DATAdesign

programming documentation

"Application Programming Interface"

PROGS, Professional & Graphical Software
Mechelbaan 344
2580 Putte
BELGIUM
tel : +32 (0)14/ 22 23 26
e-mail : PROGS@triathlon98.com
www : http://www.triathlon98.com/PROGS/

DATADesign Programming API : Table of Contents

Introduction

This is the manual for the third version of the DATAdesign engine (actually the second version never existed). The idea behind writing the DATAdesign engine was to get a powerful, multi-user database management system. The current version contains more than 80 extensions in a `DATAdesign.engine` thing which should do just that.

Please note that this manual does presume some programming expertise from the reader.

DATAdesign is what some people would call a free-form database. This means that no restrictions (ahum, as few as possible) are posed upon the creator and user of database files which are manipulated with the DATAdesign engine.

Concepts

file

A file is the entity which includes a set of related data to be used by the DATAdesign engine. When we use the word file in this manual, we don't mean a file as in "a file on disk", but we mean a file which is used by the DATAdesign engine. The conventional usage for file won't be used a lot in this manual, and the conventional usage will be called *medium-file*. A file may be on disk (*disk-based*), or it may be completely in memory (*memory-based*). When a medium-file is not used by any job in memory (so there are no buffers using the file (see later)), then this medium-file will not be called a file. Files are referenced by (hopefully) unique *filenames*. These filenames are case-dependent.

buffer

A buffer is an entry-point to a file. It contains a copy of the *current record*. If the buffer is not a *read-only buffer* then the record will be *locked*, that is, unavailable to all other buffers using this file. All operations which read or change records are done through a buffer. This means that you don't actually change the record in the file. To copy a record back into the actual file, you have to *implement* it. This makes sure that the record in the file is an exact copy of the record in the buffer if you had changed it. It doesn't change anything to the buffer, only to the file. The buffer can also be cleared, so as to obtain a new record etc. Last but not least all operations need a bufferid as this is the only way to let the DATAdesign engine know which file is affected or queried by a certain operation (but there are defaults).

Each buffer has a special property, a *bufferid* which is unique and defines every buffer. Buffers can only be accessed by the job who created them.

Please note that if you create a buffer (by using or creating a file), you also have to release it, as it will otherwise clog up some part in memory. Even when the job which uses the buffer is released, the buffer will keep existing (and nobody can access is). This can be solved with garbage collection.

index

An index is a special entry-point to a buffer, and thus a file. Indexes are used only for file navigation and fast searching. Indexes are the only way in which you can sort a file, or filter it, that is, specify an order in which the record are available, and/or select which records are available and which are not. Indexes are however restricted by memory. This should not be a big problem as each entry in an index uses a maximum of 94 bytes, and usually much less (that is 14 bytes plus the bytes need for each sort level, being 2 bytes for char or word, 4 bytes for long, and 8 bytes for text or double).

Each index has a special property, an *indexid* which is unique and defines every index. Indexes always define the buffer which was passed when they were created, and can't be shared by buffers. Some commands may be passed an indexid instead of a bufferid.

record

Records are parts of a file which combine related data. If you go to a library and you want to find a book, you search the register, which is a database. In those libraries where you still have to find them manually, there will be a place where you can find a card for each book. Each of these cards is a record, and all the cards together are the file.

All record have a special property, a *recordid* which is unique and defines every record. No two records in a file can ever have the same recordid, and a recordid never changes, even if the record is changed. Furthermore, even after a record has been deleted, the recordid won't be assigned to another record for a long while. This ensures that a recordid is the safest way to make links between records of different files.

More information on the programming approach to records can be found under buffer and in the explanation of the specific commands.

A recordid is a long integer (4 bytes). All values are possible except -1, which is used to denote "not found" or other problems. Recordids are assigned cyclic. So typically only one record per each cycle of 4*10^9 records which are created will get the same recordid.

field

Fields are subdivisions of records, and these subdivisions are available in all records. To use the example of the index at the library, the fields are the subdivisions of the cards, like `author, title, publisher,...`

field type

In the DATAdesign engine, all fields are typed. Five basic types are provided and these types should allow you to put any kind of data you want in a field. It is up to the author of a program to determine what a certain value in a field is supposed to mean.

All field types can be sorted except one. It is impossible to sort raw fields, as these can represent just about anything.

type	code	element size	usage
raw	1	1 byte	graphics, fonts, ...
char	2	1 byte	text
short	3	2 bytes	small integer values selections, statuses, ...
long	4	4 bytes	large integer values dates, ...
ieee	5	8 bytes	ieee double any numerical value

file-status

This is a special property which each file has. It indicates whether a file is disk-based (1) or memory-based (0). By default all files are memory based when you create them. But that can be changed. There is no automatic switching between the two statuses.

disk-based

This file-status is included for two reasons. Firstly to allow for very long files to be used, even files which are much longer than memory will permit. However there is always an index with references to the place which has to fit in memory. (This shouldn't be a problem, every record only takes 18 bytes in this index). Secondly it is the safest way you can work on a file. Even if a system-crash would occur for some reason, then a maximum of one record will be lost. There are a few commands which are actually not that safe, but it will be mentioned when they are discussed.

inter-record-space

When a file is disk based and a change to a record makes that record grow a bit, then it would have to be moved to the end of the file as it would not fit in the medium-file at the old place. To prevent this from happening too often, you can make sure that there is always a bit of empty space after each record. This space is called the inter-record-space. This is not relevant when a file is memory-based.

Even when a large inter-record-space is used, it may not be always be enough. That way large empty gaps will be created in the file. These gaps can be removed with garbage collection. A large inter-record-space is not advisable as it can waste a lot of disk space. We advice small values, e.g. 10.

lock

As DATAdesign is a fully multi-user database, it is necessary that records which are edited by a read-write buffer can't be accessed by other read-write buffers. So each record which is accessed by another read-write buffer is locked, unless this buffer is view only. View only records can always access *all* existing records.

Interfaces

There are three language interfaces for the DATAdesign engine. You should only read about those interfaces that you are going to use. However it may sometimes be interesting to read about the other interfaces as well, as this may give you some additional information (especially for assembler programmers). The source code for the C interface is provided on disk, so that may also be interesting to have a look at, and this for both C and Assembler programmers. The commands are all treated together, explaining what they do, and giving the specific details for each of the interfaces.

- SuperBASIC interface
    - parameters
- Assembler "interface"
    - parameters
- C interface
    - parameters

SuperBASIC interface

Actually, there isn't much which has to be mentioned before listing all the commands. You just interface through the Sbasic commands. But you have to know some details.

All routines which need a buffer- or indexid have an optional parameter at the start. If you explicitly specify this buffer- or indexid, then it has to be preceded by a hash (#). This is similar to the passing of channel ids to the i/o commands like print. If you don't pass a buffer- or indexid, then the default bufferid and/or indexid will be used.

You also have to know how errors are dealt with. All of the routines will only interrupt the program in either of two occasions, that is if the DATAdesign engine is not initialized, or when a bad parameter is encountered. Parameters can only be bad when the type of the parameter is completely wrong, or in the case of a fieldid, when it is passed as the fieldname and no field with the given name exists. Do note that fieldnames are case dependent.

In all other cases that errors are encountered, they will be returned in a special variable called `dd_err`. Three important remarks have to be made about this.

- This variable has to be spelled exactly like that, in lowercase.
- This variable has to be initialized, preferable before any call to the DATAdesign engine, i.e. at the start of the program.
- The fact that this variable is set at each call to any of the routines in the DATAdesign engine Sbasic interface is a side-effect. So the line

```
PRINT dd_err, recordID
```

will print the error generated by the call to recordID and its return value, and *NOT* the value dd_err had before the line was executed.

parameters

The Sbasic interface allows for some parameters to be left out, or just not specified, or to have several types. If there are default values for a parameter, then the default will be mentioned after the name in square brackets. If there is no default, but the parameter doesn't have to be specified, then the type will be put in square brackets. The possible parameter types are :

bufferid

always has to be a float. Can be left out (like with channel id's). If not stated the default will be used.

indexid

always has to be a float. Can be left out (like with channel id's). If not stated the default index of the default buffer will be used. If the given id was actually a bufferid, then the default index for that buffer will be used.

field

This parameter can be specified either as the fieldid, or as the fieldname. Fieldnames are case relevant. An error will be reported if the fieldname does not exist. The fieldid can be passed as an integer or as a float. If no value is given for this field, then the default will be used.

compare

This parameter can be passed either as an integer or float, or as a string. When passed as a numerical value, you have to add the values you want together. When passed as a string you just combine the specific characters. So "-C" or 48 would give the same result : reverse order and case dependent.

string

This parameter always has to be specified between quotes. If no string is specified, than a NULL string (that is "", the empty string) will be used.

short

This parameter can always be specified either as an integer or as a float. If this parameter is not optional and is not specified, then an error will be reported.

long

This parameter can only be passed as a float. If this parameter is not optional and is not specified, then an error will be reported.

char

This parameter is always specified as a string. The first character in the given string is the passed character.

float

This parameter has to be passed as a float, and will internally be converted to an IEEE double.

double

This parameter is passed as a pointer to 8 bytes in memory which should contain an IEEE double.

pointer

This is a parameter, passed in a float, which has to point to a piece of memory which can be written into, i.e. a piece of memory which was allocated with a call to `ALCHP` or `RESPR` or similar.

special

There is also a parameter which can have any of the types string, long, short, char, float. The actual type is then dependent. on some other part of the parameter parsing, or some internal structures of the DATAdesign engine. Parsing these parameters will cause an error when the wrong type is used (not in dd_err), so you should be careful.

pointer to special

This parameter is passed as a pointer to 8 bytes in memory which should contain either 8 characters (each in one byte, value 0 used for filling when less than 8 chars used), a long (in the first four bytes), a short (in the first 2 bytes), a char (also the second byte, first byte 0), or an IEEE double (conversion of a float to a double can be done with the `SETfloat` and `GETdouble` commands on a field in an empty record) (all eight bytes).

If a parameter is followed with *updated*, then the value of the parameter will be changed according to the command. If the parameter is not passed as a variable, and is not a pointer, then nothing is updated. If you want to compile your SuperBASIC programs which use DATAdesign with Qliberator, you MUST use the names option. If you don't do this, DATAdesign will not be able to find the dd_err variable ! The SuperBASIC interface can not handle strings which are longer than 256 characters. This is not a limitation of the DATAdesign engine, but only of the SuperBASIC interface !

Assembler "interface"

The entire DATAdesign engine is provided as an extension thing, and so you can hardly discuss an assembler interface. All the actual interfacing can be done with the standard thing system. If you want more information about this, you should buy "QDOS Reference Manual", available from Jochen Merz (and probably some others as well).

