expected to contain the files listed in `contents.pfs`. Copy all files from the subdirectories of the distribution disc to the associated directories of the host's file system. If you use a PC, this can be done for example by

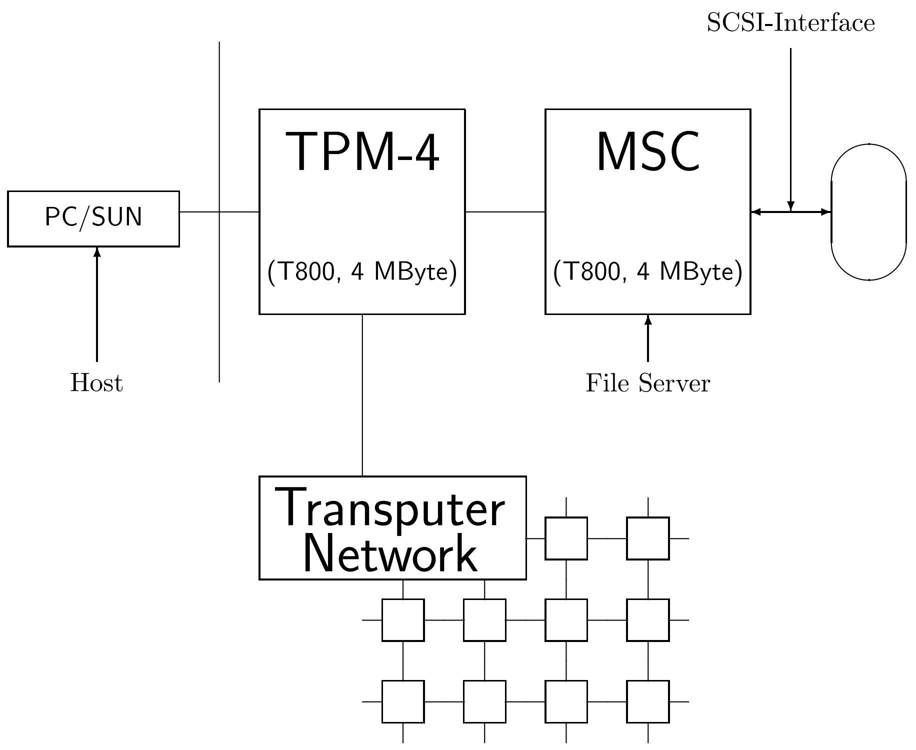

Figure 3.1: Sample network

```
>xcopy a:  c:\helios /s
```
(under MS-DOS)

or

```
% cp -r /a/* /helios
```
(under Helios)

SUN users perform

```
cd /helios
tar xvf /dev/rst0
```

## 3.3 Device configuration

This section describes how to establish your PFS. For that reason we will create the file `/helios/etc/devinfo.src` and compile it via the `gdi`-command to `/helios/etc/devinfo`. The file server will use this file to get technical information about your hardware and about the logical structure you want to give to your file system.

**NOTE:**

- In addition to `devinfo`, `scsiinfo` contains information about the connected SCSI devices. Refer chapter 4 if you want to add SCSI devices of a not yet supported type.

- If you have used HFS v1.x or PFS v2.0.x before, see chapter 2 and execute the changes do your `devinfo.src` first.

### 3.3.1 Conditions

There are three different kinds of information you have to write down in your `/helios/etc/devinfo.src` file (henceforth called `devinfo` to save space):

- technical facts concerning your hardware (e g a Winchester's SCSI-address). For these you should be armed with a guide to your devices.
- strategical information about the method you want to manage your file system (e g the size and apportionment of your buffer cache). For these you should consult the file `bible.src` (introduced below) to get some convenient standard values.
- arbitrary but necessary information about your taste to handle your volumes. For this you should have (if you prefer to name your volumes with your friends' names) at least five friends if you want to establish five volumes, for example.

**NOTE:**

The Parsytec File System, once configured, allows you to forget detailed information about the SCSI drives connected to the MSC board. You only work with some volumes (data pools with a name), whereat there can be some logical volumes on one hardware drive, or some partitions of different drives can be combined to one volume. Since `devinfo` is the essential way to configure your file server, you should not do a 'quick and dirty'-hack. As well as some values cannot be changed without loss of data, you will pay for every minute you save creating the `devinfo` with much more time in your daily work, because inconsiderate partition of your buffer cache can slow down data transfer significantly, for example.

### 3.3.2 General structure

The `devinfo` contains a sequence of entries consisting of a keyword, a name, and a description. The keywords currently supported are *fileserver* and *discdevice*. The name is used to identify the entry and has no other meaning. The description is enclosed in braces ('`{`' and '`}`') and consists of a sequence of keyword/value pairs. The character '`#`' introduces a comment which extends to the end of the line. Keywords expect to be given a value of a certain type, this may be either a name, a file name, a number or a description. Names consist of a leading alphabetic character followed by alphanumerics or periods. File names are a sequence of names separated by slashes ('`/`'). Numbers are sequences of numerals preceded by an optional minus sign. By default, numbers are interpreted in decimal, but the standard C-syntax may be used to provide values in octal or hexadecimal.

### 3.3.3 The developer's box - an example

To make you familiar with creating your own `devinfo` we will watch step by step the formation of the `devinfo` that was used by the programmer of the file server to test his program.

**NOTE:**
It's easier to describe the configuration buttom-up, although the notation in our `devinfo` is top-down.

### 3.3.4 The hardware resources

#### 3.3.4.1 Description of the equipment

On our desktop we see a big box containing

| | |
|---|---|
| CDC WREN VI | a 600 MB Winchester drive, |
| Exabyte EXB-8200 | an 8mm Cartrige Tape Subsystem, |
| Sony SMO-D501 | a rewritable 5.25" optical disc drive, connected to a SMO-C501 SCSI controller, |
| Rodime 40 | a 40 MB Winchester drive, |
| Tandberg TDC 3600 | a conventional tape streamer, |

all SCSI devices. First we want to tell our file server about these five hardware drives. We do this by using the *drive* keyword, followed by information about the SCSI address (*id*) and a *type* identification, both together enclosed in braces. The value that is given to the *type* keyword refers to entries in the `scsiinfo` file, which is described more detailed in section 4.3.

**NOTE:**
Similar to `devinfo`, `scsiinfo` can be changed by the user to make it possible to use non-standard hardware in nexus with the PFS, whereas 'non-standard' means SCSI devices 'not delivered by Parsytec'. All drives that you purchase at Parsytec are supported by the standard `scsiinfo` (which is updated in case of enlargement of our hardware assortment). In the following all *type* values refer to the original `scsiinfo.src` and have to be corrected if that file is changed.

Let's start creating `devinfo` with writing down the drive list (that consists of five *drive* blocks which are numbered by the file server automatically):

```
# Remember, a #-character introduces a comment which ends here ->
# The first drive is the Rodime 40MB. It will be referenced by number 0 (in
# our drive list) by the file server. (NOTE: the sequence of the drives
# stipulates their numbering)

drive       # the drive keyword indicates the start of some hardware device
```

```
            # information
{           # the opening brace
  id   0x00 # this drive occupies SCSI address 0 (represented by the first
            # hexadecimal digit), and Logical Unit Number 0 (specified by
            # by the latter hexdigit)
  type 1    # this drive is of type 1 (which means Rodime drive when working
            # with standard scsiinfo)
}           # the closing brace

# The second drive is the WREN VI. Since its description follows drive 0's
# specification, it will get number 1 in the drive list

drive       # hey file server, here's an info about my hardware
{           # from here
  id   0x10 #   SCSI address 1, LUN 0
  type 0    #   getting information from device block #0 in scsiinfo, which
            #   represents a WREN VI device
}           # to here

# Optical disc drive, number 2 in drive list

drive
{
  id   0x30 # SCSI address 3, LUN 0
  type 2    # optical disc
}

# Number 3:

drive{id 0x40 type 3}#this version to declare an Exabyte streamer works too,
#of course, but doesn't satisfy our esthetic necessaries

# Number 4:

drive         # so let's return to this pretty styled version to explain the
{             # fifth drive, the Tandberg tape streamer
  id    0x50  # at SCSI address 5, LUN 0
  type  4     # described in scsiinfo at position #4
}
```

**NOTE:**

If you remove one of your drives and delete its description, all following *drive* blocks will get a new number automatically. This has consequences in the *partition* block (see below).

Figure 3.2 reflects the situation after writing down all drive descriptions.

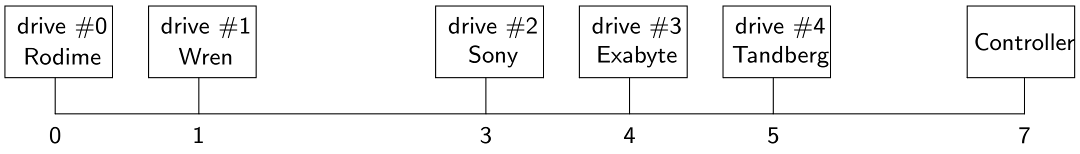

Figure 3.2: Device connection to the SCSI bus

#### 3.3.4.2 From drives to partitions

Now, the file server knows the SCSI address, Logical Unit Number, and the drive *type* of every SCSI device in our system. The second thing we want to do is to install a low level logical structure on all the drives. This structure is a division in subsections, called partitions. A *partition* (use this keyword to unite the following information) is defined by the drive number where it takes place, the first and the last block (the corresponding keywords are *drive*, *start* and *end*). Afterwards, partitions can be combined to larger structure, called volumes. These volumes are the data pools the user works with.

**NOTE:**

- The *drive* keyword must be given a value.
- The *start* and *end* keywords are only allowed if the medium that keeps the partition is structured (can be random accessed) and is unremoveable (the medium cannot be changed). In other terms: you need a Winchester drive to use *start* and *end* (which is given the last block by default).
- The values of *start* and *end* are interpreted in *addressing* size. This keyword is described below.
- As the *drive* list, the *partition* list will be numbered by the file server automatically - with the same consequences when deleting one *partition* block (following partitons' number will decrease by one, so that an eror occurs in a *volume* definition block, see below).

