ISS 45

ICL

QuickDex Technical Reference

Version 7.7

ISS45 7.7 QuickDex Technical Reference

Date of Issue	Product Identification Number	Part Number	Brief Description
April 1995	45001/009	80303886	Version 7.0
February 1999	45001/009	Electronic Library 80602986	Version 7.6
August 2000	45000/009	89000056	Version 7.7 (Unchanged)

Address comments and corrections to:

ICL Retail Systems
ISS45 Program Director
2933 Bunker Hill Lane
Suite 101
Santa Clara, CA 95054

This documentation is designed for placement in an ICL binder that can be ordered separately. To order the binder, contact your sales representative. Indicate PIN 45007/002 and/or part number 80192817.

This sheet contains spine cards that can be used to identify the binder for the appropriate documentation. Cut one of the cards along the dotted lines and insert it in the binder's spine pocket. Discard the remaining cards or save them for later use.

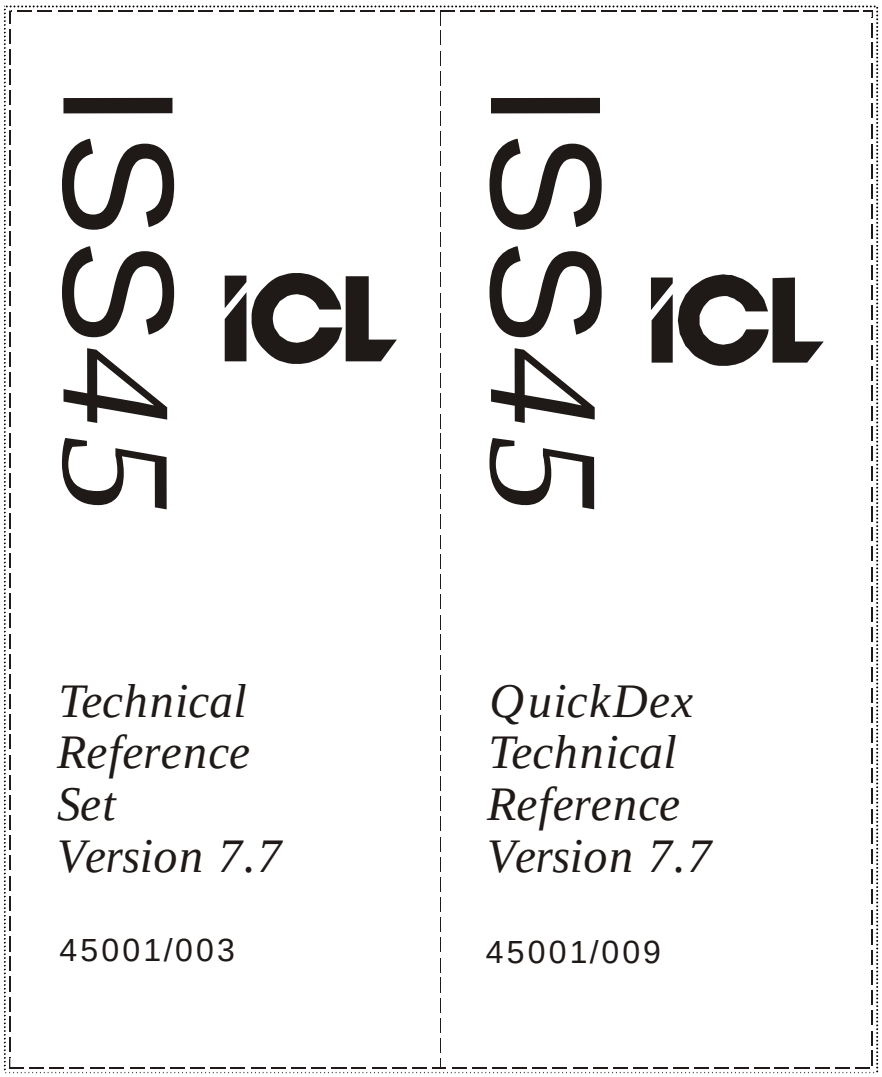


Table of Contents

Introduction

This manual explains the use of QuickDex. It describes:

- Organization of files and record structures
- QuickDex programs
- Calls from BASIC and C
- Assembly interface
- Memory allocation
- Benchmark tests
- Release changes

General

The QuickDex 4 driver handles up to 255 different files, (version 3 handled up to 64 files), and its internal parameters structure has increased.

The parameter file contains information on each file such as:

- Record length
- Block size
- Index position in record
- Index length
- Flag byte position (bit 7 on means DELETED item, bit 6 on means FREE BLOCK)
- # of free records per block during initialization or re-organization

The file can be an INDEX file, a RELATIVE file or a FIFO file. Interface to QuickDex is via the DRVPOS device driver (installed in config.sys).

Guidelines

The QuickDex driver Version 4 reads/writes via DOS (in order to gain control of virtual disks and use smartdrv cache efficiently).

Earlier QuickDex versions read/write directly by absolute access to the cylinder/sectors of the QuickDex file and bypass operating system file handling.

The QuickDex file driver handles inserts and deletes online, with no MERGE procedure.

Timing tests showed that the optimum is to read up to one CLUSTER (1 to 8 sectors, depending on disk type) and to write one SECTOR (usually 512 bytes) at a time.
However, the amount of bytes read at a time is a parameter called BLOCK (an integer multiple of sector but not bigger than a cluster). The BLOCK must be contiguous disk space.

The size of RECORD is also a parameter, but a BLOCK built in a way that record cannot overlap between blocks.

The QuickDex file is a normal DOS file, and does not contain any absolute information, so it can be copied and re-copied.

The last two blocks read are kept in special cache buffers, so next read operations of these blocks do not use slow disk access, but rather the data in the buffers. However, every write operation is written to the disk.

QuickDex file organization

QuickDex file types

QuickDex handles the following file types:

Index file

Relative file -Flat sequential/random access file.

FIFO file - Where the file is considered to be cyclic FIRST IN FIRST OUT file.

Index Expanded files - Consists of two QuickDex files. An Index file linked to a Flat relative file.

QuickDex index files

Every QuickDex Index file is constructed from:

Data Section -The actual data held on the disk or in memory (expanded or regular), dependent on parameters.

Memory Index -Built by Q-LOAD after every system start up, it is always resident in the regular part of PC memory. It contains the first key of each block of data and includes a pointer to the physical location of the block on the disk.

Memory Free Blocks List -Also built by Q-LOAD, contains pointers to Free Blocks in the Data section.

Data section

The new QuickDex file is built from units called blocks. Block size can be 512, 1024, 2048, or 4096 bytes.

If the file is an Index file, every block has reserved space for inserts located at the end of the block, marked by a deleted field .The amount of free space is a parameter. The block is sorted by KEY number (including the deleted records, which have dummy big KEY numbers). All index bytes are 0FFH.

The Index file also contains reserved space located after existing keys for new inserts, called 'free blocks'.

The first two blocks of an Index file are reserved for internal use. FIFO and Relative files do not contain leading blocks.

The first block is reserved for future purposes.

The second block is reserved for file re-organization. It is used to ensure that the file won't corrupt if power failure occurs in the middle of the process. (More information on re-organization in the Q-LOAD description).

This disk file is mapped (by Q-LOAD program), to determine the absolute location of every block. The result of this mapping is the memory index and free blocks list.

Mapping also performs a logical test of data (checking numeric keys etc.) and calculates file checksums for redundancy tests.

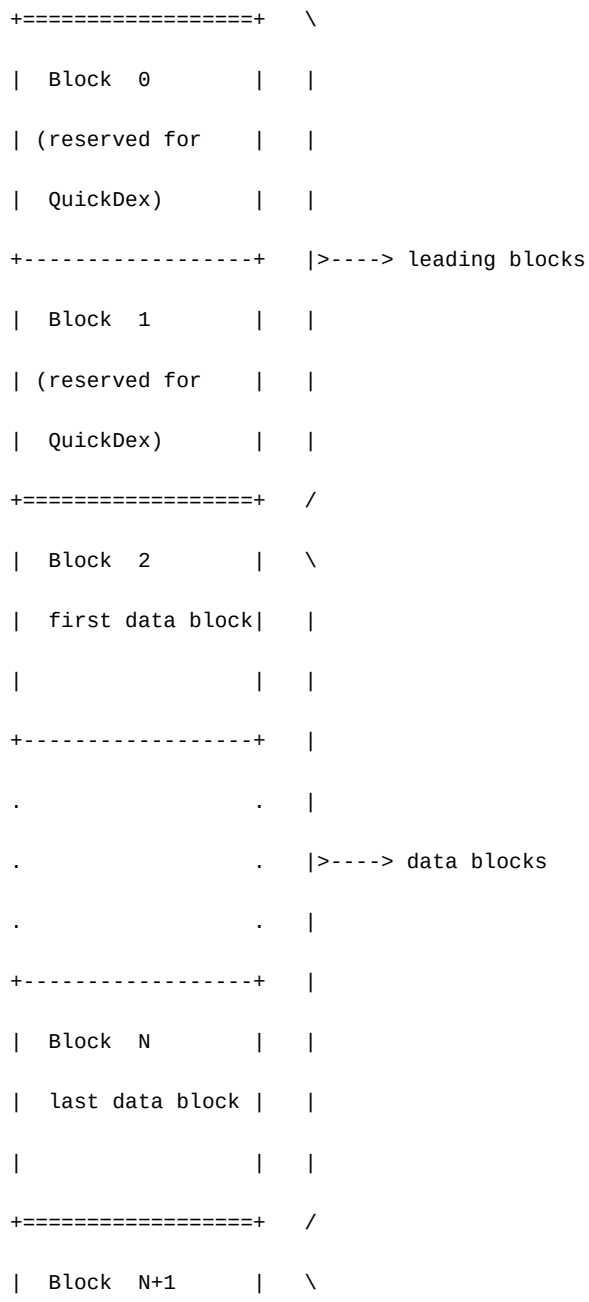
QuickDex protects against accidental access of wrong sectors by the following methods:

- QuickDex does not write to Physical sectors that are less then 12. These sectors are used by the system for the FAT and other information.
- During the OPEN command, which is performed by the application before using the file, the file directory pointer is compared to QuickDex's file location pointer, to avoid the

situation where the file was deleted or moved after Q-LOAD was performed.

- On every access, in an Indexed type file, the first key in the block should match the key in the memory. If this is not the case, QuickDex returns the error code (FDh) and NO write is performed.