But we have also provided a few routines which are actually very similar to the access routines for the Menu Extension (also from Jochen Merz; also an extension thing). These routines are used to call and release one specific extension at a time. It is however advisable to keep one of the extensions "in use" during the execution of your program, as this prevents your buffer(s) from getting lost during execution of your program (this can only happen if someone gets the stupid idea to either remove the DATAdesign files job or DATAdesign.engine thing). You could use the "INFO" extension for this. This is an empty extension, which only contains some data which is used by the engine itself, and which is always the first extension in the list. Never actually call this extension !

So here are the access routines:

```
Use DATAdesign engine DDE_USE
        entry                       exit
d0                                  return
d2.l    extension id
a1                                  address of thing extension
error return:
-thing not implemented
-engine not found
-any other error
```

```
Free DATAdesign engine DDE_FREE
no parameters, no return

Call DATAdesign engine extension jsr $18(a1)
         entry                       exit
d0                                   return
a0       address of thing extension
a1       ptr to parameter list
```

The address of thing extension is the return value of the call to `DDE_USE` and can also be placed in any other address register.

These routines can be found in the library file `engine_lib`.

Naturally, it is also possible to use the C-interface directly from assembler if you prefer to do so.

parameters

The standard extension thing protocol is used.

bufferid

always passed in two long words. The second long word is the actual bufferid. If the MS (most significant) word of the first long word is zero, then the default buffer will be used, and the second long word becomes irrelevant.

indexid

always passed in two long words. The second long word is the indexid. If the MS word of the first long word is zero, then the default index of the default buffer will be used, and the second long word becomes irrelevant. If the given id was actually a bufferid, then the default index for that buffer will be used.

The engine will automatically recognize whether the given parameter was an indexid or a bufferid.

field

always passed as a long word. Only the LS word is important. If the value -1 is passed then the default field will be used.

string

strings are always passed as two long words. There are two possibilities. If you want to pass a standard QL string, then the MS word of the first long word has to be $0100, and the second long word is the pointer to the string. If you want to pass a substring (just a series of chars), then the MS word of the first long word has to be $0200, and the LS word has to be the length. The second long word contains the pointer to the substring.

short

this parameter is passed in a long word. The value is in the LS word.

long

this parameter is passed in a long word.

char

this parameter is passed as the LS byte of a long word.

double

this parameter is passed as an 8 bytes IEEE double.

return/update string

the returned string can be either a QL string, or a C string (null terminated). The C string is treated as return substring. The parameter is passed as two long words. The second long word is a pointer to the place where the string must be filled in. The first long word contains the type in the MS word, $a100 for QL string, $a200 for substring, and the maximum number of characters in the LS word.

return/update xxx

passed as two long words. The MS word of the first long word is $a000, the second long word must be a pointer to the place where the return value must be filled in, or where the value can be read and written after processing. If the update parameter doesn't have to be passed, and you don't want it updated, you should clear the first long word.

special

there is also a parameter which can have any of the types string, long, short, char, double. The actual type is then dependent. on some other part of the parameter parsing. It is passed in two long words, and the value is left justified. Char is then considered as an unsigned word. A string is max 8 chars, unused bytes must be zero.

pointer to special

this parameter is passed as a pointer to 8 bytes in memory which should contain either 8 characters (each in one byte, value 0 used for filling when less than 8 chars used), a long (in the first four bytes), a short (in the first 2 bytes), a char (also the second byte, first byte 0), or an IEEE double (all eight bytes). It is passed as two long words, the pointer has to be in the second long word.

C interface

There isn't much which has to be said about the C interface except all the commands. But before any of the commands can be called, you have to initialize the DATAdesign variable which contains the address of the DATAdesign.engine thing and which is used by all the other commands and macros. This can be done with the `DDinit()` routine. This also makes sure that your job uses the DATAdesign.engine thing. The thing can also be released with a call to `DDfree()`.

To be able to use the C interface in your C program files you have to include the "DATAdesign.h" file. You will also have to link the `DATAdesign_lib` file with your own source code files.

parameters

All parameters are of a standard C data-type. There is however one exception. Some routines require a pointer to a special 8 byte parameter. This parameter can be considered as a union like this:

```
union special {
    char string_par[8];
    long long_par;
    short short_par;
    short char_par;
    double double_par;
};
```

It may also be interesting to have a look at the *special* data type in the Assembler interface, and at the source of the C interface.

Furthermore, bufferid and indexid parameters are passed as a long. If a zero is passed, then the default buffer- or indexid will be used.

All C interface functions return the error which occurs during the execution of the engine.

Creating a buffer/file

Although the commands in this section are called NEWfile and USEfile, these are the commands which actually create new buffers, and which may – if that is the case – create a new file.

- NEWFile
- USEfile

NEWFile

This command always creates a new file. The new file will be empty (no records in it), memory-based, and won't have any fields, except one. All files always have a *MEMO* field. This is a character field, which is supposed to be invisible. It always has fieldid zero, and can never be deleted. It is included to make sure a file always has a field, and so you can have *invisible* notes.

When a file is created, you immediately have the possibility to set the filename. You don't have to do this, but it is always interesting to do so. It makes sure that you haven't got several un-named files flying about in memory, which can't be accessed by another job. You can create a file with a name that is already used by another file which is used, so be careful.

The new buffer will immediately be the default buffer for the current job.

```
Sbasic
    NEWfile filename
    filename : [string]

Assembler
    NEWF
    filename : [string]
    bufferid : return long

C
    long DDnewfile(char *filename);

errors, code,   meaning
imem    -3      insufficient memory
```

USEfile

This command creates a new buffer for a file with the given name. If there is no file in use with the given name, then the file will be loaded from the given device, or from the *data_use* device if the file can't be found. If there already was a file with given name, then another buffer to access that file will be created. To determine if a file already is in memory, the name is compared case dependent. If there is more than one file in memory with the same name, then you will probably get the last created one.

There is also a view parameter which can be set (<>0). If it is set, then the buffer will be read only, and no writing can be done through that buffer. Also the records that are viewed through such a buffer won't be locked by that buffer, and even record which are locked by another buffer can be viewed (you will get the version in the file, not the version in that other buffer).

The new buffer will immediately be the default buffer for the current job, and there will be no record in the buffer (as after `NEWrec`).

The medium-file which is loaded needs the `_ddf` extension. This extension should not be specified or it will be included in the filename.

If a bufferid is passed, then a new buffer which accesses the same file will be created.

```
Sbasic
    USEfile #bufferid, filename, filedevice, view
    filename : string
    filedevice : [string]
    view : short[0]

Assembler
    USE
    bufferid
    filename : string
    filedevice : [string]
    view : short

C
    long DDusefile(long bufferid, char *filename,
                   char *filedevice, short view);

errors, code,   meaning
imem    -3      out of memory
fdnf    -7      file not found
                maybe you just forgot to give a filename
isyn    -21     this is not a DATAdesign file
iexp    -17     the given DATAdesign file is too old
...             any other file error
```

Buffer maintenance

- UNUSebuffer
- CYCle Buffers for this job

UNUSebuffer

When you don't need a certain buffer any more, you have to call this command to remove it from memory. If the given buffer was the only buffer with access to a certain file, then the file will be closed as well.

Note that all buffers have to be removed before removing a job, as the buffers will otherwise remain in memory until the next garbage collection.

```
Sbasic
    UNUSEbuffer #bufferid

Assembler
    UNUS

C
    long DDunusebuff
    bufferider(long bufferid);

errors, code,   meaning
no errors possible
```

CYCle Buffers for this job

If you want to find out which buffers are in use by your job, you can cycle through them with this command. To get the first buffer in the list, you have to call it without specifying the bufferid. In any other case, the bufferid of the next buffer is returned, or zero if there is no next buffer. In Sbasic this can be used to remove all buffers like this :

```
    dd_err=0
    REPeat loop
        UNUSEbuffer #CYCLEbuffer
        IF dd_err THEN EXIT loop
    END REPeat loop
```

```
Sbasic
    ret=CYCLEbuffer(#bufferid)

Assembler
    CYCB
    bufferid
    return long bufferid
```

```
C
    long DDcyclebuffer(long bufferid, long *return);

errors, code,   meaning
ITNF    -7      invalid bufferid
```

File maintenance

These commands are all specific to the file. You only have to specify a bufferid so that the engine knows which file you want to change something for. The changes are however relevant for all buffers which use that file.

- GARBage collection
- Set Inter-Record-Space
- Set FileSTatus
- Set ViewSTatus
- CYCLe files

GARBage collection

Garbage collection can be quite important when using the DATAdesign engine. It always makes sure that any buffers whose owning job is no longer present will be removed from memory, possibly also releasing the file that this buffer was using (if it was the only buffer for that file). In Sbasic it can be used together with the routine to release all buffers which may not have been unused by Sbasic programs that were in memory before the current one.

It can be important not only to release the memory which is occupied by the buffer, but also because such buffers may keep a record locked. The only way to make sure no records are locked by buffers from jobs which no longer exist is exactly by collecting garbage.

If you pass a bufferid and the file that the given buffer accesses is disk-based, then any unnecessary white-space between records will be removed. If there was more white-space between two record than the IRS, then it will be made shorter. Garbage collection never increases the space between records. Please note that this command is not as safe as the others. If a power failure would occur during garbage collection, then it would probable be very difficult to recover your file !

```
Sbasic
    GARBAGE #bufferid
    the bufferid always has to be passed explicitly, the
    default buffer is not used automatically. So to call
    it for the default buffer, you should type
        GARBAGE #bufferID

Assembler
    GARB
    bufferid
```

```
C
    long DDgarbage(long bufferid);

errors, code,   meaning
itnf    -7      invalid bufferid
...             any file i/o error (should not occur)
```

Set Inter-Record-Space

You can set the inter-record-space for a certain file with this command. Note that only records which are implemented after this call will be imbedded in the given inter-record-space. As inter-record-space can never grow, not even with garbage collection, it is best to set the inter-record-space immediately when you create a file.

The inter-record-space is also considered when saving.

The inter-record-space has to be a value between zero and 255.

```
Sbasic
    SET_IRS #bufferid, irs
    irs : short

Assembler
    SIRS
    bufferid
    short irs

C
    long DDset_irs(long bufferid, short irs);

errors, code,   meaning
itnf    -7      invalid bufferid
orng    -4      inter-record-space not in 0-255 range
```

Set FileSTatus

This routine allows you to determine whether the file is to be disk-based or memory based. You can give a filename and filedevice in case you want to move the file to disk.

When you make a file disk-based (status=1), then the file is overwritten, so be careful. when giving the filename and filedevice.

When you move the file to memory (status=0), then the file-status is not changed on the medium-file. So if you don't save the file afterwards, and you load it again later (with a `USEfile` command), then the file will still be disk-based.

When an error occurs during the moving of the file from disk to memory (or vice versa), then *BIG* problems may occur. This can be tested by comparing the record count before and after the move. The buffer will be unused if problems occur when making the file memory-based.