The partitions:

```
# The first partition occupies the first 20,000 blocks of drive 0 (that
# was the Rodime 40MB).

partition      # number 0 in partition list
{
  drive 0      # this partition takes place on drive 0,
  start 0      # starts at block 0 and
  end   19999  # ends at block 19999.
               # NOTE:
               #   The size of a block is defined by yourself. You can do
               #   that by giving a value to the keyword addressing as
               #   described below. So, if addressing equals 1024, this
               #   partition (number 0) takes the first 20MB on drive 0.
}

partition      # number 1 in partition list
{
  drive 0      # the same drive as partition 0
  start 20000  # starts beyond partition 0 (that ended at block 19,999)
               # Since we want to get the rest of drive 0, the end keyword is
               # obsolete.
}

# Now we start to devide drive 1 (the WREN VI) in different partitions:
# the first with 20,000 blocks (0 - 19,999),

partition      # number 2 in list
{
  drive 1
  start 0
  end   19999
}

# the two following with each 100,000 blocks

partition      # number 3
{
  drive 1
  start 20000
  end   119999
}

partition      # number 4
{
  drive 1
  start 120000
  end   219999
}

# and the 6th that takes the rest of drive 1.

partition      # number 5
{
  drive 1
  start 220000
}

# The Tandberg streamer occupies a complete partition

partition     # number 6
{
  drive 4     # a streamer is changable and not structured, so we have to give
              # up to divide it in smaller partitions
}
```

```
# as well as the optical disc

partition     # number 7
{
  drive 2     # an optical disc is changable, so the start and end keywords
              # are not allowed
}

# and the Exabyte streamer.

partition     # number 8
{
  drive 3
}
```

#### 3.3.4.3 A complete discdevice

Enclose the nine *partition* blocks and the four *drive* blocks together in braces and place the keyword *discdevice*, followed by a name for the group of drive resources, in front of that block. There are three specifications left to add in the *discdevice* block to complete the hardware declaration: the *name* of the file containing the device drivers (placed in the `/helios/lib` directory), the address of the SCSI *controller* and the addressing size for the devices in this group (following the keyword *addressing*, interpreted in byte). Let's have a look at our complete *discdevice* block:

```
discdevice msc21        # Mass Storage Controller, driver version 2.1
{
  name        msc21.dev # FIXED and expanded to /helios/lib/msc21.dev
  controller  0x70      # the controller's SCSI address and LUN
  addressing  1024      # FIXED: we want all disc devices to handle with an
                        # addressing size of one KByte
  partition             # number 0
  {
    :
  }

    :                    see above
    :

  drive                # number 0
  {
    :
  }

    :                    see above
    :
}                      # end of discdevice msc21 block
```

See figure 3.3 for the effects.

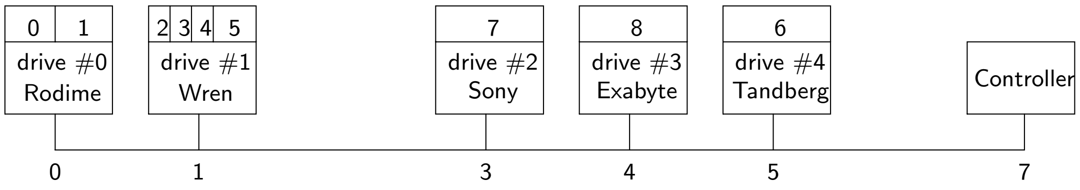

Figure 3.3: Partitioning of the drives

### 3.3.5 Piling up the file server

#### 3.3.5.1 From partitions to volumes

At this point we want to attach our nine partitions to seven volumes, whereby two volumes consist of each two partitions and the other volumes claim only one. Since the names of my friends are not to you, the seven volume names will be provided by the bible. In addition to the *name* of a *volume* you must enter at least two keywords and their values: *partition* and *type*. The simpliest volume blocks look like these:

```
volume           # number 0
{
  name      pride
  partition 8     # the complete Exabyte streamer
  type      raw   # in opposite to partitions on random accessable drives like
                  # optical disc drives or Winchesters which are of type
                  # structured
}
```

```
volume           # number 1
{
  name      anger
  partition 6     # This is the Tandberg device
  type      raw
}
```

The next volume makes use of the *minfree* keyword. This causes your file server to leave some blocks free for the `/lost+found` directory that will be created by the checker.

```
volume                   # number 2
{
  name      covetousness
  partition 7             # optical disc drive (complete)
  minfree   100           # all but 100 blocks are used by the file server
                          # NOTE:
                          #   The value of minfree is interpreted in the
                          #   fileserver's blocksize (see below) and NOT in
                          #   discdevice's addressing size.
  type      structured
}
```

Let's try to construct our first Winchester volume now using two more optional keywords, *cgsize* and *ncg*. The blocks of a partition are combined to larger structures called cylinder groups. You are able to choose the number of blocks per cylinder group (via *cgsize*) and/or the number of cylinder groups per *partition* (via *ncg*). The remaining value(s) is/are calculated by the file server.

**NOTE:**

- There are maxima for both, the *cgsize* and the *ncg* values (see below).
- It's no error if the product of *cgsize* and *ncg* is smaller than the number of blocks your partitions offer. The remaining blocks are **not** used.

```
volume                  # number 3
{
  name      lust
  partition 4           # in the middle of WREN
  cgsize    256         # One cylinder group consists of 256 blocks.
                        # NOTE:
                        #   -Once more: the size of THIS blocks are fixed by
                        #    the blocksize keyword.
                        #   -The ncg keyword was not used, its value will be
                        #    calculated.
                        #   -The maximum for cgsize is 3072.
  minfree   100
  type      structured
}
```

Ok, well done. Immediately the next one (using both cylinder group parameters):

```
volume                  # number 4
{
  name      gluttony
  partition 1           # second part of the Rodime
  cgsize    256         # assuming that the blocksize is 4096 bytes, a
                        # cylinder group occupies 1024 KBytes,
  ncg       20        # so volume 4 claims 20 MBytes
                      # NOTE:
                      #   -Partition 1 must offer that 20 MBytes, of course,
                      #    but here it does
                      #   -The maximum for ncg is 300.
  minfree   100
  type      structured
}
```

The two-partitions-on-one-volume-version:

```
volume                # number 5
{
  name      envy
  partition 3         # second part of the WREN
  partition 5         # fourth part of the WREN
                      # neither the ncg nor the cgsize keyword is used, the
                      # file server has to do the job
  minfree   100
  type      structured
}
```

Last but not least: two partitions on two drives combined to one volume:

```
volume                # number 6
{
  name      sloth
  partition 0         # Rodime's first
  partition 2         # WREN's first
  ncg       40        # We want the file server to install 40 cylinder
                      # groups.
                      # NOTE:
                      #   -The cgsize keyword was not used, its value will
                      #    be calculated.
  minfree   100
  type      structured
}
```

The volume structure is shown in figure 3.4—I hope we survive.

#### 3.3.5.2 Complete?

Almost. Two kind of information are left to write down: the type of *discdevice* you want to handle with your file server (following the keyword *device*) and some information about your buffer cache's apportionment. There are three different kinds of packages, and it's your's to determine their number. Here's the complete *fileserver* block:

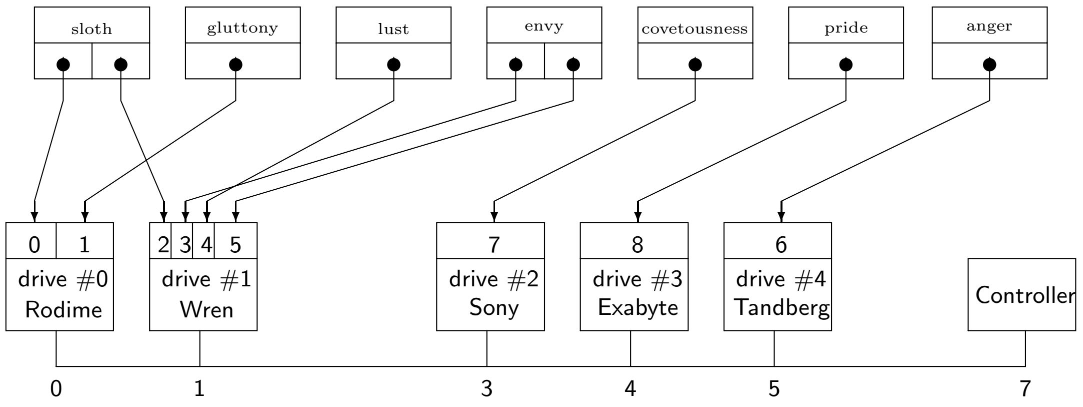

Figure 3.4: Combining partitions to volumes

```
fileserver msc21
{
  device      msc21 # use the discdevice msc21 block (defined above)
  blocksize   4096  # FIXED to 4 KBytes
  syncop      0     # 0 turns the synchronous mode OFF (1 turns ON).
                    # In Synchrous mode all data are written on disc
                    # immediately, else the sync process does that job every
                    # 20 seconds.
                    # NOTE:
                    #   The syncop switch concerns user data. System data
                    #   (like directory information) are always written at
                    #   once.
  smallpkt    1     # FIXED to 1  -- blocks per small cache packet
  mediumpkt   4     # FIXED to 4  -- blocks per medium cache packet
  hugepkt     16    # FIXED to 16 -- blocks per huge cache packet
  smallcount  20    # number of small  cache packets (1 <= smallcount  <= 30)
  mediumcount 8     # number of medium cache packets (1 <= mediumcount <= 15)
  hugecount   28    # number of huge   cache packets (1 <= hugecount   <= 42)
                    # NOTE:
                    #   The number of blocks that are reserved for the
                    #   buffer cache must be <= 750.
  volume            # number 0
  {
    :
  }
    :                 see above
    :

}
```

## 3.4 Running the multi-volume file server

The following example checklist should help you to setup a new file system and work with it. For this example we use a network with a node named “MSC” which offers a T805 and at least 4 MBytes RAM to keep the file server with all data structures. We assume that you've already copied the files from the distribution disc as described at the beginning of this chapter.

**NOTE:**

There are two ways to start the Parsytec File System. Note that the the MSC node must be of attribute HELIOS (declared in your resource map) on both cases.

1. To get the first experience, it's recommended to start it via `remote` from your shell. This allows the execution of the file server under the full control of the user.
2. Later, if everything is working well, the file server is capable of getting started in the general Helios startup sequence. The shell script `/helios/etc/startrc` is looking for `pfsbook` and `pfsrc` (both placed in the `/helios/etc` directory) and excutes them. `pfsbook` might look like

   ```
   domain get /MSC
   domain book /MSC
   ```

   whereas `pfsrc` could be

   ```
   test -d MSC
   if (0 == $status)
       remote -d MSC fs msc21
   else
       echo "No processor MSC to start PFS."
   endif
   ```

   Creating this two files will guarantee that the file server is up and running when the user login prompt appears.

### 3.4.1 Startup

Boot the Helios network as usual

```
> server
```

(on PC systems)

or

```
% helios
```

(on SUN systems)

After logging in, you can prepare an empty file system based on the information which is kept in `devinfo`. To change some of the file system parameters, you have to edit `devinfo.src` as described above and recompile it with the `gdi`-command.

```
% pushd /helios/etc
% emacs devinfo.src

     <editing>

% gdi devinfo.src devinfo
% popd
```

If the `devinfo` is successfully compiled, we start the file server on our MSC-board via

```
% remote -d MSC fs -f msc21
```

Now the server is active and all volumes specified in the `devinfo` are waiting for being loaded. When using a volume the first time, an empty file system must be created. We edit `ldvol1st`

```
% emacs ldvol1st
```

```
load -m $1
makefs $1
```

and start it for every structured volume:

```
% ldvol1st /covetousness
% ldvol1st /lust
% ldvol1st /gluttony
% ldvol1st /envy
% ldvol1st /sloth
```

**NOTE:**

Do not run `ldvol1st` if your volume contains any data. To ease daily work we create the `loadall` shell script

```
% emacs loadall
```

```
load /pride
load /covetousness
load /lust
load /anger
load /gluttony
load /envy
load /pride
```

and start its execution in every future session.

Now, our network could look like this:

```
/Cluster/MSC/pride/...
            /covetousness/...
            /lust/...
            /anger/...
```

```
/gluttony/...
/envy/...
/sloth/...
/tasks/loader
       /procman
       /fs
⋮
```

All volumes are started, you are now able to run our copy-demo (see chapter 7), format an optical disc or do what you are payed for. If you like to format a disc, create a script like

```
% emacs formtopt
```

```
unload $1
load -m $1
format $1
makefs $1
```

**NOTE:**
Formatting an optical disc takes approximately 45 minutes. So be aware before typing

```
% formtopt /covetousness
```

The last shell script we offer to you leads to terminating all your volumes and thereby the complete file server.

```
% emacs termall
```

```
termvol /pride
termvol /covetousness
termvol /lust
termvol /anger
termvol /gluttony
termvol /envy
termvol /pride
```

If you like to sin again, restart the file server.

