Software and Utilities for Trident Disks: Tfs, Tfu, Triex

1. Introduction

This document describes Bcpl-based software for operating any of the family of Trident disk drives attached to an Alto using a "Trident controller card" (the software presently deals with the T-80 and T-300 models). Hardware and diagnostic information can be found in the document "Trident disk for the Alto" (on <ALTODOCS>TRIDENT.EARS), by Roger Bates.

The software documentation is divided into three parts: (1) a brief "how-to" section describing the software package available for operating the Trident; (2) a section describing two useful utility programs, Tfu and Triex; and (3) a section describing the software package in more detail. There is a short revision history at the end.

The Tfs package and utilities all assume that the disk is to be formatted with 9 sectors per track, 1024 data words per sector. Thus a T-80 disk has a capacity (815 tracks, 5 surfaces, 9 sectors, 1024 words per sector) of 36,675 pages or 37,555,200 words. A T-300 (19 surfaces rather than 5) has a capacity of 139,365 pages or 142,709,760 words; however, due to the restriction of virtual disk addresses to 16 bits, a single file system may utilize only about 47 percent of this capacity, and it is necessary to construct multiple file systems in order to make use of the entire disk.

Because of bandwidth limitations, it is unwise to operate the Trident disk while the Alto display is on. Although the Tfs package will save the display state, turn it off, run the disk, and restore the display for every transfer, the user may prefer to turn the display off himself. The Tfs management of the display causes the screen to flash objectionably whenever frequent calls to Tfs are underway.

2. Trident File System (Tfs) software package

The software for operating the Trident disk is contained in <Alto>Tfs.Dm, and consists of the following relocatable files: TfsInit.Br, TfsBase.Br, TfsA.Br, TfsWrite.Br TfsCreate.Br, TfsClose.Br, TfsDDMgr.Br, TfsNewDisk.Br, TfsSwat.Br, and TriConMc.Br. The definitions file Tfs.D is also included. The LoadRam.Br file, formerly included as part of the Tfs, is now available as a separate package.

2.1. Initializing the microcode

Operating the Trident requires special microcode that must be loaded into the RAM before disk activity can start. The procedure LoadRam will load the RAM from a table loaded into your program (it is actually
↑L

part of TriConMc.Br). It will then "boot" the Alto in order to start the appropriate micro-tasks in the RAM. (This booting process is "silent" -- it does not re-load Alto memory from the file Sys.Boot, but instead lets your program continue.) The standard way to call LoadRam to load the Trident disk microcode is:

```
external DiskRamImage
external LoadRam

let result=LoadRam(DiskRamImage, true)  //Load and boot
if result ls 0 then
        [
        Ws("The Alto has no RAM or Ethernet board.")
        Ws(" Cannot operate Trident")
        finish
        ]
```

After LoadRam has returned successfully, the code of LoadRam and TriConMc may be overlaid with data -- they are no longer needed.

When exiting a program that has micro-tasks active in the RAM, it is helpful to "silently" boot the Alto so that all micro-tasks are returned to the ROM. If this is not done, subsequent use of the RAM may cause some running micro-task to run awry. To achieve the "silent boot," simply call the procedure TFSSilentBoot() at 'finish' time or as part of a 'user finish procedure'.

For further information, consult the LoadRam package documentation.

2.2. Initializing the Trident drive

Once the RAM has been loaded, the Trident disk can be initialized. The procedure TFSInit will do this, provided that a legal file structure has previously been established on the drive (see Tfu Erase, below). The procedure returns a "disk object," a handle which can be used to invoke all the disk routines. This disk object (or "disk" for short) can be passed to various Alto Operating System procedures in order to open streams on Trident disk files, delete Trident disk files, etc.

```
tridentDisk = TFSInit(zone, allocate [false], driveNumber [0], ddMgr
        [0], freshDisk [false])
```

zone — You must provide a free-storage pool from which memory for the disk object and possibly for a buffer window on the disk bit table can be seized. The zone must obey the normal conventions (see Alto Operating System Manual); zones created by InitializeZone are fine.

