PROGRAMMING FOR LYNX DOS DISKS

25th. July 1987

CONTENTS

PROGRAMMING FOR LYNX DOS DISKS

1. INTRODUCTION

Lynx DOS provides various commands for handling whole BASIC program files, e.g. EXT LOAD, EXT SAVE. The Disk Utilities add the ability to read and write, by index, fixed length records within a single file at a time, e.g. EXT PUT, EXT GET.

However it is possible to access disks, as provided in other Operating Systems like CP/M and PC DOS, in a more versatile and direct way than this so that, for instance, you can access any part of the disk as a whole regardless of files or handle 2 or more files simultaneously.

This paper documents various hitherto unpublished information about the way Lynx Dos Disks are accessed by software within the DOS itself, what use you can make of this and also provides means by which you can call the routines in the DOS from BASIC programs.

In order to make use of these routines, you first need to understand how Lynx DOS disks are structured. This is described in Section 2. This information was first published in Personal Computer World, May 1986.

Disks may be accessed at various levels; each progressively more primative and therefore more versatile. There are routines in the DOS for each level; each calling upon the routines at the level below :-

File-level handles files or parts of them,
Block-level handles the logical blocks,
Sector-level handles the physical sectors within tracks,
Port-level handles the disk controller directly by addressing its ports.

The next 4 sections discuss each of these levels; firstly block and sector levels in detail (these are equivalent to CP/M's BDOS and BIOS System calls, respectively) as these are not known to be publically documented, followed by summeries of the other two levels, for completeness, as they are described in existing documentation.

Section 7 shows how to access disks from other systems; particularly how to read or write 200K disks in a 800K drive. Finally Section 8 gives some other hints on disk usage.

The information presented here is described in terms of the 96K Lynx; I believe it applies equally to the 128K Lynx except that the Bank-Switch port number and values are different.

2. DISK STRUCTURE

This section describes the way Lynx DOS disks are logically structured. This information could be useful in seeing how efficiently the disk is organised or to access the disk at a more primative level than whole files as in Lynx DOS or even to navigate round the disk using the Disk Editor. I was prompted to "discover" this structure when I was beset by

mysteriously corrupted discs (which turned out to be due to having a disk in the drive when powering on or off) in order to locate the corruption and correct it.

Nearly all the terms used in this section were invented by myself as there is no reference material on this subject and this structure was gleaned entirely by viewing discs through the Disk Editor (DISKED).

2.1 TRACKS, SECTORS & BLOCKS

On the Lynx, disks are laid-out (formatted) in the following overall way:-

- 40 Tracks on 1 side of a disk for the 200 Kb drive
  or
- 80 Tracks on each side of a disk for the 800 Kb drive
  where a Track is a concentric ring on the disk,

- 10 Sectors of 512 bytes on each Track
  where a Sector is the addressable unit of the disk.

However as far as the structure of the disk is concerned, it is expressed in terms of Half-Sectors of 256 bytes; also called Blocks, as used by EXT DIR in its used/free space summary.

Accordingly I shall refer to Blocks from now on.

Blocks are numbered on a disk from zero to either:-
799 (&031F) on the 200 Kb drive
or 3199 (&0C7F) on the 800 Kb drive

(& indicates a hex. number)

2.2 FILES

At any time a disk holds a number of files; these are the files you created and saved, together with some special "System Files" that are on every disk. These System Files control the organisation of the disk.

Not every file has a name (your's do) but each file is assigned internally an Index number, starting from zero, by which it can be found.

Each file is stored in a whole number of Blocks, not necessarily consecutive. There could be fragmentation with the file scattered throughout the disk in groups of Blocks, due to gaps left by the erasure of previous files. It follows that the structure is fairly efficient on space since there is no need to find a consecutive group of Blocks big enough to hold the whole file. Every Block on the disk is either part of one file or unused.

2.3 INDEX FILE

The key to the disk structure is what I call the Index File.

This file starts at Block 4 of each disk and contains an Entry for each file (including itself) on the disk, giving information about the location of that file on the disk.