The medium-file will get the `_ddf` extension. This extension never has to be specified. When it is specified, it also be included in the filename.

```
Sbasic
    SETFILEstatus #bufferid, status, filename, filedevice
    status : short
    filename : [string]
    filedevice : [string]

Assembler
    SFST
    bufferid
    short status
    optional string filename
    optional string filedevice

C
    long DDsetfilestatus(long bufferid, short status,
                         char *filename, char *device);

errors, code,   meaning
itnf    -7      invalid bufferid
ipar    -15     status has to be 0 or 1
                no filename
...             any file i/o error (see SAVE and USE)
```

Set ViewSTatus

This routine allows you to change the view status of the given buffer.

It will always clear the current record (as in NEWrec).

```
Sbasic
    SETVIEWstatus #bufferid, status
    status : short

Assembler
    SVST
    bufferid
    short status

C
    long DDsetviewstatus(long bufferid, short status);

errors, code,   meaning
itnf    -7      invalid bufferid
```

CYCLe files

If you want to find out which files are in use on your system, then you can cycle through them with this command. To get the first file in the list, you have to call it with zero as seed. In any other case, the updated value should be used. If you have reached the last file, then the seed will be updated to zero, and the string will be unchanged (null string for Sbasic).

The filename is always returned, so that you can USE the file if you want. A filename is maximum 32 characters long and enough space has to be provided for the return string (no problem in Sbasic).

In Sbasic you can print the filenames of all files like this :

```
    seed = 0
    REPeat loop
        name$ = CYCLEfile$(seed)
        IF NOT seed THEN EXIT loop
        PRINT name$
    END REPeat loop
```

Sbasic

```
    filename$ = CYCLEfile$(seed)
    seed : long
```

Assembler

```
    CYCL
    update long seed
    return string filename
```

C

```
    long DDcyclefile(long *seed, char *return);
```

errors,	code,	meaning
itnf	-7	seed invalid

File Navigation

All these commands (except Get RECord) only require one parameter, an indexid. If you specify an indexid, then the order of the records in that index will be used. If you specify a bufferid, then the order of the records in the default index for that buffer will be used. If there is no index for that buffer or the default indexid is zero, then the order of the records in the file will be used. That is the order specified by the recordid. Not specifying an indexid is the same as specifying the default bufferid.

- FiRST record
- LAST record
- NEXT record
- PREVious record
- Get RECord
- FORWard
- BaCKWard

FiRST record

Get the first available record. That is the first record if the given buffer is view only, or if the first record is not locked, else it is the first record which is not locked.

```
Sbasic
    FIRSTrec #indexid

Assembler
    FRST
    indexid

C
    long DDfirstrec(long indexid);

errors, code,   meaning
itnf    -7      invalid index- or bufferid
                no first record found
imem    -3      insufficient memory (only possible with indexes)
```

LAST record

Get the last available record. That is the last record if the given buffer is view only, or if the last record is not locked, else it is the last record which is not locked.

```
Sbasic
    LASTrec #indexid
```

```
Assembler
    LAST
    indexid

C
    long DDlastrec(long indexid);

errors, code,   meaning
itnf    -7      invalid index- or bufferid
                no last record found
imem    -3      insufficient memory (only possible with indexes)
```

NEXT record

Get the next record in the list which is not locked.

```
Sbasic
    NEXTrec #indexid

Assembler
    NEXT
    indexid

C
    long DDnextrec(long indexid);

errors, code,   meaning
itnf    -7      invalid index- or bufferid
                no next record found
imem    -3      insufficient memory (only possible with indexes)
```

PREVious record

Get the previous record in the list which is not locked.

```
Sbasic
    PREVrec #indexid

Assembler
    PREV
    indexid

C
    long DDprevrec(long indexid);

errors, code,   meaning
itnf    -7      invalid index- or bufferid
                no previous record found
imem    -3      insufficient memory (only possible with indexes)
```

Get RECord

Considering that recordids are the safest way to make links between files, there must be a way to get a record with a specified recordid. And that is exactly what `GETrec` does. View only files always get the requested record (unless it was deleted or never existed). When the current record is requested, then nothing happens (no error, but the record is not 'truncated' either).

```
Sbasic
    GETrec #bufferid, recordid
    recordid : long

Assembler
    GREC
    bufferid
    long recordid

C
    long DDgetrec(long bufferid, long recordid);

errors, code,   meaning
itnf    -7      invalid bufferid
                or record not found (invalid recordid)
imem    -3      insufficient memory (record too long)
fdiu    -9      record in use by another buffer (read/write buffers only)
```

FORWard

Get a certain following record. This command allows you to skip some records, that is to move through the file with a relative displacement. When given a displacement of one, this command is the same as `NEXTrec`.

If the requested record was locked, it will take the NEXT record.

The displacement should always be positive. Any value smaller than one will be treated as if it was one.

```
Sbasic
    NEXTrec #indexid, displacement
    displacement : short

Assembler
    FORW
    indexid
    short displacement

C
    long DDforward(long indexid, short displament);

errors, code,   meaning
itnf    -7      invalid index- or bufferid
                no next record found
imem    -3      insufficient memory (only possible with indexes)
```

BaCKWard

Get a certain preceding record. This command allows you to skip some records, that is to move through the file with a relative displacement. When given a displacement of one, this command is the same as `PREVrec`.

If the requested record was locked, it will take the PREVious record.

The displacement should always be positive. Any value smaller than one will be treated as if it was one.

```
Sbasic
    PREVrec #indexid, displacement
    displacement : short

Assembler
    BCKW
    indexid
    short displacement

C
    long DDbackward(long indexid, short displacement);

errors, code,   meaning
itnf    -7      invalid index- or bufferid
                no next record found
imem    -3      insufficient memory (only possible with indexes)
```

File Information

These commands can give you the information you may need on a certain file.

- FileNAMe
- FileDEVice
- CouNT All records
- Get Inter-Record-Space
- Get FileSTatus
- Get ViewSTatus

FileNAMe

Get the filename of the file this buffer has access to.

A filename is maximum 32 characters long, and enough space has to be provided for the return string (no problem in Sbasic).

```
Sbasic
    name$ = FILEname$(#bufferid)

Assembler
    FNAM
    bufferid
    return string filename

C
    long DDfilename(long bufferid, char *return);

errors, code,   meaning
itnf    -7      invalid bufferid
```

FileDEVice

Get the filedevice of the file this buffer has access to.

A filedevice is maximum 32 characters long, and enough space has to be provided for the return string (no problem in Sbasic).

```
Sbasic
    device$ = FILEdevice$(#bufferid)
```

```
Assembler
    FDEV
    bufferid
    return string filedevice

C
    long DDfiledevice(long bufferid, char *return);

errors, code,   meaning
itnf    -7      invalid bufferid
```

CouNT All records

Find out how many records are in this file. This function returns the total amount of records, and doesn't take any index into consideration.

```
Sbasic
    number = COUNTall(#bufferid)

Assembler
    CNTA
    bufferid
    return long number
C
    long DDcountall(long bufferid, long *return);

errors, code,   meaning
itnf    -7      invalid bufferid (or indexid actually)
```

Get Inter-Record-Space

Find out the current value for the inter-record-space.

```
Sbasic
    irs = GET_IRS(#bufferid)

Assembler
    GIRS
    bufferid
    return short irs

C
    long DDget_irs(long bufferid, short *return);

errors, code,   meaning
itnf    -7      invalid bufferid
```

Get FileSTatus

Find out whether the file this buffer works on is memory-based (status=0), or disk-based (status=1).

```
Sbasic
    status = FILEstatus(#bufferid)
```

```
Assembler
    GFST
    bufferid
    return short status

C
    long DDfilestatus(long bufferid, short *return);

errors, code,   meaning
itnf    -7      invalid bufferid
```

Get ViewSTatus

Find out whether this buffer is read/write (status=0), or view only (status=1).

```
Sbasic
    status = VIEWstatus(#bufferid)

Assembler
    GVST
    bufferid
    return short status

C
    long DDviewstatus(long bufferid, short *return);

errors, code,   meaning
itnf    -7      invalid bufferid
```

File saving

Entire file (SAVE)

This command can only be used if your file is memory-based. It makes sure that there is an up to date version of your file on disk. The record in the buffer is implemented first. The filename and devicename are changed if passed. If no filedevice passed and no default filedevice exists, and saving to just the filename doesn't work, then the file is saved to the *data_use* device.

You can also state the overwrite status (over). If this is zero and the medium-file already exists, an error will be reported. If over is set, then the medium-file will be overwritten.

The medium-file will get the `_ddf` extension. This extension never has to be specified.

```
Sbasic
    SAVEfile #bufferid, filename, filedevice, over
    filename : [string]
    filedevice : [string]
    over : short[0]

Assembler
    SAVE
    bufferid
    optional string filename
    optional string filedevice
    short overwrite status

C
    long DDsavefile(long bufferid, char *filename,
                    char *filedevice, short over);

errors, code,   meaning
itnf    -7      invalid bufferid
drfl    -11     drive full
fex     -8      file already exists
...             any other file i/o error
```

BACKup

This command can be used to make a backup of a file. There are two reasons for this: making a backup of a disk-based file, or to preserve the filename and filedevice (which SAVE overwrites when changed).

The other way to make a backup of your files is by closing the file and copying it manually, this is however not as easy as using this command. When making a backup of a disk-based file, it is impossible to put the backup on another disk in the same drive.

This routine always makes a backup of the entire file, and is actually just an implementation of this Sbasic program :

```
DEFine PROCedure backup(bid, name$, dev$, over)
    keep=bufferID : DEFbuffer=bid
    keepi=indexID : DEFindex=0
    keepv=VIEWstatus : SETVIEWstatus 1
    dd_err=0
    SAVEinit name$, dev$, over
    IF dd_err THEN RETURN
    FIRSTrec
    REPeat loop
        SAVErec : IF dd_err THEN EXIT loop
        NEXTrec : IF dd_err THEN EXIT loop
    END REPeat loop
    SAVEfinish
    SETVIEWSTATUS keepv : DEFbuffer keep : DEFindex keepi
END DEFine backup
```

This program changes the viewstatus to view only as this makes sure that all records are actually saved. This program does however not return any error, contrary to the routine in the engine. It does show how to make sure that all statuses are preserved when the routine ends.

A similar routine could be implemented if you want to save all the record in a certain index, or if you want to save only certain fields.

```
Sbasic
    BACKfile #bufferid, filename, filedevice, over
    filename : [string]
    filedevice : [string]
    over : short[0]

Assembler
    BACK
    bufferid
    optional string filename
    optional string filedevice
    short overwrite status

C
    long DDbackfile(long bufferid, char *filename,
                    char *filedevice, short over);
```

```
errors, code,   meaning
itnf    -7      invalid bufferid
drfl    -11     drive full
fex     -8      file already exists
fdiu    -9      Save Record Sequence channels still open
...             any other file i/o error
```

Save record sequence

Save record sequence is very important as is can be used to save only a part of a file, with preservation of recordids, and also as it is the only way in which you can make a backup of a disk-based file without removing all buffers for that file.

Note that there can only be one save record sequence for each buffer at any given moment. If you want to start another save record sequence on the same buffer, then the previous save record sequence has to be finished.