# Chapter 4

# MSC device driver

This chapter gives you a short introduction to the SCSI bus, the internal details of the MSC device driver and explains how to add new device descriptions in the `scsiinfo` file.

## 4.1 The SCSI bus

The SCSI (Small Computer System Interface) as defined in the ANSI X3.131-1986 standard is a widely used interface between host computers and peripheral devices, such as disc drives, tape streamers and printers. It supports data rates up to 5 megabytes per second and provides some device independence for host computers.

The Interface protocol differs between two types of participants: initiators (devices capable of initiating an operation) and targets (devices capable of responding to a request to perform an operation). Up to 8 devices of different type may be connected to a single bus system, as the architecture of SCSI includes a priority-based distributed arbitration scheme.

The Parsytec MSC board uses the single-ended version of the SCSI bus, thereby allowing maximum cable lengths of 6.0 meters. The onboard WD 33C93A SCSI controller satisfies the ANSI X3T9.2 specifications, with the exception of device disconnection/reconnection.

## 4.2 Internal details

This section gives a deeper view into the MSC device driver. It examines the inner structure of the driver, describes the usage of SCSI commands and explains how the Helios requests are performed.

### 4.2.1 Device driver internals

A Helios device is a piece of software which contains code to access a specific hardware device. It is loaded dynamically by the program which uses it, typically a server, which then uses the hardware through a defined interface and provides a GSP interface for it.

This MSC device driver is written to support the Parsytec File System in connection with the MSC Mass Storage Controller board. It is highly optimised for throughput and flexibility on the SCSI interface.

### 4.2.2 Process structure

The `OpenDevice()` call loads the driver and calls its Open function. The process structure after successful completion is shown in figure 4.1.

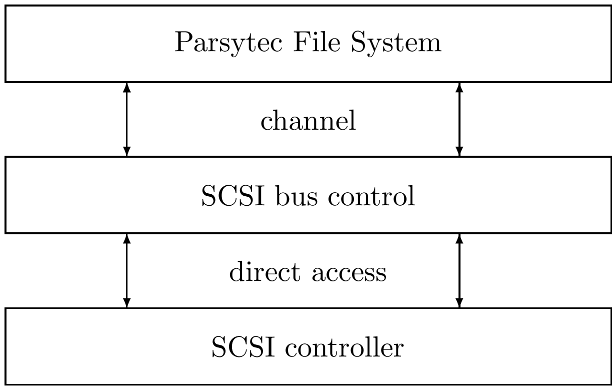

Figure 4.1: Process structure after `OpenDevice()`

The driver forks a SCSI bus control process, which is responsible for interaction with the controller hardware. This process runs at high priority to reduce reaction times. Communication to this process uses hardware channels to allow the use of the occam-style Alternate construct. Opening a SCSI device generates a new device control process, which knows about commands and sequences and handles single Helios requests. The `DevOperate()` function puts all forthcoming requests from the file system into request queues, from which the device control process extracts them as they are handled individually. For random access devices, the device control process uses a special algorithm to optimize head movement and to reduce seek delays. Figure 4.2 shows the process structure after openening two SCSI devices.

### 4.2.3 Supported Helios requests

Due to the demands of the Parsytec File System, the driver supports 8 different device requests. Originally, these requests refer to partitions, but they are mapped onto physical drives within the device driver. Valid requests are:

- `FG_Open`
  Open a SCSI device for further use. This request is implemented by executing a sequence of SCSI commands called the initialisation sequence, which has to be defined in `scsiinfo`. If none of the commands fails and all necessary drive parameters have been evaluated, the device control process is forked off. Opening an already

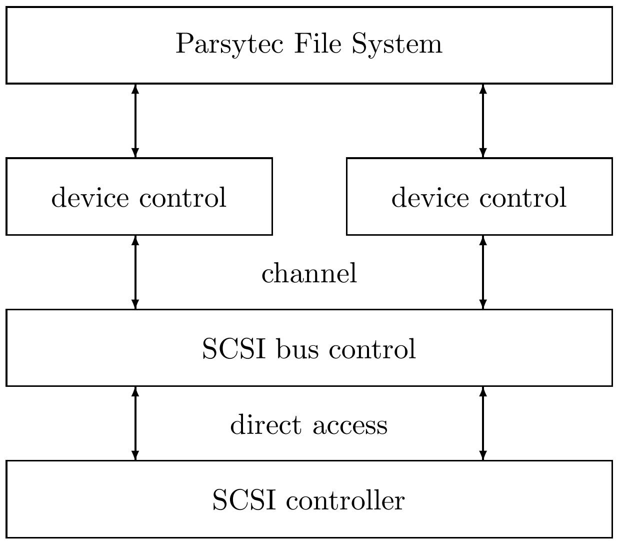

Figure 4.2: Process structure after opening two SCSI devices

opened device will only succeed if the device is a fixed disc device, otherwise the request is deemed erroneous.

- `FG_Close`
  Release an opened device and terminate its device control process. If the device has been opened more than once, the device control process stays alive until all users have closed the device.

- `FG_Read`
  Read some data from the SCSI device. This request is directly mapped onto the Read SCSI command.

- `FG_Write`
  Write some data to the SCSI device. Like Read, this request maps onto the Write SCSI command.

- `FG_GetSize`
  Report the block size and the number of blocks for a partition. Using the information from the initialisation sequence, this request can be fulfilled without interaction with the SCSI device.

- `FG_Format`
  Format a medium and prepare it for further usage. This request is implemented in several phases: First, the format sequence is executed, then the driver issues the Format SCSI command. After completion, the whole medium is verified and defective blocks are reassigned. For tape drives, the Format request may be used to erase a tape.

- `FG_WriteMark`
  Write filemarks to a tape using the Write Filemarks SCSI command.

- `FG_Seek`
  Position a tape at the specified position. This request performs the Rewind SCSI command if both the position and size values of the request are zero. Otherwise, a Space SCSI command will be issued using the lowest 2 bits of the position value as the command code. The size value (positive or negative) specifies the range of medium movement.

### 4.2.4 Error handling

If a SCSI command returns a CHECK CONDITION status, the device driver issues a Request Sense command and evaluates an error code from the replied sense data. The class of this error code is interpreted in the following way:

- `EC_Recover`
  The driver retries the command up to five times before reporting the error code. For Read or Write SCSI commands, `EC_Recover` codes are suppressed to allow retries upon corrected ECC errors.
- `EC_Warn`
  If a Read or Write SCSI command on a disc device returns an `EC_Warn` code, the driver tries to detect the number of the block which caused the problem. If possible, the defective block will be reassigned. In case of success, the driver returns a 'Recoverable, broken Medium' error code.
- `EC_Error`, `EC_Fatal`
  Error codes belonging to these classes cause the driver to abort the request immediately.

## 4.3 Adding new SCSI devices

This section gives a guideline to add new devices to the `scsiinfo.src` file. It explains the principles of driver operation and presents tools for device inspection and verification of a modified `scsiinfo` file. Imagine you have got your brand new DICKENS Pickwick-1836 drive and you plan to use it with the Parsytec File System.

1. You have to find out how the drive acts on the SCSI bus and how the different commands work.
2. You should add a new entry to your `scsiinfo.src` file to reflect the new drive.
3. You should check whether the new drive can work together properly with the other drives on the SCSI bus.

### 4.3.1 Required documents

For testing the new drive, you should have the Reference Manual or User's Manual at hand (the manual which defines the SCSI command subset supported by the drive).

### 4.3.2 Testing single SCSI commands

To explore the behaviour of a new SCSI drive, you might want to issue single SCSI commands and have a look at the drive's respose. This can be achieved using the `testdrv` utility, which lets you define the low-level parameters and data values for a SCSI request and shows the returned data.

**NOTE:**

In most cases it's not necessary to test each SCSI command that's defined for a new drive. Usually you first create the new device description and try to run the `testinfo` utility. Only if that fails, go step by step through the requests using the `testdrv` utility.

It's presumed that you have connected the new device to the SCSI bus, have adjusted an unequivocal SCSI address via jumpers or dip switches and have installed/removed the SCSI terminators.

**CAUTION:**

The `testdrv` utility does **not** preserve any data previously contained in the tested drives. Prove that you have a complete set of backups for all tested drives.

Start `testdrv` on your MSC board via

```
% remote MSC testdrv <DiscDeviceName>
```

The screen changes to

```
Current Request parameters:

 1)      Target ID   : 0
 2)      Target LUN  : 0
 3)      Sector size : 0
 4)      Read        : 0
 5)      Blockmove   : 0
 6)      Timeout     : 0 sec
 7)      CDB size    : 0
 8)      CDB         : 00 00 00 00 00 00 00 00 00 00 00 00
 9)      Size        : 0
         Rest        : 0
         Status      : 0x0
 A)      Data        : 00 00 00 00 ...

 1..A: edit field  R: send request Q: quit
```

and you have the choice wether to construct a request by editing the different fields, to perform the request or to leave the program. Let's send the first request to a new SCSI device. Type '1' to be asked for the

```
New Target ID :
```

Type in the SCSI address of your new device. Change the values of Target LUN, Sector size, Read and Timeout, and your screen is expected to look like

```
Current Request parameters:

 1)      Target ID   : 1
 2)      Target LUN  : 0
 3)      Sector size : 1024
 4)      Read        : 1
 5)      Blockmove   : 0
 6)      Timeout     : 5 sec
 7)      CDB size    : 0
 8)      CDB         : 00 00 00 00 00 00 00 00 00 00 00 00
 9)      Size        : 0
         Rest        : 0
         Status      : 0x0
 A)      Data        : 00 00 00 00 ...

 1..A: edit field  R: send request Q: quit
```

To be sure to work with the correct drive and not to destroy any useful data, it's recommended to perform the Inquiry SCSI command first. On this account we consult the "Guide to the famous Pickwick drive", and on page 69 we find the entry shown in figure 4.3.

| | |
|---|---|
| Command: | INQUIRY |
| Group: | 0 |
| OP-code: | 12H |
| Command descriptor block: | |

| Bit / Byte | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 01 | Logical Unit Number | | | Reserved | | | | |
| 02 | Reserved | | | | | | | |
| 03 | Reserved | | | | | | | |
| 04 | Allocation Length | | | | | | | |
| 05 | X | X | Reserved | | | | Flag | Link |

Figure 4.3: "Guide to the famous Pickwick drive", page 69

That's all the information we need to give the correct values to CDB size, CDB and size:

- CDB size
  INQUIRY is a group 0 command, thus it's command descriptor block size is 6.
- CDB
  - The first byte of the CDB contains the opeartion code, which is 12H in our example.
  - The three most significant bits of the second byte have to contain the LUN (which is zero here).
  - The remaining bits of byte #01 are reserved and must be set to zero, as well as the bytes #02 and #03.
  - The allocation length has to be fixed in byte #04. Since we want to make the drive to transfer the complete response to the INQUIRY command, its maximum length has to be entered here (the Pickwick drive answers with 51 (33H) bytes).
  - The first two bits of the last byte are not checked by the drive (marked as X's in the CDB), so we can leave their values at 0. The next four bits are reserved and therefore expected to be 0. The last two bits are set to zero, too, because we don't want to perform several linked requests but only a single one.
- Size
  The expected data is of length 33H, so `testdrv` must reserve this 51 bytes for the response.

