Disc Diagnostic

Contents

[1 Purpose 4](#_Toc374438554)

[2 Overview 4](#_Toc374438555)

[3 Implementation 4](#_Toc374438556)

[4 Files in the project 5](#_Toc374438557)

[5 Commands 5](#_Toc374438558)

[6 Workings of the diagnostic 6](#_Toc374438559)

[7 Starting discdiag 7](#_Toc374438560)

[8 Reads, writes and compares 9](#_Toc374438561)

[9 Patterns 10](#_Toc374438562)

[10 Patterning the write buffer 10](#_Toc374438563)

[11 Verifying patterns in the read buffer 12](#_Toc374438564)

[12 Miscompares and controlling error actions 13](#_Toc374438565)

[13 Values 14](#_Toc374438566)

[14 Expressions 14](#_Toc374438567)

[15 Variables 14](#_Toc374438568)

[16 Printing 15](#_Toc374438569)

[17 Program control 15](#_Toc374438570)

[17.1 The loop command 15](#_Toc374438571)

[17.2 The if command 16](#_Toc374438572)

[17.3 The while-end loop 17](#_Toc374438573)

[17.4 The repeat-until loop 17](#_Toc374438574)

[17.5 The for loop 17](#_Toc374438575)

[17.6 The select statement 17](#_Toc374438576)

[17.7 The goto statement 17](#_Toc374438577)

[17.8 Procedures 17](#_Toc374438578)

[18 Program store 17](#_Toc374438579)

[18.1 Line entry 17](#_Toc374438580)

[18.2 Execution 18](#_Toc374438581)

[18.3 Load and save 18](#_Toc374438582)

[19 The initialization script 18](#_Toc374438583)

[20 Fully automated scripts and errors 19](#_Toc374438584)

[21 The Acceptance test 19](#_Toc374438585)

[22 Construction of the diagnostic 19](#_Toc374438586)

[22.1 The I/O specific calls 20](#_Toc374438587)

[23 Compiling for different targets 20](#_Toc374438588)

[24 C language coding standard 21](#_Toc374438589)

[25 Running and using Doxygen 21](#_Toc374438590)

# Purpose

The disc diagnostic is a tool for disc drive producers to check out the hardware and software involved in disc drive production.

# Overview

Discdiag is a command line program designed to read and write sectors from a disc. It can be used to check out disc drivers as well as hardware. It uses simple commands to read, write and show data within sectors, as well as automatically check it. It is fully scriptable, meaning that it can be both used immediately from the command line, as well has have those commands written into complex scripts that run fully automated tests against the disc drive.

# Implementation

Disçdiag features a separate I/O interface module that implements opening, closing, read and write sectors, show the size of the disc, and show what disks are online for testing.

It has been ported to Windows, Linux and DOS/BIOS. The diagnostic uses the fact that all of these systems can access disc sectors directly.

# Files in the project

ccodestandard.docx C code standard used in the source.

cdiscdiag Batch file that creates the Linux executable.

cdiscdiag.bat Batch file that creates the Windows executables.

cdiscdiag\_dos.bat Batch file that creates the DOS/BIOS executable.

cdiscdiag\_stub.bat Batch file that creates the pseudo executable.

discdiag The linux executable.

discdiag.c The main source code file. Contains all of the diagnostic except the OS specific portions.

discdiag.docx The Word document for discdiag.

discdiag.exe Windows executable.

discdiag.ini The automatically loaded script file for both the Windows and Linux versions.

discdiag\_dos.exe The DOS executable.

discdiag\_dos.ini The DOS automatic script (has different default numbers to account for smaller buffer sizes).

discdiag\_stub.exe "pseudo executable" (accesses a test disc in memory).

discio.h General header file for OS specific functions.

dosio.c The DOS I/O and specifics library.

linuxio.c The Linux I/O and specifics library.

readme.txt A quick explanation of the files in the project.

stubio.c A test/no-op I/O implementation that reads and writes an array.

winio.c The Windows I/O and specifics library.

# Commands

The diagnostic executes lines from the console. Each command is of the format:

command [parameter]…

The command is an alphanumeric label. Any number of parameters can appear. They may be labels, quoted strings, numeric values, or even full expressions.

Multiple commands can appear on a line as:

Command [; command]

At any place on a command line, the character “!” introduces a comment. The rest of the line will be ignored. If the “!” character appears within a quoted string, it will not be treated as a comment.

Each command usually has multiple forms, a short form and a long form, such as:

p

Or

print

The purpose of this is to allow short commands to be typed in immediately, and then the longer versions can be used for building scripts that are more “self documenting”. Of course the use of the long versions is entirely optional.

# Workings of the diagnostic

Discdiag centers around two commands, read and write:

read [lba [num]]

write [lba [num]]

Which can be shortened to “r” and “w”. Discdiag contains two buffers, the write and the read buffers:

Write buffer

Read buffer

The write buffer contains all data that is to be written to the disk. The read buffer contains all data that is read from the disk. For each of the read and write commands, some number of sectors starting at buffer location 0 are written to the disk or read from the disk.

The reason for the buffer is that disk operations are more efficient in blocks of sectors. Also, this matches the way the disk is actually used. It is written and read in varying size blocks. The buffers are sized to be fairly large, 256 sectors of 512 bytes each (on Windows and Linux) so that disk write and read tests and be performed as efficient whole buffer operations.