SAVe record sequence Initialize

Open the file for the save record sequence for the given buffer. If no filedevice passed, and opening just the filename doesn't work, then the medium-file is opened on the DATA_USE device.

You can also state the overwrite status (over). If this is zero and the medium-file already exists, an error will be reported. If over is set, then the medium-file will be overwritten.

The medium-file will get the `_ddf` extension. This extension never has to be specified.

```
Sbasic
    SAVEinit #bufferid, filename, filedevice, over
    filename : string
    filedevice : [string]
    over : short[0]

Assembler
    SAVI
    bufferid
    string filename
    optional string filedevice
    short overwrite status

C
    long DDsaveinit(long bufferid, char *filename,
                    char *filedevice, short over);

errors, code,   meaning
itnf    -7      invalid bufferid
drfl    -11     drive full
fex     -8      file already exists
fdiu    -9      in use, there is still a channel open, should call SAVEfinish
...             any other file i/o error
```

SAVe Record

The current record will be saved. You have to make sure that every record is only implemented once, or you will have problems when loading. A record has to have been implemented if you want to save it (it has to have a recordID).

This routine also takes the IRS into account. Also note that the channel should be closed with SAVEfinish if an error occurs.

```
Sbasic
    SAVErec #bufferid

Assembler
    SAVR
    bufferid

C
    long DDsaverec(long bufferid);

errors, code,   meaning
itnf    -7      invalid bufferid
drfl    -11     drive full
orng    -4      this is a new record, not accepted
...             any other file i/o error
```

SAVe record sequence Finish

Close the file for the save record sequence for the given buffer.

```
Sbasic
    SAVEfinish #bufferid

Assembler
    SAVF
    bufferid

C
    long DDsavefinish(long bufferid);

errors, code,   meaning
itnf    -7      invalid bufferid
```

Merging another file

On some occasions it may be necessary to merge in another file to the current one, and although there is no extension to do this, we hereby supply a routine which should do the trick. It is the routine which was actually incorporated in the DATAdesign main program.

This routine has the advantage of putting fields which already existed (same name) at the same place, and creating new fields for the others. So you should take care with the fieldnames (even more because they are case dependent.). On the other hand the routine we present here lacks some error trapping, and it also doesn't check whether the fields in the different files have actually got the same field type. This may cause some strange effects. Only the actual merging is handled here. The file 'merge' is merged in with the current file. Two more variables have to be set: 'buffer' which contains the address of some memory which can be used to copy the fields, and 'maxlen' which is the length of this buffer. This buffer can be allocated and released with the lines

```
buffer=ALCHP(maxlen)
RECHP buffer
```

So here is the actual routine:

```
dd_err=0
nrf=NRfields(#merge)
REMark array with fieldid in old file, fieldid in current file
DIM ref(nrf,2): ref(0,1)=0 : ref(0,2)=0
REMark add the fields
j=0
FOR i=1 TO nrf
    name$=CYCLEfields(#merge,j,t)
    ADDfield name$,t
    ref(i,1)=j : ref(i,2)=fieldID(name$)
END FOR i
REMark copy all the fields
FIRSTrec #merge
REPeat loop
    NEWrec
    FOR i=0 to nrf
        length=GETfield(#merge, ref(i,1), maxlen, buffer)
        SETfield ref(i,2), length, buffer
    END FOR i
    IMPLEMENT
    NEXTrec #merge
    IF dd_err THEN EXIT loop
END REPeat loop
```

Field manipulation

- ADDField
- FieLD Clear
- FieLD Delete
- Field Rename

ADDField

At any given time, you can add a field with a certain name and type. Note that all fieldnames have to be unique (but comparing is case dependent., so "Name" is not the same as "name").

The possible value for the field type are :

type	code	element size	usage
raw	1	1 byte	graphics, fonts, ...
char	2	1 byte	text
short	3	2 bytes	small integer values selections, statuses, ...
long	4	4 bytes	large integer values dates, ...
ieee	5	8 bytes	ieee double any numerical value

```
Sbasic
    ADDfield #bufferid, name, type
    name : string
    type : short[2]

Assembler
    ADDF
    bufferid
    string fieldname
    short fieldtype

C
    long DDaddfield(long bufferid, char *fieldname, short type);
    *fieldname doesn't have to be at an even address
```

errors,	code,	meaning
itnf	-7	invalid bufferid
orng	-4	fieldtype has to be in the 1-5 range
inam	-12	invalid fieldname (null name not accepted)
fex	-8	fieldname already exists
drfl	-11	too many fields already (max 256 fields)

FieLD Clear

This command can be used to clear a field in the current or in all records. A field is considered as cleared when it is not present in the record. An empty field is not considered as cleared, i.e. all non-existing fields are cleared, while a field is empty when it is in the record, but there is no data in it. If you try to read a value from an empty field, the engine gives an out of range error, but when you try to read from a cleared field, a “not found” error is returned. All fields in a record are cleared until somebody puts some data in it.

When you select all, and the requested field exists, then the record will be implemented first. It is impossible to know which record will be in the buffer after this command has finished. The given field will not be cleared in locked records (except the current record).

```
Sbasic
    fieldCLEAR #bufferid, field, all
    all : short (<>0 -> all, 0 -> current record only)

Assembler
    FLDC
    bufferid
    field
    short all status

C
    long DDfieldclear(long bufferid, short field, short all);

errors, code,   meaning
itnf    -7      invalid bufferid
...             any error from IMPLement
```

FieLD Delete

This command can be used to remove a field from a file. It can only be called if the given buffer is the only buffer which has access to the file. You can't delete the *memo* field.

The record will be implemented first. It is impossible to know which record will be in the buffer after this command has finished.

```
Sbasic
    fieldDELETE #bufferid, field

Assembler
    FLDD
    bufferid
    field

C
    long DDfielddelete(long bufferid, short field);

errors, code,   meaning
itnf    -7      invalid bufferid
fdiu    -9      this is not the only buffer using this file
...             any error from IMPLement
```

Field Rename

This command can be used to change the name of a field in the file. You can't rename the *memo* field.

```
Sbasic
    fieldRENAME #bufferid, field, newname

Assembler
    FLDR
    bufferid
    field
    newname : string

C
    long DDfieldrename(long bufferid, short field, char *newname);
```

errors,	code,	meaning
itnf	-7	invalid bufferid
ipar	-15	field doesn't exist
inam	-12	invalid name (null name nor accepted)
fex	-8	fieldname already exists

Field Information

General

CouNT Fields

Get the number of fields in the file.

```
Sbasic
    number = Nrfields(#bufferid)

Assembler
    CNTF
    bufferid
    return short number of fields
```

```
C
    long DDnrfields(long bufferid, short *result);

errors, code,   meaning
itnf     -7     invalid bufferid
```

CYCle Fields

This command allows you to find out all you want to know about all the fields in as few calls as possible. Given a seed fieldid, you can get fieldname, type and fieldid all in one go.

A fieldname is maximum 16 characters long and that amount of space has to be available in the return string (no problem in Sbasic).

This command can't give any information about fieldid zero, but that is no problem as that field *always* is the *memo* field.

```
Sbasic
    name$ = CYCLEfield$(#bufferid, seed, type)
    seed : short, updated with fieldid
    type : updated short

Assembler
    CYCF
    bufferid
    short seed, can be updated
    short type, can be updated
    return string fieldname

C
    long DDcyclefield(long bufferid, short *seed,
                      short *type, long *return);

errors, code,   meaning
itnf    -7      invalid bufferid
orng    -4      a fieldid always has to be in the 0-255 range
eof     -10     no fields left (seed updated to zero)
```

FieLD Type

Get the type of the given field.

```
Sbasic
    type = fieldTYPE(#indexid, field)

Assembler
    FLDT
    indexid
    field
    return short field type

C
    long DDfieldtype(long indexid, short field, short *result);

errors, code,   meaning
itnf    -7      invalid bufferid or indexid
ipar    -15     field doesn't exist
```

FieLDName

Get the name of a field with given id.

A fieldname is maximum 16 characters long and that amount of space has to be available in the return string (no problem in Sbasic).

```
Sbasic
    name$ = fieldNAME$(#bufferid, field)

Assembler
    FLDN
    bufferid
    field
    return string fieldname

C
    long DDfieldname(long bufferid, short field, char *result);

errors, code,   meaning
itnf    -7      invalid bufferid
ipar    -15     fieldid doesn't exist
```

ID of FIeld

Get the id of the field with the given name (case dependent).

```
Sbasic
    id = fieldID(#bufferid, name)
    name : string

Assembler
    IDFI
    bufferid
    string fieldname
    return short fieldid

C
    long DDfieldtype(long bufferid, short field, short *result);

errors, code,   meaning
itnf    -7      invalid bufferid
                or fieldname not found
inam    -12     null name invalid
```

LENgth of Field

Get the length of the given field, that is the number of elements in the field.

```
Sbasic
    len = fieldLEN(#bufferid, field)

Assembler
    LENF
    bufferid
    field
    return long length
```

```
C
    long DDfieldlen(long bufferid, short field, long *result);

errors, code,   meaning
itnf    -7      invalid bufferid
                or field not found
```

LENgth of String in field

Get the length of a line in a character field.

```
Sbasic
    length = lineLEN(#bufferid, field, line)
    line : short[1]

Assembler
    LENS
    bufferid
    field
    short line
    return long length
C
    long DDlinelen(long bufferid, short field, long *result);

errors, code,   meaning
itnf    -7      invalid bufferid
                or field not present (i.e. field is clear)
                or line not present
ipar    -15     field doesn't exist
                or isn't a character field
orng    -4      line has to be >=0, but has to exist
```

NumbeR of LiNes in field

Get the number of lines in a character field.

```
Sbasic
    number = NRlines(#bufferid, field)

Assembler
    NRLN
    bufferid
    field
    return long number of lines

C
    long DDnrlines(long bufferid, short field, long *result);

errors, code,   meaning
itnf    -7      invalid bufferid
                or field not present (i.e. field is clear)
ipar    -15     field doesn't exist
                or isn't a character field
```

Extracting data

This is probably the most important part of the DATAdesign engine. These are the commands you need to examine the contents of a record. Of course you have to be able to find out what information is contained in your file, and here is the way to do so.

Get FieLD

Get the contents of an entire field at once. This is also the only way you can get the contents of a 'raw' field.

If the field is not found (that is cleared or invalid), then zero is returned as length.