Diagram of a QuickDex index file



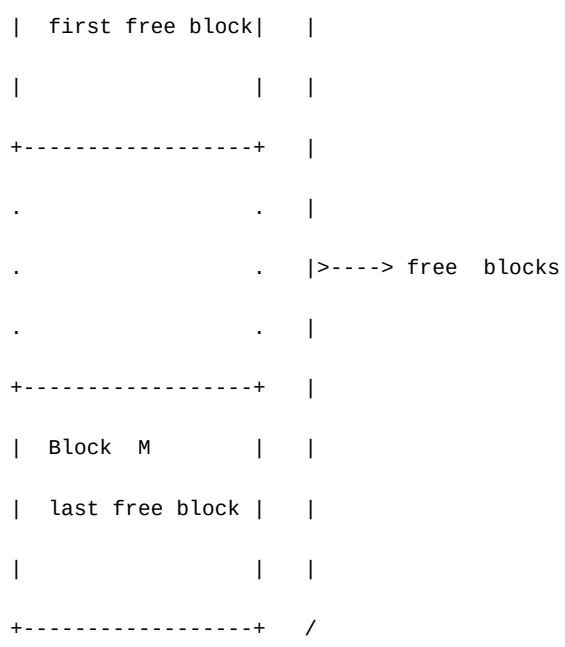
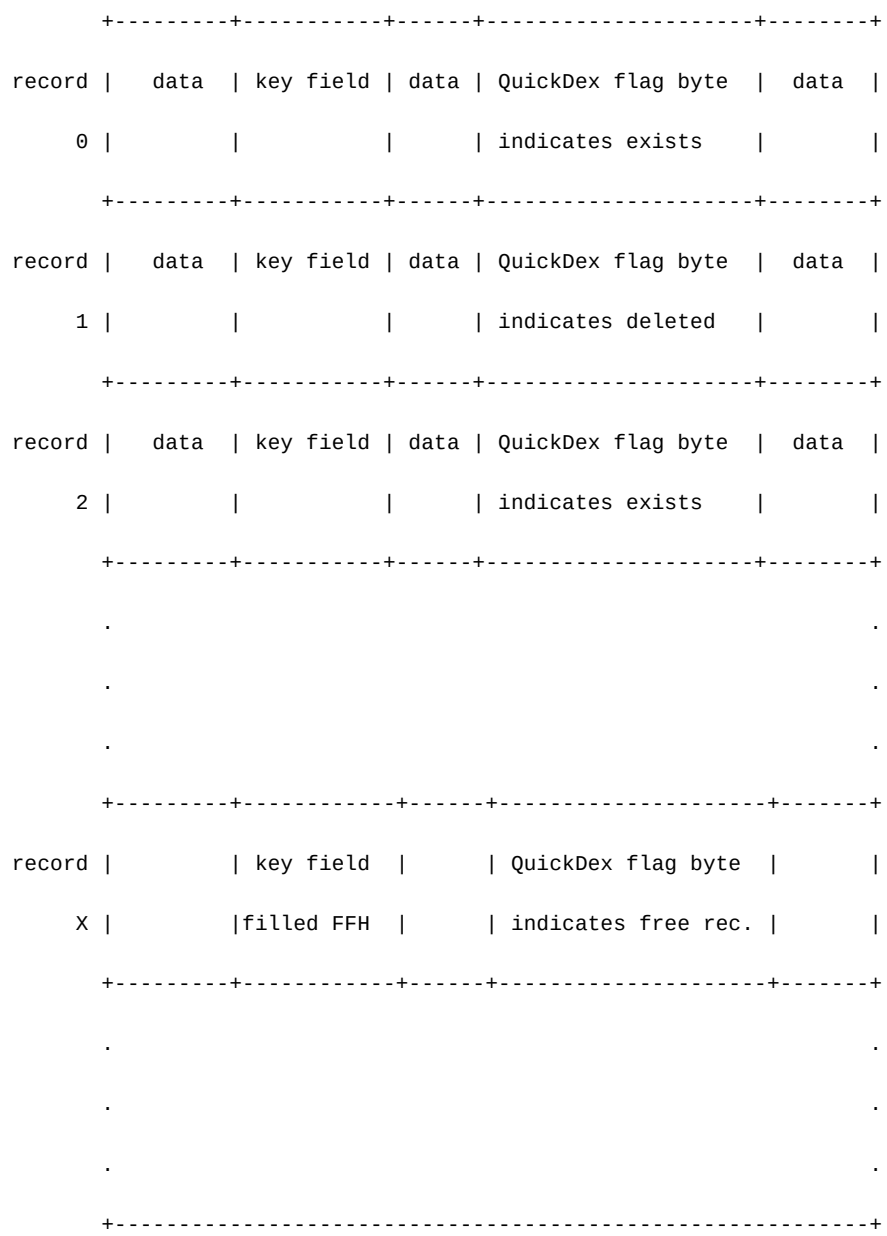

| first free block|
|
+------------------+
.
.
.
.
+------------------+
| Block M
| last free block |
|
+------------------+
|
|
|>----> free blocks
|
|
/

Diagram of a data block



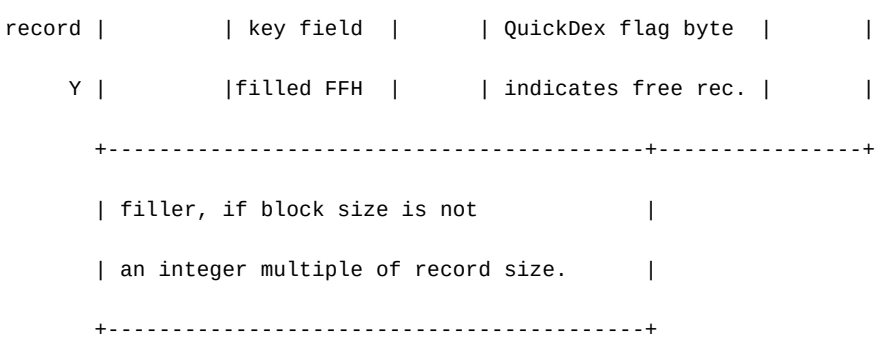
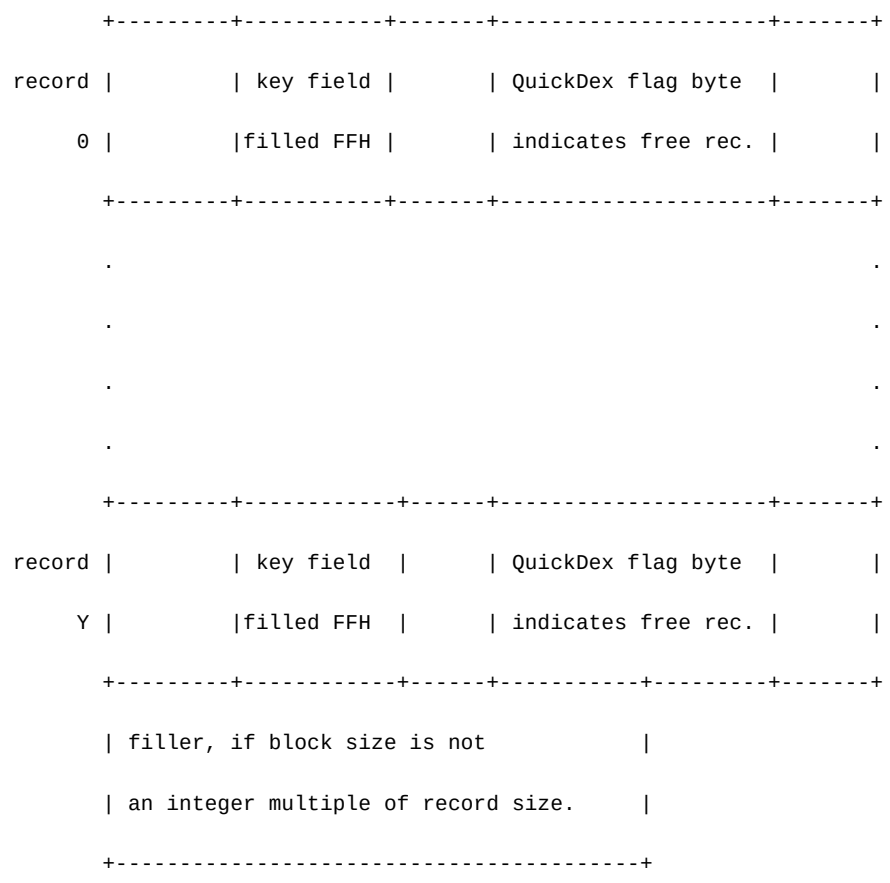

record | | key field | | QuickDex flag byte | |
Y | |filled FFH | | indicates free rec. | |
| filler, if block size is not |
| an integer multiple of record size. |

Diagram of a free block



Memory index

The index is built from the relative file and stored in resident memory. The index contains the first KEY of every block and its absolute sector number.

Index format example for 6 bytes PLU file:

```
| 6 byte plu number | 2 byte sector  number |  Lowest PLU block

-----------------------------------------

| 6 byte plu number | 2 byte sector  number |

-----------------------------------------

| 6 byte plu number | 2 byte sector  number |

-----------------------------------------

.                   .                     .

.                   .                     .

-----------------------------------------

| 6 byte plu number | 2 byte sector  number |  Highest PLU block

-----------------------------------------
```

Free blocks list

The Free blocks list is built and stored in resident memory. When disk space is needed for insert, the driver searches the list for a free block. The driver handles a pointer to the next free block.

Free blocks format:

```
------------------------

| 2 byte sector number |  1st sector of 1st free block

------------------------

| 2 byte sector number |

------------------------

.                      .

.                      .

------------------------

| 2 byte sector number |  1st sector of last free block

------------------------  (last block of file)
```

For relative and FIFO files, the memory index contains only the sector numbers of the first sector of every block.

Leading block pointer

The absolute sector of the leading block is saved in memory. The block contains information on the file and scratch information on unfinished split insert, etc.

Implementation of QuickDex functions for index files

QuickDex open

Checks for valid and loaded file.

QuickDex close

Updates directory date and time if file was modified.

QuickDex read

Index search:

Binary search in memory index till key >= requested is found.

if key = requested key: search the block.

if key > requested key: search the block pointed to by the previous index entry. If this was the first index entry, then NOT FOUND.

Block search:

Read the block, search sequentially for the required key. If found, check Deleted field. Return NOT FOUND or all RECORD information to application. Search is done until key > requested key. Tests showed that a sequential search is preferable over binary search. If successful, save key sector and index record for read next, read previous functions.

QuickDex start

get key >= requested key: same as read, but if not found return next key. If successful, save key sector and index record for read next, read previous functions.

QuickDex read next

Get following key after last successful read. If successful, save key sector and index record for read next, read previous functions.

QuickDex read previous

Get previous key after last successful read. If successful, save key sector and index record for read next, read previous functions.

QuickDex write

Same search as read. If PLU found, the driver writes the data to disk. Most times this is one sector (512 bytes) writen to disk, except for the case where the record within the block is positioned to cross a sector boundary. In this case two sectors are written to disk.