allocate — This flag is true if you wish the machinery for allocating or de-allocating disk space enabled. If it is enabled, a small DDMgr object and a 1024-word buffer will be extracted from the zone in order to buffer the bit table (unless you supply a ddMgr argument, described below).

driveNumber This argument, which defaults to 0, specifies the number of the Trident disk drive being initialized. If the drive is a T-300, the left-hand byte specifies the number of the file system to be accessed on that drive, in the range 0 to

↑L

2. (For further information, consult the section entitled 'Disk Format'.)

ddMgr This argument, which defaults to 0, supplies a handle on a 'DiskDescriptor Manager' (DDMgr) object, whose responsibility it is to manage pages of the DiskDescriptor (bit table), which, on the Trident, must be paged into and out of memory due to its considerable size. If this argument is defaulted, a separate DDMgr will be created upon each call to TFSInit, at a cost of a little over 1024 words. If you intend to have multiple Trident drives open simultaneously, you may conserve memory by first issuing the call 'ddMgr = TFSCreateDDMgr(zone)' and then passing the returned pointer as the ddMgr argument in each call to TFSInit, thereby permitting the single ddMgr to be shared among all drives. (This argument is ignored unless the allocate argument is true.)

freshDisk Normally, TFSInit attempts to open and read in the DiskDescriptor file in order to obtain information about the file system. However, if freshDisk is true, this operation is inhibited and the corresponding portions of the disk object are set up with default values. This operation is essential for creating a virgin file system.

tridentDisk The procedure returns a disk object, or 0 if the Trident cannot be operated for some reason. The most likely reasons are:

1. No Trident disk controller plugged into the Alto.

2. No such disk unit, or disk unit not on-line.

3. Can't find SysDir, can't open DiskDescriptor, or DiskDescriptor format is incompatible. (These errors can't happen if freshDisk is true.)

After TFSInit has been executed, the code can be overlaid, as it is not used for normal disk operation.

2.3. Closing the Trident disk

When all operations on the disk are completed, the TFSClose procedure will insure that any important state saved in Alto memory is correctly written on the disk. This step can be omitted if the 'allocate' argument to TFSInit was false (assuming you don't mind the loss of the storage that was extracted from 'zone' by TFSInit).

```
TFSClose(tridentDisk, dontFree [false])
```

The second argument is optional (default=false), and if true will not permit the DiskDescriptor Manager (DDMgr) to be destroyed. This option is useful in conjunction with the 'ddMgr' argument to TFSInit.
↑L

2.4. Example

Following is an example that uses the Trident disk system and demonstrates the procedures described above. Note that the calls on operating system disk stream routines all pass a private zone to use for stream structures, rather than the default sysZone. The reason is that streams on Trident disks require large buffers (1024 words) which quickly exhaust the available space in sysZone. In addition, the stream routines will consume more stack space when operating the Trident disk than they do when operating the standard Alto disk.

