FUJITSU COMPUTERS SIEMENS

Reliant UNIX ONLINE Documentation

Reliant UNIX 5.44

Asynchronous Disk Access

Edition September 1997

Identification: U25613-J-Z915-1-7400

Copyright and Trademarks

1 Preface title>

This manual explains how to install and use the AIO interface from Siemens Nixdorf (AIO = Asynchronous I/O).

With most UNIX® operating systems, a process has to wait after an I/O request has been issued until this request has been processed. However, if you are using the AIO interface on your system under Reliant UNIX®, a process can continue operating after an I/O request has been issued. In this way, you avoid bottlenecks being created as a result of a number of write operations, which would negatively affect the throughput. This is a particularly useful feature with DBMS systems which only use one backend process for disk I/O: If this type of process had to wait with every write operation, this would severely impair the performance of the entire database.

The AIO interface provides the processes with a simple alternative, i.e. the possibility to perform asynchronous read and write operations for a specific disk in raw mode. A process can start an I/O operation and continue running without being blocked until the I/O operation has been completed. The process then later has to query the status of all previous I/O requests issued.

The process uses the same devices (including virtual devices) that would normally also be used for disk I/O in raw mode. However, the *ioctl*(2) command must be called in order to actually issue the I/O request and to query the status of the previously issued requests.

Furthermore, it is possible to receive asynchronous notification of completed I/O requests. The advantage of this is that completed requests do not have to be polled, rather notifications are generated for all requests. If this feature is activated when a request is issued, the *SIGEMT* signal is sent to the relevant process following the request.

1.1 Target group

This manual is intended for system administrators and application or system programmers who already have experience of using and administering the Reliant UNIX operating system.

1.2 Changes since the last version of this manual

- New version number *AIOVERSION* = 3
- The maximum number of AIO requests can be configured using the *AIO_MAX* tuning parameter.

1.3 Summary of contents

The next chapter describes the procedure for installing the AIO software on the system, how AIO is used as well as how it is invoked with *ioctl* control functions. Details are also provided of the AIO error messages. The final chapter contains programming examples in which the AIO interface is used to improve performance in different situations.

1.4 Related publications

Further information on the AIO interface and the Reliant UNIX operating system can be found in the following manuals:

- System Administrator's Guide
- System Administrator's Reference Manual: *aio*(7), *sdisk*(7) and *vdisk*(7)
- Programmer's Reference Manual: *ioctl*(2), *fcntl*(2), *read*(2), *write*(2), *fcntl*(5)

1.5 Notational conventions

The following notational conventions are used throughout this manual:

Italics	in the main body of text denote file names, programs, commands, variables, options, and software references such as input fields, text fields, menus, variables in

	examples, etc. (e.g. *filename)*
Fixed-pitch type	denotes system output (error messages as well as other messages) and file entries
	provides additional information and tips

2 Installing and using AIO

2.1 Loading the AIO software package

The AIO interface is shipped on tape or CD-ROM. The software is loaded like any other optional software package under Reliant UNIX. The *pkgadd*(1M) command, which was supplied with the tool for installing the Reliant UNIX operating system, is used to install the package. You have to invoke the following command, for example:

pkgadd aio.

At the prompt, type *y*:

Do you wish to rebuild the kernel after *specaio* installation?□
(default: n) [y,n,?]

The installation tool automatically regenerates the system kernel of the Reliant UNIX operating system with the AIO interface. You should refer to the documentation for system administrators for further details of how to install software packages.

You can also manually regenerate the Reliant UNIX kernel. To do this, use the *idbuild*(1M) command. If you decide in favor of this option, you have to ensure that the *sdevice* file is processed (replace *N* with *Y*), so that *specaio* will be activated.

2.2 Configuring the AIO software

As before, all relevant configuration parameters are defined with default values following installation of the AIO software package.

In order to ensure greater flexibility, some new tuning parameters have been introduced which allow individual adaptations to be made if required. These are the parameters:

AIO_VERSION, AIO_HIWAT, AIO_LOWAT, AIO_MAX

The current default values for the tuning parameters can be found in the *mtune* file in the */etc/conf/cf.d* directory.