QuickDex delete

Same search as read. If the key is found, the driver marks the delete file and writes the record to disk (see write command). The index does not change.

UNDELETE feature: The delete is a logical delete. While the item remains on the disk it can be restored. The physical delete is done during Q-LOAD. Delete decrements the active QuickDex active items counter.

QuickDex undelete

Same search as read. If found, the driver resets the delete flag and writes the record to disk. The memory index does not change. UNDELETE decrements the active QuickDex active items counter

QuickDex insert

Index search:

Binary search in memory index until key >= requested key is found.

If key = requested key: Insert in this block.

If key > requested key: Insert in the block of the previous index entry.

If this was the first index entry then insert in the first block.

If no key >= requested key, insert in the last index entry block.

Block insert:

Read the block, searching sequentiallyfor the key number. If found, check Deleted field.

If not deleted, return: ALREADY EXISTS.

If deleted: update record information and write to disk.

If not found: find key location to place the new record. Location must be between the item that is smaller, and the item that is larger than the requested key insert.

If one of the two records is a deleted record: put the inserted record in the place of the deleted record and write to disk.

If none of the two records are deleted records and the block is not full (other records are marked as deleted or the block is not full), update the last deleted record in the block with the new record information, or put it in the correct location in respect to its order. Sort the block by key number. Write the changed sectors to disk.

If the first record in the block changed, update the memory index.

If the block is full, do Split procedure:

- Get a free block from the free blocks list.
- Advance pointer of next free block.
- Copy X% of the record to the new block and mark the rest of the block as deleted.
- Add the first record in the new block to the memory index.
- Mark remaining part of the records in the split block as deleted and write the block to disk.
- Update number of active keys and number of occupied blocks (in memory only).

Note

To increase reliability when power failures occur, the load key utility detects split fail, and performs the necessary recovery function.

QuickDex close

Update the file's FCB with system date and time if data has changed.

QuickDex relative files

QuickDex Relative files are random access files with variable record length or sequential files. Every QuickDex Relative file is constructed from A-Data section: resides in memory for memory files, resides on disk file for other files.

B-Memory Blocks list: built by Q-LOAD, contains pointers to Blocks in the Data section.

QuickDex FIFO files

QuickDex FIFO files are relative files, where their records are accessed on a First In First Out basis. There is a special command to view the file, i.e.: to read records without deleting them from the FIFO.

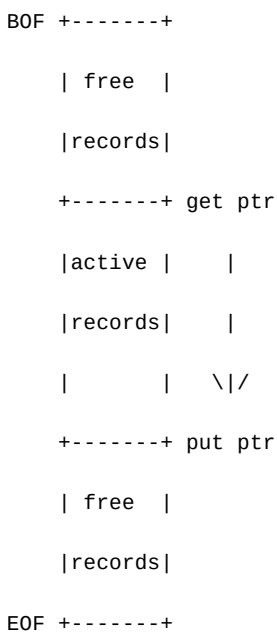
Every QuickDex FIFO file is constructed from:

- Data section : Resides on memory for memory files, resides on disk file for other files.
- Memory Blocks list : Built by Q-LOAD, contains pointers to Blocks in the Data section.
- Put record pointer : Set by Q-LOAD, contains the number of the next record to write
- Get record pointer :Set by Q-LOAD, contains the number of the next record to read.

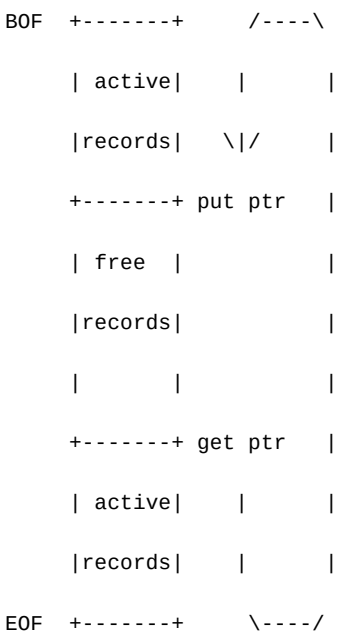
Organization of QuickDex FIFO files

The basic nature of FIFO file organization, is a group of active records and a group of free, unused records. By definition, there cannot be gaps between active records. Only two forms of organization are valid:

1. FIFO without wrap-around



2. FIFO with wrap-around



During Q-LOAD program execution, the FIFO file is checked for consistency with one of the two forms. If gaps or holes are found, Q-LOAD tries to delete the gaps and concatenate the active records into one of the valid forms, displaying a message on the screen:

fixing non contiguous FIFO file - pass nn .

If after 10 passes the file cannot be fixed, it is considered as a corrupt file and must be deleted. Q-LOAD returns to DOS after displaying the message:

FATAL ERROR: Cannot fix non contiguous FIFO file - pass nn

Delete file xxxx and run Q-LOAD again...

Gaps in FIFO files can occur if the files are corrupt or if an existing wrapped-around FIFO file is expanded.

Q-LOAD detects this situation and fixes the file in one pass.

Index expansion files

An index file PLUs record expansion in a relative file. This type enables:

- Long records (more than current length of 2048) up to 4096. It may be more if the QuickDex internal buffer length is increased or tampered with. However, DRVFILE limits the actual record size to 1024 bytes.
- Smaller spare space overheads allocated for future key inserts.
- The record data is divided into two groups:

  1. Fast data - resides in key file (up to one disk access)
  2. Slow data - resides in expansion file (up to two disk accesses)

The drawback is a slower access time. If desired, only the fast key data may be read or written by resetting bit 5 of option word in q_parm.

Records structure

The data record of a key file is like a regular key file but with a 2.5 byte pointer which contains the record number of the expansion record in the relative file. 2 bytes of the pointer are before the QuickDex flag byte, and the upper byte is in the 4 low bits of the QuickDex flag byte.

The data record of an expansion file is like a regular FIFO file, but an empty record has a 2.5 byte pointer which contains the next free record number. 2 bytes of the pointer will be before the QuickDex flag byte, and the upper byte will be in the 4 low bits of QuickDex flag byte. To increase data integrity, the free pointers linked are calculated on every Q-LOAD process. The pointer to the first free record is held in memory. The key is also written to the expansion record. This is redundant, but may help in diagnostics with reverse pointing mechanisms. The user supplies only the key to the key record part, and QuickDex duplicates the key to the expansion part.

Key record:

	key field data		2 bytes LSB data ptr to exp record LSB ----SB	QuickDex flag byte 1/2 byte MSB ptr to exp		data

Expansion record:
Empty record:

unused data	2 bytes ptr to next free record LSB ---- MSB	QuickDex flag byte indicates deleted record ½ byte MSB ptr to nxt free	unused data

Full record:

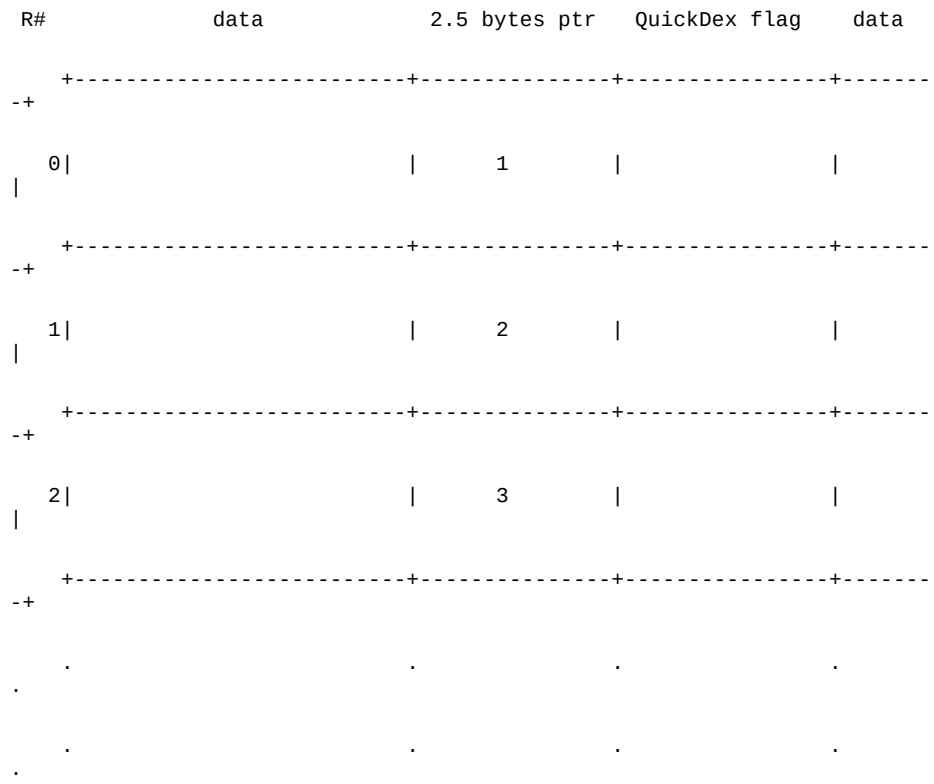
data	key field		QuickDex flag byte indicates active record	data

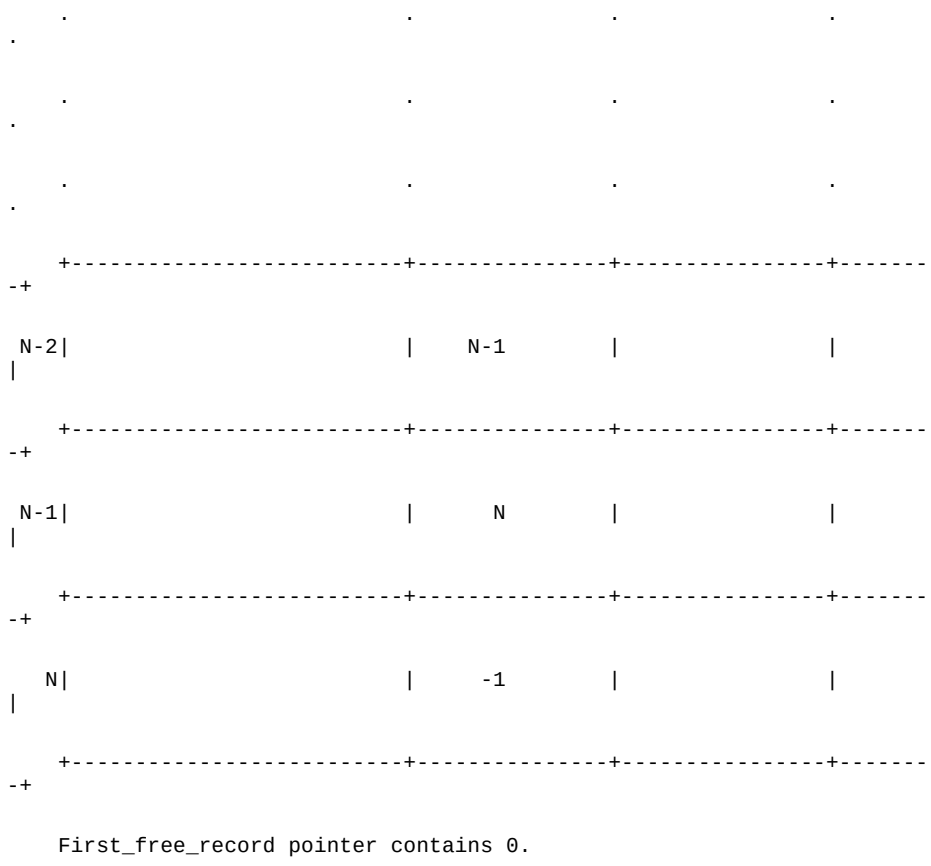
Load process

Regular load of key file. The first_free_record pointer is initialized to -1. The expansion file is searched for free records. The first_free_record pointer is set to the first free record number. On every free record found, the pointer in the last free record found is set to the current record number. The first record of the file is therefore inserted on the first insert.