Since the Alto OS does not know about Trident disks, a call to Swat will not properly wait for all Trident transfers to complete, with consequent undefined results. This problem is easily remedied through use of an assembly-language Swat context-switching procedure TFSSwat, which is included as part of the TFS package. The example shows how it is set up.

```
//Example.bcpl -- TFS Example
//Bldr Example TfsBase TfsA TfsWrite TfsCreate TfsClose TfsDDMgr
//  TfsSwat TfsInit LoadRam TriConMc

get "streams.d"

external [
        TFSInit
        TFSClose
        TFSSilentBoot
        LoadRam
        DiskRamImage

        OpenFile
        Closes
        Puts
        DeleteFile

        InitializeZone
        SetEndCode
        TFSSwatContextProc
        lvUserFinishProc
        lvSwatContextProc
        ]

static [ savedUFP; savedSCP; TFSdisk = 0 ]

let TryIt() be
[
        let driveNumber=0
        let zonevec= vec 3000
        let TFSzone = InitializeZone(zonevec, 3000)

//Initialize the RAM:
        let res=LoadRam(DiskRamImage, true)
        if res ls 0 then [ Ws("Cannot load the RAM."); finish ]
```

```
//Set up to cleanly finish or call swat
        savedUFP = @lvUserFinishProc
        @lvUserFinishProc = MyFinish
        savedSCP = @lvSwatContextProc
        @lvSwatContextProc = TFSSwatContextProc
↑L
```

```
//Initialize the disk:
        TFSdisk = TFSInit(TFSzone, true, driveNumber)
        if TFSdisk eq 0 then
                [ Ws("Cannot operate Trident disk"); finish ]

//Reclaim space used by initialization code:
        SetEndCode(TFSInit)     //Overlay TFSinit, LoadRam, TriConMc

//Now we are ready to operate the disk:
        DeleteFile("Old.Bad", 0, 0, TFSzone, 0, TFSdisk)

        let s=OpenFile("New.Good", ksTypeReadWrite, 0,0,0,0,
                                TFSzone, 0, TFSdisk)

        for i=1 to 1000 do
         for j=1 to 1000 do Puts(s, $a) //Write a million bytes!

        Closes(s)

        finish
]

and MyFinish() be
[
        if TFSdisk ne 0 then TFSClose(TFSdisk)
        @lvUserFinishProc = savedUFP
        @lvSwatContextProc = savedSCP
        TFSSilentBoot()
]
```

3. The Tfu and Triex utilities

3.1. Tfu

The Tfu utility (saved on <Alto>Tfu.Run) is used to initialize a Trident pack with a virgin file system and to perform various file copying, deleting, directory listing operations. Commands are given to Tfu on the command line: immediately following the word "Tfu" is a sub-command name (only enough characters of a sub-command are needed in order to distinguish it from other sub-commands), followed by optional arguments. Several subcommands may appear on one command line, separated by vertical bars. Thus "TFU Drive 1 | Erase" will erase drive 1. There must be a space on each side of the vertical bar.

In what follows, an "Xfile" argument is a filename, perhaps preceded by a string that specifies which disk is to be used:

```
        s:name.extension        -- use standard Alto system disk
        tn:name.extension       -- use Trident drive n (n=0 to 7)
        name.extension          -- use default disk (Trident)
```

The "default disk" is always a Trident drive; the identity of the drive is set with the Drive command.

```
TFU DRIVE driveNumber
↑L
```

This command sets the default Trident drive number to use for the remainder of the command line. The default drive is effectively an 'argument' to the ERASE, DIRECTORY, CONVERT, and BAD commands. (On a T-300, file systems 0, 1, and 2 are specified as 'x', '40x', and '100x', where 'x' is the actual unit number.)

TFU ERASE [tracks]

This command re-initializes the default Trident drive, after asking you to confirm your destructive intentions. The tracks argument specifies how many tracks of the drive are to be included in the file system; it defaults to the maximum possible. If smaller numbers are used, the initialization is correspondingly faster. In any case, tracks beyond the one specified are available for use outside the confines of the file system. (Note that one "track" is 45 pages; this corresponds to one cylinder on a T-80 and to nothing in particular on a T-300.)

TFU COPY Xfile ← Xfile

This command copies a file in the direction of the arrow. The destination file may be optionally followed by the switch /C, in which case (provided it is a Trident disk file), the file will be allocated on the disk at consecutive disk addresses. (Note: More precisely, an attempt will be made to perform such an allocation. If the attempt fails, you will sometimes get an error message. The best way to verify that a file is contiguous is to use the "address" command, below.)

TFU CREATEFILE Xfile pages

This command creates a contiguous file named Xfile with length "pages."

TFU DELETE Xfile

This command deletes the given file.

TFU DIRECTORY [Xfile]

This command lists the directory of the default Trident drive on the file Xfile; if Xfile is omitted, each entry will be typed on the display. When the display fills up or the listing is finished, Tfu waits for you to type any character before proceeding. A somewhat more verbose listing can be achieved with TFU DIR/V.