The object of having the separated buffers is that a predefined pattern can be set up in the write buffer, then written to disk, read back, and then checked for correctness. This can either be done by simply comparing the buffers to each other, or by comparing the read buffer to a known pattern.

Everything in the diagnostic is enumerated in terms of sectors. Thus, the number of sectors to be read or written is specified. The disks are listed by the size in terms of sectors. A sector contains 512 bytes, a convention that goes back to the PDP-11 days.

The lbas, or Logical Block Address, number the sectors from 0 to N, where N is the size of the disk-1.

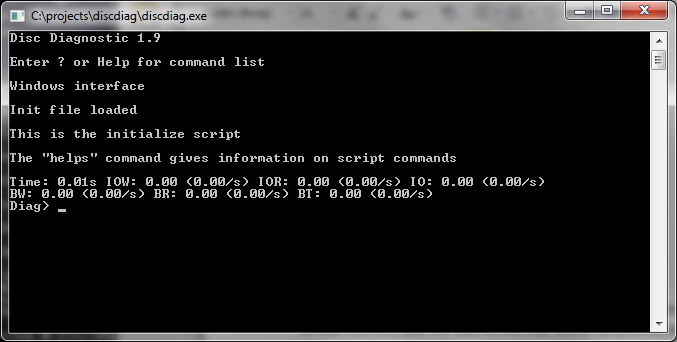
Thus:

Diag> r 10 5

Means read 5 sectors starting at LBA 10 from the disk to the read buffer starting at location 0, and proceeding to location 9.

# Starting discdiag

Discdiag is executed as a command line program as a cmd, term or dos shell:



It must be executed with administrator privileges in order to see the raw drives in use. This is done in Windows by right clicking the discdiag.exe file and selecting “Run as administrator”. It is done in Linux by running the command as sudo (root permissions). In DOS it is run as normal, since that system has no permissions needed.

The first command you want to run is to list the available disks using the “listdrives” or “ld” command:

Diag> ld

Physical drives available:

Drive 0 (\\.\PhysicalDrive0) available 3907029168 lbas

This lists the available drives and the number of sectors they contain (lbas). Drives in discdiag are numbered from 0 to 9. The logical name of the drive is listed next, and then the size in sectors (lbas).

On all systems drive 0 is typically the boot drive and contains the operating system. For this reason, selecting drive 0 generates a warning:

\*\*\* Warning: You have selected the system drive

This is to remind you that you can damage the disk you are standing on with discdiag. In addition, all drives are write protected by default. To enable write access, the “unprot” command must be used.

The operating system name of the drive also appears in the ld command. In Linux, this is the path, /dev/sd?, of the drive in the device tree. In Windows, it is a special path name used to reference the physical drive. In DOS, there are no names of drives, only numbers, so the drive is simply listed as “driveN”, where N is the drive number.

In Windows and DOS, the drive is commonly associated with a “drive letter”, such as C:. This is not the same thing as a disk drive, because in Windows and DOS, there can be multiple “logical drives” per disk. Discdiag always deals with physical disks, no matter what they contain.

To access a drive, it must be set active. This is done with the “drive” command:

Diag> drive 1

Only one disk is active at a time. Once the drive is selected, all other commands dealing with the disk refer to the active drive.

Here’s a typical sequence dealing with the active drive, and reading from the MBR (Master Boot Record):

Disc Diagnostic 1.9

Enter ? or Help for command list

Windows interface

Init file loaded

This is the initialize script

The "helps" command gives information on script commands

Time: 0.00s IOW: 0.00 (0.00/s) IOR: 0.00 (0.00/s) IO: 0.00 (0.00/s)

BW: 0.00 (0.00/s) BR: 0.00 (0.00/s) BT: 0.00 (0.00/s)

Diag> ld

Physical drives available:

Drive 0 (\\.\PhysicalDrive0) available 3907029168 lbas

Diag> drive 0

\*\*\* Warning: You have selected the system drive

Time: 0.00s IOW: 0.00 (0.00/s) IOR: 0.00 (0.00/s) IO: 0.00 (0.00/s)

BW: 0.00 (0.00/s) BR: 0.00 (0.00/s) BT: 0.00 (0.00/s)

Diag> r

Time: 0.01s IOW: 0.00 (0.00/s) IOR: 1.00 (100.00/s) IO: 1.00 (100.00/s)

BW: 0.00 (0.00/s) BR: 512.00 (50.00k/s) BT: 512.00 (50.00k/s)

Diag> dr

Contents of sector:

00000000: 33 c0 8e d0 bc 00 7c 8e c0 8e d8 be 00 7c bf 00 "3@.P<.|.@.X>.|?"

00000010: 06 b9 00 02 fc f3 a4 50 68 1c 06 cb fb b9 04 00 "..9..|s$Ph..K{9."

00000020: bd be 07 80 7e 00 00 7c 0b 0f 85 0e 01 83 c5 10 ".=>..~..|......E"

00000030: e2 f1 cd 18 88 56 00 55 c6 46 11 05 c6 46 10 00 ".bqM..V.UFF..FF."

00000040: b4 41 bb aa 55 cd 13 5d 72 0f 81 fb 55 aa 75 09 ".4A;\*UM.]r..{U\*u"

00000050: f7 c1 01 00 74 03 fe 46 10 66 60 80 7e 10 00 74 ".wA..t.~F.f`.~.."