Each Entry is 32 bytes long with the Entry for the file with Index 0 stored in the first 32 bytes of the Index File, file 1's Entry in the next 32 bytes, and so on.

The files being Indexed are ordered as follows :-

Index	File	Blocks Occupied	Use
0	Boot & Copyright	0 to 1	Boot program & a copyright statement
1	Not used		
2	Usage	2 to 3	Used / Free Space
3	Index	4 to 7 initially	Pointers to files
4	Directory	8 to 11 initially	File names & Indices
5	Not used		
6 on	User's	Where-ever	Your files

To find the position of one of your files you first need to look up the position of the Directory (Index 4) in the Index File, then look up the file's Index in the Directory, and then count down the Entries in the Index File to find the file's Pointers.

You can see from above that the System Files occupy Blocks 0 to &0B. Hence when a disk is freshly formatted there are just 12 Used Blocks.

2.3.1 INDEX ENTRY

Each Entry, of 32 bytes, contains the following information of interest:-

```
Byte  0       = &FF if a System File
                  1 if a user file
                  0 if unused
Byte  1 Bit 0= 0 if file pointed at directly
               1 if file pointed at via Pointer Blocks (see below)
Byte  2       = File Lock
Bytes 8/9     = File size in bytes
Bytes 16/17   } Pointers : from 1 to 4 (dependent on file size)
Bytes 20/21   }            Block addresses of either :
Bytes 24/25   }       - actual Blocks of the file if size ≤ &0400
Bytes 28/29   }       - Blocks pointing to the Blocks of file otherwise.
                      Unused Pointers have the value &FFFF.
```

If the file takes less than 1K (&0400) bytes, i.e. requires no more than 4 Blocks, the Pointers are 1 to 4 Block addresses of the actual Blocks of the file. For example the Entry for the Index File itself

initially contains (in hex.) :-

```
Byte 0  : FF 00 00 XX XX XX XX XX      (XX means doesn't matter
Byte 8  : 00 04 XX XX XX XX XX XX
Byte 16 : 04 00 XX XX 05 00 XX XX
Byte 24 : 06 00 XX XX 07 00 XX XX
```

meaning that the Index File is &0400 bytes long and is stored in Blocks 4, 5, 6 and 7.

However if a file requires more than 4 Blocks (as many do) this method cannot cope with it, so a further level of pointers are required.

If a file takes more than 1K bytes but no more than 16K (&4000) bytes then bytes 16/17 of its Entry contains the Block address of a Pointer Block which in turn contains the Block addresses of the actual Blocks of the file.

The Pointer Block is laid out with each Block address stored in the first 2 bytes of each 4 bytes.

For example a file of size &0999 bytes requires to be stored in 10 Blocks and so its Entry in the Index File might look like :-

```
Byte 0  : 01 01 05 XX XX XX XX XX
Byte 8  : 99 09 XX XX XX XX XX XX
Byte 16 : 25 00 XX XX FF FF XX XX
Byte 24 : FF FF XX XX FF FF XX XX
```

meaning that its Pointer Block is stored in Block &25; the Pointer Block might look like :-

```
Byte 0  : 21 00 XX XX 22 00 XX XX
Byte 8  : 23 00 XX XX 24 00 XX XX
Byte 16 : 26 00 XX XX 27 00 XX XX
Byte 24 : 28 00 XX XX 29 00 XX XX
Byte 32 : 2A 00 XX XX 2B 00 XX XX
Byte 40 : FF FF FF FF FF FF FF FF
          ..........
```

meaning that the file is actually stored in the 10 Blocks &21 to &24 and &26 to &2B. Notice that this Pointer Block occurs after the first 4 Blocks of the file. This often happens, as you might expect, because whilst the file has no more than 4 Blocks it can be directly addressed from the Index File Entry. Once a 5th. Block occurs, a Pointer Block is needed; and where better to store it than after the first 4 Blocks of the file, with the 5th. and subsequent Blocks going after it.

If a file takes more than 16K bytes then up to 4 Pointer Blocks are created with their addresses in the Index Entry; so that a file can be no bigger than 64K (&FFFF) bytes. No further indirection is possible or needed beyond this.