**NOTE:**
There are some useful keys when editing the CDB resp. the Data field:

| | |
|---|---|
| ↑, Ctrl-P | Move cursor to previous line |
| ↓, Ctrl-N | Move cursor to next line |
| ←, Ctrl-B | Move cursor to previous byte/character |
| →, Ctrl-F | Move cursor to next byte/character |
| PgUp | Move cursor to previous page |
| PgDn | Move cursor to next page |
| Tab | Change to byte/ASCII area |
| R | Reset all bytes (set to 0x00) |
| S | Set all bytes (set to 0xFF) |
| P | Preset all bytes (set to 0x00 ... 0xFF) |

If your screen equals to

```
Current Request parameters:

 1)      Target ID   : 1
 2)      Target LUN  : 0
 3)      Sector size : 1024
 4)      Read        : 1
 5)      Blockmove   : 0
 6)      Timeout     : 5 sec
 7)      CDB size    : 6
 8)      CDB         : 12 00 00 00 33 00 00 00 00 00 00 00
 9)      Size        : 51
         Rest        : 0
         Status      : 0x0
 A)      Data        : 00 00 00 00 ...

 1..A: edit field  R: send request Q: quit
```

perform the request by pressing '`r`'. If everything was done in the right way, the value of status is `0x0` and the Data field contains `01 80 01 00 ...` . Press '`a`' to have a look at the complete data block.

```
New Data

xxxxxx | x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 xA cB xC xD xE xF | ascii
-------+-------------------------------------------------+-----------------
000000 | 01 80 01 00 2E 00 00 00 44 49 43 4B 45 4E 53 20 | ........DICKENS
000010 | 50 69 63 6B 77 69 63 6b 2D 31 38 33 36 20 20 20 | Pickwick-1836
000020 | 2D 30 37 3A 30 36 43 52 45 41 54 45 44 31 31 31 | -07:06CREATED111
000030 | 00 00 17 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
000040 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
000050 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
000060 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
000070 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
000080 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
000090 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
0000a0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
0000b0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
0000c0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
0000d0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
0000e0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
0000f0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
```

If the status is `0x1` (which means CHECK CONDITION), you have to perform the REQUEST SENSE command to get information about the error that has occured. That command is described in our handbook, too (see figure 4.4).

The CDB is followed by a description of the Extended Sense Byte Definitions (including the Sense Key Value Definitions), which has a max. length of 26 (1AH) bytes. So we transform our screen to

```
Current Request parameters:

 1)      Target ID   : 1
 2)      Target LUN  : 0
```

|  |  |
|---|---|
| Command: | REQUEST SENSE |
| Group: | 0 |
| OP-code: | 03H |
| Command descriptor block: | |

| Bit / Byte | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 01 | Logical Unit Number | | | Reserved | | | | |
| 02 | Reserved | | | | | | | |
| 03 | Reserved | | | | | | | |
| 04 | Allocation Length | | | | | | | |
| 05 | 0 | 0 | Reserved | | | | Flag | Link |

Figure 4.4: “Guide to the famous Pickwick drive”, page 42

```
3)      Sector size : 1024
4)      Read        : 1
5)      Blockmove   : 0
6)      Timeout     : 5 sec
7)      CDB size    : 6
8)      CDB         : 03 00 00 00 1A 00 00 00 00 00 00 00
9)      Size        : 26
        Rest        : 0
        Status      : 0x0
A)      Data        : 00 00 00 00 ...

1..A: edit field  R: send request Q: quit
```

and perform the request. Interpreting the received data byte by byte will help us to find the error.

### 4.3.3 Creating the `scsiinfo` configuration file

The device driver needs some technical information about the connected SCSI devices. This information is provided in the file `/helios/etc/scsiinfo`. Run `gsi` with the shipped file `/helios/etc/scsiinfo.src`, and `gsi` will print a list of the supported drives. `scsiinfo.src` is ASCII text and can be edited to add new device descriptions. It is recommended to copy the file `pattern.src` to `<NewDevice>.src`, to change that copy afterwards like described in itself and in this manual, and to append the new device description afterwards to `scsiinfo`.

#### 4.3.3.1 General structure

The `scsiinfo.src` file contains a sequence of entries consisting a keyword, a name and a description. The currently supported keyword is *device*. The name is used to identify the entry within the file and has no further meaning. The description is enclosed in braces (‘`{`’ and ‘`}`’) and consists of a sequence of keyword/value pairs. The character ‘`#`’ introduces a comment which extends to the end of the line.

Each keyword expects a value of a certain type. This may either be a name, a number, a description or a list. Names are either character sequences enclosed in double quotes or consist of a leading alphabetic character followed by alphanumerics or underscores. Numbers are sequences of numerals according to the standard C syntax, optionally preceded by a minus sign. Lists are sequences of numbers in the range from 0 to 255, enclosed in braces.

See appendix A.2 for a formal description of the `scsiinfo` file. The next paragraphs will give you a top-down introduction.

#### 4.3.3.2 Toplevel

A `scsiinfo` file is a list of device descriptions and consequently has the following form:

```
device "Dickens Pickwick Series 18"  # This is the 1st device description,
{                                    # so you have to refer to it with
  :                                  # "type 0x0n" from the devinfo file
  :                                  # The gsi command which compiles
  :                                  # scsiinfo.src to scsiinfo will
  :                                  # report the string after the device
  :                                  # keyword when translating this
  :                                  # device description block. The
  :                                  # string has no other meaning.
}

:                                    # more devices...
:

device "Dickens Tupman Device"       # This is the last device description
{                                    # block within scsiinfo.
  :
}
```

#### 4.3.3.3 Device level

Let’s have a look at a typcial device description block:

```
device "Dickens Pickwick Series 18"
{
  type        random # The device type specifies the set of mandatory
                     # commands for device operation. Other possible
                     # values are "structured" (wich means the same like
                     # "random" and points out a disc device), "raw" and
                     # "sequential" (wich both hint a tape device).

  ident       "DICKENS Pickwick-1836" # The specified name is expected as
                     # the reply to the Inquiry SCSI command and is used
                     # to verify the correct application of a description
                     # onto a device.

  command            # A SCSI command description as described below. A
  {                  # device-type dependent set of commands has to be
    :                # defined.
  }

  :

  command            # End of the command description list.
  {
    :
  }

  error              # An error condition which can be extracted from the
  {                  # reply to the Request Sense SCSI command. Conditions
    :                # which are not defined result in an unspecified
    :                # error.
  }

  :

  error              # End of the error description list.
  {
    :
  }

  request            # Both the Helios FG_Open and FG_Close requests are
  {                  # processed with a SCSI command sequence (which may
    :                # be empty). For the Helios FG_Format request, it is
    :                # also possible to define an initialisation sequence
    :                # to prepare the device for the Format command.
  }

  :

  request            # End of the request description list.
  {
   :
  }

}                    # End of the device description block.
```

#### 4.3.3.4 Command/error/request level

**4.3.3.4.1 Commands** A command description block is of the following form:

```
command
{
  name       Mode_Select # The command name is used to identify the
                         # standard SCSI commands, which are mandatory for
                         # a device. Command names are also referred in
                         # sequence definitions. Command names are not
                         # case-sensitive, and underscores are treated as
                         # spaces.

  read       no          # The direction of data flow is necessary for
                         # correct driver operation.

  blockmove  yes         # The device driver can use a fast blockmove
                         # operation to speed up data transfer. This
                         # operation is only possible if the device
                         # performs sector transfers under all
                         # circumstances.

  cdbsize     6          # The size of the Command Descriptor Block in
                         # bytes. Default values are calculated for groups
                         # 0, 1, 2 and 6, other commands need this value.

  cdb                    # The Command Descriptor Block defines the SCSI
  {                      # command to be executed. For the Request Sense,
    :                    # Inquiry, Mode Sense, Read, Write, Verify, Write
    :                    # Filemark and Space SCSI commands position and
    :                    # size parameters are inserted at runtime. For
    :                    # all other commands, the CDB should contain the
    :                    # complete description including sizes.
  }

  datasize    0x10       # Only the Read, Write, Verify and Space SCSI
                         # commands use calculated data sizes. For all
                         # other commands, the transfer size in bytes
                         # should be supplied.

  data                   # Some SCSI commands, e g Mode Select, need
  {                      # user-supplied parameters. Up to datasize bytes
    :                    # may be provided here.
  }

}
```

**4.3.3.4.1.1 Mandatory SCSI commands** The device driver supports disc (random access) and tape (sequential access) drives. Due to their different characteristics, both types use different SCSI commands and different parameters. The following list names those SCSI commands which have to be defined for either drive type.

- Common commands
  These commands have to be defined for both device types:

  **Test Unit Ready** checks whether a logical unit is ready for media access. It takes no parameter and has no reply.

  ```
  command
  {
    name  Test_Unit_Ready
    read  yes
    cdb   { 0x00 0x00
            0x00 0x00
            0x00 <VendorUnique> }
          }
  }
  ```

  **Request Sense** requests that the target shall send sense data to the initiator, thus providing additional information in case of errors. Both extended and nonextended sense data format is supported. The expected `<DataSize>` should be supplied in `scsiinfo`. Sense data is checked for known error conditions as defined in `scsiinfo` (see below).

  ```
  command
  {
    name      Request_Sense
    read      yes
    cdb       { 0x03       0x00
                0x00       0x00
                <DataSize> <VendorUnique> }
    datasize  <DataSize>
  }
  ```

  **Inquiry** requests that information regarding the target parameters and identity is sent to the initiator. The expected `<DataSize>` should be supplied in `scsiinfo`. Inquiry data is used to check the device identity against the ident value of the `scsiinfo` device entry. Further information regarding the device type and medium removability is also extracted.

  ```
  command
  {
    name      Inquiry
    read      yes
    cdb       { 0x12       0x00
                0x00       0x00
                <DataSize> <VendorUnique> }
    datasize  <DataSize>
  ```

```
}
```

- Commands for direct-access device types
  Disc devices are always used in blocked mode. Therefore, a transfer size value refers to a specific number of blocks. The following commands are mandatory for disc devices:

**Format Unit** ensures that the medium is formatted so that all data blocks are accessible. This does **not** require a physical formatting, and there is no guarantee that the medium will be altered. The CDB is taken as is from `scsiinfo`, any format mode and interleave parameters have to be named in the CDB definition there. Any defect lists should be supplied as data to the Helios request. The command has no reply.

```
command
{
  name        Format
  read        no
  cdb         { 0x04              <Mode>
                <VendorUnique>    <Interleave(MSB)>
                <Interleave(LSB)> <VendorUnique> }
}
```

**Reassign Blocks** requests the target to reassign defective blocks to unused spare blocks reserved for this purpose. The required defect list is generated by the device driver.

```
command
{
  name        Reassign_Blocks
  read        no
  cdb         { 0x07 0x00
                0x00 0x00
                0x00 <VendorUnique> }
}
```

**Read** requests the target to transfer user data from the medium to the initiator. Both logical block address and transfer size are calculated from the Helios request and inserted into the CDB.

```
command
{
 name        Read
 read        yes
 cdb         { 0x08 0x00
               0x00 0x00
               0x00 <VendorUnique> }
}
```