The pointer to the place where the contents of the field has to be filled in doesn't have to be even. You have to indicate the size of the buffer as it is used to prevent a buffer overflow. If the buffer is not long enough, then it will be filled and the length of the buffer will be returned.

```
Sbasic
    length = GETfield (#bufferid, field, length, place)
    length : short
    place : long, ptr to buffer

Assembler
    GFLD
    bufferid
    field
    long length of buffer
    pointer to buffer
    return long length of field

C
    long DDgetfield(long bufferid, short field, long length,
                    char *buffer, long *return);

errors, code,   meaning
itnf     -7     invalid bufferid
                field not found (cleared or invalid)
bffl     -5     buffer full
```

Get CHaRacter

Get a character from a character field.

```
Sbasic
    char$ = GETchar$(#bufferid, field, element)
    element : short[1]

Assembler
    GCHR
    bufferid
    field
    long element
    return char
```

```
C
    long DDgetchar(long bufferid, short field,
                   long element, char *return);

errors, code,   meaning
itnf    -7      invalid bufferid
                or field not present in this record
ipar    -15     invalid fieldid
                or field has wrong type
orng    -4      element has to be >=1, but may not exist
```

Get SHoRt

Get a short from a field of shorts.

```
Sbasic
    value = GETshort (#bufferid, field, element)
    element : short[1]

Assembler
    GSHR
    bufferid
    field
    long element
    return short

C
    long DDgetshort(long bufferid, short field,
                    long element, short *return);

errors, code,   meaning
itnf    -7      invalid bufferid
                or field not present in this record
ipar    -15     invalid fieldid
                or field has wrong type
orng    -4      element has to be >=1, but may not exist
```

Get LoNG

Get a long from a field of longs.

```
Sbasic
    value = GETlong (#bufferid, field, element)
    element : short[1]

Assembler
    GLNG
    bufferid
    field
    long element
    return long

C
    long DDgetlong(long bufferid, short field,
                   long element, long *return);
```

```
errors, code,   meaning
itnf    -7      invalid bufferid
                or field not present in this record
ipar    -15     invalid fieldid
                or field has wrong type
orng    -4      element has to be >=1, but may not exist
```

Get DouBLe

Get a double from a field of ieee doubles.

```
Sbasic
    value = GETfloat (#bufferid, field, element)
    element : short[1]
  or if you want to get the value as an ieee double :
    GETdouble #bufferid, field, element, place
    element : short[1]
    place : long pointer to 8 byte space to fill in double

Assembler
    GDBL
    bufferid
    field
    long element
    return ieee double

C
    long DDgetdouble(long bufferid, short field,
                     long element, double *return);

errors, code,   meaning
itnf    -7      invalid bufferid
                or field not present in this record
ipar    -15     invalid fieldid
                or field has wrong type
orng    -4      element has to be >=1, but may not exist
```

Get STRing

Get a line from a character field.

```
Sbasic
    line$= GETline$(#bufferid, field, line)
    line : short[1]

Assembler
    GSTR
    bufferid
    field
    long line
    return string

C
    long DDgetline(long bufferid, short field, long line,
                   char *return, short space);
    space : the space which is available for the string
```

```
errors, code,   meaning
itnf    -7      invalid bufferid
                or field not present in this record
ipar    -15     invalid fieldid
                or field has wrong type
orng    -4      line has to be >=1, but may not exist
bffl    -5      buffer full (line is longer than return string can contain,
                the return buffer will be filled)
```

Setting data

Set FieLD

Set the contents of an entire field. The type of the field is not considered. This is the only way you can set the contents of a 'raw' field.

```
Sbasic
    SETfield #bufferid, field, length, place
    length : short
    place : long, ptr to buffer

Assembler
    SFLD
    bufferid
    field
    long length of data in buffer (in bytes)
    pointer to buffer

C
    long DDsetfield(long bufferid, short field, long length, char *buffer);

errors, code,   meaning
itnf    -7      invalid bufferid
ipar    -15     invalid fieldid
imem    -3      insufficient memory
rdo     -20     read only buffer
```

Set CHaRacter

Set a character in a character field. If the field was not present, it will be created. If the element already existed, it will be overwritten. If there were fewer elements in the field, then new spaces will be created as filling.

```
Sbasic
    SETchar #bufferid, char$, field, element
    char$ : character
    element : short[1]

Assembler
    SCHR
    bufferid
    field
    char value
    long element

C
    long DDsetchar(long bufferid, short field, long line, char value);
```

```
errors, code,   meaning
itnf    -7      invalid bufferid
ipar    -15     invalid fieldid
                or field has wrong type
orng    -4      element has to be >=1
imem    -3      insufficient memory
rdo     -20     read only buffer
```

Set SHoRt

Set a short in a field of shorts. If the field was not present, it will be created. If the element already existed, it will be overwritten. If there were fewer elements in the field, then new zero elements will be created as filling.

```
Sbasic
    SETshort #bufferid, value, field, element
    value : short
    element : short[1]

Assembler
    SSHR
    bufferid
    field
    short value
    long element

C
    long DDsetshort(long bufferid, short field, long line, short value);

errors, code,   meaning
itnf    -7      invalid bufferid
ipar    -15     invalid fieldid
                or field has wrong type
orng    -4      element has to be >=1
imem    -3      insufficient memory
rdo     -20     read only buffer
```

Set LoNG

Set a long in a field of long. If the field was not present, it will be created. If the element already existed, it will be overwritten. If there were fewer elements in the field, then new zero elements will be created as filling.

```
Sbasic
    SETlong #bufferid, value, field, element
    value : long
    element : short[1]

Assembler
    SLNG
    bufferid
    field
    long value
    long element

C
    long DDsetlong(long bufferid, short field, long line, long value);
```

```
errors, code,   meaning
itnf    -7      invalid bufferid
ipar    -15     invalid fieldid
                or field has wrong type
orng    -4      element has to be >=1
imem    -3      insufficient memory
rdo     -20     read only buffer
```

Set DouBLe

Set a double in a field of doubles. If the field was not present, it will be created. If the element already existed, it will be overwritten. If there were fewer elements in the field, then new zero elements will be created as filling.

```
Sbasic
    SETfloat #bufferid, value, field, element
    value : float
    element : short[1]
    or if you want to set the value as an ieee double :
    SETdouble #bufferid, place, field, element
    place : long, pointer to 8 bytes where the double can be found
    element : short[1]

Assembler
    SDBL
    bufferid
    field
    double value
    long element

C
    long DDsetdouble(long bufferid, short field, long line, double value);

errors, code,   meaning
itnf    -7      invalid bufferid
ipar    -15     invalid fieldid
                or field has wrong type
orng    -4      element has to be >=1
imem    -3      insufficient memory
rdo     -20     read only buffer
```

Set STRing

Set a line in a character field. If the field field was not present it will be created. If the line already existed it is overwritten. If there were fewer lines in the field, then some empty lines will be created for filling.

```
Sbasic
    SETline #bufferid, line$, field, line
    line$ : string
    line : short[1]
```

```
Assembler
    SSTR
    bufferid
    field
    string line to fill in
    long line

C
    long DDsetline(long bufferid, short field, long line, char *string);

errors, code,   meaning
itnf    -7      invalid bufferid
ipar    -15     invalid fieldid
                or field has wrong type
orng    -4      line has to be >=1
imem    -3      insufficient memory
rdo     -20     read only buffer
```

Changing

As you may have noticed, you can think of a field as an array of a certain type. This array can grow or shrink without limitation (except memory of course). However, it may be necessary to remove a part of this array, or insert a part, and this either at the end, or somewhere in the middle of this array.

INSert Elements

This command allows you to insert some array elements in a field. Inserted elements will be cleared to zero. The elements will be inserted *before* the element given as place.

```
Sbasic
    INSERTel #bufferid, field, place, number
    place : short[1]
    number : short[1]

Assembler
    INSE
    bufferid
    field
    short place
    short number

C
    long DDinsertel(long bufferid, short field, short place, short number);

errors, code,   meaning
itnf    -7      invalid bufferid
ipar    -15     field doesn't exist
                place has to be >=0
                number has to be >=0
imem    -3      insufficient memory
rdo     -20     read only buffer
```

ReMoVe Elements

This command allows you to remove some array elements from a field.

The first elements to be removed is given in place.

```
Sbasic
    REMOVEel #bufferid, field, place, number
    place : short[1]
    number : short[1]

Assembler
    RMVE
    bufferid
    field
    short place
    short number

C
    long DDremoveel(long bufferid, short field, short place, short number);

errors, code,   meaning
itnf    -7      invalid bufferid
ipar    -15     field doesn't exist
orng    -4      place has to be >=0
                number has to be >=0
imem    -3      insufficient memory
rdo     -20     read only buffer
```

Record Manipulation

Now that we know how we can change the data in a record, we also have to be able to copy them back into the file (implement them), to create new records,...

- DELEte record
- DUPLicate record
- IMPLement record
- NEW Record

DELEte record

Delete the current record from the file. The record will remain in the buffer, but the recordid will have changed if you re-implement it! When followed by a NEWrec or GETrec or similar command, the record can't be retrieved.

```
Sbasic
    DELrec #bufferid

Assembler
    DELE
    bufferid

C
    long DDdelrec(long bufferid);

errors, code,   meaning
itnf    -1      invalid bufferid
rdo     -20     read only buffer
...             any file i/o error (only when disk-based)
```

DUPLicate record

This command can be used if you need a new record which is similar to the current record. The record in the buffer will be treated as a new one. However the buffer will stay the same. But when you implement, the old record will be the same, and you will have a new record with another recordid and any changes you made to the record.

```
Sbasic
    DUPLICATE #bufferid

Assembler
    DUPL
    bufferid
C
    long DDduplicate(long bufferid);

errors, code,   meaning
itnf    -7      invalid bufferid
```

IMPLement record

Implement the given buffer in the file. Necessary if you want the changes you made in a the buffer to appear in the file. The contents of the buffer isn't changed, but the newly created record will get a place in the file and a recordid.

If the file is disk-based, then the record will immediately be written to disk to ensure complete safety in case of a power failure or a system crash.

If the file is disk-based and a "drive full" error may occur, then the record is *not* (re-)implemented, and an error is reported.

Note that other file-errors may cause problems.

```
Sbasic
    IMPLEMENT #bufferid

Assembler
    IMPL
    bufferid

C
    long DDimplement(long bufferid);

errors, code,   meaning
itnf    -7      invalid bufferid
imem    -3      insufficient memory
drfl    -11     there was risk of a "drive full"
                too many records (impossible, indicates problems)
nimp    -19     not implemented, only occurs when using the demo version
...             any file i/o error (should not occur)
```

NEW Record

Make sure the record in the buffer is a new one. This means that the current record will be empty (all fields cleared), and without recordid just yet.

```
Sbasic
    NEWrec #bufferid

Assembler
    NEWR
    bufferid

C
    long DDnewrec(long bufferid);

errors, code,   meaning
itnf    -7      invalid bufferid
```

Record Information

- DATE of last change to record
- ID of current REcord
- LENgth of Record

DATE of last change to record

Get the date and time when the current record was last implemented.

This routine returns zero if the current record hasn't been implemented just yet. The returned value is a long word which is derived from the internal clock of the QL. So the date of the current record in the default buffer can be printed like this

```
    PRINT DATE$(recordDATE)
```

```
Sbasic
    date = recordDATE(#bufferid)

Assembler
    DATE
    bufferid
    return long date

C
    long DDrecorddate(long bufferid, long *result);

errors, code,   meaning
itnf    -7      invalid bufferid
...             any file i/o error
```

ID of current Record

Get the recordid of the current record.