The most important parameter in this context is *AIO_MAX*. *AIO_MAX* defines the maximum number of asynchronous I/O command structures that can be assigned dynamically in the operating system. Each active asynchronous I/O command occupies precisely one command structure. When the maximum number has been reached, and no more free command structures are available, the *EWOULDBLOCK* error message is issued. Should an application require more than the standard number of parallel asynchronous I/O commands defined, *AIO_MAX* can be set to a higher value.

Further information on the remaining parameters and on configuring the operating system can be found in the Tuning Guide.

2.3 Using AIO

A general algorithm follows which describes the correct use of asynchronous I/O:

1. The process can define whether the AIO interface is supported for a specific file descriptor (with the *DKIOCAIOVERS* control function of the *ioctl* command).
2. The process assigns buffer space for transferring data from and to disk.
3. The *FRAIOSIG* flag can be set optionally for asynchronous notification and a signal handling routine set up for *SIGEMT*.
4. The process now issues several I/O requests with the *DKIOCASTRT* control function of the *ioctl* command. The status of these requests is then polled with the *DKIOCASTAT* control function of the *ioctl* command.

2.4 Control functions of the ioctl command

If the AIO interface is to be used instead of regular I/O, the I/O interface of the user program has to be modified. The program has to contain the following definition file in any case:

```
#include <sys/async.h>
```

The following control functions are available for processing AIO requests (see *aio*(7)).

DKIOCAIOVERS

This control function determines whether the AIO interface is supported by the respective file descriptor and whether the system kernel version for the AIO interface corresponds to the interface used by the user program.

```
long version;
ioctl(fd, DKIOCAIOVERS, &version);
```

If the *ioctl* system call returns the value 0, the AIO version contains the number for the standard version of the AIO interface (*version* = AIOVERSION).

If the *ioctl* system call returns the value *-1*, either the AIO interface is not configured, or the specified file does not support AIO access.

DKIOCMLOCK

This control function is offered for reasons of compatibility. It need no longer be used because the *DKIOCASTRT* control function locks the corresponding buffer area for the individual requests. Locked memory is freed again with the *DKIOCASTAT* control function.

DKIOCASTRT

This control function is used by the process when issuing an asynchronous I/O request:

```
struct areqbuf reqbuf;
ioctl(fd, DKIOCASTRT, &reqbuf);
```

The *areqbuf* data structure contains the following information: The command to be executed, the source and target destinations for transfer to disk and in the memory as well as the scope of the data to be transmitted:

```
long au_cmd;     /* command              */
long au_daddr;   /* location on disk     */
char *au_maddr;  /* location in memory   */
```

```
long au_size;    /* bytes to transfer       */
char *au_ref;     /* caller specified reference */
```

The parameters relevant for the command are as follows: *AU_READ*, *AU_WRITE* and *AU_ORDWRITE*. The ordering of the sort queue for the disk drive can generally be handled optimally for a write request. The sequence of the physical write operations is independent of the order in which the operations were issued. The difference between *AU_WRITE* and *AU_ORDWRITE* is that with the second command, all subsequent write operations are only executed physically when the request (*ordered write*) has been completed.

If the *AU_ORDWRITE* command is used incorrectly, the disk performance may be considerably impaired. There are restrictions in using virtual disks.

The position on the disk is specified in units of 512 bytes. If, for example, the second sector on a disk is to be read with sectors of 1024 bytes in size, *au_daddr* must have the value *2* (2 * 512). If *1* is specified for *au_daddr*, the *EINVAL* error will result in this case.

The position in the memory has to be adapted accordingly for the respective device. A word alignment generally suffices for the buffer. If an incorrectly aligned address is specified for *au_maddr*, the *EIO* error results for this request. Details of specific devices can be found under *sdisk*(7).

The scope of data transmitted has to be specified as an integer in disk sectors. The value must not exceed a certain size so that it can be processed for the device with a single I/O request. You can query the size of the physical sector and the maximum size of a single request for a disk with the *DKIOCGETTYPE* control function:

```
#include <sys/types.h>
#include <sys/dkio.h>
{
struct dktype dkt;
ushort_t SECTORSIZE; /* physical sector size in bytes */
ulong_t MAXREQUEST;  /* maximum request size in bytes */
ioctl(fd, DKIOCGETTYPE, &dkt);
SECTORSIZE = dkt.dkt_bsize;
MAXREQUEST = dkt.dkt_maxbsize;
}
```

The reference value given in the command can be used to assign an end status for the original request. The value assigned is arbitrary, provided it is valid for a pointer of type 'character'. This value is not evaluated by the system kernel, but is simply returned in the *asyncstatus* data structure, which contains the end status for this request.

