Parte xxxii

File «.DBF»

«

File «.DBF»: dBase III e derivati

Il software basato sui file in formato '`.DBF`', ovvero quelli di dBase III, negli anni 1980 è stato molto importante nell'ambito del sistema operativo Dos. Nel suo piccolo ha permesso agli utenti di quel sistema operativo di realizzare delle strutture di dati che si avvicinavano alle potenzialità di una base di dati relazionale.

Ancora oggi si trovano programmi applicativi gestionali basati su questo formato, scritti probabilmente con il famoso compilatore Clipper. Attualmente è disponibile il compilatore Harbour, che si ripromette di offrire un ambiente totalmente compatibile con il passato; tuttavia è possibile leggere il contenuto di questi file attraverso alcuni piccoli programmi.

Dbview

Il programma '**`dbview`**' [1] consente di leggere il contenuto dei file '`.DBF`' di dBase III e probabilmente anche le versioni di dBase IV.

```
dbview [opzioni] file_dbf
```

Se viene avviato senza opzioni, si ottiene la visualizzazione del contenuto del file indicato nel formato predefinito, come si vede dall'esempio seguente:

```
Articolo   : 1
Descr      : bicicletta uomo
Prezzo u   : 500.00
Import     : T
Scadenza   : 20011120
Note       : 2

Articolo   : 2
Descr      : bicicletta donna
Prezzo u   : 550.00
Import     :
Scadenza   : 20011120
Note       : 3

Articolo   : 3
Descr      : bicicletta uomo/donna leggera
Prezzo u   : 600.00
Import     :
Scadenza   : 20011120
Note       : 4
```

In realtà, così facendo, i nomi degli attributi vengono mostrati in modo diverso dal reale, utilizzando anche le lettere minuscole ed eliminando i trattini bassi. Utilizzando l'opzione '**`-r`**', la prima tupla apparirebbe così:

```
ARTICOLO   : 1
DESCR      : bicicletta uomo
PREZZO_U   : 500.00
IMPORT     : T
SCADENZA   : 20011111
NOTE       : 2
```

È necessario osservare che gli attributi booleani (in questo caso si tratta di quello intitolato '**`IMPORT`**') mostrano solo la lettera '**`T`**' per il valore *Vero*, altrimenti non si ha alcuna indicazione; inoltre, le date vengono espresse secondo il formato ***aaaammgg***. Infine, dall'esempio non si intuisce, ma l'attributo '**`NOTE`**' è di tipo «memo» e in questo caso si sono persi i dati.

I dati contenuti nei file '`.DBF`', dal momento che sono stati memorizzati presumibilmente con un sistema operativo Dos, utilizzano molto probabilmente un insieme di caratteri ristretto e incompatibile con gli standard comuni; pertanto, è probabile che sia necessario rielaborare ciò che si ottiene con '**`dbview`**' attraverso un programma di conversione come Recode (sezione 47.8.1). Tuttavia, è bene considerare che nella storia dei file '`.DBF`' sono state usate anche codifiche differenti dal solito IBM437 e di questo occorre tenerne conto quando ci si accorge che la conversione non funziona come ci si aspetterebbe.

Tabella u135.3. Alcune opzioni.

Opzione	Descrizione
`--browse` `-b`	Se si utilizza questa opzione, le tuple vengono mostrate su una sola riga per volta, separando gli attributi con un simbolo, il separatore, che di solito è costituito dai due punti ('`:`').
`--delimiter` ***x*** `-d` ***x***	Con questa opzione è possibile specificare il simbolo da utilizzare per separare gli attributi delle tuple che vengono visualizzate. Il simbolo di separazione predefinito sono i due punti ('`:`')
`--description` `-e` ***x***	In questo caso, oltre a mostrare il contenuto del file, nella parte iniziale vengono riepilogate le caratteristiche degli attributi contenuti.
`--omit` `-o` ***x***	Non elenca il contenuto del file, ma si limita a dare le altre informazioni se richieste attraverso le opzioni opportune.

Opzione	Descrizione
`--reserve` `-r` ***x***	Mostra i nomi degli attributi così come sono stati memorizzati.

Segue la descrizione di alcuni esempi.

- `$` **`dbview articoli.dbf`** [*Invio*]

  Elenca il contenuto del file '`articoli.dbf`' nella forma predefinita.

- `$` **`dbview -b articoli.dbf`** [*Invio*]

  Mostra le tuple utilizzando una sola riga per ognuna.

- `$` **`dbview -b articoli.dbf | recode ibm437:latin1`** [*Invio*]

  Come nell'esempio precedente, ma utilizza '**`recode`**' per trasformare i caratteri speciali che altrimenti non sarebbero visibili correttamente (per esempio le lettere accentate).

DBF2pg

«

Il programma '**`dbf2pg`**' [2] consente di leggere il contenuto di un file '`.DBF`' e di inserire i dati relativi in una relazione di una base di dati di PostgreSQL.

```
dbf2pg [opzioni] file_dbf
```

In base alle opzioni che vengono indicate, i dati possono essere aggiunti a una relazione esistente, oppure possono sostituire le tuple di

tale relazione, oppure si può creare una relazione da zero. Quello che conta è che i permessi fissati attraverso PostgreSQL consentano l'accesso e le operazioni che si intendono svolgere.

'**dbf2pg**' non è in grado di trasferire gli attributi «memo», quelli che tradizionalmente venivano creati utilizzando file con estensione '`.DBT`'.

Tabella u135.4. Alcune opzioni.

Opzione	Descrizione

`-v`
 	
`-vv`
	Permette di avere informazioni sulle operazioni svolte, ottenendo un dettaglio maggiore nel secondo caso.

`-h` ***nodo***	Permette di specificare il nodo a cui accedere per connettersi con il servente di PostgreSQL. In mancanza di questa indicazione, viene tentato l'accesso a *localhost*.
`-d` ***base_di_dati***	Permette di specificare il nome della base di dati a cui ci si vuole connettere. In mancanza di questa indicazione, viene tentata la connessione con la base di dati '**`test`**'.
`-t` ***relazione***	Permette di specificare il nome della relazione in cui si vogliono trasferire i dati del file '`.DBF`'. In mancanza di questa indicazione, viene tentato l'inserimento nella relazione '**`test`**'.

Opzione	Descrizione
`-D`	Con questa opzione, si fa in modo di cancellare il contenuto della relazione di destinazione, prima di iniziare l'inserimento dei dati.
`-c`	Richiede espressamente che sia creata la relazione di destinazione. In mancanza di questa opzione, la relazione deve essere già disponibile, altrimenti l'operazione fallisce. Nel caso si utilizzi questa opzione mentre una relazione con lo stesso nome esiste già, si ottiene la cancellazione del suo contenuto prima di iniziare, come se fosse stata usata al suo posto l'opzione '**`-D`**'.
`-f`	Prima di procedere, converte i nomi degli attributi in modo che questi siano scritti utilizzando solo lettere minuscole.
`-l` `-u`	Con l'opzione '**`-l`**' si fa in modo che il contenuto degli attributi venga convertito in lettere minuscole, mentre con l'opzione '**`-u`**' si ottiene una conversione in maiuscole.
`-s` ***nome_vecchio*=*nome_nuovo*** [***, nome_vecchio*=*nome_nuovo***] ...	Con questa opzione si può stabilire la sostituzione di alcuni nomi degli attributi della relazione. Ciò può essere particolarmente utile nel caso in cui i nomi originali siano incompatibili con PostgreSQL.

Opzione	Descrizione
`-s` ***n_riga_iniziale*** `-e` ***n_riga_finale***	Le opzioni '**`-s`**' e '**`-e`**' permettono di definire l'intervallo di righe da trasferire, dove nel primo caso si indica la riga iniziale e nel secondo quella finale. Se non si indicano, il trasferimento parte dall'inizio e prosegue fino alla fine.

Segue la descrizione di alcuni esempi.

- `$` **`dbf2pg -d Anagrafe -c -t Indirizzi address.dbf`**[*Invio*]

  Crea la relazione '**`Indirizzi`**' nella base di dati '**`Anagrafe`**' disponibile presso l'elaboratore locale, prelevando i dati dal file '`address.dbf`'.

- `$` **`dbf2pg -h localhost -d Anagrafe -c -t Indirizzi address.dbf`**[*Invio*]

  Esattamente come nell'esempio precedente, con l'indicazione precisa del nodo locale.

[1] **Dbview** GNU GPL

[2] **DBF2pg** software libero con licenza speciale

nanoBase 1997

An old, but free xBase for Dos.[1]

What is it

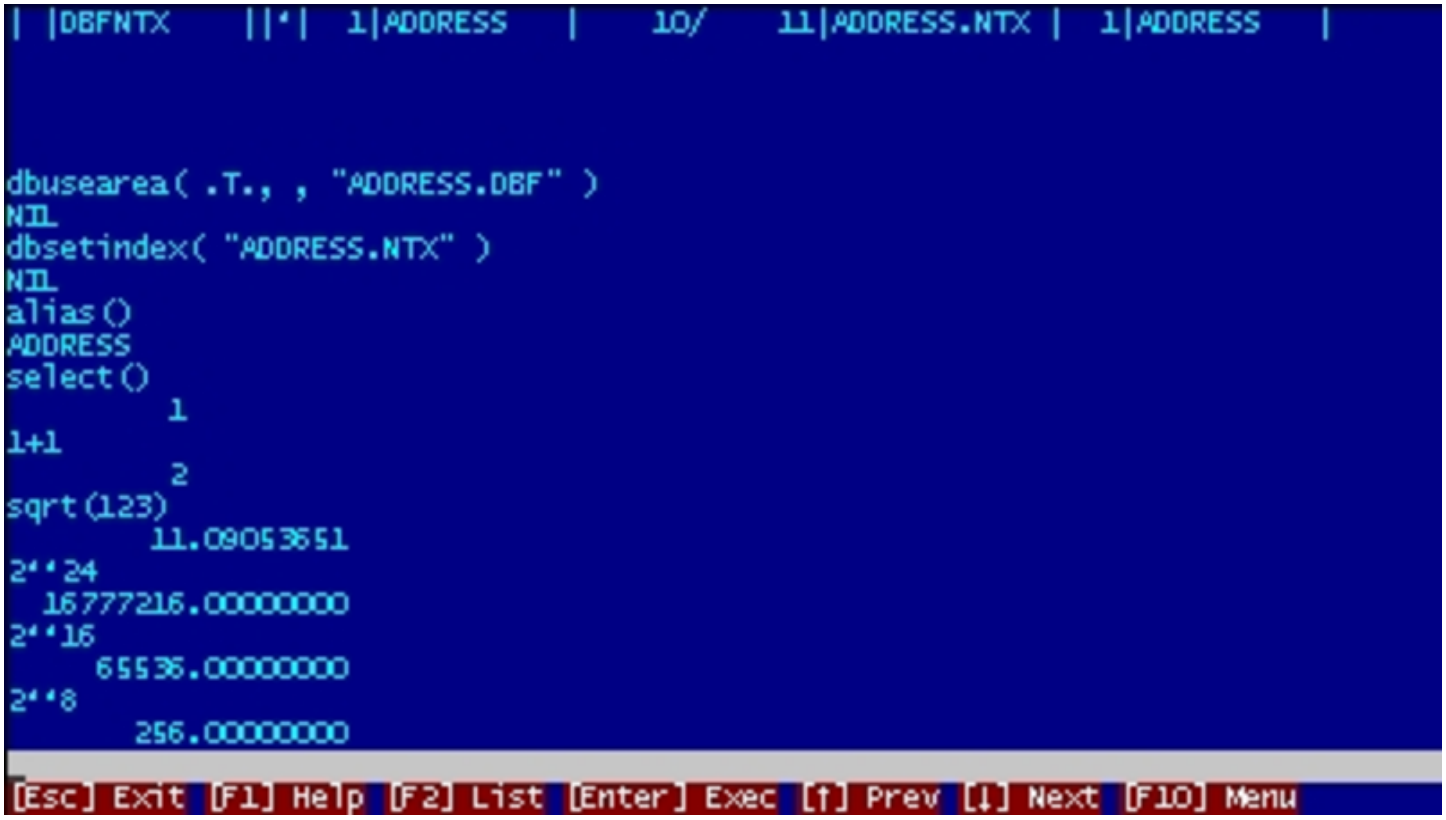
nanoBase [2] is a Dos program that works essentially as:

- a dot command line xBase,
- a menu driven xBase,
- a xBase program interpreter.

nanoBase 1997 is compiled in two versions: a small one to be used with old computers (x86-16 with 640 Kibyte RAM), and a second one to be used with better computers, at least i286 (or better) with 2 Mibyte RAM.

The dot command line

Figure u136.1. The dot line.

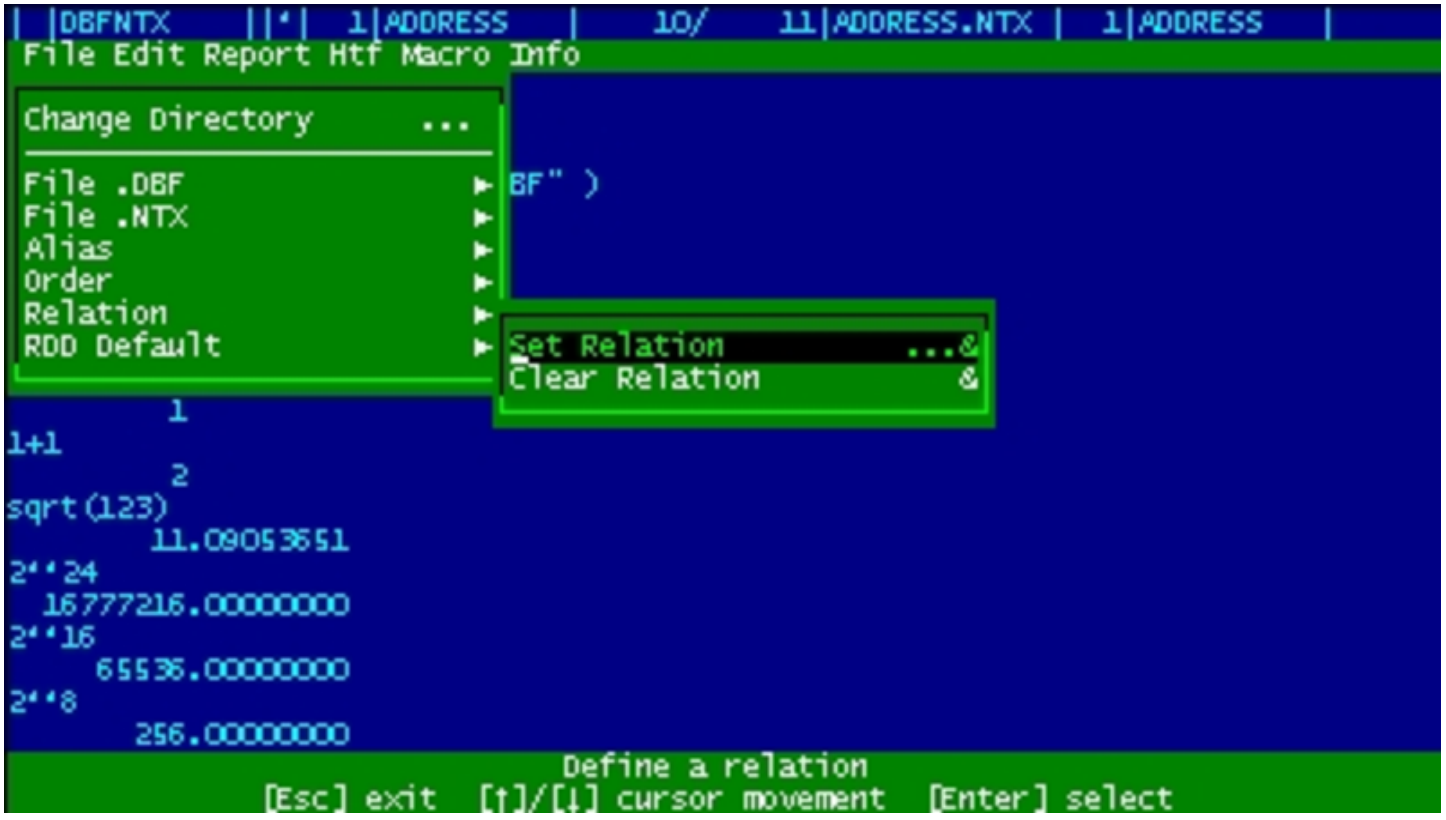


The dot command line is the first face of nanoBase, the one that appears starting the program normally. It recalls the dot line command of the old xBases.

Please note that **nanoBase recognise only expressions** (that is: no commands).

The menu

Figure u136.2. The file menu.



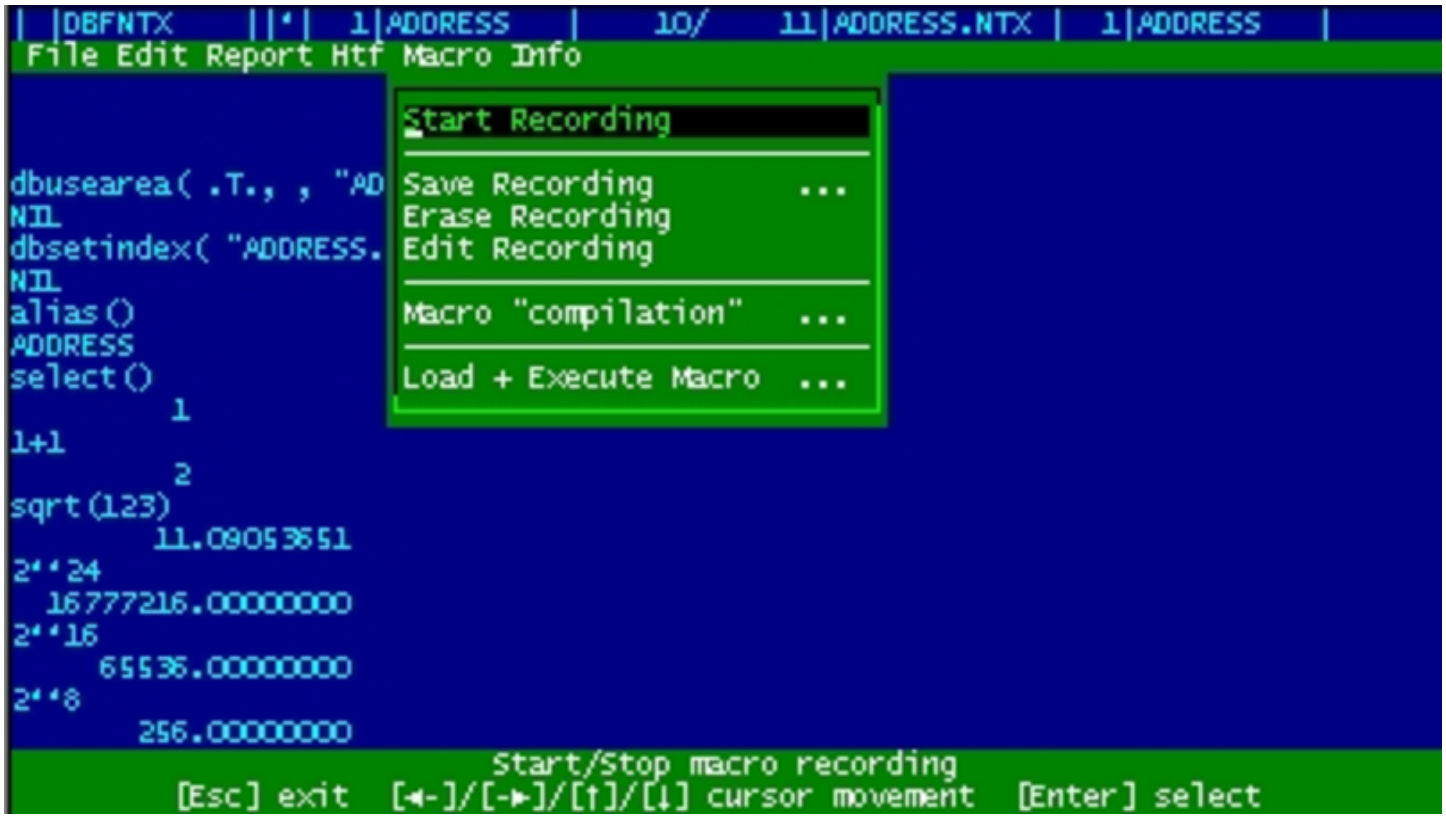
Pressing [*F10*] the nanoBase menu appears.

From this menu the operations are easier than writing all commands on a prompt line, but it is always possible to come back to the dot line to do an operation not available from the menu.

The macro recording, compiling and execution

«

Figure u136.3. The macro menu.



nanoBase is able to record some actions made with the menu and all what is correctly typed from the dot prompt. This may be the begin for a little program (called macro inside nanoBase) that can be executed as it is (ASCII), or compiled into another format, faster to execute.

Macros for nanoBase are made with a reduced set of the Clipper syntax. The statements recognised from nanoBase are:

```
PROCEDURE procedure_name
      statements...
      [RETURN]
      statements...
ENDPROCEDURE
```

```
DO PROCEDURE procedure_name
BEGIN SEQUENCE
     statements...
     [BREAK]
     statements...
END
```

```
DO CASE
CASE lCondition1
     statements...
[CASE lCondition2]
     statements...
[OTHERWISE]
     statements...
END
```

```
WHILE lCondition
     statements...
     [EXIT]
     statements...
     [LOOP]
     statements...
END
```

```
IF lCondition1
      statements...
      [ELSE]
      statements...
END
```

- the '**FOR**' loop is not available (too difficult to implement),
- there may be no user defined functions (code blocks may be created instead),
- procedure calls cannot transfer variables,
- there are only public (global) variables.