To increase speed, the file is created with the correct pointer values, and the load updates the pointers only if errors are found.

Empty file diagram





First_free_record pointer contains 0.

Expansion read process

First read key record. If non-deleted, test expansion pointer. If not -1 value read relative from expansion file according to pointer value. Assemble the two records into one long record and copy its data to user.

Expansion write process

Start by expansion read process. If ok, split long user record into key and expansion parts. Write key part to key file and expansion part to expansion file.

Expansion insert process

If first_free_record contains -1, return EXP_FILE_FULL status. Split long user record into key and expansion parts. First insert key record and update expansion pointer field with first_free_record pointer value. Read expansion file, record number of first_free_record. Set first_free_record pointer to next_free_record value and read from expansion record. Set QuickDex flag to non-deleted value and write expansion data.

Expansion delete process

Split long user record into key and expansion parts. Delete key record and get expansion pointer. Write to expansion file, record number expansion_pointer and update QuickDex flag byte to deleted value. Write in the previous two bytes first_free_record value and set first_free_record to expansion record number.

Memory files

In QuickDex version 4, memory files are handled by DOS commercial RAM disk utilities such as RAMDRIVE, VDISK etc. The files can reside on regular, extended or expanded memory. The files are fast access temporary files (about ten times faster than a disk file). It is implemented for fast data management applications. The disadvantage of these files is that data is lost on power failure or booting.

Q-load program

The Q-LOAD program scans QuickDex files on the disk, maps their physical location and loads the memory index and the free blocks list. If a file does not exist, it is created before loading. During load, index files are tested to match QuickDex requirements for sorted blocks, legal key values, etc. If a record is corrupt, it is erased from the file and written to a log file named Q-LOAD.LOG.

Index files can be re-organized. Re-organization is required when there are few free blocks in a file that contains many deleted records. This can lead to a file full situation when there are actually several blocks in the file that contain many deleted records. During re-organization the index file is read and each block is written as a full block, so the number of free blocks increases. Data integrity is maintained even if power failures occur in the middle of re-organization, because QuickDex uses the second leading block as a scratch backup area. On the next load, QuickDex detects that the last re-organization did not succeed, and uses the data from the leading block to reconstruct the file.

Re-organization is done automatically when less than 10% of the blocks are free, and the number of non-active records (free or deleted), would free at least two more blocks.

Re-organization can be forced by executing Q-LOAD /O. In a 'DUAL' system, the forced re-organization must be done on both PCs at the same time.

USAGE: Q-LOAD [switches] <ENTER>

If no switches are specified, Q-LOAD loads all QuickDex files, automatically reorganizing a file if necessary.

A log file of discarded bad records Q-LOAD.LOG is written.

Switches:

/help -Display help on Q-LOAD

/O -Force re-organization on files

/Ln -Load QuickDex file number n only (n=1 to 16)

/F -Fast load, no checks on keys, no re-organization (switches must be delimited with blanks).

/Backup- Back up QuickDex indexes to disk for fast load later

/restore- Restore QuickDex indexes from disk after backup

Q-LOAD sets DOS ERRORLEVEL to 0 if no errors.

Error Level 1:

If index file(s) fragmented. In this case performance may decrease. Running a disk optimizing utility causes the file to be contiguous and increase performance.

Error Level 2:

If a fatal error occurs during load.

Q-setup program

The Q-SETUP program sets QuickDex parameters and should be run before QuickDex.

The Current Q-SETUP programs handle both DRVFILE and QuickDex parameters and replaces the Q-SETUP and CFSPRM old programs.

Q-SETUP can work on some files groups. If the command line is Q-SETUP <ENTER> then Q-DEX.PRM and FILE.PRM are modified. (This is the default).

You can run Q-SETUP xxx <ENTER>. Then FILE.xxx and Q-DEX.xxx are modified. For example: Q-SETUP MFS <ENTER>, modifies MFS files Q-DEX.MFS and FILE.MFS

Main menu:

```
F1 - Define QuickDex files

F2 - Define QuickDex general
parameters

F3 - Define DRVFILE parameters
```

General Parameters:

```
1. DRVPOS Entry Number

2. Maximum Number Of Files Used
[1-255]

3. Rlt Super indexes path

4. Fast load indexes path
```

1. DRVPOS is a special driver that handles communication between applications and
other drivers. Each driver is identified by a specific number. The QuickDex driver number is 5 and must not be changed. DRVPOS is activated through a command in the CONFIG.SYS file:

   'DEVICE=C:\path\DRVPOS.COM'

2. Number of files used (1 to 255).
3. The path for super indexes. For better performance, use a Ram-Drive directory.
4. The path for backing up indexes for fast load.

Define a QuickDex file:

```
1. Remark:

2. File name

3. File type    [] wrap
around

4. Record size

5. Key length

6. Key offset

7. Qdx flag offset

8. Maximum records in file

9. Block size

10.Split percent

11.Relative linked file

12.Drvfile Entry number:

     [local, read/write
backup LAN

13.Use Super index for this
file

14.Super index Number and
name
```

1 File remark comment.

2 File name including path (37 bytes).

3 File type: 0 -Index file.
1 -Relative file: Flat sequential/random access file.

4 FIFO file: Where the file is considered to be a cyclic FIRST IN FIRST OUT file.

5 Relative expansion file. A Flat file which is linked to an Index file.

6 Super index of another index file.

Wrap - Effective only for FIFO files (parameter #11 set to 2): If set to YES, and the file is full, new data overwrites the oldest data in the file. If set to NO, and the file is full, insert is denied and the 'file full' error code is returned to the application.

7 File record size (up to 1024 bytes).

8 Key length (up to 128 bytes) for FIFO. Relative files must be one byte.

9 Key location in the record.

10 Location of QuickDex reserved byte in the file record.

11 The total amount of records in the file. This parameter affects the size of the whole
file. For index files, this parameter value must be bigger then the number of active
records in the file, to allow inserts of new records. For small files (with few blocks), use up to double the number of active records. For large files add 10 to 50 percent to
the active records.

12 Block size - dependent on the disk type. Must not exceed DOS Cluster size or 4096 bytes. Usually DOS sets the cluster size to 2048 or 4096 bytes for hard disks and 1024 bytes for double sided floppy disks. The biggest valid value is the best choice for consuming less memory for the index. Use value 0 for automatic setting of the optimal block size. The QIDXCONV utility program can convert an existing QuickDex file to a different block size. Block size must be greater than or equal to 512 bytes, because every disk causes the writing of at least one sector (sector size is 512 bytes).

13 This parameter defines how many records reside in the old block after split is performed. If most of the new record inserts are sequential, use 100%. If the inserts are random use 50%.

The effect of split percent is the following:

After insert to an index file is done and the block is full, QuickDex keeps split percent records in the old block, allocates a new block, and copies the rest of the records to the new block.

When split percent is 50%, it guarantees that a random order of inserts keep QuickDex blocks balanced. (i.e. there will only be a few blocks with few valid records).

If you know that the inserts are sorted according to the key, then you get a more efficient use of blocks by increasing the split percent. The blocks contain more than 50% records and you have more free blocks for next insert. However, if split percent is bigger, and the order of inserts is not sorted, you will get worse results with fewer free blocks.

Examples:

Let's estimate the blocks usage in a test file with 10 records in a block, 100 blocks total.

- Split 50%, 100 inserts in ascending order: Result is 20 blocks used, 80 blocks free.
- Split 50%, 100 inserts in descending order: Result is 20 blocks used, 80 blocks free.
- Split 70%, 100 inserts in ascending order: Result is 14 blocks used, 86 blocks free.
- Split 70%, 100 inserts in descending order: Result is 33 blocks used, 67 blocks free.

14 The number of the file linked to this file if a relative expansion index file pair.

0-254: Number of the linked QuickDex file.

9999: No relative expansion of index.

15 DRVFILE entry number - associate this file to a DRVFILE entry defining how DRVFILE reads/writes/backs up this file to the LAN.

16 Marked if Super index is used for this file

17 The number of the super index linked to this file:
0-254: Number of the linked Super index QuickDex file
255: No super index

18 A checkbox indicates whether this file is guaranteed write. If yes, every write to the file is followed by a flush of the cache buffers of Smartdrv and DOS, to guarantee a write to the disk. The penalty of using this is slower performance.

Q-setup has two new parameters:

/F - Allows working on q-dex.xxx and file.xxx where xxx can be MFS, LFS, etc.
The default is q-dex.prm and file.prm.
work on q-dex.mfs and file.mfs, run q-setup /Fmfs

/O - Helps to update parameters in batch mode, so that you can update other PCs with dates made on another PC.

Step 1 First you have to create an output file, run q-setup normally. Update the parameters you need and press the 'Output File' button. Tis creates an ASCII file q-setup.out with the current parameter values.

Step 2 You can edit this file, delete parameters that you do not want to change, and keep only the 'file number=xxx' line and the parameters you want to update.

Step 3 Update the target system with the parameters set in the output file. Run q-setup /Oq-setup.out.

Q-prm program

The Q-PRM program sets QuickDex parameters in memory only, and should be run after QuickDex.

Usage: `Q-PRM /SPLITxx=nn%`

Change split percentage of file no. xx to nn%

This utility is good for efficient QuickDex file creation (first time inserts). Usually the split percentage is 50% to 80%, but if the file is empty and the inserted records are sorted, the split percentage should be set to 100% so that all the space of the blocks in the file can be used by the command:

Q-PRM /SPLIT0=100%

The records are then inserted, and afterwards the split percentage is restored by the command:

Q-PRM /SPLIT0=50%

Note

Only the memory parameter is changed. Booting restores the values set by Q-SETUP utility.