This stratagy applies equally to any file on the disk including the

System Files. So although the Index and Directory Files start off at 4 Blocks, they can be extended if necessary. Initially the Index File of 4 Blocks is capable of holding Entries for 32 files of which 26 are yours. If you have more than 26 files of your own then the Index File will extend to 8 Blocks; you can often see this.

It is surprising that 4 bytes are reserved for pointers in the Index Entries and Pointer Blocks; I can only suppose that allowance was made for hard disk, since 2 bytes will accomodate 64K Blocks i.e 16 megabytes which could be limiting for larger hard disks.

2.4 DIRECTORY FILE

This file relates file names to their Indices in order to look up the Index Entry.

It starts at Block 8 and, like the Index File, contains a 32 byte Entry for each user file and itself, giving the file name and its Index.

Each Entry contains the following information :-

```
Byte  0-27   = File Name in ASCII
               Byte 0 has 80h added to it if the file has been erased;
                    hence the ease of UNERASing it.
Byte  30/31  = File Index.
```

The Directory itself is the first Entry; it has a name of "." followed by spaces and an Index of 4. So your first file is the second Entry and has an Index of 6.

2.5 USAGE FILE

This file keeps track of which Blocks on the disk have used and which are free to use.

It occupies Blocks 2 and 3, and consists of 1 bit for every Block on the disk. Each bit is either { 0 if Block unused
{ 1 if Block used.

It is this file that is used by EXT DIR to report on the disk usage and immediately after formatting it contains :-

```
      Byte 0  : FF 0F 00 00 00 00 00 00
      Byte 8  : 00 00 00 00 00 00 00 00
                ........
```

showing just 12 Blocks used; Blocks 0 to &0B.

Byte 0 refers to Blocks 0 to 7 from right to left,
Byte 1 refers to Blocks 8 to &0F from right to left and so on.

This file is also compared by EXT CHECK with the structure found through

the Index File. In fact it is rewritten at the end of EXT CHECK to reflect the structure found, with one of the messages :-

"Directory OK" or
"Directory was corrupt but is now OK"

Thus if EXT CHECK is done on a write protected disk, it will fail with an error to this effect at the end.

2.6 BOOT FILE

This file contains various information on the structure and type of the disk, as well as any boot code. It is usually read by DOS at the start of any DOS command to see what type of disk it is or if it has been has been changed.

It occupies Blocks 0 and 1, and it contains :-

Byte 0 : Disk Format; 0 = CPM, 1 = Lynx DOS
Bytes 2-3 : Boot address as a JR instruction, or 0 if no boot
Byte 5 : Disk type; &81 = 200K disk, &8C = 800K disk:
Bits 0-1 = stepping motor rate (0-3);
1 for 200K, 0 for 800K.
Although different for the 2 types, does not seem to make any difference, except that it must not be 0 for 200K, else it only moves 1 track at a time.
Neither type of drive is capable of more than one actual stepping rate.
Bit 2 = 2 sides?
Bit 3 = 80 track?
Bytes 6-7 : Pre-compensation track limits;
&00, &1B for 200K,
&40, &40 for 800K
Bytes 8/9 : Block no. of Index = 4
Bytes 12/13: No of Blocks on disk; &320 for 200K, &C80 for 800K
Bytes 16-19: Unique disk no. for sensing change-over.
Bytes 20-61: Copyright statement
Bytes 62 on: Available for BOOT code.

2.7 VIEWING DISK THROUGH DISKED

As a disk is structured in Blocks it is best to use Half-Sector addressing when using the Disk Editor, DISKED.

Thus the command GHn ,where n is a Block number in hex., will read Block n (and its Sector neighbour) into the buffer and position on the half-buffer holding the Block.

If for some reason the structure of a disk gets corrupted it may be possible, using the information given here, to identify and read the suspect Block using GHn, correct the Block in the buffer and carefully rewrite the Block using PHn, having first taken a copy, maybe.