**Write** requests the target to transfer user data from the initiator to the medium. Both logical block address and transfer size are calculated from the Helios request and inserted into the CDB.

```
command
{
  name        Write
  read        no
  cdb         { 0x0A 0x00
                0x00 0x00
                0x00 <VendorUnique> }
}
```

**Mode Sense** provides a means for the target to report medium, unit or device parameters to the initiator. The expected `<DataSize>` should be supplied in `scsiinfo`. Mode data is used to check medium write protection. If a block descriptor is available, it will be used to get information about the number and the size of logical blocks on the medium.

```
command
{
  name      Mode_Sense
  read      yes
  cdb       { 0x1A       0x00
              0x00       0x00
              <DataSize> <VendorUnique> }
  datasize  <DataSize>
}
```

**Read Capacity** causes the target to send information about the medium capacity to the initiator. Capacity data is used to determine the disc size, therefore the PMI bit of the CDB has to be cleared (set to zero).

```
command
{
  name      Read_Capacity
  read      yes
  cdb       { 0x25 0x00
              0x00 0x00
              0x00 0x00
              0x00 0x00
              0x00 <VendorUnique> }
  datasize  0x08
}
```

**Verify** command requests the target to verify data written on the medium. This shall be a medium verification (no data will be transferred for this command), and the BytChk bit of the CDB has to be cleared (set to zero).

```
command
{
  name        Verify
  read        no
  cdb         { 0x2F 0x00
                0x00 0x00
                0x00 0x00
                0x00 0x00
                0x00 <VendorUnique> }
}
```

- Commands for sequential-access device types
  Tape devices may be used in blocked or nonblocked mode. The operating mode has to be set up before the first medium access command is issued, usually during the initialisation sequence. Transfer sizes and position values in variable length mode will be in bytes. Now, here are the commands which are needed for tape devices:

**Rewind** requests that the target rewind the logical unit to the beginning-of-medium or load-point. Depending on the `<Immediate>` bit, the status is returned as soon as the operation is initiated or after the operation is completed. It is recommended to clear this bit (set to zero).

```
command
{
  name        Rewind
  read        yes
  cdb         { 0x01 <Immediate>
                0x00 0x00
                0x00 <VendorUnique> }
}
```

**Read** requests the target to transfer user data from the medium to the initiator. The transfer length is calculated from the Helios request and inserted into the CDB. The `<Fixed>` bit has to be set according to the selected mode. In variable block mode, the transfer size should meet the block size found on the medium.

```
command
{
  name        Read
  read        yes
  cdb         { 0x08 <Fixed>
                0x00 0x00
                0x00 <VendorUnique> }
}
```

**Write** command requests the target to transfer user data from the initiator to the medium. The transfer length is calculated from the Helios request and inserted into the CDB. The `<Fixed>` bit has to be set according to the selected mode.

```
command
{
  name        Write
  read        no
  cdb         { 0x0A <Fixed>
                0x00 0x00
                0x00 <VendorUnique> }
}
```

**Write Filemarks** causes a specified number of filemarks to be written to the medium.

```
command
{
  name        Write_Filemarks
  read        no
  cdb         { 0x10 0x00
                0x00 0x00
                0x00 <VendorUnique> }
}
```

**Space** provides a variety of positioning functions determined by `<Code>` and the count value, which is adapted to meet the block size if fixed mode is used.

```
command
{
  name        Space
  read        yes
  cdb         { 0x11 <Code>
                0x00 0x00
                0x00 <VendorUnique> }
}
```

**Mode Sense** provides a means for the target to report medium, unit or device parameters to the initiator. The expected `<DataSize>` should be supplied in `scsiinfo`. Mode data is used to check medium write protection. If a block descriptor is available, it will be used to get information about the number and the size of logical blocks on the medium. A logical block size of zero marks variable length mode and is interpreted as a block length of one.

```
command
{
  name      Mode_Sense
  read      yes
  cdb       { 0x1A       0x00
```

```
                0x00       0x00
                <DataSize> <VendorUnique> }
  datasize  <DataSize>
}
```

**4.3.3.4.1.2 Other SCSI commands** Those commands described above are essential for driver operation, and the presence of their definitions is checked by the `gsi` compiler. Various other commands may be defined in the `scsiinfo` file, they can be used for device initialisation or to prepare formatting. Some examples are the following commands:

- Commands for direct-access device types

  **Mode Select** provides a means for the initiator to specify medium, unit or device parameters. Its parameter list has to be declared in `scsiinfo`.

```
command
{
  name      Mode_Select
  read      no
  cdb       { 0x15                  0x00
              0x00                  0x00
              <ParameterListLength> <VendorUnique> }
  datasize  <ParameterListLength>
  data      { 0x00
              <MediumType>
              0x00
              <BlockDescriptorLength>
              <BlockDescriptor(s)>
              <VendorUnique> }
}
```

  **Start/Stop Unit** causes the target to enable or disable the unit for further operation. A Winchester disc drive might spin down if a Stop Unit command is issued, hereby reducing its power consumption. The Start Unit command is typically part of the initialisation sequence to ensure correct drive operation.

```
command
{
  name  Start_Unit
  read  no
  cdb   { 0x1B 0x00
          0x00 0x00
          0x01 <VendorUnique> }
}

command
{
  name  Stop_Unit
```

```
    read  no
    cdb   { 0x1B 0x00
            0x00 0x00
            0x00 <VendorUnique> }
  }
```

- Commands for sequential-access device types

**Mode Select** provides a means for the initiator to specify medium, unit or device parameters. Its parameter list has to be declared in `scsiinfo`.

```
command
{
  name      Mode_Select
  read      no
  cdb       { 0x15                 0x00
              0x00                 0x00
              <ParameterListLength> <VendorUnique> }
  datasize  <ParameterListLength>
  data      { 0x00
              0x00
              <Mode>
              <BlockDescriptorLength>
              <BlockDescriptor(s)>
              <VendorUnique> }
}
```

**Erase** requests part or all of the medium to be erased, which means that the medium shall appear as gap or unwritten. This command might be used as a replacement to the Format command to put a medium into a known state.

```
command
{
  name  Format          # Do not use Erase!
  read  no
  cdb   { 0x19 <Long>
          0x00 0x00
          0x00 <VendorUnique> }
}
```

**Load/Unload** causes the target to enable or disable the unit for further operation. A tape drive e g might eject the medium if a Unload command is issued, hereby reducing its power consumption. This SCSI command is mostly defined twice in `scsiinfo` to load the medium within the initialisation sequence and to unload it after usage. It may also be used to request the re-tension function on peripheral devices that support this function.

```
command
{
  name        Load
  read        no
  cdb         { 0x1B                0x00
                0x00                0x00
                (<ReTension> | 0x01) <VendorUnique> }
}

command
{
  name        Unload
  read        no
  cdb         { 0x1B       0x00
                0x00       0x00
                <ReTension> <VendorUnique> }
}
```

- Commands for removable medium device types
  **Prevent/Allow Medium Removal** requests that the target should enable or disable the removal of the medium from the unit. Usually, this SCSI command is defined twice in `scsiinfo` to distinguish between allowing and preventing medium removal.

```
command
{
  name        Prevent_Medium_Removal
  read        yes
  cdb         { 0x1E 0x00
                0x00 0x00
                0x01 <VendorUnique> }
}

command
{
  name        Allow_Medium_Removal
  read        yes
  cdb         { 0x1E 0x00
                0x00 0x00
                0x00 <VendorUnique> }
}
```

**4.3.3.4.2 Errors** After a Request Sense SCSI command, the reply will be scanned for known errors, which are defined with the following error description block:

```
error
```

```
{
  code        0x47110815  # This value is returned as the Helios error
                          # code if all of the following conditions are
                          # met.

  condition               # A condition description block, see below. Any
  {                       # number of these may appear.
    :
  }

  :

  condition
  {
    :
  }

}
```

**4.3.3.4.2.1 Conditions** Conditions are evaluated in order of appearance. If all conditions for an error code are fulfilled, the associated error code will be returned.

```
condition
{
  offset  0x08  # Byte offset into the Request Sense reply data.
  mask    0xFF  # Bit mask to apply to the addressed byte.
  value   0x15  # Value to be expected after masking out unused bits.
                # If this value is found, the condition is satisfied.
}
```

**4.3.3.4.3 Requests** These keywords are used to specify a SCSI command sequence:

```
request
{
  fncode 0x06060842  # Helios function code, for which the sequence shall
                     # be submitted.

  item  Mode_Select  # One of the earlier defined commands, which shall be
                     # executed.
}
```

#### 4.3.3.5 Compiling

If this source file finally meets your environment, compile it to binary form with:

```
% gsi scsiinfo.src /helios/etc/scsiinfo
```

### 4.3.4 Testing `scsiinfo/devinfo` entries

After recompiling both the `scsiinfo` file and the `devinfo` file, you should ensure that the driver will work correctly with the new configuration. Another test utility named `testinfo` allows you to simulate Helios requests as if they were comming from the file server. `testdrv` doesn't support multivolume/multipartition `devinfo`s. Only volumes that consist of one partition will be tested correctly.

**CAUTION:**
The `testinfo` utility does **not** preserve any data previously contained in the tested drives. Prove that you have a complete set of backups for all tested drives.

Starting `testinfo` is as follows:

```
% remote MSC testinfo msc21
```

```
Defined Volumes : 3
        0 :     tandberg
        1 :     exabyte
        2 :     wren

(O)pen Volume  open (A)ll  (Q)uit :
```

The program starts and displays the list of volumes as defined in `devinfo` (imagine that each volume occupies a complete drive). Now you may select a command by pressing one of the keys enclosed in brackets. When a volume has been opened successfully, the program creates a new window for that volume, using the volume name for the window name. After opening e g the Tandberg tape streamer, the main window will look as follows:

```
Defined Volumes : 3
        0 :     tandberg
        1 :     exabyte
        2 :     wren

(O)pen Volume  open (A)ll  (Q)uit : o

Which Volume : 0
```

```
   sending Open request for Volume #0 ( tandberg )
   time = 36 ms, Actual 0, Result 0x0

   Load result :   Error 0x0

           Raw 1, Removable 1, Loaded 1
           Protected 0, Formatted -1, NotLocked 0

   Defined Volumes : 3
           0 : *   tandberg
           1 :     exabyte
           2 :     wren

   (O)pen Volume  open (A)ll :
```

In this example, the `FG_Open` request was completed successfully within 36 ms. The Load result gives further information about the drive's state and type: The drive is of type raw and uses removable media. The medium which has been loaded is not write-protected, but no access check has been made. In the volume list the tandberg drive is marked open with the asterisk. Furthermore, `(Q)uit` is disabled until all volumes are closed. There are two types of volume windows, for disc drives and for tape drives, reflecting the different command sets for each drive type. For the Tandberg streamer, a tape window is opened and a menu offers several requests to execute and a test to perform. We choose to perform the test and are asked for the `File Size` and the `Files per Loop`. After typing in '20' and '8', we make the tape to rewind first, and the test begins. After a few minutes, we stop the test by pressing a key, the tape is rewound a short summary is printed to the screen.

```
                                                                        tandberg

HandleTape #0 (tandberg) :
(G)etSize  (R)ead  (W)rite  (S)eek  (E)rase  (F)ilemark  (T)est  (C)lose : t
File Size ( 4 MByte ) : 20
Files per Loop ( 8 ) :
1: Space reverse, test whole tape  0: Rewind, test from start : 0
Testting Tape tandberg: From start with 8 files of 20 MByte, loop 1:
Rewind...                     53 ms
Write Filemark...            687 ms, file avg 77 KByte/sec
Write   501...               643 ms,   99 KByte/sec, bus avg  103 KByte/sec
Test stopped.
Rewind...                  38577 ms
Maximum value after 1 Loop:
Write count : 502, Reda Count 0, Tape Size 41024 units of 1024 bytes
Number of Requests with data errors : 0

HandleTape #0 (tandberg) :
(G)etSize  (R)ead  (W)rite  (S)eek  (E)rase  (F)ilemark  (T)est  (C)lose : c
Executing close request ...
```

```
time: 101 ms, Actual: 0, Result 0x0
press any key to close the window...
```

There are some more items in the menu to choose from, here is a short description:

‘G’ Perform a GetSize request. `testinfo` reports the number of blocks ans the block size.