TFU BAD virtualDA

This command marks a given virtual disk address (on the default drive) as "used," so that the file system will not endeavor to assign this page to a new file. This command has become obsolete and its use is not recommended because it does not record the address in all the appropriate places; it merely marks it in the bit table. There will shortly be available a version of the Triex program that automatically records bad pages that it discovers.

```
TFU ADDRESS Xfile
↑L
```

This command reads the entire file and prints a list (in octal) of virtual disk addresses of the file pages. Typing any character will proceed to the next output line.

TFU CONVERT

An incompatible change in the format of DiskDescriptor was made in the Tfs release of July 24, 1977. The current Tfs software will refuse to access Trident disks written in the old format (specifically, TFSInit will return zero). The TFU CONVERT command reformats the DiskDescriptor to conform to current conventions (it is a no-op if applied to a disk that has already been converted). Once you have converted all your Trident disks, you should take care to get rid of all programs loaded with the old Tfs, since the old Tfs did NOT check for version compatibility.

TFU EXERCISE passes drive drive drive ...

This command embarks on a lengthy "exercise" procedure; it is repeated 'passes' times (default=10), and uses the disk drives listed after 'passes' (if none are specified, all drives that are on-line are used). It operates by making a series of files (test.001, test.002 etc.) on the disk packs, and performing various copying, deleting, writing and positioning operations. The files are deleted when the exercise finishes. It is not essential that the packs be fully erased initially; the procedure for building test files will try to fill up the disk, just short of overflowing. The test takes 20 to 30 minutes per full pack per pass.

One or more of the following global switches may be specified (i.e., a command of the form TFU/switch EXER...):

/W Use a systematic data pattern when writing files, rather than arbitrary garbage.

/C Carefully check the data read from the disk (implies /W). Use of this switch makes the test run considerably slower than normal.

/D Leave the display on during Trident disk transfers. This causes data late errors to occur and thereby exercises the error recovery logic.

/E Turn the Ethernet on during Trident disk transfers, with results similar to /D.

3.2. Triex

The documentation in this section is almost entirely obsolete. Words of wisdom from Roger Bates will be inserted here when they become available.

The Triex "Trident disk exerciser" program can be found on <Alto>Triex.Run. This program is designed for initializing and testing new disk packs before using with Tfu and Tfs. The program has essentially three operations:
↑L

1) Initialize headers and labels for use by Tfu.

2) Writing and reading of random data over the entire disk for locating and recording of sectors on the disk pack which have "bad" spots on them.

3) Writing of headers when you think just the header has been clobbered.

The program is run by typing commands as appropriate once the program is running. An example of the sequence for a new disk pack is:

```
TRIEX

Header initialization (yes) cr
Data testing
   number of repeats (10) cr

Quit (yes) cr

GEARS packerr
```

This sequence will first write headers and labels over the entire disk pack and then read them back to verify the headers. Next, multiple passes are made over the disk where random data is written on the entire disk and read back. Any disk errors are stored in the table and written into the file on the standard Alto disk. A record of the pack errors can be saved by quitting Triex and printing "packerr." The bad spots found while running Triex can be "mapped out" with the "Tfu Bad" command, using the virtual disk address reported in the file packerr. A record of bad spots should be kept for future reference so they can be re-mapped out should it be necessary to rebuild the entire file system with "Tfu Erase."

Each pass over the entire disk takes 90 seconds for writing and 90 seconds for reading. In the above example, Header initialization takes 3 minutes and 10 passes of Data testing takes 30 minutes.

Whenever Triex is running the disk, the screen must be turned off due to bandwidth considerations. In this mode, the cursor displays a "W" while writing and an "R" during reading; the vertical position represents the cylinder address being tested; and the horizontal position indicates the proportion of passes toward completion. Any operation can be suspended by typing any key. When this is done, error messages and status information will become visible. The particular operation can be terminated or continued by typing "N" or any other key.