3. BLOCK-LEVEL ACCESS

Section 2 described the way Lynx DOS disks are structured in blocks. This section discusses how blocks of these disks may be directly read and written from within any program, BASIC or machine-code, by calling a very useful routine in the DOS itself which has not been previously made public.

As a result of my previous work I guessed that there must be often-called routines in the DOS that would read/write a particular block to/from a given buffer. On this basis I have located a single routine in DOS that handles all block-level access to the disk; particularly reading and writing.

In order to call this DOS routine easily from BASIC, I also provide a machine-code interface routine that can be used in any BASIC or machine-code program.

This information will be of use to those who want to access disks at a block level by program, rather than interactively as provided by the disk Editor (DISKED). As an illustration, I have used the block read function to produce a file map by blocks.

3.1 BLOCK-LEVEL ACCESS ROUTINE

Routine Call

The Block-level DOS routine is located at address &5975 in the data RAM and accordingly is normally hidden by the ROM. Therefore it cannot be called directly from BASIC and must be called from machine-code. For compatability with possibly different issues of DOS, it is also vectored from a standard address, &FFF2, in the visible RAM, and it is this address that should be called in preference to &5975.

Thus any machine-code calling up this DOS block-level disk routine should, besides handling its parameters (see below), do the following in sequence :

- switch out ROM,
- CALL &FFF2 ,
- switch in ROM again,
- return.

Function Code

Immediately after the call instruction there must be a single byte between 1 and 9 to determine which block function should be carried out by the routine. These functions are as follows:-

Code	Function
1	Resets the disk and reads bytes 8-15 of block 0, containing disk size, to a buffer
2 & 3	Read a block to a buffer
4 & 5	Write a block from a buffer
6	Internal buffering
7	Formats disk and initialises all of it to &E5 except blocks 0 & 1, but does not leave it properly structured.
8	Verifies a given track
9	Obeys BOOT code from JR instruction at byte 2 of block 0. Code should leave register A=0 on exit.

Whilst there seems to be duplication of the Read/Write functions, the odd numbered code of each runs considerably slower than its even counterpart, and therefore presumably carries out validation on the transfer. Experience shows that these odd codes are safer.

When swapping between disks in the same drive, always use the Reset function (1) before first accessing any disk; otherwise false data, from the previous disk, may be read or written.

Parameters

Depending on the function the routine may require some parameters to be set up in the Z80 registers BC and DE prior to its call, as follows :-

Code	BC	DE
1	C= drive no. (0 to 3)	Address of 8-byte buffer
2} 3} 4} 5}	Address of 5-byte area consisting of drive no. followed by 4-byte block no. (stored byte-reversed)	Address of 256-byte buffer
6	-	-
7	B= drive	Address of 7000-byte buffer the 1st. two of which are either: &0000 for sequential sectoring, &0001 for skew sectoring (as normal)
8	B= drive, C= track	0
9	C= drive	-

'-' means not required. Where a parameter is required, it should not be given an invalid value; else it may either do nothing or crash the Lynx.

Result Code

The routine sets register A as a result code on exit, as follows :-

Result	Meaning
0	O.K.
&80-&EF	Disk failure (see DOS Manual)
&F0-&FF	Parameter error

3.2 BLOCK-LEVEL INTERFACE

The following machine-code routine, designed to easily fit in with BASIC, can be used to interface to the DOS routine from any BASIC or machine-code program :-

```
3E 10          BLOCKIO LD A, &10
01 7F FF               LD BC,&FF7F
ED 79                  OUT (C),A       :Switch out ROM
C5                     PUSH BC
EB                     EX HL,DE        :Second para. from HL
01 00 00               LD BC,&0000     :First para.
CD F2 FF               CALL &FFF2      :DOS Block Routine
00                     DEFB  0         :Function
6F                     LD L,A          :Reply to HL
AF                     XOR A
67                     LD H,A
C1                     POP BC
ED 79                  OUT (C),A       :Switch in ROM
C9                     RET             :Exit
```

This interface routine can be put in a BASIC CODE line or anywhere in RAM; in the following 'blockio' represents its address e.g. LCTN (10).