‘W’ Perform a Write request. You have to enter the number of units (whereas each unit contains `devinfo`’s *addressing* bytes) and the comparison seed. A sequence of random number (initialized with the comparison seed) will be written do tape. The time needed, the number of bytes actally written and the Helios error code is reported. In case of failure a Request Sense will be executed and its result will appear, too.

‘R’ Perform a Read request. You have to enter the number of units and the comparison seed, too. Based on that seed, the same sequence of random numbers is generated again and is expected to match the sequence written to tape before. `testinfo` reports time, bytes, error code and sense data (if required).

‘S’ Perform a Seek request. You have to choose wether to look for blocks, for file marks, or for sequential file marks. The number of blocks/marks to skip has to be entered.

‘E’ Perform a Format request.

‘F’ Perform a Write Filemark request. Please enter the number of file marks to write.

‘T’ Conduct a test like described above.

‘C’ Perform a Close request and close the window.

To finish this mini-session, press ‘`c`’, which causes `testinfo` to close the `tandberg` window and to return to the main menu.

A test session with a random access drives is quite similar, an example could be as follows:

```
                                                                      wren
executing GetSize request ...
Device has 647926 block of 1024 bytes.

HandleDisc #2 (wren) :
(R)ead  (W)rite  (F)ormat  (G)etSize  (T)est  (C)lose : t
Test disc wren :
Number of requests ( 16 ): 16
Request size ( 1 .. 64 units ) : 64
Allocted 16 requests of 64 units.
Test: (S)ame (C)ontiguous op(T)imised (R)andom (O)ptimised (E)nd : o
Press any key to stop :
```

```
Random Optimised :
wwwwwwwwwwwwwwww
16 requests of 64 KByte each written in 1790 ms ( 571 KBytes/sec )
rrrrrrrrrrrrrrrr
16 requests of 64 KByte each read in 1781 ms ( 574 KBytes/sec )
Random Optimised :
wwwwwwwwwwwwwwww
16 requests of 64 KByte each written in 1730 ms ( 591 KBytes/sec )
rrrrrrrrrrrrrrrr
16 requests of 64 KByte each read in 1731 ms ( 591 KBytes/sec )
Test: (S)ame (C)ontiguous op(T)imised (R)andom (O)ptimised (E)nd : e
```

In addition to the tape menu the disc menu offers some different kind of tests. After entering the number of requests and the size of each request, you have the choice between 5 tests:

- ‘S’ Repeat the last test.
- ‘C’ Read/write some blocks that are in series on disc. Wait for the completion of each request.
- ‘T’ Like ‘C’ , but its waited for the undevided completion of the requests.
- ‘R’ Read/write some blocks placed in different disc areas.
- ‘O’ Like, ‘R’, but wait for undevided completion.

# Chapter 5

# Commands

The following commands can be executed from the command line and are supplied to make better use of the Parsytec File System.

## 5.1 access

| | |
|---|---|
| **Purpose:** | Reports access rights of a file. |
| **Format:** | `access <File>` |
| **Description:** | `access` reports the user's access rights of file. For more details see "The Helios Parallel Operating System", section 3.4, "Protection: a tutorial". |
| **See also:** | `chmod`, `matrix`, `refine` |

## 5.2 chmod

| | |
|---|---|
| **Purpose:** | Alter the protection bits of a file. |
| **Format:** | `chmod [vxyz][+-=] [rwefghvxyzda] <File>` |
| **Description:** | `chmod` is used to enable and disable the protection bits of a `<File>`. For more details see "The Helios Parallel Operating System", section 3.4, "Protection: a tutorial". |
| **See also:** | `access`, `matrix`, `refine` |

## 5.3 finddrv

| | |
|---|---|
| **Purpose:** | Looks for SCSI devices. |
| **Format:** | `finddrv <DiscDevice>` |
| **Description:** | `finddrv` reads `/helios/etc/devinfo` and loads the driver specified in the `<DiscDevice>` block. The driver reads the `/helios/etc/scsiinfo` file and tests every SCSI address for devices. |

## 5.4 format

| | |
|---|---|
| **Purpose:** | Format a disc. |
| **Format:** | `format <PathToVolume>` |
| **Description:** | This command works with optical discs only. The specified volume is tried to be formatted physically. After formatting each sector is verified and reassigned on error. The volume has to be loaded with the `-m` option otherwise `format` returns an error. Depending on medium size the execution of this command may last some time because the `format` command scans every physical sector on disc. |
| **See also:** | `makefs`, `mksuper` |

## 5.5 fs

**Purpose:** Start the Parsytec File Server.

**Format:** `fs [-c][-o][-f|-b|-n] [<DevInfoName>] <FileServer>`

**Description:** The file server is started, loads `<DevInfoName>` (when given), tries to install the file server specified in the `<FileServer>` (usually of value '`msc21`') block. The file server has to be started on a MSC in the background. This is normally done with

```
remote -d MSC fs msc21
```

Options:

- `-c`
  The use of the buffer cache checksum is enabled. Using this mode reduces speed of file server by factor three. Default is working without buffer cache checksum.

- `-o`
  If this option is given the file server reports open requests to the server window.

- `-f`, `-b`, `-n`
  These options work with structured volumes only. The checking mode is determined. This option is overridden by the checking mode option of a `load` command. Default checking mode is `-f`.

  - `-f`
    Full checks; file system data and directory trees are checked.
  - `-b`
    Basic checks; file system data is checked and on occurrence of errors directory trees are checked.
  - `-n`
    No checks; checker is bypassed completely.

**NOTE:**
`fs` allocates memory for all volumes specified in `devinfo`. So you have to execute a `termvol` command for those volumes to terminate the file server and to clean the memory, even if some volumes have not been loaded.

**See also:** `load`, `unload`, `termvol`

## 5.6 fsync

| | |
|---|---|
| **Purpose:** | Toggle between partly and fully synchronous mode. |
| **Format:** | `fsync <PathToVolume> [-as]` |
| **Description:** | `fsync` allows the selection between two operation modes: at volume load time the default mode is the partly synchronous mode (`-a`) which means that all data-blocks are written with a certain delay (of max. 20 seconds) to disc, when the “sync process”—which is part of the server—becomes active and detects some of them. To guarantee that all blocks are written directly to disc (“write-through-cache”), the user has the alternative to switch to fully synchronous mode (`-s`) , which eliminates all delayed-write operations. |
| **See also:** | `sync` |

## 5.7 gdi

| | |
|---|---|
| **Purpose:** | Generate a "device information file". |
| **Format:** | `gdi <Input> <Output>` |
| **Description:** | `gdi` is a simple compiler which generates a binary object from the given device information `<Input>` file. The default filename which is searched by the server is `/helios/etc/devinfo`, so `gdi`'s `<Output>` should be that file. When using other values for `<Output>` (necessary if you are using several MSCs with different `devinfos`), the file server has to be given the correct `devinfo` as parameter. |
| **See also:** | `gsi` |

## 5.8 gsi

| | |
|---|---|
| **Purpose:** | Generates a "SCSI-information file". |
| **Format:** | `gsi <Input> <Output>` |
| **Description:** | `gsi` is a simple compiler which generates a binary object from the given SCSI-information `<Input>` file. The file name which is searched by the MSC device driver is `/helios/etc/scsiinfo`, so `gsi`'s `<Output>` has to be that file. |
| **See also:** | `gdi` |

## 5.9 load

**Purpose:** Load a volume.

**Format:** `load [-v][-l][-m][-f|-b|-n] <PathToVolume>`

**Description:** The volume specified by `<PathToVolume>` is loaded (mounted) and for structured volumes the checker is called and a file system is tried to be set up. After file server startup this has to be done with every volume explicitly. Changeable media are locked after they have been loaded so that they cannot be removed until an unload command is given.

Options:

- `-v`
  The `load` command waits for the completion of the load and reports about the results. On success `load` reports about the number of cylinder groups and blocks per cylinder group of the loaded file system. This option has no effect if the `-m` option is given. Default is not to wait for completion.

- `-m`
  This option works with structured volumes only. The specific volume is loaded but the checker is not called and no file system is tried to be set up. This option must be applied before using the `makefs`, `format` or `mksuper` command. This option disables the `-v` option.

- `-f`, `-b`, `-n`
  These options work with structured volumes only. The checking mode is determined. If none of these options is given the checking mode determined in the file server commandline is used. If no specific checking mode was given there the default mode (`-f`) is used.

  - `-f`
    Full checks; file system data and directory trees are checked.
  - `-b`
    Basic checks; file system data is checked and on occurrence of errors directory trees are checked.
  - `-n`
    No checks; checker is bypassed completely.

- `-l`
  This option only has an effect during a full check. If there are 'hanging' symbolic links detected after a full check these links will be destroyed. Default is not to destroy 'hanging' links.

**See also:** `unload`, `termvol`

## 5.10 makefs

**Purpose:** Create a file system.

**Format:** `makefs <PathToVolume>`

**Description:** This command works with structured volumes only. A new file system is tried to be created on the volume depending on the volume/partition description in `/helios/etc/devinfo` (that was created via the `gdi` command). The volume has to be loaded with the `-m` option and physically formatted, otherwise `makefs` returns an error. On success `makefs` reports about the number of cylinder groups and blocks per cylinder group of the file system it has installed.

**CAUTION:**

When accidently running on a data containing file system, all files will be destroyed.