4. The Tfs software package in more detail

If programmers wish to interface the the Trident disk at levels lower than Operating System streams, the Tfs package provides an additional interface. The "disk" object created by TFSInit has a number of abstract operations defined on it, which the Tfs package implements. Documentation for these operations can be found in the Alto Operating System Manual in the section labeled "Disks and Bfs." The catalog of available procedures is:
↑L

```
In TfsBase.Br and TfsA.Br:
        ActOnDiskPages(disk, CAs, DAs, ....)
        RealDiskDA(disk, vda, ....)
        VirtualDiskDA(disk, ....)

In TfsWrite.Br:
        WriteDiskPages(disk, CAs, DAs, ....)
        AssignDiskPage(disk, vda)*

In TfsCreate.Br
        CreateDiskFile(disk, name, ....)*
        DeleteDiskPages(disk, CA, ....)*
        ReleaseDiskPage(disk, vda)*
```

The items with *'s following may be invoked only if the disk object was created with the 'allocate' argument set to true. WriteDiskPages may be invoked even if 'allocate' is false, provided it never allocates new disk space. It should be noted that the standard Alto Streams package invokes WriteDiskPages even for files opened for reading only, and that TFSInit uses Streams to read in the DiskDescriptor. Hence it is necessary that all of the Tfs modules (TfsBase, TfsA, TfsWrite, TfsCreate, and TfsDDMgr) be loaded in order to avoid undefined 'external' references. However, after initialization is complete, the space occupied by TfsCreate and TfsDDMgr may be reclaimed if you do not intend to allocate or delete pages, and TfsWrite may be discarded if you are not using streams but rather are calling ActOnDiskPages directly.

The TfsWrite and TfsCreate modules require that TfsDDMgr.Br (or some equivalent) be loaded. This module provides the standard primitives necessary for managing the DiskDescriptor. The DDMgr is an 'object', so it may be replaced by one of your own devising so long as it provides equivalent operations. An example of this would be to manage pages of the DiskDescriptor as part of a more general virtual memory mechanism (perhaps through use of the Alto VMem package). A complete description of the required DDMgr operations may be found as comments at the beginning of TfsDDMgr.Bcpl.

In addition to the standard "actions" defined in Disks.d, Tfs permits the following. These actions are defined in Tfs.d and are available only on Trident disks.

```
        DCreadLnD   Read header, read label, no data.

        DCreadnD    Check header, check label, no data.

        DCwriteLnD  Check header, write label, no data.
```

These actions neither read nor write the data record and therefore do not require a buffer to be provided.

CreateDiskFile has a special feature for operating the Trident disks -- an optional seventh argument. If this argument (pageBuf) is present, it is assumed to point to a 1024-word buffer that will be used to create the leader page for the file. This feature may be used to save stack space in CreateDisk file and/or to write interesting data into the portion of the leader page not used by the file system (only the first 256 words are used by the file system; the remainder has no standard interpretation).
↑L

VirtualDiskDA returns fillInDA as the virtual address for a real disk address that is either illegal or outside the confines of the file system.

The procedures for creating and destroying the disk object, TFSInit and TFSClose, were explained above. The procedure TFSWriteDiskDescriptor(disk) will write out onto the disk all vital information about the disk that is presently saved in memory. If you write programs that run the disk for extremely long periods of time, it is wise to write the disk descriptor occasionally. The only automatic call on TFSWriteDiskDescriptor is performed by TFSClose.

TfsInit.Br contains a procedure TFSDiskModel(disk) that returns the model number (80 or 300) of the drive referenced by the disk handle. This is useful in deciding whether to open a second or third file system on a T-300.

A lower level of access is permitted with the routines TFSInitializeCbStorage, TFSGetCb, and TFSDoDiskCommand, analogous to the Bfs routines described in the Operating System Manual. Users of these routines may wish to retrieve source files for the Tfs package and examine the definitions in Tfs.D and the actual disk operation in some detail. Sources are on <AltoSource>TfsSources.Dm.

4.1. TFSNewDisk

The TFSNewDisk procedure, defined in TfsNewDisk.Br, "erases" a disk (formatting it and making all its pages appear free) and creates a virgin Alto file system (SysDir and DiskDescriptor). It is called by:

```
success = TFSNewDisk(zone, driveNumber [0], diskSize [default])
```

The zone passed to TFSNewDisk must be capable of supplying about 3500 words of storage. If the drive is a T-300, the driveNumber may include a file system number (0 to 2) in its left byte, as is the case for TFSInit. The diskSize argument is the number of disk pages to be included in the file system; it defaults to the maximum possible, which is all of a T-80 or a little less than half of a T-300. TFSNewDisk returns true if successful.