It should be called with HL containing the second parameter (the one for DE) if any; having first done the following pokes :-

```
DPOKE blockio+10 ,  value of BC parameter
POKE  blockio+15 ,  function code.
```

It will reply with the result code in HL.

For reading or writing blocks it can be used from BASIC or the machine-code equivalent, by initially setting up the 5-byte block address area :-

```
20      CODE 00 00 00 00 00
30      LET C=LCTN (20)
40      DPOKE blockio+10 , C
```

and then, whenever a disk read or write is required of block n on drive d to/from the 256-byte buffer starting at b, doing the following :-

```
POKE blockio+15 , function
POKE C , d
DPOKE C+1 , n
CALL blockio , b
IF HL THEN PRINT "Disk Failure " #HL
```

4. SECTOR-LEVEL ACCESS

This section discusses how sectors may be directly read and written from within any program, BASIC or machine-code, by calling a very useful routine in the DOS itself which has not been previously made public, in a similar way to block-level access.

In order to call this DOS routine easily from BASIC, I also provide a machine-code interface routine that can be used in any BASIC or machine-code program.

This information will be of use to those who want to access disks at a sector level by program; for example to copy disks regardless of structure or access disks from other systems.

4.1 SECTOR-LEVEL ACCESS ROUTINE

Routine Call

The Sector-level DOS routine is located at address &5D1A in the data RAM and accordingly is normally hidden by the ROM. Therefore it also cannot be called directly from BASIC and must be called from machine-code, which must switch ROM in and out around the call to this routine. It is not vectored from normal RAM.

The routine also requires some parameters to be set up in the Z80 registers B, D ,HL and IY prior to its call.

Function Code

Register B must be loaded with a code which determines which sector function should be carried out by the routine. These functions are as follows:-

	Code in B	Function
*	&80	Read a sector
	&90	Read consecutive sectors (within 1 track)
*	&A0	Write a sector
	&C0	Read disk address as track, side & sector plus 3 other unimportant bytes.
*	&F0	Write a whole track (7000 bytes), both data & overhead; as for initial formatting

These in fact are a subset of the controller read/write function codes described in Section 6. Those marked by * are actually used by Lynx DOS; the others also appear to be effective.

PROGRAMMING FOR LYNX DOS DISKS

Sector Address

Register IY must contain the address of an 6-byte control area holding the address of the sector as follows :-

Byte 0	:	Drive no. (0 to 3)
Byte 1, Bit 7	:	Side no. (0 or 1)
Bits 0-6	:	Track no. (0 to 39 or 79)
Byte 2	:	Sector no. (1 to 10)
Bytes 3-5	:	Same as bytes 5-7 of the disk's Block 0 in the Boot file i.e. for 200K : &81, &00, &1B for 800K : &8C, &40, &40

Buffer Address

Register HL must contain the address of a buffer in RAM big enough to hold the data being transferred, as follows :-

Code	Size in bytes
&80, &A0	512
&90	up to 2048 (see below)
&C0	6
&F0	7000 in correct format for initialisation, but we do not need to go into that detail here.

Amount to read

Register D is only relevent to the read functions &80 and &90 and is rather difficult to explain.

For normal sector reads it should be set to 3.

However it is actually bit-significant with each bit, from low to high, representing a 256-byte half-sector (block). If the bit is set (1) the next block will be read into the next position in the buffer; otherwise if the bit is unset (0) the next block will be skipped over and a corresponding gap left in the buffer.

For a normal single sector read (function &80) only the 2 low bits are relevent i.e. $D \leq 3$.

For a multi-sector read (function &90) all the bits are significant and, for example, D = &FF will read the maximum 4 sectors (8 blocks); providing they are all on the same track. It is possible to read up to 10 sectors using the controller directly but this gets rather involved.