00000060: 26 66 68 00 00 00 00 66 ff 76 08 68 00 00 68 00 "t&fh....f⌂v.h..h"

00000070: 7c 68 01 00 68 10 00 b4 42 8a 56 00 8b f4 cd 13 ".|h..h..4B.V..tM"

00000080: 9f 83 c4 10 9e eb 14 b8 01 02 bb 00 7c 8a 56 00 "...D..k.8..;.|.V"

00000090: 8a 76 01 8a 4e 02 8a 6e 03 cd 13 66 61 73 1c fe "..v..N..n.M.fas."

000000a0: 4e 11 75 0c 80 7e 00 80 0f 84 8a 00 b2 80 eb 84 "~N.u..~......2.k"

000000b0: 55 32 e4 8a 56 00 cd 13 5d eb 9e 81 3e fe 7d 55 ".U2d.V.M.]k..>~}"

000000c0: aa 75 6e ff 76 00 e8 8d 00 75 17 fa b0 d1 e6 64 "U\*un⌂v.h..u.z0Qf"

000000d0: e8 83 00 b0 df e6 60 e8 7c 00 b0 ff e6 64 e8 75 "dh..0\_f`h|.0⌂fdh"

000000e0: 00 fb b8 00 bb cd 1a 66 23 c0 75 3b 66 81 fb 54 "u.{8.;M.f#@u;f.{"

000000f0: 43 50 41 75 32 81 f9 02 01 72 2c 66 68 07 bb 00 "TCPAu2.y..r,fh.;"

00000100: 00 66 68 00 02 00 00 66 68 08 00 00 00 66 53 66 "..fh....fh....fS"

00000110: 53 66 55 66 68 00 00 00 00 66 68 00 7c 00 00 66 "fSfUfh....fh.|.."

00000120: 61 68 00 00 07 cd 1a 5a 32 f6 ea 00 7c 00 00 cd "fah...M.Z2vj.|.."

00000130: 18 a0 b7 07 eb 08 a0 b6 07 eb 03 a0 b5 07 32 e4 "M. 7.k. 6.k. 5.2"

00000140: 05 00 07 8b f0 ac 3c 00 74 09 bb 07 00 b4 0e cd "d....p,<.t.;..4."

00000150: 10 eb f2 f4 eb fd 2b c9 e4 64 eb 00 24 02 e0 f8 "M.krtk}+Iddk.$.`"

00000160: 24 02 c3 49 6e 76 61 6c 69 64 20 70 61 72 74 69 "x$.CInvalid part"

00000170: 74 69 6f 6e 20 74 61 62 6c 65 00 45 72 72 6f 72 "ition table.Erro"

\*\*\* Hit return to continue \*\*\*

00000180: 20 6c 6f 61 64 69 6e 67 20 6f 70 65 72 61 74 69 "r loading operat"

00000190: 6e 67 20 73 79 73 74 65 6d 00 4d 69 73 73 69 6e "ing system.Missi"

000001a0: 67 20 6f 70 65 72 61 74 69 6e 67 20 73 79 73 74 "ng operating sys"

000001b0: 65 6d 00 00 00 63 7b 9a 62 53 2f 8e 00 00 80 20 "tem...c{.bS/...."

000001c0: 21 00 07 df 13 0c 00 08 00 00 00 20 03 00 00 df " !..\_....... ..."

000001d0: 14 0c 07 fe ff ff 00 28 03 00 00 58 dd e8 00 00 "\_...~⌂⌂.(...X]h."

000001e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 "................"

000001f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 aa "...............U"

Time: 0.66s IOW: 0.00 (0.00/s) IOR: 0.00 (0.00/s) IO: 0.00 (0.00/s)

BW: 0.00 (0.00/s) BR: 0.00 (0.00/s) BT: 0.00 (0.00/s)

Diag>

The fact that it is the MBR can be seen by the standard message “Invalid partition table”, and similar messages. On the PC, all MBRs look this way, on all of DOS, Windows and Linux systems.

# Reads, writes and compares

Discdiag can be used to simply explore the drive sector by sector, but there are better tools for this, and there are tools that understand the file structure of the disk as well. To read and write a disk, any data on the disk is going to be corrupted. Before doing that, you need to take the disk off line from the file system. If you don’t do this, the operating system is going to attempt to read and write from the same drive, causing problems.

By type of operating system, this is what you need to do before executing the diagnostic:

Windows From the start menu, right click on computer, select “manage”, then in the computer manager select “storage”, then “disk manager”. Then right click on the disk to be tested and perform “delete volume”. This removes the file system and takes it offline.

Linux Unmount the target drive (“umount”).

DOS DOS does not really have a way to demount drives, nor does it need it. Drives that are not being accessed are not touched, nor are there (usually) any background tasks. Thus you simply need to reformat the drive with a file system after testing to set it back on line.

When reading from the disk drive (“r” or “read” commands), as mentioned, the data goes to the read buffer. You can then view it with the “dr” or “dumpread” command. This command optionally takes the number of sectors to dump, or uses the default of 1 sector.