4.2. DiskFindHole

The procedure DiskFindHole, in DiskFindHole.Br, can be used to locate a "hole" of available space in the disk bit table. The call:

```
virtualDA=DiskFindHole(disk, nPages)
```

will attempt to locate a contiguous hole nPages long. If it fails, the procedure returns -1, otherwise the virtual disk address of the first page of the hole.

In order to create a contiguous file, it is first necessary to create the minimal file with a leader page at the given disk address and then to use Operating System or Tfs routines to extend the file properly. The first step is achieved by calling TFSSetStartingVDA(disk, vda), where 'vda' is the desired disk address (i.e., the result returned by DiskFindHole). This value will be used to bias the selection of an
↑L

initial disk address for the leader page. Once the file is created, it is wise to extend it to its final length immediately, as other disk allocations might encroach on the "hole" that was located.

For example, if we are using the Operating System, we might proceed as follows:

```
let nPages=433          //Number of data pages needed.
let vda=DiskFindHole(TFSdisk, nPages+2)
                        //(+2= 1 for leader, 1 for last page)
if vda eq -1 then [ Ws("Cannot find a hole big enough") ]
TFSSetStartingVDA(TFSdisk, vda)

let s=OpenFile("New.Contiguous",ksTypeWriteOnly,0,verNew,0,0,0,
                TFSzone, 0, TFSdisk)
PositionPage(s, nPages) //Make the file the right length
Closes(s)
```

5. File structure on the Trident disk

The file structure built on the Trident disk by Tfs (Trident File System) is as exact a copy of the Alto file structure built Bfs (Basic File System) as is possible. Certain exceptions are present due to hardware and microcode differences. The Alto Operating System Reference Manual should be consulted for all file formats and internal information not presented here.

5.1. Disk Format

The Trident disk unit and pack, as it comes from Calcomp, is set up to run with the following parameters:

number of cylinders:	815
number of surfaces:	5 (T-80), 19 (T-300)

Triex will format each surface in the standard Tfs format:

number of sectors per track:	9
header words per sector:	2
label words per sector:	10
data words per sector:	1024

Thus, a T-80 disk will have 9*5*815 = 36,675 sectors = 37,555,200 words. Sector 0 will not be used by Tfs. All but sector 0 will be available to the file system.

Ordinarily, Tfs utilizes only the first 383 cylinders (= 65,493 sectors = 67,064,032 words) of a T-300 disk. This is the largest integral number of cylinders that can be addressed using a 16-bit virtual disk address. The 16-bit virtual address limitation is deeply embedded in all existing higher-level Alto file system software, so changing the Tfs interface to permit a larger virtual address space would be impractical.

Instead, Tfs permits one to obtain another, entirely independent disk
↑L

object for referencing the second 383 cylinders of the same T-300, thereby permitting a separate, self-contained file system to be constructed. This is done by passing a '1' in the left byte of the 'driveNumber' argument to TFSInit or TFSNewDisk (that is, drive '#400' refers to the second file system on a T-300 pack mounted on drive 0). A third file system (number '2', drive '#1000') may also be constructed, but it contains only 49 cylinders (= 8379 pages, only 6 percent of the disk's total capacity), so doing so is probably not worthwhile.