This routine returns -1 if there is no recordid for the current record just yet (-1 is an impossible recordid).

```
Sbasic
    id = recordID(#bufferid)

Assembler
    IDRE
    bufferid
    return long id

C
    long DDrecordid(long bufferid, long *result);
```

```
errors, code,   meaning
itnf    -7      invalid bufferid
```

LENgth of Record

Get the length of the current record. This is the length in bytes, it is the added length in bytes of the field which are not cleared and their fieldheaders. Such a fieldheader contains the fieldid and the length of the field (6 bytes).

The record length should only be used for comparing, this is not a very useful routine.

```
Sbasic
    len = recordLEN(#bufferid)

Assembler
    LENR
    bufferid
    return long length
C
    long DDrecordlen(long bufferid, long *result);

errors, code,   meaning
itnf    -7      invalid bufferid
```

Marking of Records

All records have a special *mark status*. This status is the same for all buffers which have access to that record, and can have a value between zero and 255. If a mark status is zero, we say it is cleared.

- Get MaRK status
- Set MaRK status
- Clear all MaRK statuses

Get MaRK status

```
Sbasic
    value = GETmark(#bufferid)

Assembler
    GMRK
    bufferid
    return byte value

C
    long DDgetmark(long bufferid, char *result);

errors, code,   meaning
itnf    -7      invalid bufferid
```

Set MaRK status

The mark status will only be changed in the file after implementing !!

The mark status is an unsigned byte, so a value between 0-255, and is called clear when it is zero.

```
Sbasic
    SETmark(#bufferid, value)
    value : short

Assembler
    SMRK
    bufferid
    byte value

C
    long DDsetmark(long bufferid, char value);

errors, code,   meaning
itnf    -7      invalid bufferid
```

Clear all MaRK statuses

You can also clear the mark status for all records in the file. Note that locked records may not be affected when they are re-implemented.

```
Sbasic
    CLEARmark(#bufferid)

Assembler
    CMRK
    bufferid
C
    long DDclearmark(long bufferid);

errors, code,   meaning
itnf    -7      invalid bufferid
isyn    -21     internal file structure corrupted (should not occur)
...             any file i/o error (should not occur)
```

Searching

This section describes how you can search your file in a slow, linear, but flexible way. The following commands can search in one field only if you give a fieldid, or in all fields with a suitable type if you don't specify a field. The value which is searched can be anywhere in the field(s), and the place where the value was found can be returned on request.

The `FIRSTrec`, `LASTrec`, `NEXTrec`, `PREVrec` commands are used internally for navigation, so the default indexid is important! However, these commands only accept a bufferid as parameter. So if you want to use a given indexid you should use some code like:

```
    keep = indexID(#bid)
    DEFindex #bid,iid
    FINDxxx #bid,...
    DEFindeb #bid,keep
```

for any given indexid id, and bufferid bid. Of course you don't have to specify the bufferid in all commands.

When a matching record is found, it will be in the buffer. If no such record is found, then the last record will be in the buffer (or the first when searching backwards).

These commands also need a "compare" value which determines how they search and where they start to search. Here are the possible values. You just have to add the values together to get combinations, or combine the characters in a string (Sbasic only).

name	bit	val	char	meaning
agai	0	1	aAmM	start searching from next/previous record instead of first/last
case	4	16	cC	compare case dependent (find string only)
rvrs	5	32	-rR	reverse order (search backwards)

- FiND String
- FiND Integer
- FiND Double
- Replace

FiND String

Find a string.

```
Sbasic
    FINDstring #bufferid, field, compare, value$, rfield, rplace
    value$ : string
    rfield : [short], updated if passed
    rplace : [short], updated if passed
```

```
Assembler
    FNDS
    indexid
    field
    short compare
    string value to find
    update short fieldid where found
    update short place in field where found

C
    long DDfindstring(long indexid, short field, short cmp, char *value,
                      short *rfield, short *rplace);

errors, code,   meaning
itnf    -7      invalid bufferid or indexid
                or not found
ipar    -15     invalid fieldid (wrong type or doesn't exist)
imem    -3      insufficient memory for FRST/LAST/NEXT/PREV
```

FiND Integer

Find an integer. If the integer fits in a short, then both short and long fields can be searched. If not then only fields of longs can be searched.

```
Sbasic
    FINDinteger #bufferid, field, compare, value, rfield, rplace
    value : short
    rfield : [short], updated if passed
    rplace : [short], updated if passed

Assembler
    FNDI
    indexid
    field
    short compare
    long value to find
    long 0 (unused)
    update short fieldid where found
    update short place in field where found

C
    long DDfindinteger(long indexid, short field, short cmp, long value,
                       short *rfield, short *rplace);

errors, code,   meaning
itnf    -7      invalid bufferid or indexid
                or not found
ipar    -15     invalid fieldid (wrong type or doesn't exist)
imem    -3      insufficient memory for FRST/LAST/NEXT/PREV
```

FiND Double

Find a double.

This is a fragile command as the doubles have to match exactly. This can however not be guaranteed because of rounding errors, and because some programs use real ieee doubles (most C-programs written for DATAdesign, unless they were read from ASCII in which case they usually convert ieee floats to doubles), and some programs convert QL floats to doubles, thus loosing accuracy (mainly Sbasic programs, and also the DATAdesign main program).

Actually in Sbasic it is even worse as you can only pass the value to find as a float which is converted to a double. This should not be a problem as you will probably do everything with converted floats.

```
Sbasic
    FINDfloat #bufferid, field, compare, value, rfield, rplace
    value : float
    rfield : [short], updated if passed
    rplace : [short], updated if passed

Assembler
    FNDD
    indexid
    field
    short compare
    double value to find
    update short fieldid where found
    update short place in field where found

C
    long DDfinddouble(long indexid, short field, short cmp, double value,
                      short *rfield, short *rplace);

errors, code,   meaning
itnf    -7      invalid bufferid or indexid
                or not found
ipar    -15     invalid fieldid (wrong type or doesn't exist)
imem    -3      insufficient memory for FRST/LAST/NEXT/PREV
```

Replace

Although there is no specific command to replace some value by another in the DATAdesign engine, you can do it by combining some commands. This can be done because the find commands can return where they found the item.

So if you want to replace the value 10 by 100 in all occurrences in the default buffer with the default index, in all integer fields, it can be done like this:

```
dd_err= 0
FINDinteger ,,10,f, p
REPeat loop
    IF dd_err THEN EXIT loop
    IF fieldTYPE(f)=3 THEN
        SETshort 100,f, p
    ELSE
        SETlong 100,f, p
    END IF
    FINDinteger, 1,10,f,p : REMark or FINDinteger ,'a',10,f,p
END REPeat loop
```

When replacing strings you may also need to use the `REMOVEel` or `INSERTel` commands when the length of the two strings is different. Note that the string will probably have to be set character by character.

Defaults

The DATAdesign engine allows you to use defaults for bufferids, indexids and fieldids. There is however no guarantee that any default is actually valid.

- Default buffer
- Default field
- Default index

Default buffer

You can always set or get the bufferid of the default buffer. Reading is done like this (but you don't know if that buffer still exists) :

```
Sbasic
    id= bufferID

Assembler
    IDBF
    return long bufferid

C
    long DDbufferid(long *result);

errors, code,   meaning
none
```

Or you can set it like this :

```
Sbasic
    DEFbuffer id
    id : long

Assembler
    DEFB
    long bufferid

C
    long DDdefbuffer(long bufferid);

errors, code,   meaning
itnf    -7      invalid bufferid
ijob    -2      this buffer is owned by another job
```

Default field

Every buffer can have a default field. You have to keep track of this, as you can't find out what the default field is at a given moment.

```
Sbasic
    DEFfield #bufferid, field

Assembler
    DEFF
    bufferid
    short field

C
    long DDdeffield(long bufferid, short field);

errors, code,   meaning
itnf    -7      invalid bufferid
```

Default index

Every buffer also has a default index for it. Reading is done like this :

```
Sbasic
    id= indexID (#bufferid)

Assembler
    IDIN
    bufferid
    return long indexid

C
    long DDindexid(long bufferid, long *result);

errors, code,   meaning
itnf    -7      invalid bufferid
```

Or you can set it like this :

```
Sbasic
    DEFindex #bufferid, id
    id : long

Assembler
    DEFI
    bufferid
    long indexid

C
    long DDdefindex(long bufferid, long indexid);

errors, code,   meaning
itnf    -7      invalid bufferid or indexid
```

Index Manipulation

Here are the commands to create indexes and/or find out how they are sorted or filtered.

- SORT
- FILTer
- Index UPDate
- Get Index Sort Level
- Set Index Sort Level
- Get Index Filter Level
- Set Index Filter Level

SORT

Routine to determine how the records in an index have to be sorted.

Indexes can be sorted with up to ten levels. If you don't need that many levels a compare value of -1 indicates that only previous levels have to be taken into consideration (Assembler only).

If an indexid is passed, than the sort data will be overwritten. If a bufferid is passed or default used, then a new index will be created. This new index will automatically become the default index. The index itself has to be built with a call to `indexUPDATE`.

Default fields are not accepted by this command.

Possible values for compare:

```
name    bit val char    meaning
mark    0   1   mMaA    mark records if equal (get value 255)
line    2   4   lL      place is a line number
text    3   8   tT      sorted as text (character field only)
case    4   16  cC      compare case dependent (character field only)
rvrs    5   32  -rR     reverse order (sort backwards)
```

If 'text' is set in compare, then eight characters are used with place as offset in the character array (considered as groups of eight characters). The characters will be sorted properly (not by their character code). If 'line' is also set, then the place will be the start of the given line.

When a character field is sorted and 'text' is not indicated, then the text is considered as raw data and sorted by the character code (no case conversion).

If an element is not provided, it will be put at the end, except when 'text' is indicated, then it will be put at the start (so that "Jo" will come before "Jona").

Equal records can be marked with mark value 255 if 'mark' is set in compare for the first level.

```
Sbasic
    SORTfile #indexid, compare, field, place,...
    field : short
    place : short[1]
    the last three parameters can be repeated up to ten times
    fields can't be passed by name

Assembler
    SORT
    indexid
    ten times : short compare
                short field
                short place
    return long indexid

C
    long DDsortfile(long indexid, short number, short compare,
                    short field, short place,...);
    the last three parameters have to be repeated 'number' times
```

errors,	code,	meaning
itnf	-7	invalid bufferid or indexid
imem	-3	insufficient memory

FILTer

Routine to determine which records should be in the index and which not.

Indexes can be filtered with up to ten levels. If you don't need that many levels a compare value of -1 indicates that only previous levels have to be taken into consideration (Assembler only). The levels are evaluated from left to right.

If an indexid is passed, then the filter data will be overwritten. If a bufferid is passed or default used, then a new index will be created. This new index will automatically become the default index. The index itself has to be built with a call to `indexUPDATE`.

Default fields are not accepted by this command.