Beside these limitations, there are many added functions to the standard language that make the programming easier.

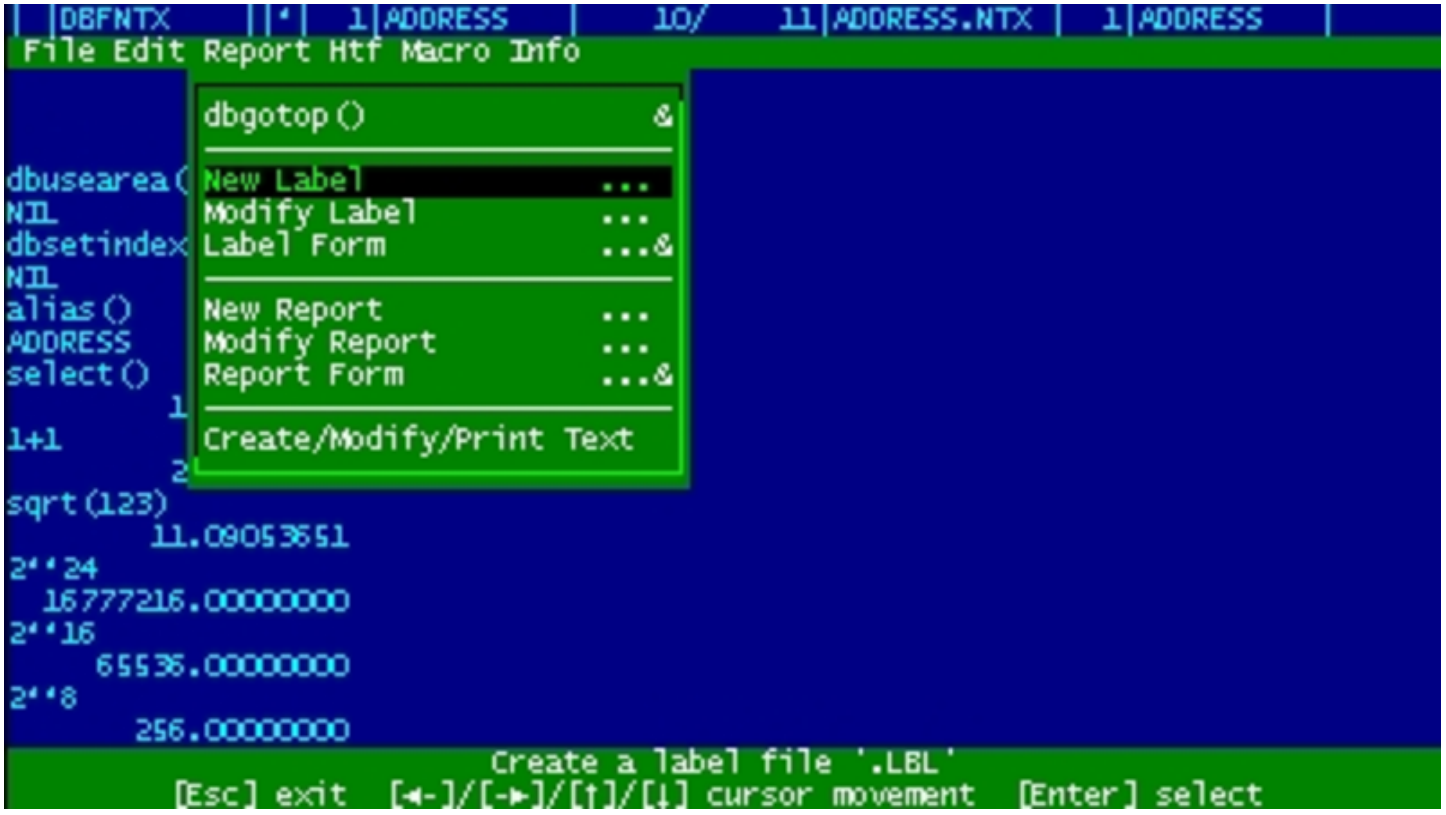
All you need is inside '**NB.EXE**':

- the utility to handle manually the data,
- the macro compiler,
- the macro executor.

The report system

«

Figure u136.4. The report menu.



nanoBase can handle label (‘.LBL’) and form (‘.FRM’) files in the dBaseIII format. Labels and forms may be created and edited inside nanoBase. Beside these old report system there is another way to make a little bit complicated reports without making a complex macro: it is called RPT.

A RPT file is a ASCII file with text mixed with code. The text may contain variables (usually a field or an expression containing fields).

To make a complex report some work is needed, but surely less than the time needed to make a report program.

The main purpose of it was to be able to print text with variables (typically names and addresses) for every record of a particular ‘.DBF’ file. Now the RPT system makes something more.

The integrated text editor

«

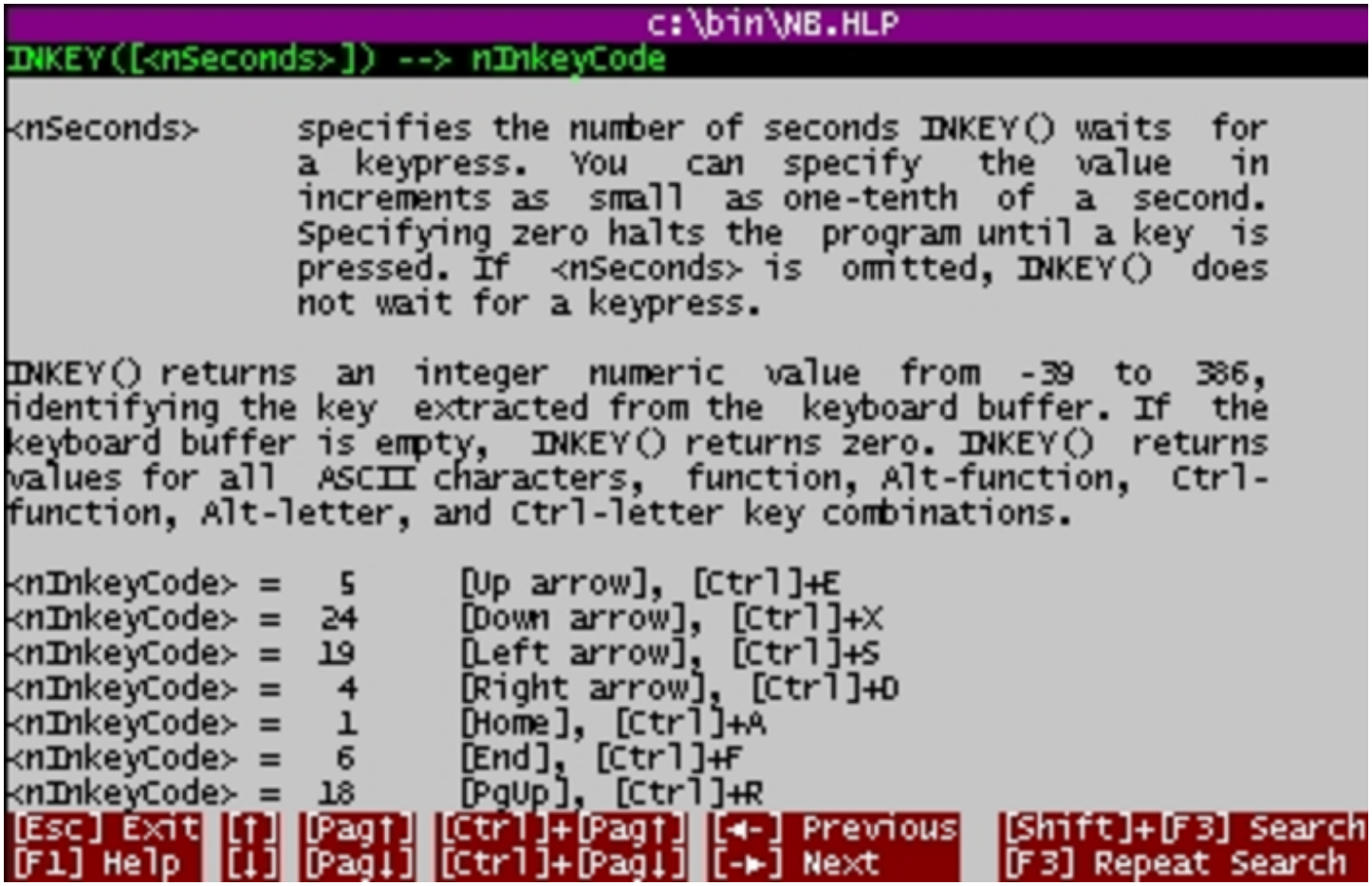
Figure u136.5. The integrated text editor.

nanoBase contains an integrated text editor not particularly good, but very usefull for RPT files (as the expression insertion is very easy with the use of the [*F2*] key) and whenever there isn't any other editor there.

The internal documentation

«

Figure u136.6. The internal documentation.



nanoBase's documentation si translated also inside the HTF format: 'NB.HLP'. Pressing [*F1*], normally, a contextual piece of the manual appears.

Some standard functions have its own internal help, contained inside the '.EXE' file. This was made to help programming with nanoBase.

Download it

«

Here is the 1997 edition of nanoBase.

- EXE for small computers.
  ftp://ftp.simtel.net/pub/simtelnet/msdos/database/nbase7a1.zip
  http://www.google.com/search?q=nbase7a1.zip

- EXE for i286 with more than 2 Mibyte.
  ftp://ftp.simtel.net/pub/simtelnet/msdos/database/nbase7a2.zip
  http://www.google.com/search?q=nbase7a2.zip
- Runtime for small computers.
  ftp://ftp.simtel.net/pub/simtelnet/msdos/database/nbase7a3.zip
  http://www.google.com/search?q=nbase7a3.zip
- Documentation in many different formats.
  ftp://ftp.simtel.net/pub/simtelnet/msdos/database/nbase7a4.zip
  http://www.google.com/search?q=nbase7a4.zip
- Macro programming examples.
  ftp://ftp.simtel.net/pub/simtelnet/msdos/database/nbase7a5.zip
  http://www.google.com/search?q=nbase7a5.zip
- Source for version 96.06.16, without mouse support (1996).
  ftp://ftp.simtel.net/pub/simtelnet/msdos/database/nbase7a6.zip
  http://www.google.com/search?q=nbase7a6.zip
- Source for version 1997.
  ftp://ftp.simtel.net/pub/simtelnet/msdos/database/nbase7a7.zip
  http://www.google.com/search?q=nbase7a7.zip

Bugs and known problems

«

Here is the list of known bugs and problems.