Q-PRM /GWxx=1

Set guaranteed write mode for file no. xx

Q-PRM /GWxx=0

Do not guarantee write for file no. xx

Qidxconv program

The QIDXCONV program is obsolete. Use the Q-CONVRT program instead.

Q-convrt program

This program converts a QuickDex index file to match a different block size value.

An enhanced conversion utility is provided to convert QuickDex index files to various block sizes without the need to load a file before conversion.

Usage: **Q-CONVRTV [old fname] [new fname] /NEWBLKnn /RECLENnn /KEYLENnn /KEYOFFnn /FLGOFFnn**

Converts an index file to conform with the new block size.

The new block size can be 512, 1024, 2048, 4096 or 0.

Block size = 0 is the same as 4096.

Note

Conversion is not needed for relative or FIFO files.

This utility is good for transferring an existing index file to a different block size. The best block size for QuickDex 4 is 4096 bytes, which requires minimum resident memory.

After running Q-CONVRT, the new file should be copied to the actual QuickDex file.
Q-SETUP must be run, and then update the block size to the converted value. The PC should then be rebooted.

Q-close program

The Q-CLOSE program is used to close QuickDex open files. It is used mainly while copying QuickDex files after they are loaded. If there is a need to overwrite a QuickDex file while the system is on, the file must be closed first (via Q-CLOSE utility).

For example: Replacing POS PLU file (QuickDex file no. 1):

Q-CLOSE 1

COPY A:PLUPOS.QDX C:\CFS\QDX

Q-LOAD /L1.

Redundancy and net

Recovery

Since the QuickDex file does not contain absolute information, it can be r-copied during the recovery process, and run index and free blocks list building procedures afterwards.

A special RCOPY command supports fast QuickDex file copy and load:

```
RCOPY [source fname] [destination fname] /QDXnn
```

where **nn** is QuickDex file number.

Redundancy check

By running checksum on each computer. The Back-office checksum is read by a special PCMNET function. The checksum calculation is a word XOR over the active files' blocks.

The DRVFILE driver must be loaded to enable remote access to QuickDex files.

Working with QuickDex

1. Make sure you have activated DRVPOS in the CONFIG.SYS file. The following line must be included in the CONFIG.SYS file:

   DEVICE=[drive:][path]drvpos.com
2. Run setup utility Q-SETUP.EXE .
3. Run QuickDex driver Q-Dex.EXE .
4. Run Q-LOAD.EXE to load memory index with keys, or create new files.
5. Run Q-TEST.EXE to test QuickDex or other application programs.

Note

After parameters change, you MUST remove QuickDex from memory by running
Q-REMOVE or booting. Booting is preferred. You then perform steps 3-5 again!

Calls from BASIC and C

General

All calls except QCHK use a parameters block.

q.param of 5 words (10 bytes) formatted:

```
  byte- QuickDex file number (0-63)
  byte- reserved
  word- options normally 0
                        bit0: 1=reserved for system use,return 1st in
block
                              on qstart
                        bit1: 1=dont update user record
                        bit2: 1=dont use buffer,always read disk
                        bit3: 1=write zeros instead of user data
                        bit4: 1=multi-user environment i.e. commands
                              like read-next will not be affected by
                              other user read commands
                        bit5: 0= use only index part of expanded index
file
                              1= use whole record of expanded index file
                        bit 6:1=exchange high and low parts of offset
  word- high word of write_part_offset (for relative file functions)
  word- low word of write_part_offset,used as offset of string
        in partial write and other commands
  word- write_part_length,length of string in partial write
        20 bytes - reserved
```

Example: q.parm in BASIC:

```
q.param$ = mki$(file.num%) + mki$(option%) + mki$(hi.offset%)
       +mki$(low.offset%) + mki$(length%)
```

Example: q.parm in C:

```
struct q_parm_
    {
    unsigned file_num;
    unsigned option;
    unsigned hi_offset;
    unsigned low_offset;
    unsigned length;
char   filler[20];
    }q_parm;
```

Return codes :

```
 0H  =   OK
80H  =   GENERAL_ERROR
 1   =   ERR_NOT_FOUND
 2   =   ERR_INDX_START
 3   =   ERR_INDX_READD
 4   =   ERR_DELETED
 5   =   ERR_EXISTS
 6   =   ERR_DISK_READ
 7   =   ERR_DISK_WRITE
 8   =   ERR_INDX_NOT_LOADED
 9   =   ERR_INDX_WRITE
0Ah  =   ERR_INDX_DISK_MATCH
             Memory index does not match disk content.
```

```
                    Try loading again.
          0BH  =   ERR_FILE_NOT_OPENED
          0CH  =   ERR_LOAD_FAIL
                    Q-LOAD program failed. Try loading again
          20H  =   ERR_BAD_FUNCTION_TYPE
                    The function cannot be performed on this
                    kind of file, such as key insert on FIFO file..
          21H  =   ERR_FILE_FULL
                    No room for inserts in index file
          22H  =   ERR_RECORD_OVERFLOW
                    offset and length exceeds record size
          23H  =   ERR_EXP_NOT_FOUND
                    Expansion record not found
          24H  =   ERR_EXP
                    Expansion error or Expansion record not found
          25H  =   ERR_EXP_FILE_FULL
                    Expansion file full
          26H  =   ERR_EXP_DELETED
                    Expansion record deleted
          27H  =   ERR_EXP_EXISTS
                    Expansion record already exists
          28H  =   ERR_SUP_IDX
                    Error in super index file, (super index closed or
corrupt file)
          29H  =   ERR_NOT_OPENED
                    The file was not opened
          2AH  =   ERR_SEEK
                    Error during DOS seek on the file
```

```
 FDH  =   ERR_MAP
              Bad mapping ,or deleted index file
 FEH  =   ERR_DRVPOS
              DRVPOS or QuickDex not loaded
 FFH  =   bad_function_number
100H  =   QuickDex busy
```

Check if QuickDex driver loaded - QCHK

Returns 0 in ret_code if QuickDex driver in memory

Call from BASIC:

call QCHK(ret%)

Call from C:

int ret = q_chk();

Get QuickDex parameters - QGETPARMS

Get driver internal parameters and place in user's record. User's record minimum length is 6000 bytes.

Internal parameters format:

160 bytes, General parameters

160 bytes for evrey QuickDex file * 32 files

And some additional parameters such as active keys per index.

Call from BASIC:

call QGETPARM(q.parm$,record$,ret%)

Call from C:

int ret = q_get_parm((struct q_parm_ *)&q_parm,char *record);

The parameters are in QuickDex version 2 format. To get the parameters in QuickDex version 4 format, Q.parm option word must be 2.

Set QuickDex parameters - QSETPARMS

Set driver internal parameters from user's record.

User's record length is 6000 bytes.

Internal parameters format is the same as QGETPARMS

Call from BASIC:

call QSETPARM(q.parm$,record$,ret%)

Call from C:

int ret = q_set_parm((struct q_parm_ *)&q_parm,char *record);

Q.parm option word must be 2.

Open file - QOPEN

Every QuickDex file must be opened before data processing. QuickDex checks that the file is loaded and mapped well into memory. The file pointer is positioned at the beginning of the file.

Call from BASIC:

```
call QOPEN(q.parm$,record$,ret%)
```

Call from C:

```
int ret = q_open((struct q_parm_ *)&q_parm,char *record);
```

Close file - QCLOSE

Every QuickDex file must be closed after data processing. QuickDex updates the directory entry with system date and time if the file was changed since the last open command.

Call from BASIC:

```
call QCLOSE(q.parm$,record$,ret%)
```

Call from C:

```
int ret = q_close((struct q_parm_ *)&q_parm,char *record);
```

Read from index file - QREAD

The user must set the key field in user record to the desired value to be read. QuickDex fills all the record if key found.

Call from BASIC:

```
call QREAD(q.parm$,record$,ret%)
```

Call from C:

int ret = q_read((struct q_parm_ *)&q_parm,char *record);

Read matching or next key from index file - QSTART

The user must fill the key field in user record.

QuickDex fills all the record if key found. If the key is not found, QuickDex reads the next key and fills the user record with its data.

Call from BASIC:

call QSTART (q.parm$,record$,ret%)

Call from C:

int ret = q_start((struct q_parm_ *)&q_parm,char *record);

Read next key from index file - QREADN

If a multi-user environment (q_parm.option bit 4 is set), the user must set the key field in user record to the last key read.

If a stand-alone environment (q_parm.option bit 4 is not set), QuickDex maintains the next key to read.

The next key is read into the user's record

Call from BASIC:

call QREADN(q.parm$,record$,ret%)

Call from C:

int ret = q_readn((struct q_parm_ *)&q_parm,char *record);

Read previous key from index file - QREADP

If a multi-user environment (q_parm.option bit 4 is set), the user must set the key field in user record to the last key read.

If a one-user environment (q_parm.option bit 4 is not set), QuickDex maintains the previous key to read.

The previous key is read into the user's record

Call from BASIC:

```
call QREADP(q.parm$,record$,ret%)
```

Call from C:

```
int ret = q_readp((struct q_parm_ *)&q_parm,char *record);
```

Read the last key from index file - QREADLAST

The last key of the index file is read into the user's record.

Call from BASIC:

```
call QREADLAST(q.parm$,record$,ret%)
```

Call from C:

```
int ret = q_read_last((struct q_parm_ *)&q_parm,char *record);
```

Insert a new key to index file - QINSERT

The user must fill all fields in user record.

q_parm.option bit 3 can be set to zero for all fields in the record except for the key.

Call from BASIC:

call QINSERT(q.parm$,record$,ret%)

Call from C:

int ret = q_insert((struct q_parm_ *)&q_parm,char *record);

Delete key from index file - QDEL

The user must fill the key field in user record. QuickDex deletes the key from the index file. Deleted keys can be recovered by using the UNDELETE command if they were not over written by a new key insert.

Call from BASIC:

call QDEL(q.parm$,record$,ret%)

Call from C:

int ret = q_del((struct q_parm_ *)&q_parm,char *record);

Undelete key from index file - QUNDEL

The user must fill the key field in user record. If the key was a deleted key it will be recovered.

Call from BASIC:

call QUNDEL(q.parm$,record$,ret%)

Call from C:

int ret = q_undel((struct q_parm_ *)&q_parm,char *record);

Delete all file - QEMPTY

QuickDex deletes all the keys from the index file. A relative file is filled with C0 Hex value. FIFO file pointer resets to beginning of file. Deleted keys cannot be recovered by the UNDELETE command.

Call from BASIC:

```
call QEMPTY(q.parm$,record$,ret%)
```

Call from C:

```
int ret = q_empty((struct q_parm_ *)&q_parm,char *record);
```

Update existing key data (index file) - QWRITE

The user must fill all fields in user record.

q_parm.option bit 3 can be set to zero all fields in the record except for the key.