To see what is to be written, the “dw” or “dumpwrite” command is used, which does the same for the write buffer. The write buffer starts by default as all 0’s, which is not a very interesting pattern. There are a pair of commands to form and check for patterns to write and read:

Pt/pattn [pat [val [cnt]]]

c/comp [pat [val [cnt]]]

The “pt” or pattern command sets up patterns in the write buffer to be written out to the disk. The “c” or compare command checks those patterns exist in the read buffer. Thus, executing:

Diag> pt; w; r; c

1. Writes the default pattern to a single sector on the disk at sector 0.
2. Reads back sector 0.
3. Compares that the first sector of the read buffer contains the default pattern.

In fact, this is the basic test sequence, and most tests are variations on this theme.

# Patterns

There are 5 test patterns available in discdiag. One of these is a “composite pattern” meaning that it can be used with other patterns to create complex combinations. More about that later.

cnt Byte incrementing count.

dwcnt 32 bit incrementing count.

val Numeric 32 bit value, big endian.

rand Random byte value.

lba Only the first 32 bits get LBA, rest is $ff. LBA starts at [val], and increments across buffer. Note that this only writes the first dword of each sector, use another pattern to fill the background.

buffs Compare the read and write buffers to each other. This allows complex patterns to be built up in the write buffer.

# Patterning the write buffer

Here is the default pattern, a “byte count” sequence:

Diag> pt

Time: 0.00s IOW: 0.00 (0.00/s) IOR: 0.00 (0.00/s) IO: 0.00 (0.00/s)

BW: 0.00 (0.00/s) BR: 0.00 (0.00/s) BT: 0.00 (0.00/s)

Diag> dw

Contents of sector:

00000000: 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f "..............."

00000010: 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f "................"

00000020: 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f ". !"#$%&'()\*+,-."

00000030: 30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f "/0123456789:;<=>"

00000040: 40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f "?@ABCDEFGHIJKLMN"

00000050: 50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f "OPQRSTUVWXYZ[\]^"

00000060: 60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f "\_`abcdefghijklmn"

00000070: 70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f "opqrstuvwxyz{|}~"

00000080: 80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f "⌂..............."

00000090: 90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f "................"

000000a0: a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af ". !"#$%&'()\*+,-."

000000b0: b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf "/0123456789:;<=>"

000000c0: c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf "?@ABCDEFGHIJKLMN"

000000d0: d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df "OPQRSTUVWXYZ[\]^"

000000e0: e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef "\_`abcdefghijklmn"

000000f0: f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff "opqrstuvwxyz{|}~"

00000100: 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f "⌂..............."

00000110: 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f "................"

00000120: 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f ". !"#$%&'()\*+,-."

00000130: 30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f "/0123456789:;<=>"

00000140: 40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f "?@ABCDEFGHIJKLMN"

00000150: 50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f "OPQRSTUVWXYZ[\]^"

00000160: 60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f "\_`abcdefghijklmn"

00000170: 70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f "opqrstuvwxyz{|}~"

00000180: 80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f "⌂..............."

00000190: 90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f "................"

000001a0: a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af ". !"#$%&'()\*+,-."

000001b0: b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf "/0123456789:;<=>"

000001c0: c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf "?@ABCDEFGHIJKLMN"

000001d0: d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df "OPQRSTUVWXYZ[\]^"

000001e0: e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef "\_`abcdefghijklmn"

000001f0: f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff "opqrstuvwxyz{|}~"

So this is then set up in the write buffer ready to be written to disk.

For the pt command, the default number of sectors to pattern in the write buffer is the full buffer. That is, discdiag assumes you want to set up the entire buffer even if you only write part of it out. Typically you would only need to change that if you want to save the time it takes to setup the buffer. That is, if you only wanted to write 10 sectors, specifying 10 sectors only would save execution time while performing a pattern command.

To specify a pattern, the name of the pattern is used:

Diag> pt cnt

Gives the name of the (default) byte count pattern. The next parameter after the pattern name is an optional parameter to the pattern, which is only used with the “val”, “rand”and “lba” patterns. To specify the number of sectors to be patterned, you have to specify that parameter, even if it is not used. Typically you can just say “0” for the unused parameter:

Diag> pt cnt 0 10

This would set up the byte count pattern on the first 10 sectors in the write buffer.

An alternative to the byte count pattern is a 32 bit (big endian) count. The byte count pattern repeats every 256 bytes. The 32 bit count pattern repeats every 33,554,432 sectors, so it is a better check for “aliasing” of LBA addresses, or having the sector addressing mechanisms of the drive give more than one sector to a given address.

Diag> pt dwcnt 0 10

Formats the write buffer with a 32 bit big endian count for 10 sectors.

The val or value pattern places a 32 bit word that you specify, repeated into the write buffer.

Diag> pt val 0x11223344 10

Formats the write buffer with the big endian value 0x11223344 repeated 128 times per sector for 10 sectors.

The random pattern uses the random number to generate bytes within a sector. The bytes generated are always the same for each sector, meaning that the sectors anywhere on disk can be compared for the same pattern.

Diag> pt rand 0 10

Formats the write buffer with a random byte pattern for 10 sectors.

The lba pattern is used to mark the first 32 bits of each sector with the lba of that sector (big endian). This is used to identify each sector on the disk so that the addressing mechanisms of the driver/disk drive can be verified.