- Comparison with floating point numbers may fail. It is better to convert numbers into string before comparing them.

- Macros may be contained inside ASCII files or a ‘‘compiled’’ ‘`.DBF`’ file. In the second case, when nanoBase executes the macro, a work area (the last available one) is used, so it should not be closed or the macro execution will be stopped. A ‘**`dbcloseall()`**’ will stop execution of the macro. In substitution of ‘**`dbcloseall()`**’, ‘**`DBCLOSE()`**’ should be used.
- To simplify the macro interpretation, lines such as this:

```
qqout( "You can't do that // you can't do that!" )
```

  will generate an error as the interpreter will read only:

```
qqout( "You can't do that
```

- nanoBase works good also if you have a screen configuration that permits you to show more than the usual 80 columns and 25 lines, but the library used to handle the mouse is not able to work outside the 80×25 area.

[1] This material appeared originally at ‘`http://www.geocities.com/SiliconValley/7737/nanobase.html`’, in 1997.

[2] **nanoBase** GNU GPL

nB[1] (‘‘nano Base’’: ‘‘n’’ = ‘‘nano’’ = 10**(−9) = ‘‘very little’’) is a little Dos xBase written in Clipper 5.2 that can help to access ‘.DBF’ file created with different standards.

nB is:

- a dot command interpreter,
- a menu driven xBase,
- a xBase program interpreter.

Dos xBase

«

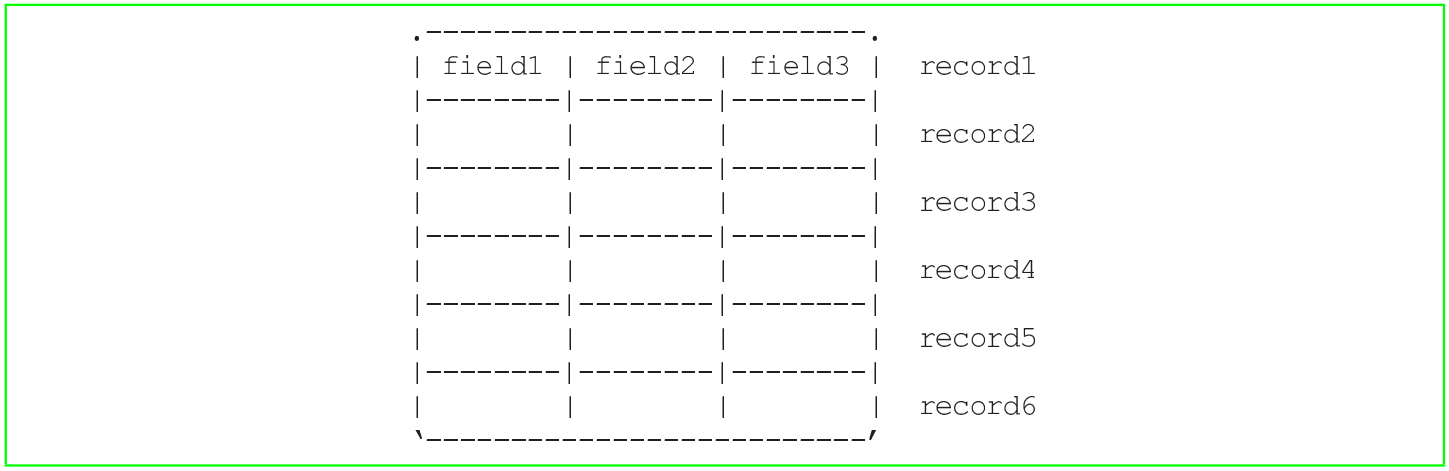
This section is a brief description of the functionality of a typical Dos xBase.

The first purpose of a xBase program is to handle data inside a ‘.DBF’ file. These files may be indexed with the help of index files and more ‘.DBF’ files may be linked with a relation to obtain something like a relational database.

.DBF files

«

‘.DBF’ files are files organised in a table structure:



The lines of this table are records and the columns are fields. Records are numbered starting from the first that is number 1.

Columns are defined as fields and fields are distinguished by name and these names are saved inside the '`.DBF`' file.

Every field (column) can contain only one specified kind of data with a specified dimension:

- '**`C`**', character, originally the maximum dimension was 254 characters, minimum is 1;
- '**`N`**', numeric, a numeric field that can contain also sign and decimal values;
- '**`D`**', date, a field dedicated to date information;
- '**`L`**', logic, a filed that may contain only '**`T`**' for True or '**`F`**' for False used as a boolean variable;
- '**`M`**', memo, a character field with no predefined dimension, not allocated directly inside the '`.DBF`', but inside a '`.DBT`' file, automatically linked.

No other field type is available for a typical xBase '`.DBF`' file.

To access the data contained inside a '.DBF' file the following list of action may be followed:

- Open a '.DBF' file inside the current area, where these areas are something like file handlers.
- After the '.DBF' file is opened, it referenced only by the alias name that usually correspond to the original filename without extention.
- Move the record pointer to the desired location.
- Lock the current record to avoid access from other users.
- Do some editing with the data contained inside the current record using the field names like they were variables.
- Release the lock.
- Move the record pointer to another desired location.
- Lock the current record to avoid access from other users.
- ...
- Close the alias.

Before you go further, you have to understand that:

- A '.DBF' file is opened using a free WORK AREA that may be associated to the concept of the file handler.
- The '.DBF' file is opened with a alias name that permit to open the same '.DBF' file more times when using different alias names.
- After the '.DBF' file is opened, we don't speak any more of file, but alias.

- If the work area "n" is used from the alias "myAlias", speaking of work area "n" or of alias "myAlias" is the same thing.

Index files

‘.DBF’ files are organised with record number, that is, you can reach a specific record and not a specific information unless that you scan record by record.

To obtain to "see" a ‘.DBF’ file somehow logically ordered (when physically it is not), index files are used.

A index file, also called INDEX BAG, is a file that contains one or more indexes

Indexes are rules by which a ‘.DBF’ file may be seen ordered.

A typical index file may contain only one index.

A index file may have the following extention:

- ‘.NDX’, single index, dBase III and dBase III plus;
- ‘.NTX’, single index, Clipper;
- ‘.MBX’, multiple index, dBase IV;
- ‘.CDX’, multiple index, FoxPro.

Every index file may be used only in association with the ‘.DBF’ for what it was made. The problem is that normally there is no way to avoid errors when the user try to associate the right ‘.DBF’ file with the wrong index.

To access the data contained inside a '.DBF' file the following list of action may be followed:

- Open a '.DBF' file.
- Open a index file.
- Select a particular order.
- Search for a key or move the record pointer on a different way.
- Lock the current record to avoid access from other users.
- Do some editing with the data contained inside the current record using the field names like they were variables.
- Release the lock.
- Move the record pointer to another desired location.
- Lock the current record to avoid access from other users.
- ...
- Close the alias.

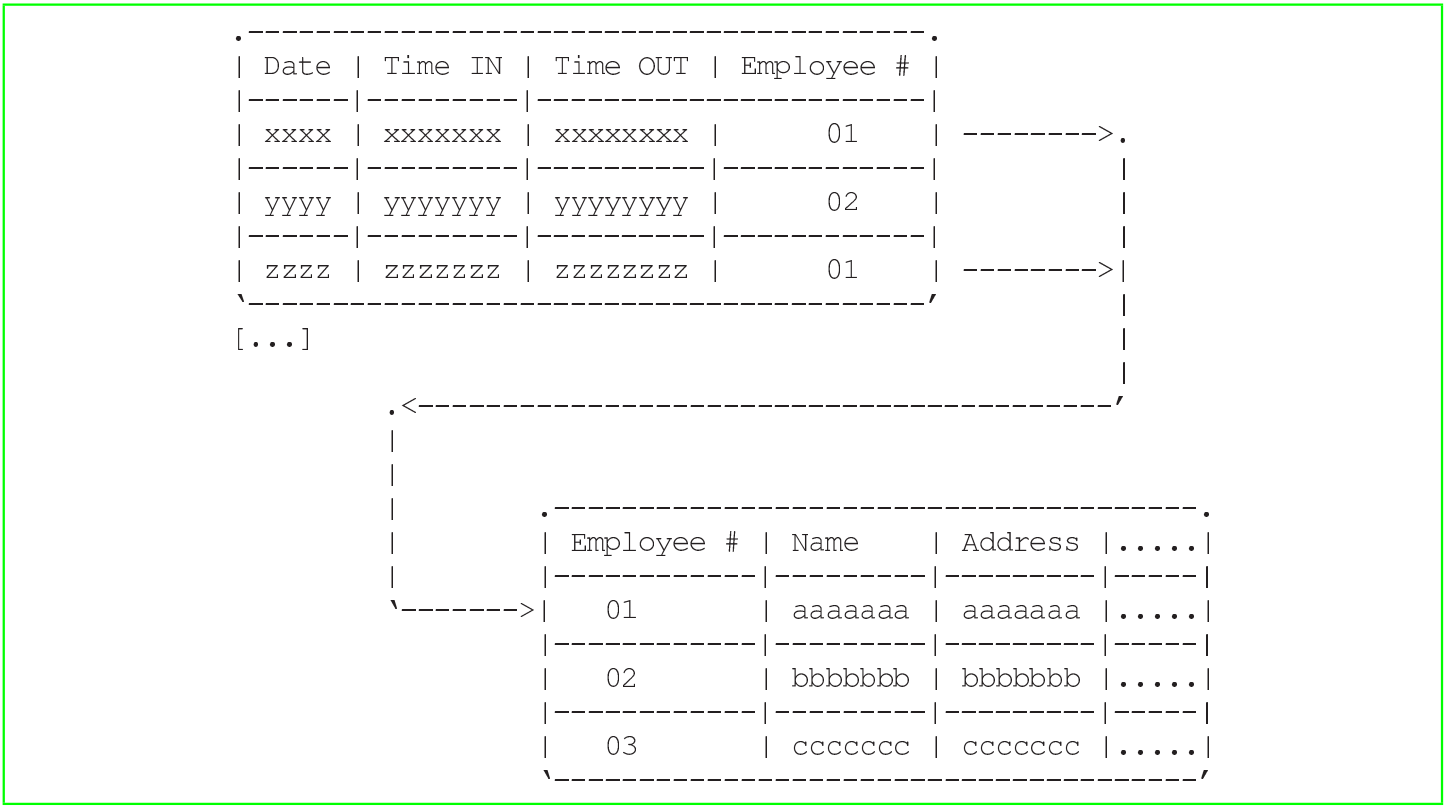
Before you go further, you have to understand that:

- As orders are contained inside a INDEX BAG file physically distinguished form the '.DBF' file, it may happen that a '.DBF' file is wrongly opened and edited without the index. In this case, the INDEX BAG is not updated and when the INDEX BAG will be opened, the records contained inside the '.DBF' file may not correspond.
- For the same reason, an improper program termination may result in an incomplete data update. That is: '.DBF' file may be all right, INDEX BAG not.