DKIOCASTAT

This control function uses the process to query the status of the completed I/O requests (the maximum number of requests is defined by the *MAXSTATUS* value):

```
struct asyncstatus iostat;□
ioctl(fd, DKIOCASTAT, &iostat);
```

The *asyncstatus* data structure is assigned values by the AIO driver. It comprises the number of requests for which the status is returned (if there are no completed requests, zero is returned) as well as an array with data structures of the type *IOSTAT* for each completed request. The *aiostat* data structure comprises the following four components:

```
short iostatus; /* completion status                */□
short iobsize;  /* bytes to transfer for the request  */□
char  *iomaddr; /* memory address for the  request    */□
char  *ioref;/* caller specified reference for the request*/
```

The *aiostat* data structure comprises the status returned by the driver for the respective request as well as three fields that can be used to identify the request. These relate to the size of the request issued, the main memory address (from the *struct areqbuf* argument for *DKIOCASTRT*) and an *ioref* value supplied with the call. The value of *ioref* is always identical to the value of *au_ref* in the original request. The value of *au_ref* /*ioref* is arbitrary provided it is valid for a pointer of the type 'character'.

2.5 The FRAIOSIG flag of the fcntl command

The *FRAIOSIG* flag can be set with *fcntl* if asynchronous notification is to follow for completed requests (see *fcntl*(5)).

```
#include <sys/file.h>□
#include <sys/fcntl.h>□
fcntl(fd, F_SETFL, FRAIOSIG);
```

If this flag is set for the specified file, the process receives a *SIGEMT* signal when an I/O request has been completed for each request for this file.

The use of signals is not very efficient. Short sleeps and queries by polling (with *DKIOCASTAT*) are often more efficient. You should note that polling without short sleeps or other system calls demands a very large amount of CPU time.

2.6 AIO-specific definitions

The following definitions are taken from *<sys/async.h>*. They describe the *ioctl* control functions as well as the related data structures required for AIO.

```
/*□□ AIO□version□
□*□□□□□□Return□the□version□number□of□the□current□interface□
□*/□
#define□AIOVERSION□□ □3□
#define□DKIOCAIOVERS□□_IOR('S',□4,□long)□
□
/*□Starting□I/O□(DKIOCASTRT)□
□*□□'S'□and□'1'□define□a□unique□command□number.□
□*□□Using□_IOW□specifies□that□the□parameters□are□for□input□only.□
□*□□□The□parameter□passed□back□to□SVR4□is□a□struct□areqbuf.□
□*/□
□
typedef□struct□areqbuf□{□
□□□□□□□□long□au_cmd;□□□□□/*□AU_READ,□etc□□□□□□□□□□□□□□□□*/□
□□□□□□□□long□au_daddr;□□□/*□destination□on□disk□□□□□□□□*/□
□□□□□□□□char□*au_maddr;□□/*□(virtual)□memory□address□□□*/□
□□□□□□□□long□au_size;□□□□/*□bytes□to□transfer□□□□□□□□□□*/□
□□□□□□□□char□*au_ref;□□□□/*□caller□specified□reference□*/□
}□AREQBUF;□
```

```
/*  command bits
 *  Note: AU_ORDWRITE is treated as AU_WRITE in the kernel
 *        They exist as separate flags for user convenience only.
 */
#define AU_READ       01  /* read request             */
#define AU_WRITE      02  /* unordered write request */
#define AU_ORDWRITE   04  /* ordered write request   */
#define AU_CMDMASK    07  /* mask of command bits    */
#define DKIOCASTRT  _IOW('S', 1, AREQBUF)

/*  Getting I/O Completion Status (DKIOCSTAT)
 *  'S' and '2' define a unique command number.
 *  Using _IOR specifies that the parameters are for output only
 *  The parameter passed back from SVR4 is a struct asyncstatus.
 */
typedef struct aiostat {
        short iostatus; /* I/O completion status for request   */
        short iobsize;  /* verification (from AREQBUF dbsize)  */
        char  *iomaddr; /* verification (from AREQBUF dbmaddr) */
        char  *ioref;   /* return caller specified reference   */
} IOSTAT;

#define MAXSTATUS  15

typedef struct asyncstatus {
        long   acount;
                      /* # of requests being returned  */
        IOSTAT astatus[MAXSTATUS];
  /* completion status per request */
} ASYNCSTATUS;

#define DKIOCASTAT  _IOR('S', 2, ASYNCSTATUS)

/*  Locking Memory for I/O (DKIOCMLOCK)
 *  'S' and '3' define a unique command number.
 *  Using _IOW specifies that the parameters are for input only.
 *  The parameters are passed to SVR4 as a struct asyncmlock.
 */

typedef struct asyncmlock {
        char     *avaddr;  /* starting virtual address  */
        unsigned asize;    /* size of area to be locked */
} ASYNCMLOCK;

#define DKIOCMLOCK  _IOW('S', 3, ASYNCMLOCK)

/* This is a new I/O error code - it really belongs in errno.h.
 *  However, since it is AIO specific, it is here.
 */
#define EIORESID  500  /* block not (fully) transferred */

#endif  /* _SYS_ASYNC_H */
```