5.2. Disk Header and Label

On the Trident, a real disk address requires two words to express, rather than the single word on the Diablo 31. Also, microcode considerations gave rise to a reordering of the entries in the Label. The result is that both the header and label formats are different for the Trident. The Trident format follows. If you are interested in this level of detail, the file Tfs.d (contained within <Alto>Tfs.dm) should be consulted.

```
// disk header
structure DH:
        [
        track word
        head byte
        sector byte
        ]

// disk label
structure DL:
        [
        fileid word lFID
        packID word
        numChars word
        pageNumber word
        previous @DH
        next @DH
        ]
        manifest lDL = size DL/16
```

5.3. Disk Descriptor

Every valid Tfs disk has on it two files which must contain the state information necessary to maintain the integrity of the file system. The Tfs system directory, "SysDir.", is identical in format and purpose with its Bfs counterpart. However the Tfs disk descriptor file, "DiskDescriptor.", while identical in purpose, is formatted differently to allow easy manipulation of the bit table (which, for the Trident, has to be paged in and out of memory). This difference in format should not be evident to even low-level Trident users (unless you write your own DDMgr), but is mentioned here for completeness.

5.4. Bad Page Table

Tfs and (soon) Triex observe the standard Alto file system convention of recording -2's in the labels of all known bad pages. However, if
↑L

this were the only location of such information, "erasing" a disk (to create a virgin file system) would require two passes over the entire disk: one to collect the addresses of all known bad pages and one to mark all remaining pages deleted. This would require an excessive amount of time, particularly on a T-300.

A duplicate table of known bad pages is therefore recorded on physical page zero (= cylinder 0, head 0, sector 0) of the disk. This page is not available to the file system for other reasons having to do with end-of-file detection. The format of the table is given by the BPL structure, which is defined in Tfs.d. Note that the entries are REAL disk addresses and can therefore refer to any page on the disk regardless of whether or not such a page is accessible through the file system. (A T-300 has only one bad page table, even if it contains several file systems.)

The Triex program is responsible for building the bad page table when a disk pack is tested. The TFSNewDisk procedure (called by Tfu Erase) is careful not to clobber this information but rather to propagate it to the other places where it is needed (namely, the disk bit table and the labels of the bad pages themselves). As a result, the bad page information, once initialized, will survive across all normal operations on the disk, including "erase" operations.

There does not presently exist any facility for manually appending to this list when new bad pages are discovered. Experience to date with the Trident disks (which provide correction for error bursts of up to 11 bits in length) has shown that such a facility is probably not needed. Thorough testing of disks (using Triex) is recommended before putting them into regular use, however.

6. Revision History

July 24, 1977

Incompatibilities:

The format of DiskDescriptor has changed. The new Tfs cannot access old disks or vice versa. See description under "TFU CONVERT".

There is now another file, TfsA.Br, that is logically part of TfsBase.Br and must be loaded along with it. It contains assembly-language code formerly included as "tables" in TfsBase.Br.

New Features:

Partial support for T-300 disks.

Conforms to new conventions for maintaining addresses of known bad pages.

TFSInit checks for valid SysDir leader page and DiskDescriptor version.

Count of bit table discrepancies added to DiskDescriptor. (These are pages falsely claimed to be free in the bit table.)

VirtualDiskDA returns fillInDA for illegal real disk addresses.
↑L

Additional Trident-specific disk actions.

Tfs is now entirely reentrant, so it is safe for the Idle() procedure to give control to another process that in turn calls Tfs procedures.

October 21, 1977

Incompatibilities:

The former TfsWrite module has been broken into four pieces: TfsWrite, TfsCreate, TfsClose, and TfsDDMgr. In most applications, all four must be loaded.

The 'sharedBT' argument to TFSInit has been replaced by a 'ddMgr' argument. The mechanism for sharing a bit table buffer among multiple drives has been entirely changed. (Programs that omit this argument are unaffected by the change.)

The TFSCreateVDA static has been removed. In its place is a new procedure TFSSetStartingVDA(disk, vda) that serves the same purpose.

The syntax of the 'Tfu Exercise' command has been changed. It is now 'Tfu Exercise <passes> <list of drives>', and <list of drives> defaults to all drives that are on-line.

New features:

Complete support for T-300 disks. In conjunction with this, the TFSDiskModel procedure has been added.

It is now possible for DiskDescriptor pages to be managed externally (perhaps through some sort of virtual memory mechanism) by use of a user-defined 'DiskDescriptor Manager' object.

TFSSilentBoot procedure added.
↑L