- This is why xBase programs are "weak" relational databases or they are not relational databases at all.
- When troubles occurs, indexes must be rebuild.

Relations

Many '.DBF' files with indexes may be opened simultaneously. Data contained inside more '.DBF' files may be somehow connected together. See the example.



The first '.DBF' file contains some data that refers to an Employee number that may appear repeated on more records.

Employee informations are stored inside another '.DBF' file that contains only one record for every employee.

Establishing a relation from the first '.DBF' file to the second, moving the record pointer of the first '.DBF' file, that is the first alias, the

record pointer of the second, the child alias, is moved automatically to the record containing the right data.

The relation is an expression that should result in a number if the child alias is opened without index, or in a valid index key if the child alias is opened with an index.

To relate two '`.DBF`' files the following list of action may be followed:

- Open the first '`.DBF`' file.
- Open a index file for the first alias.
- Select a particular order.
- Open the second '`.DBF`' file.
- Open a index file for the second alias.
- Select a particular order.
- Select the first alias.
- Define a relation form the first alias and the second alias: the child alias.
- Search for a key or move the record pointer of the first alias (don't care about the Child alias).
- Lock the current record to avoid access from other users.
- If data contained inside the Child alias should be edited (usually it doesn't happen), lock the current record of the Child alias.
- Do some editing with the data contained inside the current record using the field names like they were variables.
- Release the lock (also with the Child alias if a lock was made).

- Move the record pointer to another desired location.
- Lock the current record to avoid access from other users.
- [...]
- Release the relation.
- Close the Child alias.
- Close the first alias.

As may be seen, relations are not saved inside files, but are obtained with lines of code.

Composition

nB is composed from the following files, where ***xx*** is the the version code.

NBASExx1.ZIP	EXEs for small PCs
NBASExx2.ZIP	Runtime EXEs for small PCs
NBASExx3.ZIP	EXEs for i286 with 2M+
NBASExx4.ZIP	DOCs
NBASExx5.ZIP	EXAMPLEs
NBASExx6.ZIP	SRCs for version 96.06.16
NBASExx7.ZIP	SRCs for the current version

Every archive file contains:

'COPYING.TXT'	GNU General Public License version 2 in Dos text format.
'README.TXT'	the readme file.
'FILE_ID.DIZ'	definition.

The file 'NBASExx1.ZIP' contains also the following files.

‘NB.EXE’	the executable program for DBFNTX and DBFNDX files, linked with RTLINK.
‘NB.HLP’	this manual in "Help Text File" format.

The file NBASExx2.ZIP contains also the following files.

‘NB.EXE’	the run-time to execute macro programs for DBFNTX and DBFNDX files handling, linked with RTLINK.

The file ‘NBASExx3.ZIP’ contains also the following files.

‘NB.EXE’	the executable program for DBFCDX, DBFMDX, DBFNDX and DBFNTX files, linked with EXOSPACE.
‘NB.HLP’	the user manual in "Help Text File" format.

The file ‘NBASExx4.ZIP’ contains also the following files.

‘NB.PRN’	the user manual in printed text format.
‘NB.RTF’	the user manual in RTF format.
‘NB.TXT’	the user manual in ASCII text format.
‘NB.HTM’	the user manual in HTML format.

The file ‘NBASExx5.ZIP’ contains also the following files.

‘_ADDRESS.DBF’	an example database file.
‘_ADDRESS.NTX’	index file associated to ‘_ADDRESS.DBF’.
‘_ADDRESS.LBL’	a label form file used to print data contained inside ‘_ADDRESS.DBF’ .
‘_ADDRESS.FRM’	a report form file used to print data contained inside ‘_ADDRESS.DBF’ .
‘_ADDRESS.RPT’	a RPT text file used to print data contained inside ‘_ADDRESS.DBF’ .

‘_MAINMNU.&’	a macro program source example of a menu that executes some others macro programs. This example is made to demonstrate how nB can execute directly a source code without compiling it. This example is made only to taste it: it is very slow and only a speedy machine can give the idea of it.
‘0MAINMNU.&’	a macro program source example of a menu that executes some others macro programs. It is the same as ‘_MAINMNU.&’ but it is made to start the execution of the compiled macros.
‘0MAINMNU.NB’	compiled macro program ‘0MAINMNU.&’
‘0MAINMNU.BAT’	a batch file to show how to run the execution of ‘0MAINMNU.NB’
‘1ADDRESS.&’	a macro program source example for handling a ‘.DBF’ file containing addresses in various ways.
‘1ADDRESS.NB’	compiled macro ‘1ADDRESS.&’.
‘2ADDRESS.&’	a macro program source example for handling a ‘.DBF’ file containing addresses in various ways: a little bit more complicated than 1ADDRESS.&.
‘2ADDRESS.NB’	compiled macro ‘2ADDRESS.&’.
‘3ADDRESS.&’	a macro program source example for handling a ‘.DBF’ file containing addresses in various ways: a little bit more complicated than ‘2ADDRESS.&’.
‘3ADDRESS.NB’	compiled macro ‘3ADDRESS.&’.

‘4ADDRESS.&’	a macro program source example for handling a ‘.DBF’ file containing addresses in various ways: a little bit more complicated than ‘3ADDRESS.&’.
‘4ADDRESS.NB’	compiled macro ‘4ADDRESS.&’.
‘ABIORITM.&’	a macro program source example for calculating the personal bio wave.
‘ABIORITM.NB’	compiled macro ‘ABIORITM.&’.
‘_STUDENT.DBF’	a ‘.DBF’ file used inside the BSTUDENT macro example.
‘_STUDENT.NTX’	index file used for ‘_STUDENT.DBF’.
‘_STUDSTD.DBF’	a ‘.DBF’ file used inside the BSTUDENT macro example.
‘_STUDENT.RPT’	a RPT text file used to print data contained inside ‘_STUDENT.DBF’.
‘_STUDSTD.RPT’	a RPT text file used to print data contained inside ‘_STUDSTD.DBF’.
‘BSTUDENT.&’	a macro program source example for students evaluation: a description about students is obtained linking other standard descriptions.
‘BSTUDENT.NB’	compiled macro ‘BSTUDENT.&’.
‘CBATMAKE.&’	a macro program source example to generate a batch file to be used to back up an entire hard disk.
‘CBATMAKE.NB’	compiled macro ‘CBATMAKE.&’.
‘BROWSE.&’	a macro program source example to start an automatic browse.
‘BROWSE.NB’	compiled macro ‘BROWSE.&’.
‘BROWSE.BAT’	batch file to start a ‘.DBF’ browse with the BROWSE macro program.
‘MENU.&’	a macro program source example for a Dos menu.

'MENU.NB'	compiled macro 'MENU.&'.
'MENU.BAT'	batch file to use the MENU macro.

The file 'NBASExx6.ZIP' contains also the following files: source code for the version 96.06.16.

'NB.PRG'	the main source file for version 96.06.16.
'NB_REQ.PRG'	the source file containing links to all the standard functions.
'NB.LNK'	link file for compilation.
'NB_NRMAL.RMK'	rmake file to compile with RTLink.
'NB_EXOSP.RMK'	rmake file to compile with Exospace.
'NB_RUNTI.RMK'	rmake file to compile with RTLink defining RUNTIME to obtain a small nB runtime version.
'MACRO.LNK'	link file to compile and link a macro.
'MACRO.RMK'	rmake file to compile and link a macro.

The file 'NBASExx7.ZIP' contains also the following files: source code for the current version.

'NB.PRG'	the main source file.
'REQUEST.PRG'	the source file containing links to all the Clipper functions.
'STANDARD.PRG'	the source file for standard functions.
'EXTRA.PRG'	the source file for other standard functions.
'STANDARD.CH'	general include file that substitutes all include file normally used for normal Clipper compilations.
'NB.CH'	include file specific for nB.
'NB.LNK'	link file for compilation.
'NB_RUNTI.LNK'	link file for runtime compilation.
'NB_NRMAL.RMK'	rmake file to compile with RTLink.

'NB_EXOSP.RMK'	rmake file to compile with Exospace.
'NB_RUNTI.RMK'	rmake file to compile with RTLink defining RUNTIME to obtain a small nB runtime version.
'MACRO.CH'	include file to compile and link a macro.
'MACRO.LNK'	link file to compile and link a macro.
'MACRO.RMK'	rmake file to compile and link a macro.
'CLIPMOUSE.ZIP'	a simple free library for mouse support under Clipper (c) 1992 Martin Brousseau.

How to use nB

«

nB normal syntax is:

```
nB [nB_parameters] [macro_filename] [macro_parameters]
```

To run nB, just type the word "NB" and press [*Enter*] to execute. It will run in command mode, this means that it will look like an old xBASE command prompt.

To run the program as a macro interpreter, type the word NB followed from the macro file name with extention (no default extention is supposed). If parameters are given, after the macro file name, these will be available inside the public variables: c_Par1, c_Par2, ..., c_Par9. c_Par0 will contain the macro file name (see the macro file BROWSE.&). nB will terminate execution when the macro terminates.

These parameters are available for nB:

-c	Suppress the copyright notice. It is usefull when using nB for macro interpretation.

-w	Suppress the "Wait-Wheel" if not desired. It is the "Wheel" that appears at top-left when a macro is interpreted or other long elaborations are executed.
-?	Shows a short help.