Result Code

The routine sets register A as a result code on exit, as follows :-

Result	Meaning
0	O.K.
&C0	Drive not ready
&F2	Drive or Sector out of range

or the following controller status reponses (which may be additive) :

&80	Drive not ready
&40	Write protected
&10	Seek error - sector not found Multi-sector read (function &90) gives this, but is otherwise OK.
&08	CRC error
&04	Lost data

4.2 SECTOR-LEVEL INTERFACE

The following machine-code routine, designed to easily fit in with BASIC, can be used to interface to the DOS routine from any BASIC or machine-code program :-

```
3E 10          SECTORIO LD A, &10
01 7F FF                LD BC,&FF7F
ED 79                   OUT (C),A       :Switch out ROM
C5                      PUSH BC
FD 21 00 00             LD IY,&0000     :Sector address area
0E 00                   LD B,0          :Function
16 03                   LD D,3          :Amount to read
CD 1A 5D                CALL &5D1A      :DOS Sector Routine
6F                      LD L,A          :Reply to HL
AF                      XOR A
67                      LD H,A
C1                      POP BC
ED 79                   OUT (C),A       :Switch in ROM
C9                      RET             :Exit
```

This interface routine can be put in a BASIC CODE line or anywhere in RAM; in the following 'sectorio' represents its address e.g. LCTN (10).

It should be called with HL containing the buffer address; having first done the following pokes :-

```
DPOKE sectorio+10 , address of sector address area
POKE  sectorio+13 , function code
POKE  sectorio+15 , amount to read, if relevent.
```

It will reply with the result code in HL.

For reading or writing sectors it can be used from BASIC, or the machine- code equivalent, by initially setting up the 6-byte sector address area :-

```
20      CODE 00 00 00 81 00 1B  or 00 00 00 8C 40 40
30      LET C=LCTN (20)
40      DPOKE sectorio+10 , C
```

and then, whenever a disk read or write is required of logical sector n on drive d to/from the 512-byte buffer starting at b, doing the following :-

```
POKE sectorio+13 , function
POKE C , d
POKE C+1 , n DIV 10
IF n >= 800 THEN POKE C+1 , n DIV 10 - 80 + &80
POKE C+2 , n MOD 10 + 1
CALL sectorio , b
IF HL THEN PRINT "Disk Failure " #HL
```

Occasionally DOS gets itself somewhat lost after sector-level access has been used and gives a response of &10 (Seek error) when in fact the sector exists; this mostly happens when switching between different types of disks (see Section 7). In this case it is best to either EXT DIR direct or read Sector 0 from within a program, before trying again.

5. FILE-LEVEL ACCESS

This is described fully in Intelligent Software Ltd.'s "Lynx DOS Specification" dated 7-Dec-83.

The interface to the DOS routine is similar to that for Block-level, and is just summerised here.

5.1 FILE-LEVEL ACCESS ROUTINE

Routine Call

The File-access DOS routine is located at address &4A8A in the data RAM and is vectored from a standard address, &FFF6, in the visible RAM. Again it cannot be called directly from BASIC and must be called from machine-code between switching out and in the ROM.

Alternatively, to save switching the ROM yourself, it can be accessed by CALL &FEAC , but immediately on return you must CALL &FEC5 to switch in the ROM.

Function Code

Immediately after the call instruction there must be a single byte to determine which file function should be carried out by the routine. These functions are as follows:-

Code	Function
1	Open file
2	Create file
3	Close file
6	Read from current file
8	Write to current file
9	Get file status
10	Position within file
11	Special functions:-
	1 : Format disk
	2 : Verify track
	3 : Boot from disk
	128: Initialise directory structure
	129: Check disk
32	Delete file
33	Undelete file
34	Rename file
35	Select default drive

Some of the special functions duplicate some of the block-level functions; these really have nothing to do with files or blocks.

A weakness of the file functions is that only one file can be open at once so, for example, it would be difficult to copy a file using file-level access; however this could be done using block-level. Like block-level, further parameters are passed in the Z80 registers, depending on the function. Files are referred to by their names, just like in BASIC. Further details may be found in the document referred to.

6. PORT-LEVEL ACCESS

This level accesses the floppy-disk controller directly using Z80 IN and OUT instructions to the relevent ports.

Programming the controller is described fully in Western Digital Corps' "FD179X-02 Floppy Disk Formatter/Controller Family" Data Sheets.