The lba pattern is the only “composite” pattern. It only writes the first 32 bits/4 bytes of the sector, and leaves the rest of the sector intact. This both allows the write buffer to be patterned quickly, as well as gives the ability to form composites with other patterns.

The parameter for the pattern is the starting lba number to be used. The write buffer will be written with lba numbers starting with the parameter, then incrementing until the last sector written.

Diag> pt lba 40 10

Would format the write buffer with lbas, starting with number 40, and proceeding to number 49, for 10 sectors.

The buffs pattern is only used for comparisons.

# Verifying patterns in the read buffer

The pt or pattern command sets up patterns in the write buffer. Typically these patterns are written from the write buffer to the disk, then read back into the read buffer. The same format of command can be used to compare that the read buffer contains the patterns written into the write buffer.

Diag> c cnt 0 10

Verifies that the read buffer contains 10 sectors of the byte count pattern.

Examples of other compare commands:

Diag> c

Compares all of the sectors in the read buffer to the byte count pattern, since the default pattern is cnt, and the default length is the number of sectors in the read buffer.

Diag> c dwcnt 0 10

Verify read buffer to the 32 bit count pattern for 10 sectors.

Diag> c val 0x12345678 10

Verify read buffer contains the 32 bit pattern 0x12345678 for 10 sectors.

Diag> c rand 0 10

Diag> lba 20 10

Verifies only the lba numbers are correct for 10 sectors in the read buffer.

Diag> c buffs

Verifies the write buffer and the read buffer have the same data. So for example:

Diag> pt cnt; pt lba 0; w 0 bufsiz; r 0 bufsiz; c buffs

Would set up a byte count pattern in the write buffer, then label each sector starting with 0 lba. The Entire buffer would be written out to disk at lba 0, then read back, then the read buffer is compared to the write buffer.

This allows the composite pattern of the byte count pattern, with lba labeling, to be both written out to disk and then compared to the resulting data read back.

# Miscompares and controlling error actions

When the read buffer is checked, a miscompare appears as:

\*\*\* Error: Buffer miscompare: 00000001: 00 s/b 01

The first number is the hex address within the buffer, from 0 to N. This is followed by the byte found in the read buffer, and then the value it should have had (s/b or Should/Be).

How many bytes of miscompare are printed is controlled by the “cm” or “compare mode” command:

cm <mode>

Where mode is:

one Print only the first error encountered in the buffer.

all Print all errors found in buffer.

fail Quit the program and return to immediate mode on any error (same as “one”, but stops program).

The default is:

Diag> cm one

The fail mode is used to stop any testing on buffer miscompares. The normal mode is simply to print the miscompare and continue. Note that a miscompare in the fail mode is considered a script error, and this affects automated scripting behavior. See ***Fully automated scripts and errors.***

# Values

Any numeric parameter in the diagnostic is represented by a 64 bit signed value. These can be specified in alternate radixes as well:

10 Decimal 10

010 Octal 8

0x10 Hexadecimal 16

# Expressions

Anywhere the diagnostic accepts a number it can also take an expression. These expressions generally follow C syntax and operator priority, but do not include assignment expressions:

+a Affirmation

-a Negation

(a) Subexpression

a+b Addition

a-b Subtraction

a\*b Multiplication

a/b Integer division

a%b Modulo

a<b Comparisions

a>b

a<=b

a>=b

a=b

a!=b

Note that a=b means compare for equality, not assignment, as in C. The priorities are:

1. (a).
2. +a, -a.
3. a\*b, a/b, a%b
4. a+b, a-b.
5. a<b, a>b, a<=b, a>=b, a=b, a!=b.

As in C, comparison operators result in 0 if false, 1 if true.

# Variables

Anywhere that the diagnostic takes a number or expression, it can also take a variable. Variables are named by any of the characters a to z or A to Z followed by a to z, A to Z and 0 to 9. Example variables are:

Myval

ThisValue

Note that case is not significant in variables.

Variables keep 64 bit signed values, and substitute their value where they appear. Variables can be set by the command “s” or “set”:

Diag> s myvalue 1

Would set the variable “myvalue” to 1.

Variables are not automatically set to 0. If a variable is used before it is set, it is an error.

# Printing

Printing from diagnostic scripts can be used to annotate and format results and tests. The simplest print commands are:

echo <text>…

echon <text>…

The echo commands simply copy all of the characters after the “echo” or “echon” command and any spaces to the console, stopping on the end of line or end of command (“;”). Echo outputs a newline after the text. Echon does not.

The ability to output numbers, as well as output advanced formatting, is available with the commands:

p fmt [val]…

pn fmt [val]…

The “p” or “print” commands take a format string and zero or more numeric parameters. The format is based on C language “printf” formatting. The format characters allowed are:

%d Decimal

%x Hexadecimal

%o Octal

The format of each format specification is:

%[width[.precision]]f

Where f is the format character. The width and precision are optional.

A typical print command is:

Diag> p “Iteration count: %d” i

As for echo, the command ending indicates if a line terminator is to be output. “p” or “print” terminate the output. “pn” or “printn” does not.

# Program control

## The loop command

The most elementary program control is the loop command:

l [num]

The loop command repeats the command line before it from the beginning:

Diag> p "hi"; l 5

hi

Iteration: 1

hi

Iteration: 2

hi

Iteration: 3

hi

Iteration: 4

hi

Iteration: 5

The loop count parameter determines the number of times the loop will take place. This parameter is optional, and the default is to loop forever. Each loop counter is tracked separately, so the line:

Diag> l 10; l 10

Will loop in total 100 times (10\*10).