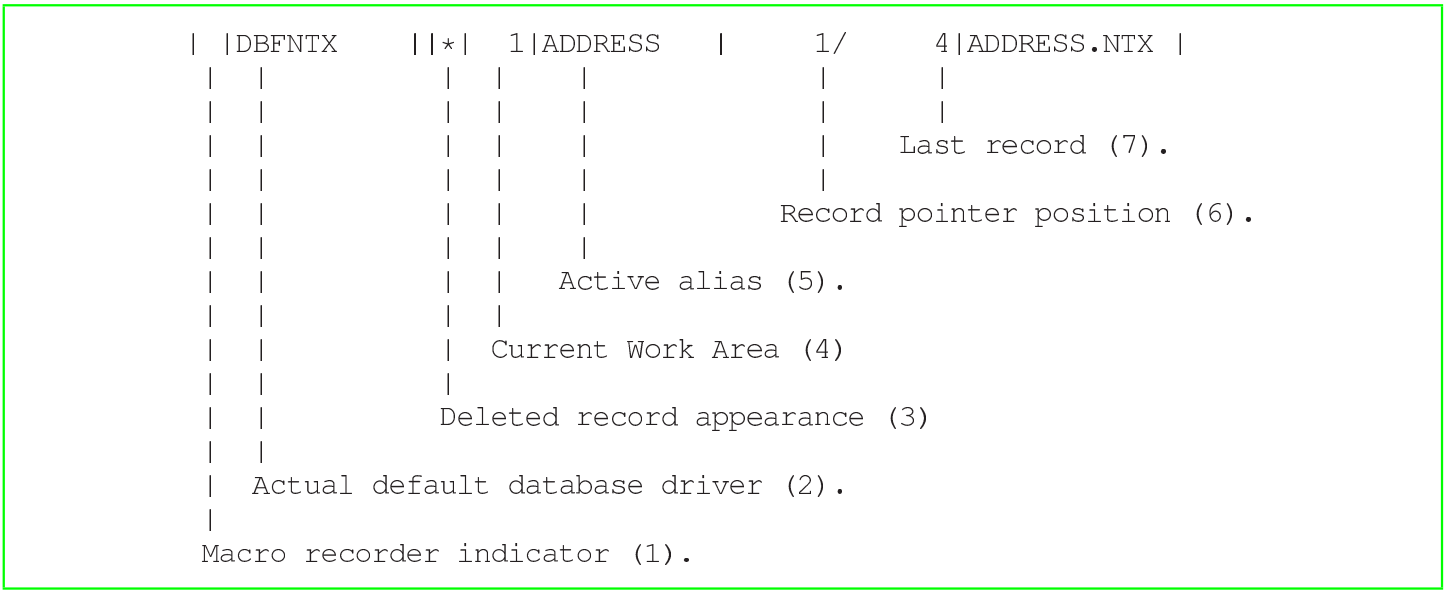
nB macro "compilation" syntax is:

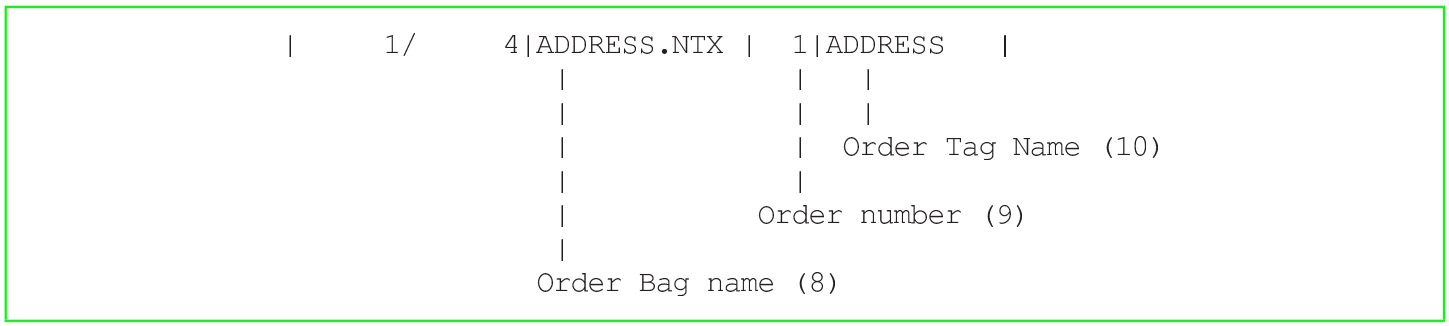
```
nB -m source_macro_filename [destination_macro_filename]
```

With the -m parameter, nB "compiles" the ASCII ***source_macro_filename*** into ***destination_macro_filename***.

Status line

nB shows a "status line" at the top of the screen when the nB command prompt or the menu system is active. It shows some important informations.





(1) This is the place for the macro recorder indicator. The symbol used is "&". Blank means that the macro recorder is OFF; & blinking means that the macro recorder is ON; & fixed means that the macro recorder is PAUSED.

(2) The name of the default database driver. It is not necessarily the database driver for the active alias; it is only the database driver that will be used for the next open/create operation.

(3) An asterisk (*) at this position indicates that SET DELETED is OFF. This means that deleted records are not filtered. When a BLANK is in this place, SET DELETED is ON, so that deleted records are filtered.

(4) The active work area number, that is, the area of the active alias.

(5) The active alias name. Note that the alias name is not necessarily equal to the '.DBF' file name.

(6) The actual record pointer position for the active alias.

(7) The number of records contained inside the active alias.

(8) The Order Bag name; that is the index file name.

(9) The order number.

(10) The order tag (name). When DBFNTX database driver is used, it correspond to the Order Bag name.

The dot line

Starting nB without parameters, the dot line appears. This is the place where commands in form of functions may be written and executed like a old xBase.

The functions written inside the command line that don't result in an error, are saved inside a history list. This history list may be recalled with [*F2*] and then the selected history line may be reused (eventually edited). Key [*up*]/[*down*] may be used to scroll inside the history list without showing the all list with [*F2*].

[*Enter*] is used to tell nB to execute the written function.

As the dot line is not an easy way to use such a program, a menu is available pressing [*F10*] or [*Alt M*]. The [*F10*] key starts the ASSIST() menu. This menu may be started also entering the name of the function: "ASSIST()".

nB includes a simple built-in text editor: DOC(). It may be started from the dot line entering "DOT()". No special key is dedicated to start this function.

The menu system

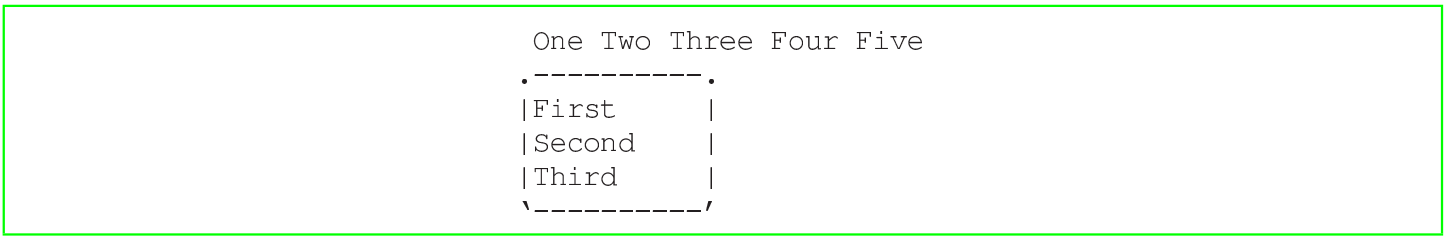
The nB menu system appears differently depending on the place where it is "called". When available, the menu system appears pressing [*Alt M*] or [*F10*].

The Menu system is organised into horizontal menu, vertical menu, and pop-up menu.

The horizontal menu contains selectable items organised horizontally:

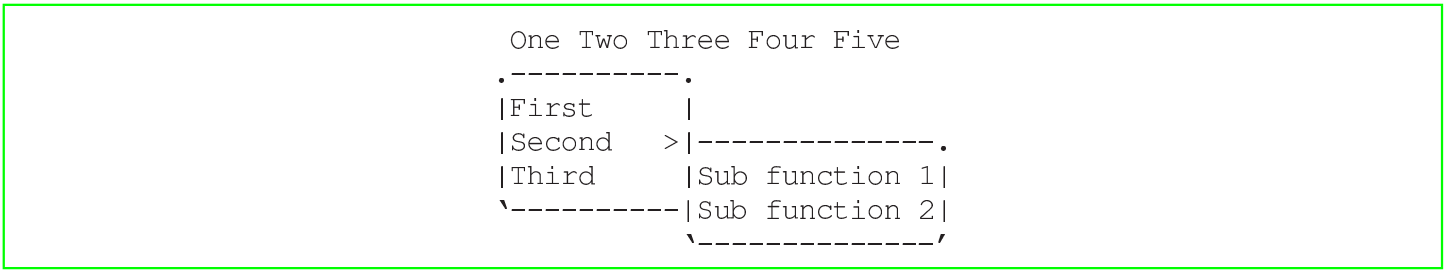
The cursor may be moved on a different position using arrow keys [*Left*]/[*Right*]; [*Esc*] terminates the menu; [*Enter*] opens a vertical menu.

The vertical menu contains selectable items organised vertically:



The cursor may be moved on a different position using arrow keys [*Up*]/[*Down*]; the arrow keys [*Left*]/[*Right*] change the vertical menu; [*Esc*] closes the vertical the menu; [*Enter*] starts the selected menu function.

The vertical menu contains selectable items organised vertically:



The cursor may be moved on a different position using arrow keys [*Up*]/[*Down*]; [*Esc*] closes the pop-up the menu; [*Enter*] starts the selected menu function.

The following sections describe the menu system.

Menu File

«

The menu File contains important function on ‘.DBF’ file, indexes, relations and Replaceable database drivers.

For database files are considered two aspects: the physical aspect,

and the logical alias. When a '.DBF' file is opened, it becomes a alias.

Indexes are considered as index files and index orders.

It follows a brief menu function description.

Change directory

Changes the actual drive and directory.

File .DBF

Contains a pop-up menu for '.DBF' operations.

New .DBF

A '.DBF' file is a table where columns, called Fields, must be specified and lines, called records, are added, edited and deleted by the program.

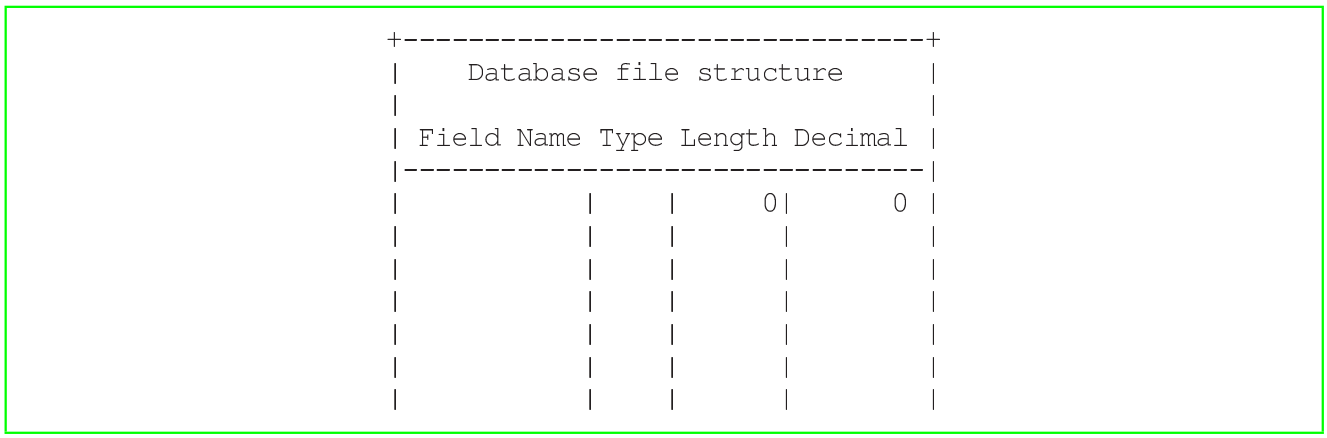
Field characteristics are:

NAME	the field name must be unique inside the same file, it is composed of letters, number and underscore (_), but it must start with a letter and it is not case sensitive.
TYPE	the field type determinates the type of data it can hold.
LENGTH	is the field total length in characters; it doesn't matter of the type of data.
DECIMAL	is the length of positions after decimal point. This information is used normally for numeric fields. In this case, take note that the DECIMAL length, together with the decimal point, will subtract space for the integer part of the number from the total LENGTH of the filed.