Call from BASIC:

```
call QWRITE(q.parm$,record$,ret%)
```

Call from C:

```
int ret = q_write((struct q_parm_ *)&q_parm,char *record);
```

Partial update existing key data (index file) - QWRITEPART

The user must fill the key and some consecutive fields in user record. Then fill q_parm.offset with the offset of the first byte of the field to write. Finally, fill q_parm.length with the length of the fields to write.

q_parm.option bit 3 can be set to zero for these fields in the record.

Call from BASIC:

call QWRITEPART(q.parm$,record$,ret%)

Call from C:

int ret = q_write_part((struct q_parm_ *)&q_parm,char *record);

Partial add to existing record data (all file types) - QADDPART

The user must fill the key (if index file) and the field to add in user record. Then fill q_parm.offset with the offset of the first byte of the field to add. Finally, fill q_parm.length with the length of the fields to add.

if q_parm.length equals 1 then a byte (C unsigned char type) add is performed.

if q_parm.length equals 2 then a word (C unsigned int type) add is performed.

if q_parm.length equals 4 then a 4 byte (C unsigned long type) add is performed.

Call from BASIC:

call QADDPART(q.parm$,record$,ret%)

Call from C:

int ret = q_add_part((struct q_parm_ *)&q_parm,char *record);

Get keys information on file - QACTIVKEYS

Get information into user's record.

User's record minimum length is 80 bytes formatted:

```
    4 bytes, long C format , the number of active keys (non
deleted
          records in Index or FIFO file.
    Word, total number of blocks for this file.
    Word, number of free blocks if Index file,-1 if FIFO or
          relative file.
    Word, block size in bytes.
    Word, record size in bytes.
    4 bytes, long C format, FIFO GET POINTER location.
    4 bytes, long C format, FIFO PUT POINTER location.
    Byte, file status,  bit 0:1=loaded, 0=not loaded
                        bit 1:1=Q-LOAD failed
                        bit 2:1=file written since last
OPEN
    4 bytes, long C format, total records in block
    2 bytes, unsigned int C, last file checksum
    2 bytes, unsigned int C, record length of relative
expansion
             linked file
    2 bytes, unsigned int C, key length
    2 bytes, unsigned int C, key offset
    2 bytes, unsigned int C, Q-dex flag offset
```

```
    other bytes reserved.
```

Call from BASIC:

```
call QACTIVEKEYS(q.parm$,record$,ret%)
```

Call from C:

```
int ret = q_active_keys_num((struct q_parm_ *)&q_parm,char
*record);
```

Perform file checksum (all file types) - QCHKSUM

The file is read and a checksum is calculated by word adding. The result is returned in the first 2 bytes of user record.

If the file is an Index file only active blocks are calculated.

If the file is a FIFO file only active records are calculated.

If the file is a relative file all the file is calculated.

Call from BASIC:

call QCHKSUM(q.parm$,record$,ret%)

Call from C:

int ret = q_file_chksum((struct q_parm_ *)&q_parm,char *record);

Perform partial file checksum (all file types) - QMASKCHKSUM

The file is read and a checksum is calculated by word adding. Part of each record is masked and not calculated in the checksum. q_parm.offset must be filled with the offset of the first byte of the masked field.

q_parm.length must be filled with the length of the masked field.

The result is returned in the first 2 bytes of user record.

If the file is an Index file only active blocks are calculated.

If the file is a FIFO file only active records are calculated.

If the file is a relative file all the file is calculated.

Call from BASIC:

```
call QMASKCHKSUM(q.parm$,record$,ret%)
```

Call from C:

```
int ret = q_mask_chksum((struct q_parm_ *)&q_parm,char
*record);
```

Sequential read from relative file - QSREAD

Read from relative files starting from current file pointer position, q_parm.length must be filled with the number of bytes to read. The actual number of bytes read is returned in q_parm.length. The file pointer is positioned at the end of the data read.

If q_parm.option bit 0 is set to 1, a SEEK operation is done before the read, the file pointer is moved to location defined by a combination of setting:

q_parm.write_part_offset and

q_parm.hi_word_writ_part_offset

FIFO files may be also read by QSREAD command.

Call from BASIC:

```
call QSREAD(q.parm$,record$,ret%)
```

Call from C:

```
int ret = q_sread((struct q_parm_ *)&q_parm,char *record);
```

Sequential write to relative file - QSWRITE

Write to relative files starting from current file pointer position, q_parm.length must be filled with the number of bytes to write. The file pointer is positioned at the end of the data read.

If q_parm.option bit 0 is set to 1, a SEEK operation is done before the write, the file pointer is moved to a location defined by a combination of setting:

q_parm.write_part_offset and

q_parm.hi_word_writ_part_offset

FIFO files may be also written by QSREAD command, but take care not to overwrite the QuickDex reserved flag byte!

Call from BASIC:

```
call QSWRITE(q.parm$,record$,ret%)
```

Call from C:

```
int ret = q_swrite((struct q_parm_ *)&q_parm,char *record);
```

Get file pointer position (relative files) - QTELL

The position of the file pointer is returned in the first 4 bytes of user record, in long 'C' format.

Call from BASIC:

```
call QTELL(q.parm$,record$,ret%)
```

Call from C:

```
int ret = q_tell((struct q_parm_ *)&q_parm,char *record);
```

Set file pointer position (relative files) - QSEEK

The new position of the file pointer is read from the first 4 bytes of user record, in long 'C' format.

If q_parm.offset equals 0, this position is taken as an offset from the beginning of the file.

If q_parm.offset equals 1, this position is taken as an offset from the current position of the file pointer.

Call from BASIC:

```
call QSEEK(q.parm$,record$,ret%)
```

Call from C:

```
int ret = q_seek((struct q_parm_ *)&q_parm,char *record);
```

Random read from relative file - QRREAD

q_parm.length must be filled with the record size to read. q_parm.offset low and high parts must be filled with the record number to read.

The exact read location is (record_size * record_number). The first record in the file is record number 0. The actual number of bytes read is returned in q_parm.length. The file pointer is positioned at the end of the data read.

FIFO files may be also read by the QRREAD command.

Call from BASIC:

```
call QRREAD(q.parm$,record$,ret%)
```

Call from C:

```
int ret = q_rread((struct q_parm_ *)&q_parm,char *record);
```

Random write to relative file - QRWRITE

q_parm.length must be filled with the record size to write. q_parm.offset low and high parts must be filled with the record number to write.

The exact write location is (record_size * record_number). The first record in the file is record number 0.

The file pointer is positioned at the end of the data written.

FIFO files may also be written by the QSREAD command, but with great care not to overwrite the QuickDex reserved flag byte!

Call from BASIC:

```
call QRWRITE(q.parm$,record$,ret%)
```

Call from C:

```
int ret = q_rwrite((struct q_parm_ *)&q_parm,char *record);
```

Read from FIFO file - QFREAD

Read the first record of FIFO file, the location of this record is kept in FIFO GET POINTER.

User record is filled with the data of this record, the record is deleted from the file and FIFO GET POINTER is updated to point to the next record in the file.

Call from BASIC:

```
call QFREAD(q.parm$,record$,ret%)
```

Call from C:

```
int ret = q_fread((struct q_parm_ *)&q_parm,char *record);
```

Write to FIFO file - QFWRITE

Write user record after the last record of FIFO file, the location to write to is kept in FIFO PUT POINTER.

FIFO PUT POINTER is updated to point to the first free record in the file.

If the file is full, and if wrap around is permitted (depending on Q-SETUP parameters), the first record is deleted and the record is written. If wrap around is not permitted then the record is not written and the user receives a file full error code.

Call from BASIC:

```
call QFWRITE(q.parm$,record$,ret%)
```

Call from C:

```
int ret = q_fwrite((struct q_parm_ *)&q_parm,char *record);
```

Multiple write to FIFO File - QBLOCKFWRITE

Write n records after the last record of FIFO file, the location to write is kept in FIFO PUT POINTER.

n- Number of FIFO records to write is read from the first 2 bytes of user record, then n FIFO records are read from the rest of the record and written to the file.

FIFO PUT POINTER is updated to point to the next record in the file.

If the file is full and if wrap around is permitted, (depending on Q-SETUP parameters) the first records are deleted and all the requested records are written. If wrap around is not permitted then the overflowing records are not written and the user receives a file full error code.

Call from BASIC:

```
call QBLOCKFWRITE(q.parm$,record$,ret%)
```

Call from C:

```
int ret = q_block_fwrite((struct q_parm_ *)&q_parm,char *record);
```

View FIFO file - QFVIEW

Read the n'th record of FIFO file, the user sets this in q_parm offset, low and high parts.

n is treated as an offset to FIFO GET POINTER. First record number is 0.

User record is filled with the data of this record. Unlike QFREAD, the record is not deleted from the file and FIFO GET POINTER is not changed.

Call from BASIC:

```
call QFVIEW(q.parm$,record$,ret%)
```

Call from C:

```
int ret = q_fview((struct q_parm_ *)&q_parm,char *record);
```

Read QuickDex block - QGETBLOCK

Read block number n from file into user's record. User's record minimum length is the file's block size n- block number is filled by the user in q_parm.offset fields.

Call from BASIC:

```
call QGETBLOCK(q.parm$,record$,ret%)
```

Call from C:

```
int ret = q_get_block((struct q_parm_ *)&q_parm,char *record);
```

Dismount QuickDex from memory - QDISMOUNT

When QuickDex is dismounted from memory, the memory is returned to DOS. DOS is able to use this memory only if QuickDex was the last resident program loaded.

Call from BASIC:

call QDISMOUNT(q.parm$,record$,ret%)

Call from C:

int ret = q_dismount((struct q_parm_ *)&q_parm,char *record);

Set keys information on file - QSETACTIVEKEYS

Set information from user's record into QuickDex parameters.

User's record minimum length is 80 bytes formatted:

4 bytes, long C format, the number of active keys (non deleted records in Index or FIFO file.

Word, total number of blocks for this file.

Word, number of free blocks if Index file,-1 if FIFO or relative file.

Word, block size in bytes.

Word, record size in bytes.

4 bytes, long C format, FIFO GET POINTER location.

4 bytes, long C format, FIFO PUT POINTER location.

Byte, file status, bit 0:1=loaded, 0=not loaded

bit 1:1=Q-LOAD failed

bit 2:1=file written since last OPEN

4 bytes, long C format, requested maximum number of records in file

other bytes reserved.

Call from BASIC:

```
call QSETACTIVEKEYS(q.parm$,record$,ret%)
```

Call from C:

```
int ret = q_set_active_keys((struct q_parm_ *)&q_parm,char *record);
```

Note

Be careful when calling this function. Incorrect data may cause serious problems.