Possible values for compare:

name	bit	val	char	meaning
mark	0	1	mMaA	mark records if equal (get value 255)
line	2	4	lL	place is a line number
text	3	8	tT	sorted as text (character field only)
case	4	16	cC	compare case dependent (character field only)
rvrs	5	32	-rR	reverse order (sort backwards)
equ	6	64	=	value in field has to be equal (not char/raw)
smal	7	128	<	value in field has to be smaller (not char/raw)
larg	8	256	>	value in field has to be larger (not char/raw)
str	9	512	sS	string has to be in field

These compare codes allow you to test whether

- a field is or is not cleared (present).
- a certain char (any of eight per level) is or is not present in the field (place=0). Zero bytes are not evaluated.
- or is present at a certain place in the field (place>0).
- whether the mark status <, >, =, <= , >=,<> the given value (byte, left justified in special) (unsigned compare).
- whether a value in a numerical field <, >, =, <=, >=, <> the given value (if place=0 or element doesn't exist -> levelstatus=false).
- whether a string (max eight characters) is or is not present in the field (place=0). Zero bytes are not evaluated. The string can be compared case dependent.
- or is present at a certain place in the field (place>0).

```
Sbasic
    FILTER #indexid, compare, field, place, value,...
    place : short[1]
    field : short
    value : 'special', used to compare
    the last four parameters can be repeated up to ten times
    fields can't be passed by name

Assembler
    FILT
    indexid
    ten times: short compare
               short field
               short place
               'special' value to compare
    return long indexid

C
    long DDfilter(long indexid, short number, short compare,
                  short field, short place, char *value[8],...);
    the last four parameters have to be repeated `number' times

errors, code,   meaning
itnf     -7     invalid bufferid
imem     -3     insufficient memory
```

Index UPDate

This command is used to actually build the index. It can also be used to re-build it if there already is one.

Records which are locked when the index is being updated are not included in the index.

Please note that it can not be predicted which record will be the current after execution of this command. This also means that it is possible that the current record is actually not visible (meaning not in the index).

```
Sbasic
    indexUPDATE #indexid
```

```
Assembler
    IUPD
    indexid

C
    long DDindexupdate(long indexid);

errors, code,   meaning
itnf    -7      invalid indexid (or bufferid)
imem    -3      insufficient memory
ipar    -15     something in the sort or filter data is impossible
...             any error returned by GETrec (except fdiu)
```

Get Index Sort Level

Get the sort data for one specific level.

```
Sbasic
    GETsort #indexid, level, cmp, fld, place
    level : short
    cmp : short, updated compare
    fld : short, updated field
    place : short, updated

Assembler
    GISL
    indexid
    short level
    update short compare (always updated)
    update short field (always updated)
    update short place (always updated)

C
    long DDgetsort(long indexid, short level, short *cmp,
                   short *fld, short *place);

errors, code,   meaning
itnf    -7      invalid indexid (or bufferid)
orng    -4      level not in 1-10 range
```

Set Index Sort Level

Set the sort data for one specific level. Should be followed by an `indexUPDATE`.

```
Sbasic
    SETsort #indexid, level, compare, fld, place
    level : short
    fld : short
    place : short

Assembler
    SISL
    indexid
    short level
    ptr to short compare
    ptr to short field
    ptr to short place
```

```
C
    long DDsetsort(long indexid, short level, short *cmp,
                   short *fld, short *place);

errors, code,   meaning
itnf    -7      invalid indexid (or bufferid)
orng    -4      level not in 1-10 range
```

Get Index Filter Level

Get the filter data for one specific level.

```
Sbasic
    GETfilter #indexid, level, cmp, fld, place, value
    level : short
    cmp : short, updated compare
    fld : short, updated field
    place : short, updated
    value : pointer to 'special', contents updated

Assembler
    GIFL
    indexid
    short level
    update short compare (always updated)
    update short field (always updated)
    update short place (always updated)
    update 'special' value (always updated)

C
    long DDgetfilter(long indexid, short level, short *cmp,
                     short *fld, short *place, char *value[8]);

errors, code, meaning
itnf    -7    invalid indexid (or bufferid)
orng    -4    level not in 1-10 range
```

Set Index Filter Level

Set the sort filter for one specific level. Should be followed by an `indexUPDATE`.

```
Sbasic
    SETfilter #indexid, level, compare, fld, place, value
    level : short
    fld : short
    place : short
    value : 'pointer' to special

Assembler
    SIFL
    indexid
    short level
    ptr to short compare
    ptr to short field
    ptr to short place
    ptr to 'special'
```

```
C
    long DDsetfilter(long indexid, short level, short *cmp, short *fld,
                     short *place, char *value[8]);

errors, code,   meaning
itnf    -7      invalid indexid (or bufferid)
orng    -4      level not in 1-10 range
```

Index Maintenance

This section contains some commands which are quite crucial to the proper operation of all commands related to indexes. There is the command to remove an index, but also some commands to assure index integrity.

When a record which is in an index is changed in a crucial place (any of the places which are relevant for the sorting or filtering of the record in the index), then the record can't be retrieved in the index any more. However the reference in the index will still exist. There are two methods to overcome this problem. You either have to update the index occasionally, or you have to preserve index integrity with the `indexDELETE` and `indexIMPLEMENT` commands. It is always interesting to rebuild your index from time to time. Records which are changed (or created) by other buffers can't preserve the integrity of your index.

- Index DELete
- Index IMPlement
- Index ReMoVe

Index DELete

If you want to remove a record from the file, or you may change a record, then it is best to delete it from the index.

This command will only work if the record you want to remove is still identical to the one that was put in the index as records in indexes can only be retrieved by their contents (and not by recordid).

So if you want to delete a record from the current buffer and the current index you should use a line like :

```
indexDELETE : DELrec

Sbasic
    indexDELETE #indexid

Assembler
    IDEL
    indexid

C
    long DDindexdelete(long indexid);

errors, code,   meaning
itnf    -7      invalid indexid (or bufferid)
imem    -3      insufficient memory
```

Index IMPlement

If you have a new record, or a record that you have (hopefully) deleted from the index before you changed it (or wanted to), then you can put it (back) in the index with this command.

This command will only work if the record has already been implemented in the file. So a new record is created and implemented in the current file and index like this :

```
NEWrec          : REMark take a new (empty) record
...             : REMark fill in the record
IMPLEMENT       : REMark implement in the file
indexIMPLEMENT  : REMark implement in the index
```

```
Sbasic
    indexIMPLEMENT #indexid

Assembler
    IIMP
    indexid

C
    long DDindeximplement(long indexid);

errors, code,   meaning
itnf    -7      invalid indexid (or bufferid)
imem    -3      insufficient memory
```

Index ReMoVe

This command is used to entirely remove an index from memory.

```
Sbasic
    indexREMOVE #indexid

Assembler
    IRMV
    indexid

C
    long DDindexremove(long indexid);

errors, code,   meaning
itnf    -7      invalid indexid (or bufferid)
```

Index Information

- CouNT Records

CouNT Records

This is the commands which tells you how many entries there are in the given index. If there is no index, it will give the number of records in the main file.

Please note that you have to maintain your index properly if you want to get a reliable result from this command. If you don't, there may even be more records in your index as there are records in the file.

Another important remark is that this command should not be used to know how many records there are if you want to access all of them. In that case you should use some code like :

```
    dd_err=0
    FIRSTrec
    REPeat loop
        IF dd_err THEN EXIT loop
        REMark do whatever you want with the record
        NEXTrec
    END REPeat loop
```

```
Sbasic
    number = COUNTrec(#indexid)

Assembler
    CNTR
    indexid
    return long number

C
    long DDcountrec(long indexid, long *return);

errors, code,   meaning
itnf    -7      invalid indexid (or bufferid)
```

Index loading/saving

- Index LOAd
- Index SAVe

Index LOAd

Naturally, it may be necessary to load indexes from disk. This needs a medium-file-name and a device.

The file which is loaded needs the `_ddi` extension. This doesn't have to be specified.

The newly created index will become the default one.

```
Sbasic
    indexLOAD #bufferid, filename, device
    filename : string
    device : [string]

Assembler
    ILOA
    bufferid
    string filename
    optional string device
    return indexid

C
    long DDindexload(long bufferid, char *filename, char *device);

errors, code,   meaning
itnf     -7     invalid bufferid
imem     -3     insufficient memory
fdnf     -7     file or device not found
isyn     -21    this is not a DATAdesign index file
...             any other file i/o error
```

Index SAVe

This command makes sure that there is an up to date version of your index on disk. If no device is passed, and saving to just the medium-file-name doesn't work, then the file is saved to the *data_use* device.

You can also state the overwrite status (over). If this is zero and the medium-file already exists, an error will be reported. If over is set, then the medium-file will be overwritten.

The medium-file will get the `_ddi` extension. This extension never has to be specified.

```
Sbasic
    indexSAVE #bufferid, filename, device, over
    filename : [string]
    device : [string]
    over : short[0]

Assembler
    ISAV
    bufferid
    string filename
    optional string device
    short overwrite status

C
    long DDindexsave(long bufferid, char *filename,
                   char *device, short over);

errors, code,   meaning
itnf    -7      invalid bufferid
drfl    -11     drive full
fex     -8      file already exists
...             any other file i/o error
```

Index Searching

- Index FiND
- Index FinD String

Index FiND

The main reason for sorting files with indexes is that it allows for very fast searching on the field of the first sort level.

This routine needs a 'special' parameter, which has to be of the type defined by the first sort level. It returns the indexid of the first record which equals the given item which has to be found. Subsequent records with the same value can be found with the `NEXTrec` command.

```
Sbasic
    record = indexFIND(#bufferid, item)
    item : 'special'

Assembler
    IFND
    bufferid
    'special' item to find
    return long recordid

C
    long DDindexfind(long bufferid, char *special[8], long *return);

errors, code,   meaning
itnf    -7      invalid bufferid
              or not found (return=-1)
```

Index FinD String

The main reason for sorting files with indexes is that it allows for very fast searching on the field of the first sort level. As mentioned above, DATAdesign has a very fast way of finding records through the INDEXfind command. This allows you to find the first record where the indexed field is **equal** to the search criterion.

However, this could conceivably be a problem when one wants to search for a string which is not the total of the 8 bytes used in the index. Suppose that one wants to search for all records where the indexed field starts with "DATA" : one couldn't use INDEXfind to do this, as it would not find the records where the indexed field contains , e.g., 'DATAdesign' - it would only find those where the field is exactly 'DATA' because the search criterion has to match the index criterion exactly (so the search criterion for INDEXFind needs 8 bytes).

Here, INDEXFinDString may help, provided that the string to search for is a part of the string in the file, and provided that it **IS LOCATED AT THE BEGINNING OF THE STRING TO BE FOUND:** so, using INDEXFinDString, one would find "DATAdesign" when searching for "Data", but NOT when searching for "ata".