The default is for loop to print the count, or iteration it is currently on. This can be suppressed by using “lq”, “loopq” or “loop quietly”.

The loop commands give a quick and easy way to form loops in the diagnostic. More advanced looping constructs are available that span multiple lines.

## The if command

The if command takes a condition parameter as an expression:

if x=10; print “it is true”; goto stop

If the expression evaluates to true, or non-zero, the rest of the statements on the line are executed. If the expression evaluates to 0, the execution of the line is stopped, and the next line is executed.

Note that in the example, both the print statement and the goto will or will not be executed according to the expression.

## The while-wend loop

A while-wend statement appears as:

while x=y; print “x: %d” x

print “y: %d” y

The while-wend statement loops as long as the condition given in the expression is not equal to zero. The enclosed statements can be executed zero or more times.

A while-wend statement can span any number of lines, or appear completely on one line.

## The repeat-until loop

A repeat-until statement appears as:

repeat; print “performing”; s x x+1; until x > 10

The repeat-until statement loops as long as the condition in the expression is zero. The enclosed statements can be executed one or more times.

A repeat-until statement can span any number of lines.

## The for loop

The for-fend loop appears as:

for x 1 10 1; print “x: %d” x; fend

for expects a variable as the first parameter, then assigns it a series of values starting with the value of the second parameter to the value of the third parameter. If the optional forth parameter exists, that sets the step or “stride” of the variable. It can be negative, so that a downcount loop is formed. The default step is 1.

The statements contained in the for-fend loop can be executed zero or more times.

A for-fend statement can span any number of lines, or appear completely on one line.

## The select statement

A select statement appears much as it’s C language counterpart, but there are differences in syntax:

select x

case 1; print “one”; go last

case 2; print “two”; go last

default; print “some other number”

last: send

The value of x is matched to all of the cases in turn, and execution continues at the case that matches. If none of the cases match, execution continues after the select statement. If there is a default statement and none of the cases match, then that is executed instead.

Note that case and default are both statements. Thus they must either stand alone on the line or be terminated with “;”, just as other statements.

Note that you must provide your own method to terminate each of the statement series labeled by cases. In this case, a go statement is used to branch to the end of the select statement.

## The go statement

The go statement branches execution to a line that contains a matching label:

go terminate

print “Continuing”

terminate: print “done”

A label appears at the front of any line as:

label:

Labels are ignored when executing a program. They are used only by the goto statement and procedure calls.

## Procedures

A procedure is a block of code starting with a procedure label, and ending with an end statement. The procedure label is like a goto label, but can also contain parameters:

test(count):

print “testing for %d iterations”

end

The parameters are optional. A procedure call is a command that is not one of the existing, built-in commands of the diagnostic. Thus:

test 10

Would call the above procedure. Each parameter in the call can be an expression. The value calculated for the expression is assigned to a variable with the name of the parameter given in it’s declaration. This variable can then be used as any other variable in the procedure.

The procedure is terminated by end. This signals the return of execution to the caller.

For each parameter there must be a matching expression in the call. There are no “default” parameters.

All of the variables created in a procedure are removed at the end of the procedure. Variables are only created if they don’t exist outside the procedure. Thus:

doit(one, two):

s x 1

end

In doit the variables one and two are always local to the procedure, and disappear after the procedure ends. The variable x is local to doit only if it does not exist outside the procedure at the time it is called. If it preexisted, it’s a global. It has the value at procedure start it had before the procedure. If it is changed within the procedure, that change persists after the procedure ends.

This “automatic local” aspect of discdiag can be changed by “forcing” variables in a procedure to be local only:

doit(one, two):

local x

s x 1

end

Here x is local, even if there is a preexisting x at call time. If x exists before the procedure, it will be unmodified by it. Thus, local x is entirely independent of an x outside the procedure.

# Program store

## Line entry

Any line entered that has a number in front of it is not executed, but instead placed into the “program store”, which is kept in memory. The line number indicates where in the program store to place the line, but after that it is discarded.

Diag> clear

Diag> 10 print "hello"

Diag> 20

Diag> 30 exit

Diag> list

Program store:

1: print "hello"

2:

3: exit

Diag>

The command clear erases the current program store. Then several lines are entered. Notice that even a blank line (line 20) is entered. Blank lines are valid in the diagnostic, and are ignored when executing the program.

When the lines are entered, the line number is used to place the line, then discarded. The final number of the line is just it’s place in the program store, from 1 to N, N being the last line number. Notice above that the lines are all renumbered from their entry values.

When a line number matches a line that already exists in the program store, it is inserted before it. Thus:

Diag> 3 print "thats all"

Diag> list

Program store:

1: print "hello"

2:

3: print "thats all"

4: exit

Diag>

If you actually want to replace the line with a new one, insert the new line and delete the unwanted one. The command delt does this:

Diag> delt 4

Diag> list

Program store:

1: print "hello"

2:

3: print "thats all"

Diag>

Any line number that is higher than the last line number is placed after the last line:

Diag> 400 exit

Diag> list

Program store:

1: print "hello"

2:

3: print "thats all"

4: exit

Diag>

Thus you can either see which line is the last one with list, or simply enter a line number larger than any possible line in the program.