Flush file - Q_FLUSH

if q_parm.option equals 0 :

QuickDex flushes the write cache to the disk, and updates the directory entry with the system date and time if the file was changed since the last open command.

if q_parm.option equals 1:

The file is set to guaranteed write mode on.

if q_parm.option equals 2:

The file is set to guaranteed write mode off

Call from BASIC:

```
call QFLUSH(q.parm$,record$,ret%)
```

Call from C:

```
int ret = q_flush((struct q_parm_ *)&q_parm,char *record);
```

Quickdex - basic interface

Interface to Microsoft BASIC compiler version 2-7 is provided by linking with QLNKB.OBJ module.

QuickDex - C interface

Interface to Microsoft C compiler versions 3 to 8 is provided by linking with LNKC.OBJ module for small models, QLNKCM.OBJ for medium models and QLNKCL.OBJ for large models.

QuickDex - assembler interface

QuickDex can be run by calling DRVPOS entry 5.

```
--------------------------------

DI = 0  Init   (not used now)

--------------------------------

DI = 1  set parameters from user into driver

  INPUT PARAMETERS:

    DS:SI - pointer to parameter string

  OUTPUT PARAMETERS:

    AX    - 0 = OK, else = error

--------------------------------

DI = 2 get parameters from driver into user

  OUTPUT PARAMETERS:

    DS:SI - pointer to parameter string

    AX    - 0 = OK, else = error

--------------------------------

DI = 3 memory indx write (for loading only)

  INPUT PARAMETERS:

    DS:BX - ptr to index number and options

    DS:SI - pointer to key record

  OUTPUT PARAMETERS:

    AX    - 0 = OK, else = error
```

```
--------------------------------

DI = 4  key read by index

  INPUT PARAMETERS:

    DS:BX - ptr to index number and options

    DS:SI - pointer to user record

  OUTPUT PARAMETERS:

    AX    - 0 = OK, else = error

    DS:SI - pointer to updated user record

--------------------------------

DI = 5  start by index (find greater or equal item)

  INPUT PARAMETERS:

    DS:BX - ptr to index number and options

    DS:SI - pointer to user record

  OUTPUT PARAMETERS:

    AX    - 0 = OK, else = error

    DS:SI - pointer to user record

--------------------------------
```

```
DI = 6  read next by index
  INPUT PARAMETERS:
    DS:BX - ptr to index number and options
    DS:SI - pointer to user record
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error,EOF
    DS:SI - pointer to user record
--------------------------------


--------------------------------
DI = 7  item read previous by index
  INPUT PARAMETERS:
    DS:BX - ptr to index number and options
    DS:SI - pointer to user record
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error,BOF
    DS:SI - pointer to user record
--------------------------------
DI = 8  item insert by user record
  INPUT PARAMETERS:
    DS:BX - ptr to index number and options
    DS:SI - pointer to user record
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
```

```
--------------------------------

DI = 9  item write by user record

  INPUT PARAMETERS:

    DS:BX - ptr to index number and options

    DS:SI - pointer to user record

  OUTPUT PARAMETERS:

    AX    - 0 = OK, else = error

--------------------------------

DI = 0AH item delete by index

  INPUT PARAMETERS:

    DS:BX - ptr to index number and options

    DS:SI - pointer to user record

  OUTPUT PARAMETERS:

    AX    - 0 = OK, else = error

--------------------------------

DI = 0BH item Undelete by index

  INPUT PARAMETERS:

    DS:BX - ptr to index number and options

    DS:SI - pointer to user record

  OUTPUT PARAMETERS:

    AX    - 0 = OK, else = error

--------------------------------
```

```
DI = 0CH memory indx write next (for loading only)
  INPUT PARAMETERS:
    DS:BX - ptr to index number and options
    DS:SI - pointer to key record
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
--------------------------------
DI = 0Dh get drv_active flag
  returns ax = 0 not active
          ax > 0 active already,
--------------------------------
DI = 0EH open index file, test map validity,reset last read
  INPUT PARAMETERS:
    DS:BX - ptr to index number and options
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
--------------------------------
DI = 0FH dismount q-dex
  INPUT PARAMETERS:
    DS:SI - pointer to 55AAh string
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
--------------------------------
```

```
--------------------------------
DI = 10H write part of record
  INPUT PARAMETERS:
    DS:BX - ptr to index number,options,flag,string off,len
    DS:SI - pointer to key record
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
--------------------------------
DI = 11H close index file ,update leading record.
  INPUT PARAMETERS:
    DS:BX - ptr to index number and options
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
--------------------------------
DI = 12H get one block
  INPUT PARAMETERS:
    DS:BX - ptr to index number,options,flag,string off,len
            string offset is block number;
    DS:SI - pointer to block(user must define at least
            block_size)
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
--------------------------------
```

```
DI = 13H set one block (not used now)
  INPUT PARAMETERS:
    DS:BX - ptr to index number,options,flag,string off,len
            string offset is block number;
    DS:SI - pointer to block
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
--------------------------------
DI = 14H get file chksum
  INPUT PARAMETERS:
    DS:BX - ptr to index number,options,flag
    DS:SI - pointer to chksum
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
--------------------------------
DI = 15H get active keys number
  INPUT PARAMETERS:
    DS:BX - ptr to index number,options,flag
    DS:SI - pointer to data structure containing
                  long: active keys_number
                  word: total blocks in file
                  word: free blocks in file
                  word: block size
                  word: record size
```

```
                    long: fifo get pointer
                    long: fifo put pointer
                    byte: key flag-0=index not loaded,1=loaded
                                  2=load failed (bad file)
                    long: requested max records in file
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
--------------------------------
DI = 16H last key read
  INPUT PARAMETERS:
    DS:BX - ptr to index number and options
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
    DS:SI - pointer to updated user record
--------------------------------
```

```
--------------------------------
DI = 17H memory indx write direct (for loading only)
  INPUT PARAMETERS:
    DS:BX - ptr to index number and options
            string offset is block number;
    DS:SI - pointer to key
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
--------------------------------
DI = 18H get memory key data
  INPUT PARAMETERS:
    DS:BX - ptr to index number and options
    DS:SI - pointer to key
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
    DS:SI - pointer to key,word sector,double word relative
            position in memory index
--------------------------------
DI = 19H  sequential read (for relative files only)
  INPUT PARAMETERS:
    DS:BX - ptr to file number, options and length
    DS:SI - pointer to user record
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
```

```
    DS:SI - pointer to updated user record
    DS:BX - pointer to file number, options
            and actual read length
--------------------------------
DI = 1AH  sequential write (for relative files only)
  INPUT PARAMETERS:
    DS:BX - ptr to file number, options and length
    DS:SI - pointer to user record
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
    DS:SI - pointer to updated user record
    DS:BX - pointer to file number, options
            and actual write length
--------------------------------
DI = 1BH  seek (for relative files only)
  INPUT PARAMETERS:
    DS:BX - ptr to file number, options and offset
            offset=0 seek from start of file
            offset=1 seek from current file pointer position
    DS:SI - pointer to seek offset (C long format)
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
--------------------------------
DI = 1CH  tell - get file pointer (for relative files only)
  INPUT PARAMETERS:
```

```
    DS:BX - ptr to file number, options
    DS:SI - pointer to C long format
  OUTPUT PARAMETERS:
    DS:SI - pointer to file pointer position (C long format)
            offset from start of file
    AX    - 0 = OK, else = error
--------------------------------


DI = 1DH  random read (for relative files only)
  INPUT PARAMETERS:
     DS:BX - ptr to file number, options record number and
             length
     DS:SI - pointer to user record
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
    DS:SI - pointer to updated user record
    DS:BX - pointer to file number, options
            and actual read length
--------------------------------
DI = 1EH  random write (for relative files only)
  INPUT PARAMETERS:
    DS:BX - ptr to file number, options record number and
            length
    DS:SI - pointer to user record
```

```
  OUTPUT  -PARAMETERS:
    AX    - 0 = OK, else = error
    DS:SI - pointer to updated user record
    DS:BX - pointer to file number, options
            and actual write length
--------------------------------
DI = 1FH  Fifo read (for fifo files only)
  INPUT PARAMETERS:
    DS:BX - ptr to file number, options
    DS:SI - pointer to user record
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
    DS:SI - pointer to updated user record
--------------------------------
DI = 20H  Fifo write (for FIFO files only)
  INPUT PARAMETERS:
    DS:BX - ptr to file number, options
    DS:SI - pointer to user record
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
--------------------------------
DI = 21H empty file (delete all index file)
  INPUT PARAMETERS:
    DS:BX - ptr to index number and options
  OUTPUT PARAMETERS:
```

```
    AX    - 0 = OK, else = error
--------------------------------
DI = 22H  Fifo view :read without delete (for FIFO files
                    only)
  INPUT PARAMETERS:
    DS:BX - ptr to file number, options, offset from get
            ptr
    DS:SI - pointer to user record
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
    DS:SI - pointer to updated user record
 --------------------------------
DI = 23H get file chksum with masking drim write_part off
     and len.
   INPUT PARAMETERS:
    DS:BX - ptr to index number, options, flag
    DS:SI - pointer to chksum
   OUTPUT - PARAMETERS:
    AX    - 0 = OK, else = error
 --------------------------------
 DI = 24H set one sector
   INPUT PARAMETERS:
     DS:BX - ptr to index number,options,flag,string
             off,len string offset is block number;
             string length is sector number in block
```

```
    DS:SI - pointer to sector data
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
--------------------------------
DI = 25H set free blocks till EOF
  INPUT PARAMETERS:
    DS:BX - ptr to index number,options,flag,string
            off,len string offset is 1st free block
            number;
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
--------------------------------
```

```
 DI = 26H set active keys number
   INPUT PARAMETERS:
     DS:BX - ptr to index number,options,flag,string off,len
     DS:SI - pointer to data structure containing
                 long: active keys_number
                 word: total blocks in file
                 word: free blocks in file
                 word: block size
                 word: record size
                 long: fifo get pointer
                 long: fifo put pointer
                 byte: key flag-0=index not loaded,1=loaded
                              2=load failed (bad file)
                 long: requested max records in file
   OUTPUT PARAMETERS:
     AX    - 0 = OK, else = error
-----------------------------------------
 DI = 27H get one sector
   INPUT PARAMETERS:
     DS:BX - ptr to index number,options,flag,string off,
             len string offset is block number;
             string length is sector number in block
     DS:SI - pointer to sector data
OUTPUT -PARAMETERS:
```

```
     AX    - 0 = OK, else = error

 --------------------------------

DI = 28H set free block, absolute sectors mapping

  INPUT PARAMETERS:

    DS:BX - ptr to index number, options, flag, string off,
len

            string offset is free block number;

  OUTPUT PARAMETERS:

    AX    - 0 = OK, else = error

--------------------------------

--------------------------------

DI = 29H reserved

--------------------------------

DI = 2AH reserved

--------------------------------

DI = 2BH reserved

--------------------------------
```

```
DI = 2cH add to part of record

  INPUT PARAMETERS:

    DS:BX - ptr to index number, options, flag, add off,
add_len

            add_len can be 1, 2 or 4 to add byte, word or C
long

    DS:SI - pointer to key record

  OUTPUT PARAMETERS:

    AX    - 0 = OK, else = error

--------------------------------

DI = 2DH reserved

--------------------------------

DI = 2EH  Fifo Block  write (for FIFO files only)

          writes N fifo records

  INPUT PARAMETERS:

    DS:BX - ptr to file number, options

    DS:SI - pointer to user record

            first word contains N=number of records to write.

            Then follow the records data. (N*Recrod size
bytes)

   OUTPUT - PARAMETERS:

    AX    - 0 = OK, else = error

---------------------------------------


DI = 2F checksum with mask on one byte

  INPUT PARAMETERS:
```

```
    DS:BX - ptr to index number and options
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
--------------------------------

DI = 30 Flush cache buffers
  INPUT PARAMETERS:
    DS:BX - ptr to index number and options
  OUTPUT PARAMETERS:
    AX    - 0 = OK, else = error
--------------------------------
```