Field Types:

C Character	it is a text field long LENGTH characters.
N Numeric	it is a numeric field long LENGTH characters with DECIMAL characters for decimal positions. Note that if LENGTH is 4 and DECIMAL is 0 (zero), the field may contain integers from -999 to 9999; but if LENGTH is 4 and DECIMAL 1, the field may contain numbers from -9.9 to 99.9: two position for the integer part, one position for the decimal point and one position for decimal.
D Date	it is a date field: it contains only dates; the length should not be specified as it is automatically 8.
L Logic	it is a logical (boolean) field: it contains only TRUE, represented by "Y" or "T", or FALSE, represented by "N" or "F". The length should not be specified as it is automatically 1.
M Memo	it is a character field with unknown dimension. It is recorded into a parallel file with '.DBT' extention. The original '.DBF' file holds a space for a pointer inside the '.DBT' file. The length of a Memo field is automatically 10 and is referred to the memo pointer.

After the function "NEW .DBF" is selected, a table for the field specifications appears.



To navigate and to edit the table use the following keys:

[*Up*]/[*Down*]/[*Left*][*Right*]	move the cursor one position (up, down, left or right);
[*PgUp*]	move to previous screen page;
[*PgDn*]	move to next screen page;
[*Ctrl PgUp*]	move to top of table;
[*Ctrl PgDn*]	move to bottom of table;
[*Ctrl Home*]	move to first column;
[*Ctrl End*]	move to last column;
[*Ctrl Enter*]	append a new empty line;
[*Ctrl F1*]	delete (cut) the current line and save a copy into the "clipboard";
[*Ctrl F2*]	copy current line into the table "clipboard";
[*Ctrl F3*]	insert (paste) the content of the "clipboard" in the current position;
[*Enter*]	start editing in the current position;
[*Esc*]	terminate;
[***x***]	any other key will be written in the current position.

When the editing is terminated, press [*Esc*] and a dialog box will ask for the file name and the RDD.

xBase files (.DBF) are not all equal, this way, when a new '`.DBF`' file si created, the RDD (Replaceable Database Driver)

is asked. The normal RDD is DBFNTX, the one used by Clipper.

Modify .DBF structure

The modification of a '.DBF' file structure is a delicate matter if it contains data.

In fact, it is a data transfer from a source '.DBF' file to a destination '.DBF' file with a different structure. This way, the destination '.DBF' will be updated only for the fields with the same name of the source one. The position may be different, but names cannot be changed (not so easily).

Mistakes may be dangerous, so, before doing it, it is recommended a backup copy of the original '.DBF' file.

Open .DBF

When a '.DBF' file is opened, it becomes a alias, a logical file, placed inside a work area. The same '.DBF' file may be opened inside different areas with different alias names.

The required information to open the file are:

FILENAME	the physical file name.
ALIAS	the alias name. If not assigned, it becomes automatically the same of FILENAME without extention.
RDD	the Replaceable Database Driver to use to access to this file.
SHARED	a logical value: TRUE means that the file will be accessible to other users, FALSE means use exclusive.
READ ONLY	a logical value: TRUE means that the file will be only readable and no modification will be allowed, FALSE means that no restriction on editing will be made.

File .NTX

Contains a pop-up menu for physical indexes operations.

New .NTX / new tag

If the active area is used we have an active alias. In this case a index may be created. The index is a way to see the active alias ordered without changing the physical position of records.

There are two words to understand: ORDER and INDEXBAG. The index bag is the file that contains the information on the record ordering, the order is the rule followed to order the records. A index bag may contains one or more orders depending on the Replaceable Database Driver in use.

Typical '`.NTX`' file are index bag containing only one order.

Depending on the RDD in use the following field may be filled.

INDEX FILENAME	this is the name of the index bag.
KEY EXPRESSION	the expression that defines the rule for the record ordering.
ORDER NAME	this is the name to give to the order (tag) when the RDD permits to have a index bag containing more than one order. In the other case, the index bag name correspond to the order name.
FOR EXPRESSION	a FOR condition to filter records before indexing.

Open index

If a index file already exists, it can be associated to the active alias simply opening it.

Take note that the system is not able to verify if the index belong the active alias and if it is not so a error will result.

INDEX NAME	is the name of the index bag file to open.

Alias

Contains a pop-up menu for logical databases (alias) operations.

Select

Only one may be the active alias and with this function the active alias may be changed choosing from the list of used areas.

Selecting the area number zero, no alias is active.

Display structure

With this function the active alias structure may be viewed.

Close active alias

Selecting this function the active alias is closed. That is: the '`.DBF`' file and eventual indexes are closed.

Close all aliases

With this function all Aliases are closed.

Order

Contains a pop-up menu for logical indexes (orders).

Order list rebuild

This function rebuild the indexes opened and associated to the active alias.

Order set focus

This function permits to change the active order selecting form the ones opened and associated to the active alias.

Order list clear

This function closes all orders associated to the active alias.

Relation

Contains a pop-up menu for relations (links with other Aliases).

Set relation

This function permits to establish a relation between a alias and a Child alias showing as a result a unique database.

CHILD	is the alias name to connect to the active alias.
EXPRESSION	is the relation expression that specify the rule for the relation. The value of this expression is the key to access the Child alias: if this Child alias is accessed without index, it must be the record number, if this Child alias is accessed via index, it must be a valid index key.

Clear relation

This function eliminates any relation that originate form the active alias.

RDD default

Contains a pop-up menu for Replaceable Database Driver defaults.

Show actual RDD default

It simply shows the actual Replaceable Database Driver.

Set default RDD

Select a new default Replaceable Database Driver.

Menu Edit

The menu Edit contains functions to access data from the active alias (the actual area).

View

This function permits you to view the active alias with eventual relations as a table.

No edit is allowed.

To navigate the table use the following keys.

[*Enter*]	start field editing.
[*PgUp*]	show previous screen page.
[*PgDn*]	show next screen page.
[*Ctrl PgUp*]	show top of alias.
[*Ctrl PgDn*]	show bottom of file.
[*Ctrl Home*]	show the first column.
[*Ctrl End*]	show last column.

Edit/browse

This function permits you to edit the active alias with eventual relations as a table.

To navigate and edit the table use the following keys.

[*Enter*]	start field editing.
[*PgUp*]	show previous screen page.
[*PgDn*]	show next screen page.
[*Ctrl PgUp*]	show top of alias.
[*Ctrl PgDn*]	show bottom of file.
[*Ctrl Home*]	show the first column.
[*Ctrl End*]	show last column.
[*Ctrl Enter*]	append a new empty record.
[*Ctrl F2*]	copy the current record.
[*Ctrl F3*]	append and paste a record.
[*Ctrl F4*]	paste a previously copied record, overwriting the content of the current one.
[*Ctrl Y*]	delete or recall the current record.
[*Ctrl Del*]	delete or recall the current record.

When a memo field is edited:

[*Esc*]	cancel and close the memo window.
[*Ctrl Y*]	line delete.
[*Ctrl W*]	save and close the memo window.

Replace

The content of a Field of the active alias may be replaced with an expression.

The required data is:

FIELD TO REPLACE	the Field name to be replaced.
NEW VALUE EXPRESSION	the expression that obtain the new value for the selected Field.
WHILE EXPRESSION	the WHILE condition expression: the replacement continue until this expression results True. The constant '`.T.`' is ever True and is the default.
FOR EXPRESSION	the FOR condition expression: the replacement is made for all records that satisfy the condition. The constant '`.T.`' is ever True and is the default.

Recall

The records signed for deletion (deleted but still there), may be recalled (undeleted).

The required data is:

WHILE EXPRESSION	the WHILE condition expression: the record recall continue until this expression results True. The constant '`.T.`' is ever True and is the default.
FOR EXPRESSION	the FOR condition expression: the record recall is made for all records that satisfy the condition. The constant '`.T.`' is ever True and is the default.

Delete

Deletes (sign for deletion) a group of record depending on the required conditions.

The required data is:

WHILE EXPRESSION	the WHILE condition expression: the record deletion continue until this expression results True. The constant '**.T.**' is ever True and is the default.
FOR EXPRESSION	the FOR condition expression: the record deletion is made for all records that satisfy the condition. The constant '**.T.**' is ever True and is the default.

Pack

This function eliminates definitely records previously deleted (signed for deletion).

It may work only if the active alias was opened in exclusive mode.

Menu Report

«

The menu Report contains functions for data report (print). In particular, label files '.LBL' and report file '.RPT' may be created and used for printing. There is also another way to print, with the RPT() system that is available inside the nB internal editor DOC().

DBGOTOP()

Moves the record pointer for the active alias at the first logical record.

New label

With this function can be created a standard label file (.LBL under the dBaseIII standard).

Labels may be printed in more than one column and can contain 16 lines maximum.