## Execution

There are two possible ways to execute lines in the program store:

1. Execute a procedure in the store.
2. Execute the initialization script.

For the most part, the program store is designed to be a stock of procedures to be executed. The initialization script is an exception to that.

## Load and save

The diagnostic can save the program store, and load it back later.

Diag> save myscript

Diag> load myscript

The filename is entered without quotes. It is saved or loaded from the current directory, paths are not supported in this version.

The file that is saved appears as a standard text file, without leading line numbers. The ability to save and load files allows the use of an external editor to create or modify programs. discdiag is not designed to be a good text editor. It is recommended that serious program editing be done with an external editor.

# The initialization script

When discdiag is started, it looks for a file by the name “discdiag.ini” (or “discdiag\_dos.ini” for the DOS/BIOS version). This script is loaded and the procedures in it become active.

If there exists a procedure named “init”, this will be executed after the program file is loaded. It is executed with no parameters. If the init procedure terminates without the diagnostic exiting, then discdiag will begin to accept command lines from the user.

# Fully automated scripts and errors

If the init procedure executes an exit command at its end, then it will never take command lines from the user. This is the key to operating the diagnostic as a fully automated tool. To use this mode, the command exitonerror should also be executed in the init procedure. This prevents errors in the run from exiting to command line entry. Instead, any errors will cause discdiag to simply exit.

# The Acceptance test

The acceptance test is a script that executes common features of the diagnostic. It is used to verify that discdiag has compiled and is working correctly.

More details on this will follow.

# Construction of the diagnostic

The diagnostic consists of two C language code files:

discdiag.c

xxxio.c

Discdiag.c contains virtually all of the diagnostic and it’s interpreter. The xxxio.c module contains the minimal set of support routines required to adapt it to different operating environments.

The main file, discdiag.c, is about 5000 lines, which I consider manageable for a source code file (I have other programs that have 20,000 lines in one file). Its about 100 pages, or about a small novel. I am, of course, aware that some find this excessive.

The different I/O configuration modules are about 500 lines each, or 8 pages, which is quite small, and a new one can be created in a day or so.

The original set of calls that are exposed by the I/O module were what I determined to be the minimum support the diagnostic needs to do real work.

The change to use of mingw, instead of the original Visual Studio, actually helped the commonality of the different I/O modules. The reason for this is that mingw tries to emulate Unix calls as much as possible, so using Unix methods for getting elapsed time, user break-ins and similar items allows them to be the same across different systems. This was carried further by Watcom, which also does a lot of Unix/ANSI C emulation.

The result is that I could have moved a few I/O routines back into the general diagnostic from the I/O specific routines. I didn’t do this because the original set of calls represents a fair “high water mark” of what might be needed to handle incompatibilities. This decision might be revisited later.

One of the things discdiag does for I/O is abstract the disks to a of integers from 0 to 9 (10 possible drives). The I/O module keeps track of both the current logical drive number, as well as what each drive number is really connected to.

Aside from when no drive is active at the start of the discdiag run, there is always an open drive in discdiag. It can be changed to another drive, but there is no concept of a “close” command.

What the relationship is between the logical drive and an actual drive specification is entirely left up to the I/O module. In Linux, it’s quite straightforward, being a device in the /dev area. In DOS, it’s equivalent to drive letters, like 0=A, 1=B, 2=C, etc., but specified as a zero based number in DOS calls. In Windows, it’s a special string so that it can be passed to the standard open/create calls. The only place that discdiag cares about what the string equivalents of drives are is when they are printed to the user.

Most of the calls concerning drives operate on the current set drive by default. The exceptions are for getting the size and name of a drive, which we need for the ls (listdrives) command. That would be inconvienent to have to open each drive in turn to query it.

The total operations implemented in the I/O module are:

* Drive manipulation
* User break checking
* Elapsed time calculation

Break checking from the user (usually with CNTRL-C) is required to stop scripts and other operations. One of the weaknesses of discdiag is that if the operating system does not return from a disc operation, then basically the system is hung.

The elapsed time calculation was done because I got tired of using a stopwatch or using the time command to find the amount of time a disk operation takes. I found it more convienent for discdiag to allways tell you how long a given command took, and give that at fairly high resolution.

## The I/O specific calls

The calls in the I/O module are:

int setdrive(int drive);

int getdrive(void);

int testdrive(int drive);

int readsector(unsigned char \*buffer, long long lba, long long numsec);

int writesector(unsigned char \*buffer, long long lba, long long numsec);

int physize(long long \*size);

int testsize(int drive, long long \*size);

void closedrive(void);

const char\* getdrvstr(int drive);

int chkbrk(void);

long long gettim(void);

double elapsed(long long t);

void initio(void);

You will find this defined in discio.h, which also features the sector size, and the size of each of the read and write buffers.

setdrive takes a 0-9 drive number and sets that as the active drive. It returns 1 if there was an error, otherwise 0. It will also print an error message specific to the problem.

getdrive returns the current active drive. If the drive number is -1, it indicates the drive was never set.

testdrive checks if the given drive is available (online). It does not affect the current set drive. It returns 0 if the drive is available, otherwise 1. It suppresses error messages.