Q-dex.prm file structure

This file contains initial setup parameter values for the driver in this format:

```
POSITION:LENGTH, DESCRIPTION

            word refers to 2 bytes length ordered low, high.


pos  0:word, DRVPOS ENTRY number, must be 5, 2 bytes length

pos  2:word, Reserved

pos  4:word, mode flags, RESERVED

pos  6:word, Old q-dex2 maximum number of indexes used (0 to
32), RESERVED

pos  8:word, maximum number of indexes used (0 to 255)

pos 10:20 bytes, directory of super indexes

pos 30:20 bytes, directory of file backing for fast load

pos 50:206 bytes, reserved.


------------------------------------------------------------
---


Pos 256 to 511 contains information about file number 0.


pos 256: word, key length .

pos 258: 10 bytes, reserved.

pos 268: word, record size
```

pos 270: word, key offset from start of record.

pos 272: word, flag byte offset from start of record.

pos 274: word, Block size,0,512,1024,2048 or 4096

pos 276: Double word, Maximum number of items in file

pos 280: word, key programmed flag,5aa5 if programmed .

pos 282: word, number of active records in QuickDex block after

split.

pos 284: word, linked file num. if 0 this is stand alone,

otherwise it is the number of the linked file+1.

Used to link an expansion file to an index file.

pos 286: byte, reserved

pos 287: 39 bytes, ascii file name including drive and path.

must terminate with binary 0,must start with drive:

pos 326: byte, reserved

pos 327: byte, super index linked file number

pos 328: word, mode flags, 16 bit fields

bit 0: 1= relative file

bit 1: 1= test key numeric ascii during load

bit 2: 1= test key packed BCD during load

bit 3: 1= FIFO file

bit 4: 1= wrap around if Fifo full, no err

bit 5: 1= this file is a memory file.

bit 6: 1= this is relative expansion of index file

bit 7: 1= this is a SUPER INDEX file

pos 330: 155 bytes, reserved.

pos 485: word, percentage of records in block after split

pos 487: 25 bytes, remarks (ascii).

Positions 512 to 767 contain information about file number 1.

Positions 768 to 1023 contain information about file number 2.

Then follows information in the same format for files 3-255.

Steps for defining super index in high memory

QuickDex does not use directly the high memory but rather can be configured to use a super index file for a specified file, instead of the memory index. This file can be a ramdrive file residing in high memory, so the regular memory space consumed by QuickDex indexes can be freed.

To achieve super index in high memory you must do the following:

- Load any Ram-Disk program where the ram drive resides in high memory, for example MICROSOFT RAMDRIVE in CONFIG.SYS:

```
DEVICE=C:\DOS\HIMEM.SYS

DEVICE=C:\DOS\RAMDRIVE.SYS 600 512 64 /e
```

  For more details about loading these drivers into high memory and parameters see Microsoft DOS documentation. If you have physical drives C: and D: the ramdrive will be E:

- Configure QuickDex to use the ram drive for Super-Indexes:

  In Q-SETUP, [F2] - general parameters, change parameter number 6 (Rlt super index path) to the Ramdrive path E:\

- Configure the QuickDex file to use super index:

  In Q-SETUP, [F1] - (define QuickDex files), choose the appropriate file number.

Index file

If the file is an index file (0 in parameter number 11), you must define a new QuickDex file that will be the super index. Enter the super index file number and file name.

Q-setup computes and updates all necessary parameters for the super index.

Other types of files (FIFO,Relative,Relative expansion)

Other file types do not have any index or super index in version 4.

- Quit Q-SETUP program.
- Copy Q-DEX.PRM file to the second PC. Both PCs must have equal QuickDex configuration.
- Reboot both Pcs.

QuickDex memory allocation

Internal tables

QuickDex Version 3.50 and later, dynamically allocates its internal tables according to the maximum files requested (Q-Setup general parameters).

- Variables definitions for memory allocation for indexes:

record_size:	Size of record including key and QuickDex flag, defined in Q-SETUP.
ptr_size:	Size of sector pointer, 2 for file size less than 32MB, 3 for larger file sizes.
key_size:	Size of key in index file, defined in Q-SETUP.
Max_records:	Maximum records in file, defined in Q-SETUP.
records_in_block:	Number of records in block, computed.
max_keys:	Maximum number of keys in index file, computed.
INDEX_MEMORY:	Total memory needed for index (allocated in regular memory).

SUPER_INDEX_FILE_SIZE:
File size needed for super index file. Allocated in disk or ram disk, according to Q-SETUP. Ram disk can be allocated in regular extended or expanded memory according to CONFIG.SYS setting.

SUPER_INDEX_MEMORY:

Total memory needed for super index, allocated in regular memory.

- Memory allocation for Index file without super index:

  records_in_block = INTEGER(block_size / record_size)

  max_keys = INTEGER (max_records / records_in_block) + 5

  INDEX_MEMORY = (key_size + ptr_size) * max_keys

- Memory allocation for Index file with super index:

  records_in_block = INTEGER(block_size / record_size)
  max_keys = INTEGER (max_records / records_in_block) + 5
  INDEX_MEMORY = 0 --> no memory allocation for the file itself.

  SUPER_INDEX_key_size = key_size + 2

  SUPER_INDEX_record_size = key_size + 3 + ptr_size;

  SUPER_INDEX_MAX_RECORDS = ((max_records * record_size)/block_size) * 3.5 + 10 computed automatically by Q-SETUP.

SUPER_INDEX_records_in_block=INTEGER(SUPER_INDEX_block_size / SUPER_INDEX_record_size)
SUPER_INDEX_block size is 4096 for Version 4 (512 for Ram disk drives in version 3.5).

SUPER_INDEX_max_keys = INTEGER(SUPER_INDEX_max_records / SUPER_INDEX_records_in_block)+3

SUPER_INDEX_MEMORY=(SUPER_INDEX_key_size +SUPER_INDEX_ptr_size)

* SUPER_INDEX_max_keys

```
SUPER_INDEX_FILE_SIZE =  SUPER_INDEX_max_keys
* SUPER_INDEX_block_size
```

- Memory allocation for non-index files:

  Non-index files do not require memory in version 4.

Changes from Q-Dex ver 3.5 to Q-Dex ver 4

QuickDex Version 4 uses standard DOS file calls instead of absolute disk read/write calls.

- No index memory is needed for relative, FIFO and Relative expansion files.
- The memory requirement for index and super index files is reduced.
- Block size can be set to the maximum of 4096 bytes, disregarding the physical cluster size, thus reducing index memory requirements.
- Performance is also good with SMARTDRV version 4 and 5, supplied in DOS 6.0, DOS 6.2 and WINDOWS. This is a solution to some cases where smartdrv.sys caused disk errors and system hanging. Using the new smartdrv increases overall system performance (such as chaining programs etc.).

- QuickDex files can now be located on any virtual disks, such as LAN file servers. However, since the memory for index files is located in the client PC, QuickDex index files can only be accessed properly from a single client.
- An updated BOOT.EXE program is provided that enables smartdrive and DOS to flush is write cache buffers to the disk. This program must be used and the old BOOT.COM program should be removed.

- FILES=NN is config.sys must be modified. NN must be set to at least the number of QuickDex files + 20 (since Q-DEX 4 uses DOS files).

- On POSition servers, smartdrv can be used with write cache, since the transactions from the POSs are guaranteed by POSition end-to-end transaction communication protocol, i.e. even if a transaction is not written to the server's disk because of power failure, the POS is requested to re-transmit it after the servers are loaded again. However, the Q-FLUSH call should be used to guarantee write at checkpoints, depending on the application.

- If needed (for POS etc.), guaranteed write can be set on several files by Q-setup or Q-PRM programs. SMARTDRV write cache can still be enabled on other files. Setting a file to guaranteed write, slowed down random write from 30ms to 85ms on a 30,000 item test file.
- An application can change the guaranteed write settings on-the-fly with a new function Q_FLUSH.
- Calling Q_FLUSH with q_parm.option zero flushes write cache buffers to the disk immediately.
- Calling Q_FLUSH with q_parm.option 1 sets the file to guaranteed write mode.
- Calling Q_FLUSH with q_parm.option 2 sets the file to non-guaranteed write mode.
- Memory requirements example: Just installing QuickDex 4 on a standard POSition server reduced QuickDex memory from 113K to 80K. Furthermore, after tuning Q-SETUP to 4096 bytes blocks for all the files, the memory required decreased to 67KB.

Performance example:

Running a benchmark program that simulates a transaction background process of 44650 transactions from a store using a 26000 item PLU file, gave the following results:

QuickDex version 3 with DOS 5 SMARTDRV.SYS: 890 sec.

QuickDex version 4 with DOS 6 SMARTDRV.EXE *without* write cache: 314 sec.

QuickDex version 4 with DOS 6 SMARTDRV.EXE *with* write cache: 155 sec.

Note

In real life, the background task is activated by the timer, so all versions have the same process time. The advantage of the new version is that the PC would be almost free between timer ticks to perform the foreground programs.

- An enhanced conversion utility is provided to convert QuickDex index files to various block sizes with no need to load a file before the conversion.

  Usage: `Q-CONVRTV` [old fname] [new fname] /NEWBLKnn /RECLENnn /KEYLENnn /KEYOFFnn /FLGOFFnn

  Converts an index file to confirm with new block size.

  The new block size can be 512,1024,2048,4096 or 0.

  Block size = 0 is the same as 4096.

Note

Conversion is not needed for relative or FIFO files.

- A new utility, Q-CLOSE is provided to close already opened QuickDex files.

ICL Retail Systems
2933 Bunker Hill Lane, #101
Santa Clara, CA 95054

P/N 89000056
PIN 45001/009