2.7 Errors

The same errors arise as with normal system calls for I/O (e.g. *read*, *write*, *ioctl* etc.). You will find further information on these errors in the relevant manpages for the system call (section 2). The *EIORESID* error should also be noted. Since *EIORESID* is specific to AIO, this error is defined in the <*sys/async.h*> header file (and not in the <*errno.h*> file). A short description follows:

EIORESID

This error indicates that the driver accepts the request and has placed it in the queue. However, an error occurred subsequently which caused the failure of the I/O before the request could be completed. I/O was only partially performed or not performed at all.

3 Examples of AIO

3.1 Example 1

An example is given below of a simple C program which asynchronously reads the first six blocks of the */dev/ios0/rsdisk000s0* device with two requests.

For reasons of legibility, the code for error checking is not contained fully in the program overview.

```
#include□<stdio.h>□
#include□<sys/fcntl.h>□
#include□<sys/async.h>□
#include□<sys/errno.h>□
□
#define□COMMAND□□□□□AU_READ□
#define□PAGETOREAD□□0□
#define□PAGESIZE□□□□1024□□/*□disk□blk□size□*/□
#define DEVICE      "/dev/ios0/rsdisk000s0"□
□
AREQBUF□□□□□reqbuf[2];□□□□□□□□□□□□/*□request buffer□□*/□
ASYNCSTATUS□iostat;□□□□□□□□□□□□□□□/*□status buffer□□□*/□
char□□□□□□□□buffer[6*PAGESIZE];□□/*□transfer buffer□*/□
extern□int□□errno;□
□
main()□
{□
□□□□int□□□□□i,□fd,□ncomplete;□
□□□□long□□□□version;□
□□□□AREQBUF□*req;□
□□□□IOSTAT□□*reqstat;□
□
□□□□fd□=□open(DEVICE,□O_RDONLY);□
□
□□□□/*□Determine□if□AIO□is□supported□for□DEVICE□*/□
□□□□if□(ioctl(fd,□DKIOCAIOVERS,□&version)□<□0)□
□□□□{□
□□□□□□□□perror("ioctl□failed□on□returning□the□version□of□AIO");□
□□□□□□□□exit(1);□□/*□for□example:□AIO□not□configured□*/□
□□□□}□
if□(version□!=□AIOVERSION)□
□□□□{□
□□□□□□printf("AIO□is□not□supported□for□the□specified□device\n");□
□□□□□□exit(1);□
□□□□}□
□□□□/*□Setup□the□request□with□the□proper□parameters□*/□
□
□□□□reqbuf[0].au_cmd□□□=□COMMAND;□
□□□□reqbuf[0].au_daddr□=□PAGETOREAD;□
□□□□reqbuf[0].au_maddr□=□&buffer[0];□
□□□□reqbuf[0].au_size□□=□3*PAGESIZE;□
□□□□reqbuf[0].au_ref□□□=□(char□*)&reqbuf[0];□
□□□□reqbuf[1].au_cmd□□□=□COMMAND;□
□□□□reqbuf[1].au_daddr□=□PAGETOREAD□+□3*PAGESIZE/512;□
□□□□reqbuf[1].au_maddr□=□&buffer[3*PAGESIZE];□
□□□□reqbuf[1].au_size□□=□3*PAGESIZE;□
□□□□reqbuf[1].au_ref□□□=□(char□*)&reqbuf[1];□
□
□□□□/*□Start□to□read□*/□
□
□□□□for□(i□=□0;□i□<□2;)□
□□□□{□
□□□□□□□□if□(ioctl(fd,□DKIOCASTRT,□&reqbuf[i])□<□0)□
□□□□□□□□{□
```

```
□□□□□□□□□□□□□if□(errno□==□EWOULDBLOCK)□
□□□□□□□□□□□□□□□□□continue;□
□□□□□□□□□□□□□perror("ioctl□failed□on□DKIOCASTRT");□
□□□□□□□□□□□□□exit(1);□
□□□□□□□□}□
□□□□□□□□i++;□
□□□□}□
□
□□□□/*□This□is□the□point□where□the□process□normally□would□do□