PLEASE NOTE:

INDEXFinDString will only work correctly on indexes built on TEXT (char) fields. If you use it for other fields, such as shorts or doubles etc, it won't work – an error is returned

This routine needs a string parameter. It returns the indexid of the first record which equals the given item which has to be found. Subsequent records with the same value can be found with the `NEXTrec` command.

```
Sbasic
    record = INDEXFINDString(#bufferid, string$)
    string$ : string to find

Assembler
    IFDS
    bufferid
    string string to find (0 terminated)
    return long recordid

C
    long DDindexfind(long bufferid, char *string[8], long *return);

errors, code,   meaning
itnf    -7      invalid buffer/indexid
orng    -4      field was not a char/text field
ipar    -15     there was no string given
eof     -10     no matching record was found
```

Data input

For the Sbasic programmers we have added two routines in the Sbasic interface to make the input and editing of strings somewhat easier. In fact these routines are similar to some Window MANager routines, which are however not available through the QPTR interface. These WMAN routines were already available in C and Assembler, and so only the Sbasic interface is explained here. We have however implemented slightly different routines, which can be exited by some more keypresses, and there is also a small difference in the parts of the window that are cleared.

- Read string
- Edit string

Read string

The given string will be edited in the given window. The entire window width can be used. The string is printed at the current cursor position, and can only use the space between the current cursor position and the right hand side of the window. Only the part to the right of the cursor position is cleared. READstring returns on reading ENTER, ESC, UP arrow, DOWN arrow, tab, shift tab, any function key.

The cursor is positioned at the start of the given string. If the first typed character is printable, then the old string is thrown away. However if the first typed character was SPACE then READstring will treat this as ENTER.

```
Sbasic
    READstring #ch, string
    ch : Sbasic channel id [1]
    string : updated string

errors, code,   meaning
>0              termination character
...             any i/o system error
```

Edit string

The given string will be edited in the given window. The entire window width can be used. The string is printed at the current cursor position, and can only use the space between the current cursor position and the right hand side of the window. Only the part to the right of the cursor position is cleared. READstring returns on reading ENTER, ESC, UP arrow, DOWN arrow, tab, shift tab, any function key.

The cursor is positioned at the end of the given string so that the string can be edited.

```
Sbasic
    EDITstring #ch, string
    ch : Sbasic channel id [1]
    string : updated string
```

errors, code,	meaning
>0	termination character
...	any i/o system error

Appendices

- Compilation
- Versions
- Overview of extensions
- Problems

Compilation

It may be interesting for speed reasons to compile Sbasic programs which use the DATAdesign engine. But because the intelligence of the Sbasic interface depends on internal structures of Sbasic, it is not possible to use Turbo. The programs can however easily be compiled by QLiberator.

When compiling a program using QLiberator, you should think of one thing. QLiberator is an intelligent, optimizing compiler, throwing out all code which is useless. QLiberator does however assume that routines have no side-effects. The problem is that setting the *dd_err* variable is actually a side effect, so all tests for *dd_err* will be thrown out of the program. For this reason you should introduce some lines, preferably lines which are never executed. Something like this will do:

```
IF string$="xyw©" THEN print dd_err
```

where the variable preferably will never get that value, so the dd_err is never actually printed, and where the variable should not have a value which can be predicted (or at least not by QLiberator).

If you want to compile your SuperBASIC programs which use DATAdesign with Qliberator, you MUST use the names option. If you don't do this, DATAdesign will not be able to find the dd_err variable !

Versions

[v3.00]

first incarnation of v3, not compatible with v2.

[v3.01]

NEWR adjusted to work on read-only files. NEXT/PREV/LAST/FRST has problems with indexes without sort. FILT could not AND levels together properly. FILT compare as string added.

FORW/BCKW added. (02/03/1993)

[v3.02]

UNUS had a bug, the current record remained locked. This also solves this problem in GARB.

[v3.03]

There was a very bad incompatibility with Minerva rom. SAVF forgot to allow access for new SAVI. This made it impossible to BACKup more than once.

[v3.04]

FILT on numerical fields didn't work. SORT/FILT/IFND now compare doubles correctly. Problem when converting QL floats to doubles solved. NUS also closed the SAVI/SAVF channel when there was none. This caused bad problems on Minerva rom. The engine now fully works on Minerva. (18/04/1993)

[v3.05]

FILT problem when comparing both positive and negative numbers solved. IUPD did not fill the fieldids in the index data. FLDT now accepts an indexid. This caused problems when passing an indexid to the FILTER command in Sbasic. (29/04/1993)

[v3.06]

basic i/f FINDinteger now also accepts integer as search value. IUPD, ILOA forgot to test for out of memory sometimes. IMPL on disk-based file should now always work, there was a problem when checking whether there is enough free space on the disk. IMPL should no longer have problems when writing to read-only disks. IFND now returns the closest match which is smaller when not found. IUPD small change in index link-in routine (some no-op's added, because there sometimes occurred an illegal instruction, search me). (08/07/1993)

[v3.07]

General improvement, DATAdesign no longer enters supervisor mode. Mutual exclusion now assured with "DATAdesign mutex" thing.

[v3.08]

Mistake in DATAdesign mutex thing solved. There was no pointer to the actual thing. This made Qpac2 crash on TT.

[v3.09]

SORT could sometimes crash on TT (and cause problems on the rest). There was a mistake in the balance after a double rotation (p1). FIND String/Integer/Double reverse order caused infinite loop. DEFI was faulty, only allowed resetting. FILT string, position<>0 didn't work. (10/01/1994)

[v3.10]

FRST/LAST/NEXT/PREV/FORW/BCKW failed to return error on faulty buffer and indexid. (31.03.1994)

[v3.11]

DATAdesign files job now always owned by system. nextrec, prevrec now accept the parameter !! Bug fixed which caused max 2 filter levels accepted by Sbasic interface. (21.05.1994)

Handling of string parameters in Sbasic interface modified, only strings up to 256 chars long are accepted. The engine can handle longer strings, but not the interface. (Has to do with the fact that Sbasic can relocate itself.) GREC failed to increase the size of the buffer when trying to access a larger record from a disk based file. (23.05.1994)

[v3.12]

FILTER Sbasic interface handled strings wrong, this could cause unexpected results. All routines which return a short (e.g. GETshort) handled short as unsigned. This has changed to handling them as signed. (29.11.1994)

[v3.13]

IDEL should now also work properly when an indexid was passed as parameter. The Sbasic interface migrated to the `PWbasic_rext` file. The `engine_rext` file no longer links in any Sbasic commands to allow loading by any job in the system. The engine is now automatically linked when loaded.

[v3.14]

IDEL could cause problems. The deleted record could in some cases still be included in the index when saved – fixed. (13.09.1996)

Overview of extensions

ADDF
ADD Field

BACK
BACKup of file

BCKW
BaCKWard some records (= fast x times PREV)

CMRK
Clear all MaRK statusses

CNTA
CouNT All records

CNTF
CouNT all Fields

CNTR
CouNT visible Records

CYCB
CYCle through Buffers for this job

CYCF
CYCle through Fields

CYCL
CYCLe through files being edited/viewed

DATE
get DATE when record was last implemented

DEFB
DEFault Buffer setting

DEFF
DEFault Field setting

DEFI
DEFault Index setting

DELE
DELEte current record from file

DUPL
DUPLicate record

FDEV
FileDEVice we are working on

FILT
set FILTer parameters

FLDC
FieLD Clear (buffer or all)

FLDD
FieLD Delete

FLDN
FieLDName when id given

FLDR
FieLD Rename

FLDT
FieLDType when id given

FNAM
FileNAMe we are working on
FNDD
FiND Double
FNDI
FiND Integer
FNDS
FiND String
FORW
FORWard some records (fast x times NEXT)
FRST
go to FiRST editable record
GARB
GARBage collection
GCHR
Get CHaR in a character array (string)
GDBL
Get DouBLe (fp8) array element
GFLD
Get FieLD contents
GFST
Get FileSTatus (memory or disk-based)
GIFL
Get Index Filter Level parameters
GIRS
Get Inter Record Space
GISL
Get Index Sort Level parameters
GLNG
Get LoNG integer array element
GMRK
Get MaRK status
GREC
Get RECord with given id and put it in buffer
GSHR
Get SHoRt integer array element
GSTR
Get STRing as a line
GVST
Get View STatus
IDBF
ID of default BuFfer
IDEL
Index DELete
IDFI
ID of FIeld with given name
IDIN
ID of default INdex
IDRE
ID of REcord we are editing

IFND
Index FiND
IIMP
Index IMPlement
ILOA
Index LOAd
IMPL
IMPLement record in file
INSE
INSert array Elements
IRMV
Index ReMoVe
ISAV
Index SAVe
IUPD
Index UPDate
LAST
go to LAST editable record
LENF
get LENgth of a field
LENR
get LENgth of the current record
LENS
get LENgth of a String as a line
NEWF
NEW File creation
NEWR
NEW Record start
NEXT
go to NEXT editable record
NRLN
get NumbeR of LiNes in a text field
PREV
go to PREVious editable record
RMVE
ReMoVe array Elements
SAVE
SAVE file to medium
SAVI
SAVe record sequence Initialise
SAVF
SAVe record sequence Finish
SAVR
SAVe record
SCHR
Set CHaR in a character array (string)
SDBL
Set DouBLe (fp8) array element
SFLD
Set FieLD contents

SFST
Set FileSTatus (memory or disk-based)

SIFL
Set Index Filter Level parameters

SIRS
Set Inter Record Space

SISL
Set Index Sort Level parameters

SLNG
Set LoNG integer array element

SMRK
Set MaRK status

SORT
set SORT parameters

SSHR
Set SHoRt integer array element

SSTR
Set STRing as a line

SVST
Set View STatus

UNUS
UNUSe a buffer

USE
USE a file

Problems

We have noticed some problems that may occur when using DATAdesign. These problems can easily be prevented, but can cause major problems when you forget them.

write protect

If you have a disk-based file, and the medium the file is stored on is write protected, then problems have been known to occur. The fact that your medium is write protected can sometimes not be reported or only much too late, and can sometimes even crash the computer. This problem may be hardware dependent.

removing disk

If you remove a medium on which a disk-based file is based, and you try to read another medium from that drive, the computer will report a *not found* error. You will have to re-insert the original disk. The problem can then be solved by un-using all the buffers to all disk-based files on that medium.

Problems may also occur if you try to write to a disk-based file if the medium is not present. The medium-file should however not be corrupted.

not found

This error may be reported when you still have disk-based files on a medium in a drive where you have inserted another medium. Try replacing the medium with the original and remove all buffers which access that file. Also see: removing disk.

INDEXES

There a 4 different indexes here :

- The “main” index, an alphabetical index which contains most references, except for Sbasic keywords, C functions and Assembler thing extension names.
- The “SBasic” index, which contains an index to the Sbasic keywords.
- The “assembler” index, which contains an index to the assembler Thing extension names.
- The “C” index, which contains an index of the C functions.

Alphabetical Index

SBasic keywords index

Basic keyword index

Index of Assembler thing extensions

Assembler extensions

Index of C functions

C functions

PROGS, Professional & Graphical Software
last edited September 13, 1996

Converted to odt and pdf April 2015 W. Lenerz