readsector reads a given number of sectors from the given LBA to the buffer provided. It returns 1 on error, otherwise 0. It prints error messages specific to the problem.

writesector writes a given number of sectors to the given LBA from the buffer provided. It returns 1 on error, otherwise 0. It prints error messages specific to the problem.

physize returns the physical size in bytes for the current disk. It returns 1 on error, otherwise 0. It prints error messages specific to the problem.

testsize returns the physical size in bytes for a given disk. It returns 1 on error, otherwise 0. It suppresses error messages.

closedrive closes the current set drive. It is only used before the diagnostic exits.

getdrvstr returns a pointer to the string representing the current set drive. It returns 0 or NULL if there is no current drive (it was never set). Note that the drive setting may be numeric, in which case the string for the drive access is only representative. The returned string is a constant and must not be modified.

chkbrk returns true or 1 if the user has requested a break-in (like control-c), and false or 0 if not.

gettim returns a large counter with the current time. The actual meaning of the number contained is up to the I/O module.

elapsed compares the given time (mark time) with the current time, and finds the number of seconds elapsed since that time. The result is given as a floating point number.

initio is called at startup, and gives the I/O module a chance to initialize itself.

# Compiling for different targets

[To be completed]

# C language coding standard

I use a C language coding standard for all of my work. Please see the document:

ccodestandard.docx

With the files in this project.

# Running and using Doxygen

[To be completed]

# General design principles

Discdiag uses my “tripod” software design principles. The tripod design idea is based on having a software foundation of three products:

Program

Document

Tests

Like a tripod or a three legged stool, a program cannot stand unless it has the three basic elements of a working, usable program. These are the program itself, the documentation for it, and the tests used to verify it.

It has been my experience that these are all required, and in fact, if any of the elements are missing, they will “self-generate” in any working program that is in general use. If the documentation is missing, it will be created by customers with notes, or sales will create it to satisfy customers. If the tests are missing, then the actual tests will be done ad-hoc or effectively turned over to the customers to perform. In fact, the ability for a working system to self-generate any of these missing elements is a hallmark of a “live” software project. If it does not happen, it is likely the software is “dead”, and not really being used.

The tripod design phylosphy is that you are better off specifically creating all of the “legs” of the tripod than to let them follow in an ad-hoc fashion.

The simple tripod diagram can be used to illustrate many of the current (and past) software design methodologies. For example, the traditional model is document->program->test or perhaps document->test->program:

Program

Document

Tests

Program->Document->Tests

Program

Document

Tests

Program->Tests->Document

In many “agile” software methodologies, the sequence is Tests->Program->Document:

Program

Document

Tests

Tripod philosophy is not about what order the products are created. Rather, the products are seen as a continuous feedback process. You may create an initial document, then program using that, then find out that the requirements have changed, and the document needs extensive revision. Thus the tripod method views all of the products as continually feeding back to each other:

Program

Document

Tests

Tripod method

The documentation for the program can be broken down into:

* Program design documents
* User manuals
* Sales brochures

Similarly, the tests can be broken down into:

* Acceptance tests
* Rejection tests

The program design document describes the functionality of the program. It is possible, and in fact common, to have no such document. Because non-trivial programs are maintained by several people, the effect of having no such document is that the original programmer must describe it to new programmers, who may take notes. They then describe it to the next programmer, etc. This works, but generally wastes programmer’s time.

Another program design document is the comments imbedded in the program itself. In fact, the new paradigm for this is to use a tool such as doxygen to produce a working guide to the program. It works by following the program structure, creating a catalog of objects in the program, and decorating the result with the comments extracted from the program itself.

The tests for a program are divided into:

* Tests that fail if the program does not execute them correctly.
* Tests that fail if the program does not catch an error within the test.

That is, a test suite that in the first case is designed to be correct, and another that is deliberately designed to be incorrect. The importance of the second cannot be understated. Many programs are checked for what they ***should*** do, and the process of checking that they reject or properly handle incorrect operations is left to ad-hoc methods.

Finally, here are a few design principles for tripod design:

* Automate builds. Most software has several builds for the different products it is used on. Experience shows that programmers will build only the product that concerns them, leading to the program often being unbuildable for other uses. Making it easy to build all products, or better still, making it mandatory to build all products, eliminates this concern.
* Automate tests (even hardware). Making test work manual insures that regular regression tests won’t be performed on new versions. Testing products on physical hardware platforms is more difficult, but very necessary. In fact, aggressive automated testing against hardware platforms is the key to stability.
* Model target hardware. Creating simple software models of target hardware not only allows work on software to proceed without waiting for hardware, the ability to run tests against virtual hardware allows the software to be tested against stable hardware models first. It is typically faster, free of hardware resource allocation issues, and illustrates software design issues before adding the complication of hardware variability. In fact, hardware model checking and real hardware automated testing complement each other. Don’t throw away the hardware models after the real hardware arrives!
* There are NO LINES, ONLY LOOPS. When people draw software or hardware manufacturing processes, typically they think in terms of lines. Software design ships to hardware design. Hardware ships to manufacturing. Manufacturing ships to test. Test ships to customer, etc. In reality, all of these lines are actually feedback loops. The hardware design department talks back to software. Manufacturing talks back to hardware, etc. Thinking about lines of product movement encourages “over the wall” thinking.