*□something□else,however,□for this□example□we□will□just□a□
*□dummy□routine.□The function□must□return□at□some□point□to□
*□pick□up□the status□of□the□sent□requests.□
□□□□□*□
□□□□□ * -->□No□notification□of□completed□requests□is□given□!!□
□□□□□ */□
□□□□ncomplete□=□0;□
□
another_completion:□
□
□□□□iostat.acount□=□0;□
□□□□if□(version□!=□AIOVERSION)□
□□□□{□
□□□□□□printf("AIO□is□not□supported□for□the□specified□device\n");□
□□□□□□exit(1);□
□□□□}□
□
□□□□/*□Setup□the□request□with□the□proper□parameters□*/□
□
□□□□reqbuf[0].au_cmd□□□=□COMMAND;□
□□□□reqbuf[0].au_daddr□=□PAGETOREAD;□
□□□□reqbuf[0].au_maddr□=□&buffer[0];□
□□□□reqbuf[0].au_size□□=□3*PAGESIZE;□
□□□□reqbuf[0].au_ref□□□=□(char□*)&reqbuf[0];□
□□□□reqbuf[1].au_cmd□□□=□COMMAND;□
□□□□reqbuf[1].au_daddr□=□PAGETOREAD□+□3*PAGESIZE/512;□
□□□□reqbuf[1].au_maddr□=□&buffer[3*PAGESIZE];□
□□□□reqbuf[1].au_size□□=□3*PAGESIZE;□
□□□□reqbuf[1].au_ref□□□=□(char□*)&reqbuf[1];□
□□□
□□/*□Start□to□read□*/□
□
□□□□for□(i□=□0;□i□<□2;)□
□□□□{□
□□□□□□□□□if□(ioctl(fd,□DKIOCASTRT,□&reqbuf[i])□<□0)□
□□□□□□□□□{□
□□□□□□□□□□□□□if□(errno□==□EWOULDBLOCK)□
□□□□□□□□□□□□□□□□□□continue;□
□□□□□□□□□□□□□perror("ioctl□failed□on□DKIOCASTRT");□
□□□□□□□□□□□□□exit(1);□
□□□□□□□□□}□
□□□□□□□□□i++;□
□□□□}□
□
□□□□/*□This□is□the□point□where□the□process□normally□would□do□
*□something□
□    * else,□however,□for this□example□we□will□just□a□
*□dummy□routine.□The function□must□return□at□some□point□to□
□    * pick□up□the status□of□the□sent□requests.□
□□□□□*□
□□□□ □* -->□No□notification□of□completed□requests□is□given□!!□
□□□ □□*/□
□□□□ ncomplete□=□0;□
□
another_completion:□
```

```
□
□□□□iostat.acount□=□0;□
□
□□□□while(iostat.acount□==□0)□
□□□□{□
□□□□□□□□do_nothing();□
□□□□□□□□if□(ioctl(fd,□DKIOCASTAT,□&iostat)□<□0)□
□□□□□□□□{□
□□□□□□□□□□□□□□perror("ioctl□failed□on□DKIOCASTAT");□
□□□□□□□□□□□□□□exit(1);□
□□□□□□□□}□
□□□□}□
□
□□□□reqstat□=□&iostat.astatus[0];□
□□□□while□(iostat.acount--)□
□□□{□
□□□□□req□=□(AREQBUF□*)□reqstat->ioref;□
□□□□□/*□
□□□□□printf("ioref.au_daddr□=□0x0%x\n",□req->au_daddr);□
□□□□□printf("ioref.au_maddr□=□0x0%x\n",□req->au_maddr);□
□□□□□*/□
□
□□□□□/*□check□for□any□errors/inconsistencies□*/□
□
□□□□□if□(reqstat->iobsize□!=□req->au_size)□
□□□□□□□printf("READ□FAILED:□iobsize□=□%d,□expected□=□%d\n",□
□□□□□□□□□□□□□□□reqstat->iobsize,□req->au_size);□
□□□□□else□if□(reqstat->iomaddr□!=□req->au_maddr)□
□□□□□□□printf("READ□FAILED:□iomaddr□=□0x0%x,□expected□0x%x\n",□
□□□□□□□□□□□□□□□reqstat->iomaddr,□req->au_maddr);□
□□□□□else if (reqstat->iostatus)□
□□□□□□printf("READ FAILED: iostatus = %d\n",□reqstat->iostatus);□
□□□□□else□
□□□□□□□printf("SUCCESSFULLY READ\n");□
□□□□□ncomplete++;□
□□□□□reqstat++;□
□□□}□
□□□if□(ncomplete□<□2)□
□□□□□□□goto□another_completion;□
□□□close(fd);□
□□□exit(0);□
}□
do_nothing()□
{□
□□□□sleep(1);□
}
```