6.1 PORT USAGE

The ports used by the Lynx to address the controller are :-

Controller Register	Read	Write
Command	-	&54
Status	&50	-
Track	&51	&55
Sector	&52	&56
Data	&53	&57
Lynx own control,e.g. Drive, Side, Precompensation, ROM.	-	&58

6.2 COMMANDS

The controller command codes are bit significant but are summerised here according to the higher digit.

Code	Command
&0X	Restore track 0
&1X	Seek track
&2X-&3X	Step one track in last direction used
&4X-&5X	Step-in one track
&6X-&7X	Step-out one track
&8X	Read a sector
&9X	Read multiple sectors
&AX	Write a sector
&BX	Write multiple sectors
&CX	Read disk address
&DX	Force interrupt
&EX	Read a track
&FX	Write a track

Commands &0X to &7X may be used to just move the drive head, while the others, except &DX, actually read or write the disk itself.

Using the controller directly to read/write disks is rather difficult as every byte must be transferred individually at the right moment; with constant checking of status and waiting for data to be ready. However just seeking tracks is fairly easy, e.g. in BASIC :-

```
        OUT &58, &90 + drive no.
        OUT &54, &59
```

will step-in 1 track, and

```
        PRINT INP (&51), #INP (&50)
```

will print the track and status registers.

7. DISKS FROM OTHER SYSTEMS

All 5 1/4" Floppy Disks are laid out in sectors within tracks so that using Sector-Level access it is theoretically possible to read, and even write but not format, them on Lynx drives providing they conform to the same standard; namely :-

On 800K drive : Double density, 96 tracks per inch (80 tracks), single or double sided. Also called quad density.
Can also be made to access 48 t.p.i (40 tracks) disks, in particular Lynx 200K disks.

On 200K drive : Double density, 48 t.p.i. (40 tracks), single sided.
In fact half of a double sided disk may also be read.

regardless of the number of sectors per track or the size of the sector.

I am not sure if single density disks can be read; there might be timing problems. I suspect that if they can they will result in half-sized sectors.

7.1 200K DISKS IN 800K DRIVES

There are two obstacles to using 200K disks in 800K drives :-

Side 0

The main obstacle is that the first side (0) on the 800K drive is the top side; unlike the 200K drive and most other disk systems, e.g. IBM PC-DOS disks, where it is the underside.

It is strange that was designed this way; I suppose that it was done to prevent the interchanging of 200K and 800K disks, although it does give the benefit that one disk can be used in both drives, providing the 800K files do not overflow one side, by formatting it first as 800K and then as 200K. However it does prevent being able to read the first sector of the disk and deciding automatically from this what type of disk has been loaded; instead DOS needs to be told explicitly.

Another difference from most other disk systems is that on the 800K drive side 0 is used up first, whereas other systems alternate their logical tracks (e.g. IBM) or even sectors between the sides. However this helps reading 200K disks as all their data is on side 1 when viewed from a 800K drive.

Track Density

The other obstacle is that the number of tracks is 40 instead of 80 but packed into the same distance; i.e. the track density is only half, yet the 800K drive is incapable of double stepping between tracks.

To overcome this the following machine code will simulate double

stepping on the 800K drive :-

```
CB 47     TRACK  BIT 0,A
28 0C            JR  Z +12     :Skip if 800K disk
4F               LD  C,A       :Store command code
DB 51            IN (A),&51    :Get current track
47               LD  B,A       :Store track
79               LD  A,C       :Restore command
CD 5F 5C         CALL &5C5F    :Seek track routine
78               LD  A,B       :Original track
D3 55            OUT (&55),A   :Reset track reg. to original track
79               LD  A,C       :Restore command
CD 5F 5C         CALL &5C5F    :Seek track again
C9               RET
```

This code of 20 bytes should be put somewhere in RAM where it will not be over-written; not in a CODE line. It could be above HIMEM providing you do not want to load any machine-code programs which might go over it. I usually poke it to address 0, which is the data RAM, as this never seems used.