**See also:** `format`, `load`, `mksuper`

## 5.11 man

| | |
|---|---|
| **Purpose:** | Prints command description. |
| **Format:** | `man <PFScommand>` |
| **Description:** | Prints (via `more`) a short description of the given `<PFScommand>`. |

## 5.12 matrix

| | |
|---|---|
| **Purpose:** | Display the access matrix of a file. |
| **Format:** | `matrix <File>` |
| **Description:** | The utility `matrix` displays the access matrix of the given `<File>`. For more details see "The Helios Parallel Operating System", section 3.4, "Protection: a tutorial". |
| **See also:** | `access`, `chmod`, `refine` |

## 5.13 mksuper

| | |
|---|---|
| **Purpose:** | Generate a superblock. |
| **Format:** | `mksuper <PathToVolume>` |
| **Description:** | This command works with structured volumes only. A superblock (info block 0) of the specified volume is constructed depending on the `/helios/etc/devinfo` information and written to disc. The volume has to be loaded with the `-m` option, otherwise mksuper will return an error. This command should only be used if the checker failed because of a corrupted superblock. |
| **See also:** | `format`, `load`, `makefs` |

## 5.14 ptar

|  |  |
|---|---|
| **Purpose:** | Store files in an archive. |
| **Format:** | `ptar <Options> <Files>` |
| **Description:** | `ptar` allows you storing copies of files in an archive. |

Options:

- `-c`, `-d`, `-t`, `-x`
  These four option switch between the operation modes.
  - `-c`
    Create a new archive.
  - `-d`
    Compare the files in the archive with those in the file system and report about differences.
  - `-t`
    Display a list a the files in the archive.
  - `-x`
    Extract files from archive.
- `-B`, `-C`, `-f`, `-M`, `-N`, `-R`, `-T`, `-v`, `-w`
  General options.
  - `-B <Number>`
    Set blocking factor to `<Number>`.
  - `-C <Directory>`
    Change into `<Dirtectory>` before continuing.
  - `-f <Filename>`
    Archive files in `<Filename>` (instead of using the value of `TARFILE` respectively '`tar.out`').
  - `-M`
    Work on a multi-volume archive.
  - `-N <Date>`
    Work only on files whose creation or modification date is newer than `<Date>`.
  - `-R`
    Print each message's record number.
  - `-T <Filename>`
    Work on the list of files in `<Filename>`, too.
  - `-v`
    Enter verbose mode.
  - `-w`
    Wait for user's confirmation before every action.

- `-h`, `-V`
  Creation options.
  - `-h`
    Treat simbolic links as normal files or directories.
  - `-V <Name>`
    Write a volume header at the beginning of the archive.
- `-k`, `-m`, `-p`
  Extraction option:
  - `-k`
    Keep existing files in the file system.
  - `-m`
    Do not extract the modification and access date from archive.
  - `-p`
    Set access matrices as recorded in the archive.

See chapter "Backups" for detailed information.

## 5.15 refine

| | |
|---|---|
| **Purpose:** | Refine or restrict a capability. |
| **Format:** | `refine [-+=][rwefghvxyzda] <File>` |
| **Description:** | refine allows refining and restricting of capabilities associated to a `<File>`. For more details see "The Helios Parallel Operating System", section 3.4, "Protection: a tutorial". |
| **See also:** | `access`, `chmod`, `matrix` |

## 5.16 sync

| | |
|---|---|
| **Purpose:** | Force a sync operation immediately. |
| **Format:** | `sync <PathToVolume>` |
| **Description:** | The utility `sync` forces an “extra” sync operation which guarantees that all data blocks in the buffer cache with the “delayed-write” flag set are written immediately to disc. `sync` is especially useful to guarantee consistency if the file server or the whole system shall be shut down. |
| **See also:** | `fsync` |

## 5.17 termvol

**Purpose:** Terminate a volume.

**Format:** `termvol [-v] <PathToVolume>`

**Description:** The volume specified by `<PathToVolume>` is terminated. Therefore the volume is unloaded and then the volume specific central server is terminated. A volume which has been terminated cannot be loaded again before starting the file server again. If all volumes have been terminated the whole file server will terminate automatically.

**NOTE:**
With respect to safety of data there's no command to terminate all volumes (and thereby the complete file server) at once.

Option:

- `-v`
  The termvol command waits for the completion of the volume termination.

**See also:** `fs`, `load`, `unload`

## 5.18 testdrv

**Purpose:** Perform low-level SCSI commands.

**Format:** `testdrv <DiscDevice>`

**Description:** `testdrv` reads `/helios/etc/devinfo` and loads the driver specified in the `<DiscDevice>` block. It lets you compose SCSI requests and performs them on a specified drive. `testdrv` must be executed on the transputer placed on the MSC board, so you must type e g

```
% remote MSC testdrv msc21
```

**CAUTION:**
Data is **not** preserved, you have to backup the drives that are intended to be tested.

**See also:** Section 4.3.2

## 5.19 testinfo

**Purpose:** Test devices described in `devinfo`/`scsiinfo`.

**Format:** `testinfo <FileServer>`

**Description:** `testinfo` performs some menu controlled standard test with the drives described in the information files (whereas multivolume/multipartition `devinfo`s are not supported). `testinfo` must be executed on the transputer placed on the MSC board, so you must type e g

```
% remote MSC testinfo msc21
```

**CAUTION:**
Data are **not** preserved, you have to backup the drives that are intended to be tested.

**See also:** Section 4.3.4

## 5.20 unload

|  |  |
|---|---|
| **Purpose:** | Unload a volume. |
| **Format:** | `unload [-v] <PathToVolume>` |
| **Description:** | The volume specified by `<PathToVolume>` is unloaded (unmounted). Actually working processes on that volume are closed and the volume is updated. Protected media are unlocked so that they can be removed. In contrary to the `termvol` command the central server for this volume is not terminated so that the volume may be loaded again.<br>Option:<br>• `-v`<br>`unload` waits for the completion of the volume unloading. |
| **See also:** | `fs`, `load`, `termvol` |

# Chapter 6

# Backups

## 6.1 Creating archives

The Parsytec tape archive program, `ptar`, is a tool to store copies of a file or a group of files in an archive. This archive may be written directly to tape, stored as a file or sent through a pipe to another program, e g `compress`. `ptar` can also be used to list the files in an archive or to extract the files in the archive.

### 6.1.1 What is an archive?

An archive describes the names and contents of the constituent files. Archives are basically files, but may be written to and read from a tape. The format used by `ptar` is compatible with the standard tar format, and archives may be sent to other machines even though they run different operating systems. Piping one `ptar` to another is an easy way to copy a directory's contents from one place on a disc to another, hereby preserving the dates, modes and link-structure of all the files within.

## 6.2 Argument syntax

The full syntax of the `ptar` command is as follows:

```
% ptar <Options> [<Files>]
```

Options and file names may be freely mixed, because each argument starting with '`-`' is considered to be an option argument. A single option argument may contain several options, of which the last one may expect a parameter. This parameter should come immediately after the option, possibly separated from it by a space. It is not feasable to put more than one parameterised option into a single argument, since the rest of the argument following the first parameterised option is regarded as its parameter value.

## 6.3 Operation modes

`ptar` is used to create an archive, to extract files from an archive or to list the contents of an archive as well. Each time you run `ptar`, you must specify exactly one of these operation modes which should be the first option for `ptar`. Other arguments are file names to work on, files to put into the archive or the files to extract from it. If you don't specify any file name, the default will depend on the operation mode: when creating an archive, all files in the current directory are used. When reading, listing or comparing an archive, `ptar` will work on all files in the archive. If a file name argument actually names a directory, then that directory, its files and all its subdirectories are used.

Here's a list of the `ptar` operation modes:

- `-c`
  Create a new archive that contains all the files specified on the command line.
- `-d`
  Compare the files in the archive with their counterparts in the file system. `ptar` will report all differences in size, mode, access matrix and contents. If a file exists in the archive but cannot be found in the file system, it will report this. If you specify file names, those files are compared with the archive and they all must exist in the archive.
- `-t`
  Display a list of the files in the archive. If you specify file names, only those files will be mentioned if they exist in the archive.
- `-x`
  Extract the specified files from the archive. If no file names are given, all files from the archive will be extracted.

## 6.4 Other options

All other options are not compulsory. Some of them make sense with all modes, while others should only be used with particular modes.

### 6.4.1 General options

These options are always meaningful, regardless of the operation mode.

- `-b <Number>`
  Use a blocking factor of `<Number>` for the archive. The default blocking factor is 20 blocks. When reading or writing the archive, `ptar` will always read from or write to the archive in blocks of `<Number>` * 512 bytes. Larger blocking factors result in better throughput of data, but might reduce media utilisation.

- `-C <Directory>`
  Change into the `<Directory>` before continuing. This option is usually interspersed with the files `ptar` should work on. It is especially useful when you have several widely spread files that you want to store in the same direcctory. For example,

  ```
  % ptar -c <FileHere1> <FileHere2> -C <OtherDir> <FileThere>
  ```

  will place the files `<FileHere1>` and `<FileHere2>` from the current directory in the archive followed by the file `<FileThere>` from the directory `<OtherDir>`. Note that the file `<FileThere>` is recorded under the precise name `<FileThere>`, not as `<OtherDir>/<FileThere>`. Thus, the archive will contain three files which all appear to have come from the same directory; if the archive is extracted with '`ptar -x`', all three files will be created in the same directory. In contrast,

  ```
  % ptar -c <FileHere1> <FileHere2> <OtherDir>/<FileThere>
  ```

  will record the third file in the archive under the name `<OtherDir>/<FileThere>` so that, if '`ptar -x`' is used, the third file will be created in a subdirectory named `<OtherDir>`.

- `-f <Filename>`
  Use `<Filename>` as the name of the archive file. If no '`-f`' option is given but the environment variable `TARFILE` exists, its value is used, otherwise the `ptar` writes to the default '`tar.out`'. If the filename is '`-`', `ptar` writes to the standard output (when creating) or reads from the standard input (when listing or extracting). Thus, `ptar` can be used as the head or the tail of a command pipeline.

- `-M`
  Work on a multi-volume archive - an archive that will not fit on a single medium used to hold it. When this option is used, `ptar` will not abort when it reaches the end of a medium. Instead, it will ask to prepare a new volume.

- `-N <Date>`
  Work only on files whose creation or modification dates are newer than `<Date>`. The main purpose is for creating an archive; then only new files are written. If extracting, only newer files are extracted.

- `-R`
  Print, along with each message, the record number within the archive where the message occurred. This option is especially useful when reading damaged archives, since it helps to pinpoint the damaged sections.

- `-T <Filename>`
  Read a list of file names from `<Filename>` and add them to the list of files to work on. If `<Filename>` is given as '`-`', the list is read from standard input. Several '`-T`' options may be given in the command line. Note that using both '`-T -`' and '`-f -`' will not work unless you have specified the '`-c`' mode before.

- `-v`
  This option causes `ptar` to be verbose about the actions it is taking. Normally `ptar` does its work silently; this option displays the name of each file `ptar` treats.

When used with the '`-t`' option, `ptar` prints full information about each file like '`ls -l`'.

- `-w`
  Wait for user confirmation before taking the specified action. `ptar` prints a message for each operation it intends to take, and waits for a line of input. If your input line begins with '`y`', the action is performed, otherwise it is skipped. This option can only be used together with the '`-f -`' option when an archive is created.

### 6.4.2 Create options

These options are used to control which files `ptar` puts into an archive, or to control the format the archive is written in.

- `-h`
  Follow symbolic links as if they were normal files or directories and archive the linked-to object. Normally, `ptar` simply records the presence of a symbolic link. If the linked-to object is encountered again, a complete second copy will be written to the archive.
- `-V <Name>`
  Write a volume header at the beginning of the archive. If this option is used together with '`-M`', each volume of the archive will have a header of '`<Name>` Volume `<N>`', whereas `<N>` is the volume number.

### 6.4.3 Extract options

These options are useful for extracting files from the archive.

- `-k`
  Keep existing files within the file system, do not overwrite them from the archive.
- `-m`
  Do not extract the modification and access dates from the archive. The modification and access times will be the time of extraction.
- `-p`
  Set the access matrix of extracted files exactly as recorded in the archive. If this option is not used, access matrices will be set to the default matrices for directories or files.