3.2 Example 2

The second example also involves a C program which contains a very powerful copy routine with a simple, sequential copy procedure.

```
□
#include□<stdio.h>□
#include□<sys/fcntl.h>□
#include□<sys/errno.h>□
#include□<sys/ioctl.h>□
#include□<sys/async.h>□
□
#define□COMMAND□□□□AU_READ□
#define□DEVICE□□□□□"/dev/ios0/rsdisk000s0"□
#define□SECTORSIZ□□512□□/* disk□sector□size□□□□□□□□□□□□*/□
#define□READSIZ□□□□1□□□□/* number□of□sectors□per□read□*/□
#define□HIWAT□□□□□□10□□□/* read-ahead□hi□water□mark□□□*/□
#define□LOWAT□□□□□□5□□□□/* lo□water□mark□□□□□□□□□□□□□□□*/□
□
int□seqread(int□fd,□char□**pbuf);□
□
main()□
{□
□□int□□cc,□fd;□
□□char□*buf;□
□□long□version;□
□
□□fd□=□open(DEVICE,□O_RDONLY);□
□
□□/*□Determine□if□AIO□is□supported□for□DEVICE.□*/□
□
□□if□(ioctl(fd,□DKIOCAIOVERS,□&version)□<□0□||□version□!AIOVERSION)□
□□{□
□□fprintf(stderr,"mismatched□version□or□AIO□not□supported□on□%s\n",DEVICE);□
□□exit(1);□
□□}□
□□while□((cc□=□seqread(fd,□&buf))□>□0)□□/*□copy□to□stdout□*/□
□□□□□□write(1,□buf,□cc);□
□
□□if□(cc□<□0)□
□□{□
□□□□perror("seqread");□
□□□□exit(1);□
□□}□
□□exit(0);□
}□
□
/*□
*□High□performance□sequential□read□with□readahead□using□AIO.□Maintain□
*□at□least□LOWAT□requests□outstanding.□Issue□as□many□as□HIWAT□requests□
*□at□one□time.□
*/□
□
seqread(int□fd,□char□**pbuf)□
{□
□□□□static□int□□issue_seq□=□0,□collect_seq□=□0;□
□□□□static□int□□outstanding□=□0;□□□□□□□□
/*□#□of□requests□started□but□not□done□*/□
□□□□static□int□□daddr = 0;□
□□□□□□□□□□□□/*□current□block□offset□in□file□□□□□□□*/□
□□□□static□char□buffer[HIWAT][READSIZ*SECTORSIZ];□
□□/*□transfer□buffer□□□□□□□*/□
□□□□static□enum□{IDLE,□WAIT,□DONE}□busy[HIWAT]□
/*□state□of□associated□buffer□*/□
```

```
□□□□static□int□□cc[HIWAT];□
□□□□□□□□□□□□/*□character□count□for□each□request□□*/□
□□□□static□int□□Eof□=□0;□
□□□□□□□□□□□□/*□a□read□error□has□occured□□□□□□□□□□□*/□