A similar routine could be written to access every other track on a 800K disk in a 200K drive. However it would not be much use with Lynx 800K disks as only the last half of the disk could be seen anyway. It might be some use with foreign disks.

7.1.1 Patches to DOS

Any DOS command, including formatting and the Disk Utilities but not programs like DISKED which do not call DOS to access disks, can be applied to a 200K disk in a 800K drive providing the following pokes to DOS are applied first :-

Function	Address	to	200K Value	Normal Value
Track Density	DPOKE &5CE9 ,	Address of Track code above		&5C5F
Swap sides	POKE &5C94 ,		&20	&28
Formatting	POKE &5E26 ,		&28	&32

Be careful when doing these patches as they are in data RAM which you cannot see. I also give the normal values so you can set them back again. I recommend that you incorporate them in a program which you save on a 800K disk with the 200K values and on a 200K disk with the normal values so that you can easily switch between 200K and 800K disks.

You cannot use a proper 200K drive while these pokes are applied, but then if you have both drives you won't need this technique anyway.

As well as using any normal DOS command, you can also make use of Sector-level access method described in Section 4. When switching a

drive between different format disks, you should always read a sector from track 0 before applying the next pokes and changing disks.

7.2 FOREIGN DISKS

The Sector-level access method may also be used to access disks from other systems, possibly in conjunction with some of the following pokes :-

Drive	Disk Format	POKE
800K	Any (with side 0 underneath)	&5C94 as above
800K	48 t.p.i.	&5CE9 as above
	and use a sector address area as for 200K	
200K or 800K	More than 10 sectors per track	&5D44 , sectors + 1 (11 is normal)

On the 800K drive, reading double sided disks you will probably have to read sides 0 and 1 alternatively between tracks (or sectors) to get the proper logical order. For example an IBM-PC DOS disk is 48 t.p.i. and usually double sided and 9 tracks per sector. You will have to do the first 2 pokes above and could, if you like, also POKE &5D44 , 10.
The order of sectors on IBM is :

Logical Sectors 0 to 8 = Sectors 1 to 9 on track 0 of side 0, then
Logical Sectors 9 to 17 = Sectors 1 to 9 on track 0 of side 1, then
Logical Sectors 18 to 26 = Sectors 1 to 9 on track 1 of side 0, then
Logical Sectors 27 to 35 = Sectors 1 to 9 on track 1 of side 1, ...

On the 200K drive, reading double sided disks you will probably have to miss out every alternate track (or even sector) which is on side 1. For example in the IBM case above you could not see logical sectors 9 to 17, 27 to 35 etc. However an optional single sided IBM disk can be completely read, without having to do any pokes; so files could be transferred from IBM PCs by formatting a single sided disk and writing files on the IBM and then reading sectors on the Lynx.

8. OTHER HINTS

8.1 COPY WITHOUT QUERY

The EXT COPY and SCOPY utilities always ask if you want to erase a file if it already exits on the destination disk. The following patches, applied after loading "U", will enable you to either overwrite all existing files or not copy any existing file, without asking you:-

	To Overwrite	Not Copy	Normal Query
DPOKE &EC5F ,	&0020	&0020	&460D
DPOKE &F0FE ,	&793E	&6E3E	&BDCD
Action on existing file :	"y" shown	"n" shown	"y/n" query

8.2 DISKED FOR 80-TRACK

The DISKED utility was designed for 200K disks only and when used with 800K disks it will show only the first 40 tracks of side 0. However the following patches to DISKED will enable you to look at any part of a 800K disk as well as foreign disks :-

Function	POKE	to Value	Normal Value
80 Tracks	&6D06 ,	80	40
More than 10 Sectors/Track	&6D0B ,	Sectors/Track+1	11
Side 1	&7480 ,	&14	&10

To see both side 0 and side 1 you will need to keep switching between the 2 values for address &7480. You can use DISKED itself to alter these values using its . command to position and its E command to alter.

Colin I. Clayman,
9,The Cedars,
Armour Hill,
Tilehurst,
Reading.

Issue 2. 25th. July 1987.