## 6.5 Creating backups

`ptar` can be used to perform full or incremental backups. Backups should only be done when no other users or programs are modifying files in the file system. If files are modified while `ptar` is making a backup, they may not be stored properly in the archive, in which case you won't be able to restore them if necessary. You should use the '`-V`' option for full backups to name the archive, so you can tell what an archive is even without a label. For incremental backups, you will need to use the '`-N <Date>`' option to tell `ptar` to only store files that have been modified or added since date, where date should be the date and time of the last full or incremental backup. A standard scheme is to do a monthly full backup and a weekly incremental backup of everything that has changed since the last monthly. Also, perform a daily incremental dump of everything that has changed since the last monthly or weekly dump. Unless you are in a hurry (and trust in the `ptar` program and your tapes), you should compare the backup with the file system after creation using the '`ptar -d`' command to ensure that the backup has been written properly. This will also detect cases where files have been modified while or just after being archived.

# Chapter 7

# Benchmarks

To fit you for both, writing your own programs using the PFS and testing the data transfer rate of your system, we implemented two small C-programs and some shell scripts to work with them.

## 7.1 The programs

`create`
: can be used to generate a file of a given size. We used it to create `kbyte.1` (of size 1 KByte), `block.1` (of size 4 KBytes) and `packet.1` (of size 64 KBytes). In general, it's called like

```
% create <ContextDir> <FileName> <FileSize>
```

to create the file `<ContextDir>/<FileName>` that will contain `<FileSize>` bytes. See `copy.c` for the sources.

`copy`
: can be used to measure PFS' data transfer rate. It's called

```
% copy <PathToSrc> <PathToDest> <PathToVolume> <ReadCount> <WriteCount>
```

to copy the file `<PathToSrc>` to `<PathToDest>` using the file server for `<PathTo-Volume>`. If you want to measure the transfer rate from/to your host you may give values to `<ReadCount>` and `<WriteCount>` which differ from one. So you can simulate a 20 MByte file on your PC's harddisc that claims only 1 MByte by reading/writing it twenty times. See `copy.c`, `csync.c` and `pathsplt.c` for the sources.

`cppfs`
: is a shell script pattern to ease the usage of `copy`. It has to be modified to meet your environment.

`demo1-demo4`
: conduct some standard tests.

# Appendix A

# The configuration files

## A.1 Quick reference to the `scsiinfo` keywords

The following section describes each keyword, the type of value which it expects and the default value assumed if the keyword is omitted. The default value `<None>` marks keywords which must be given a value.

| keyword | member of | default | meaning |
|---|---|---:|---|
| blockmove | command | no | Turn blockmove mode on/off |
| cdb | command | `<None>` | Command Descriptor Block |
| cdbsize | command | `<GroupDependent>` | Size of the Command Descriptor Block in bytes |
| code | error | `<None>` | Helios error code |
| command | device | `<None>` | SCSI command description |
| condition | error | `<None>` | Condition description |
| data | command | `<NoData>` | Parameter list/replied data |
| datasize | command | 0 | Size of parameter list or replied data |
| error | device | `<None>` | Error description |
| fncode | request | `<None>` | Helios function code |
| ident | device | `<None>` | Reply of the Inquiry SCSI command |
| item | request | `<NoCommand>` | Command to be executed |
| mask | condition | `<None>` | Bit mask to apply to the addressed byte |
| name | command | `<None>` | Command's name |
| offset | condition | `<None>` | Byte offset into the Request Sense reply data |
| read | command | `<None>` | Direction of data flow |
| request | device | `<None>` | Request description |
| type | device | `<None>` | Selects disc or tape devices |
| value | condition | `<None>` | Expected value after masking out unused bits. |

## A.2 `scsiinfo` syntax definition

The following syntax definition specifies all items and their order of appearance, using these symbols:

| | |
|---|---|
| [ ] | Square brackets surround items that are optional. |
| { } | Braces surround optional items that can be repeated. |
| <> | Fishtail brackets enclose generic expectations. |
| ... | Three dots denote continuation of a series; for example, 12...89 is the same as the series 123456789. |
| \| | A vertical bar separates two mutually exclusive alternatives. |

```
    scsiinfo ::=  <entry> {<entry>}
       entry ::=  ‘device’ <name> <description>
 description ::=  ‘{’ <type> <ident> {<command>} {<error>} {<request>} ‘}’
        type ::=  ‘type’ ‘sequential’ | ‘raw’ | ‘random’ | ‘structured’
       ident ::=  ‘ident’ <name>
     command ::=  ‘command’ ‘{’ <cmdname> <read> <cdb> [<cdbsize>] [<datasize>]
                  [<data>] ‘}’
     cmdname ::=  ‘name’ <name>
        read ::=  ‘read’ ‘yes’ | ‘no’
         cdb ::=  ‘cdb’ ‘{’ <number> {<number>} ‘}’
     cdbsize ::=  ‘cdbsize’ <number>
    datasize ::=  ‘datasize’ <number>
        data ::=  ‘data’ ‘{’ <data_item> {<data_item>} ‘}’
   data_item ::=  <number> | <string>
       error ::=  ‘error’ ‘{’ <errcode> <condition> {<condition>} ‘}’
     errcode ::=  ‘code’ <number>
   condition ::=  ‘condition’ ‘{’ <offset> <mask> <value> ‘}’
      offset ::=  ‘offset’ <number>
        mask ::=  ‘mask’ <number>
       value ::=  ‘value’ <number>
     request ::=  ‘request’ ‘{’ <fncode> {<item>} ‘}’
      fncode ::=  ‘fncode’ <number>
        item ::=  ‘item’ <name>
        name ::=  <word> | <string>
      string ::=  ‘”’ {<any printable ascii character>} ‘”’
        word ::=  <letter> {<letter> | <digit> | <underscore>}
      number ::=  [‘-’] <ovalue> | <dvalue> | <xvalue>
      ovalue ::=  ‘0’ {<odigit>}
      dvalue ::=  <nzdigit> {<digit>}
      xvalue ::=  ‘0’ ‘x’ | ‘X’ <xdigit> {<xdigit>}
      odigit ::=  ‘0’ ... ‘7’
       digit ::=  ‘0’ ... ‘9’
     nzdigit ::=  ‘1’ ... ‘9’
      xdigit ::=  ‘0’ ... ‘9’ | ‘A’ ... ‘F’ | ‘a’ ... ‘f’
      letter ::=  ‘A’ ... ‘Z’ | ‘a’ ... ‘z’
  underscore ::=  ‘_’
```

## A.3 Quick reference to the `devinfo` keywords

There are some keywords that must be given a fixed value:

| keyword | member of | must be | meaning |
|---|---|---|---|
| addressing | discdevice | 1024 | Discdevice's addressing size |
| blocksize | fileserver | 4096 | fileserver's blocksize (interpreted in bytes) |
| hugepkt | fileserver | 1 | Size of huge cache packets (interpreted in fileserver's blocksize) |
| mediumpkt | fileserver | 4 | Size of medium cache packets (interpreted in fileserver's blocksize) |
| name | discdevice | msc21.dev | Device driver's filename (placed in `/helios/lib`) |
| smallpkt | fileserver | 16 | Size of small cache packets (interpreted in fileserver's blocksize) |

The other keyword's values a chosen by the user. A default of `<None>` forces the user to give a value to the keyword.

| keyword | member of | `gdi` default | meaning |
|---|---|---|---|
| controller | discdevice | 0 | SCSI-controller's address |
| cgsize | volume | `<BestPossible>` | Cylinder group's size (interpreted in fileserver's blocksize) |
| device | fileserver | `<None>` | Discdevice the fileserver will use |
| discdevice | | `<None>` | Start of a discdevice block |
| drive | discdevice | `<None>` | Start of a drive block |
| | partition | 0 | Drive where the partition takes place (interpreted in a drive list's number that's generated by the file server) |
| end | partition | `<LastBlock>` | Last block that is claimed by the partition (interpreted in discdevice's addressing size) |
| fileserver | | `<None>` | Start a file server block |
| hugecount | fileserver | `<Tolerable>` | Number of huge packets in the cache |
| id | drive | 0 | Drive's SCSI-address |
| mediumcount | fileserver | `<Tolerable>` | Number of huge packets in the cache |
| minfree | volume | 0 | Space to be left free for the checker's `/lost+found` directory (interpreted in fileserver's blocksize) |
| name | volume | `<None>` | Volume's name |
| ncg | volume | `<BestPossible>` | Number of cylinder groups of volume |
| partition | discdevice | `<None>` | Start of a partition block |
| | volume | `<None>` | Partition that's occupied by volume (interpreted in a partition list'snumber that's generated by the file server) |
| smallcount | fileserver | `<Tolerable>` | Number of small packets in the cache |
| start | partition | 0 | First block that is claimed by the partiton (interpreted in discdevice's addressing size) |
| syncop | fileserver | 0 | Switch to toggle between partly (= 0) or full (= 1) synchronous mode |
| type | drive | `<None>` | Drive's type (refering to `scsiinfo` entires) |
| | volume | structured | Volume's type |
| volume | fileserver | `<None>` | Start of a volume block |

# Appendix B

# Errors

There are several error messages that may occur when starting/running the file server or any of the utilitiy commands. We differ between fatal error conditions that lead to stopping a program (e g not enough memory for the file server and its buffer cache) and normal errors/warnings that result in non performing of command (e g a full disc) but keep the file server running. Most of the messages explain theirselves, here are the most important ones:

| | |
|---|---|
| Text: | `Failed to load devinfo.` |
| Description: | The file server did not find the device information file that usually has to be placed in `/helios/etc` or given as parameter. |
| Corrective measures: | Create the required file by editing the `/helios/etc/devin-fo.src` file and compiling that via the `gdi` command. |

| | |
|---|---|
| Text: | `Failed to find filesystem info.` |
| Description: | The value of the last parameter of `fs` and the value of the *fileserver* keyword in your `devinfo` are not equal. |
| Corrective measures: | Reenter your `fs` call (if there was an error in typing) or correct the value in your `devinfo` (don't forget to use the `gdi` command afterwards). |

| | |
|---|---|
| Text | `Failed to find discdevice info.` |
| Description: | In your `devinfo` there's no discdevice block corresponding to the value you gave to *device* in the *fileserver* block. |
| Corrective measures: | Complete your `devinfo` and recompile it (use `gdi`). |

Text `Failed to open device.`

Description: The `OpenDevice()` call couldn't succeed.

Corrective measures:

- Usually, the device driver's name as written in `devinfo`'s discdevice block (following the name keyword) does not correspond to an existing file.

  **NOTE:**
  The name should be `msc21.dev` and is expanded to `/helios/lib/msc21.dev`. Assert that this file contains your device driver.

- Maybe you didn't start the file server on a MSC node. The correct call is

  ```
  % remote -d MSC fs msc21
  ```

- You have an old MSC board in use and didn't perform an update of your hardware. See section 2.2 for further information.

- Else there could be problems with your SCSI devices or the SCSI bus. Check your hardware and try it again.

Text `Failed to to init device info for fileserver.`

Description: Have a look at the server window to get detailed information.

Corrective measures: Correct the errors by editing your `devinfo` or clearing the memory

Text: `Not enough memory for PFS v2.1.`
or
`Failed to allocate memory for server.`

Description: There was not enough memory for the file server's buffer cache etc.

Corrective measures:

- Have a look at your `devinfo` file. Maybe your buffer cache appotionment via *smallcount*/*mediumcount*/*hugecount* results in asking for more memory than the MSC board offers. Correct the values.

- You've already started a file server and some difficulties occured, so that you wanted to restart the file server, but it's still in memory. Note that you have to terminate **all** volumes, even though one of them was not loaded.