□
□□□□IOSTAT□□□□□□□*reqstat;□
□□□□AREQBUF□□□□□□reqbuf;□□/*□request□buffer□*/□
□□□□ASYNCSTATUS□iostat;□□/*□status□buffer□□*/□
□
□□□□int□i,□done_ref,□ret□=□0;□
□□□□extern□int□errno;□
□
□□□□/*□see□if□it□is□okay□for□us□to□issue□a□few□requests□*/□
□
□□□□if□(!Eof□&&□outstanding□<□LOWAT)□
□□□□{□
□□□□□□□□while□(outstanding□<□HIWAT□&&□busy[issue_seq□%□HIWAT]□==□IDLE)□
□□□□□□□□{□
□□□□□□□□□□□□□□reqbuf.au_cmd□□□=□COMMAND;□
□□□□□□□□□□□□□□reqbuf.au_daddr□=□daddr;□
□□□□□□□□□□□□□□reqbuf.au_maddr□=□&buffer[issue_seq % HIWAT][0];□
□□□□□□□□□□□□□□reqbuf.au_size□□=□READSIZ*SECTORSIZ;□
□□□□□□□□□□□□□□reqbuf.au_ref□□□=□(char□*)issue_seq;□
□
□□□□□□□□□□□□□□if□(ioctl(fd,□DKIOCASTRT,□&reqbuf)□<□0)□
□□□□□□□□□□□□□□{□
□□□□□□□□□□□□□□□□□□if□(errno□==□EWOULDBLOCK)□
□□□□□□□□□□□□□□□□□□□□□□break;□
□□□□□□□□□□□□□□□□□□return(-1);□
□□□□□□□□□□□□□□}□
□□□□□□□□□□□□□□daddr□+=□READSIZ*SECTORSIZ/512;□
□□□□□□□□□□□□□□busy[issue_seq□%□HIWAT]□=□WAIT;□
□□□□□□□□□□□□□□outstanding++;□
□□□□□□□□□□□□□□issue_seq++;□
□□□□□□□□}□
□□□□}□
□
□□□□/*□see□if□any□requests□are□done□*/□
□
□□□□if□(collect_seq□<□issue_seq)□
□□□□{□
□□□□□□□□while (busy[collect_seq % HIWAT]□==□WAIT)□
□□□□□□□□{□
□□□□□□□□□□□□□□if□(ioctl(fd,□DKIOCASTAT,□&iostat)□<□0)□
□□□□□□□□□□□□□□□□□□return(-1);□
□
□□□□□□□□□□□□□□reqstat□=□&iostat.astatus[0];□
□□□□□□□□□□□□□□for□(i□=□0;□i□<□iostat.acount;□i++)□
□□□□□□□□□□□□□□{□
□□□□□□□□□□□□□□□□□□if□(reqstat->iostatus)□
□□□□□□□□□□□□□□□□□□□□□□Eof = 1;□
□□□□□□□□□□□□□□□□□□done_ref□=□(int)reqstat->ioref□%□HIWAT;□
□□□□□□□□□□□□□□□□□□cc[done_ref]□=□reqstat->iobsize;□
□□□□□□□□□□□□□□□□□□busy[done_ref]□=□DONE;□
□□□□□□□□□□□□□□□□□□outstanding--;□
□□□□□□□□□□□□□□□□□□reqstat++;□
□□□□□□□□□□□□□□}□
□□□□□□□□}□
□□□□□□□□*pbuf□=□&buffer[collect_seq□%□HIWAT][0];□
□□□□□□□□busy[collect_seq□%□HIWAT]□=□IDLE;□
□□□□□□□□ret□=□cc[collect_seq□%□HIWAT];□
□□□□□□□□collect_seq++;□
□□□□}□
□□□□return(ret);□
```

```
}
```