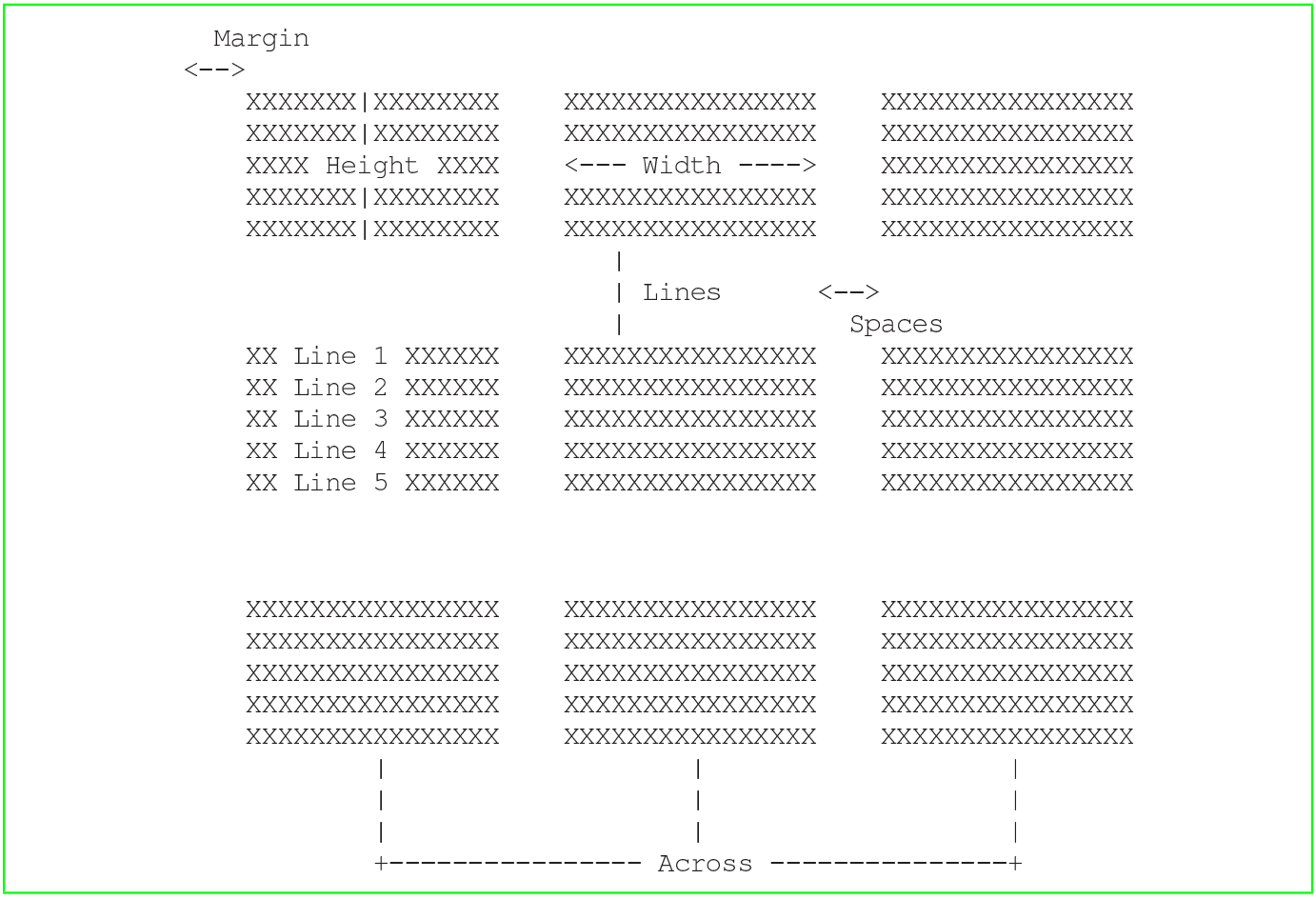
The label data is the following.

REMARK	a label remark that will not be printed.
HEIGHT	the label vertical dimension.
WIDTH	the label horizontal dimension.
MARGIN	the left margin in characters.
LINES	the vertical spacing between labels.
SPACES	the horizontal spacing between labels in characters.
ACROSS	the number of label columns.
LINE 1	The first line inside labels.
LINE n	The n-th line inside labels.
LINE 16	The 16th line inside labels.

The number of lines inside the labels depend on the HEIGHT and the maximum value is 16.

The label lines can contain constant string and/or string expressions.

See the example below.



Modify label

This function permits you to modify a label file.

Label form

This function permits you to print labels with the data provided by the active alias: one label each record.

The following data is required.

LABEL FILENAME	the label filename.
WHILE	the WHILE condition: the label printing goes on as long as this condition remain True.
FOR	the FOR condition: only the records from the active alias that satisfy the condition are used for the label print.

New report

This function permits you to create a standard report form file (.FRM under the dBaseIII standard).

The informations to create a '`.FRM`' file are divided into two parts: the head and groups; the columns.

The first part: head and groups, requires the folliwing informations:

PAGE WIDTH	the page width in characters.
LINES PER PAGE	the usable lines per page.
LEFT MARGIN	the left margin in characters.
DOUBLE SPACED?	double spaced print, yes or no.
PAGE EJECT BEFORE PRINT?	form feed before print, yes or no.
PAGE EJECT AFTER PRINT?	form feed after print, yes or no.
PLAIN PAGE?	plain page, yes or no.
PAGE HEADER	the page header, max 4 lines (the separation between one line and the other is obtained writing a semicolon, ";").
GROUP HEADER	the group title.
GROUP EXPRESSION	the group expression (when it changes, the group changes)
SUMMARY REPORT ONLY?	only totals and no columns, yes or no.
PAGE EJECT AFTER GROUP?	form feed when the group changes, yes or no.
SUB GROUP HEADER	sub group title.
SUB GROUP EXPRESSION	the sub group expression.

The second part: columns, requires the following informations structured in table form:

COLUMN HEADER	column head description (it can contain 4 lines separated with a semicolon).
CONTENT	the column expression.
WIDTH	the column width.
DEC.	the decimal length for numeric columns.
TOTALS	totals to be calculated, yes or no (usefull only for numeric columns).

To navigate and to edit the table use the following keys:

[*Up*]/[*Down*]/[*Left*][*Right*]	move the cursor one position (up, down, left or right);
[*PgUp*]	move to previous screen page;
[*PgDn*]	move to next screen page;
[*Ctrl PgUp*]	move to top of table;
[*Ctrl PgDn*]	move to bottom of table;
[*Ctrl Home*]	move to first column;
[*Ctrl End*]	move to last column;
[*Ctrl Enter*]	append a new empty line;
[*Ctrl F1*]	delete (cut) the current line and save a copy into the "clipboard";
[*Ctrl F2*]	copy current line into the table "clipboard";
[*Ctrl F3*]	insert (paste) the content of the "clipboard" in the current position;
[*Enter*]	start editing in the current position;
[*Esc*]	terminate;
[***x***]	any other key will be written in the current position.

When the editing is terminated, press [*Esc*] and a dialog box will ask for the name to give to the report form file.

Modify report

This function permits you to modify a standard report form file (.FRM under the dBaseIII standard).

Report form

This function permits you to print a report form with the data provided by the active alias.

The following data is required.

REPORT FORM FILE-NAME	the label filename.
WHILE	the WHILE condition: the form printing goes on as long as this condition remain True.
FOR	the FOR condition: only the records from the active alias that satisfy the condition are used for the report form print.

Create/modify/print text

This function activates the text editor.

Menu HTF

The menu Htf helps on creating and accessing the "Help Text Files". This name, help text file, is just the name given to it.

A text (Ascii) file prepared like this manual may be transformed into a "Help Text File" that is a simple text with pointers.

Open help text file

This function permits to open a Help Text File and browse it. The Help Text File name is required.

New help text file

This function permits to create a new "Help Text File" that is a help file under the nB style.

The source is an Ascii file where three kind of information are available: Normal text, Indexes and pointers.

Indexes and Pointers are word or phrases delimited with user defined delimiters; indexes are placed inside the text to indicate an argument, pointers are placed inside the text to indicate a reference to indexes.

Inside this manual, indexes are delimited with ## and ##, so the titles are here indexes; pointers are delimited with < and >.

Only one index per line is allowed, only one pointer per line is allowed.

The Delimiters used do identify indexes and pointers are user defined; the _start_ identifier symbol can be equal to the _end_ identifier symbol. The symbols used for indexes cannot be used for the pointers.

So, the informations required are:

SOURCE TEXT FILENAME	the filename of the text source file.
DESTINATION FILENAME	the filename of the destination Help Text File (suggested '.HLP' extention).
INDEX START CODE	the index start symbol; suggested ##.
INDEX END CODE	the index end symbol; suggested ##.
POINTER START CODE	the pointer start symbol; suggested <.
POINTER END CODE	the pointer end symbol; suggested >.

New HTML file

This function permits to create a new HTML file form a text file formatted to obtain a HTF file.

The informations required are:

SOURCE TEXT FILE-NAME	the filename of the text source file.
DESTINATION FILE-NAME	the filename of the destination Help Text File (suggested '.HLP' extention).
INDEX START CODE	the index start symbol; suggested ##.
INDEX END CODE	the index end symbol; suggested ##.
POINTER START CODE	the pointer start symbol; suggested <.
POINTER END CODE	the pointer end symbol; suggested >.
HTML TITLE	the title for the html page.

Menu Macro

«

The menu Macro helps on creating macros (programs) with a macro recorder, a macro "compiler" and a macro executor.

Start recording

This function simply starts or pause the macro recording. The menu items that end with "&", may be recorded by this macro recorder.

Save recording

A recorded macro may be saved into a ASCII file that may be later modified or simply used as it is. The filename is requested.

Erase recording

While recording or when the macro recorder is paused, it is possible to erase all previous recording with this function.

Edit recording

While recording or when the macro recorder is paused, it is possible to edit all previous recording, for example adding more comments or simply to see what the recorder does.

Macro compilation

A macro file (a program) contained inside a ASCII file, may be compiled into a different file format to speed up execution. The source filename and the destination filename are requested.

Load + execute macro

A macro file (a program) in ASCII form or compiled, may be executed.

A macro file may require some parameters.

This function asks for the macro filename to start and the possible parameter to pass to it.

Menu Info

«

The menu Info is the information menu.

ABOUT	a brief copyright notice.
MANUAL BROWSE	starts the browse of 'NB.HLP' , the nB Help Text File manual if it is present in the current directory or it is found in the PATH (the Dos SET PATH).
[F1] HELP	[F1] reminder.
[F3] ALIAS INFO	[F3] reminder. It shows all the available information on the active alias.
[F5] SET OUTPUT TO	[F5] reminder. It defines the output peripheral or file.

This menu actually appears only inside the DOC() function, the nB text editor.

New

It starts the editing of a new empty text.

Open

It opens for editing a new textfile.

Save

It saves the text file under editing.

Save as

It saves the text file under editing asking for a new name.

Set output to

It permits to change the default output peripheral: the default is the screen.

Print as it is

It prints on the output peripheral the content of the text as it is.

Print with RPT() once

It prints on the output peripheral the content of the text only once replacing possible text variables.

Print with RPT() std

It prints on the output peripheral the content of the text repeating this print for every record contained inside the archive alias.

Exit DOC()

Terminates the use of DOC() the text/document editing/print function.

The text editor DOC()

«

The function Doc() activates a simple text editor usefull to build some simple reports.

Inside this function a menu is available and is activated pressing [*Alt M*] or [*F10*]. The Doc() menu is part of the nB menu system.

DOC() may handle text files of a teorical maximum of 64K.

DOC() may be particularly useful to create formatted text with variables identified by CHR(174) and CHR(175) delimiters: when an active alias exists, [*F2*] gives a list of insertable fields.

[*Esc*]	Exit DOC().
[*F1*]	Call the help.
[*F2*]	Field list.
[*up*] / [*Ctrl E*]	Line up.
[*down*] / [*Ctrl X*]	Line down.
[*left*] / [*Ctrl S*]	Character left.
[*right*] / [*Ctrl D*]	Character right.
[*Ctrl right*] / [*Ctrl A*]	Word left.
[*Ctrl left*] / [*Ctrl F*]	Word right.
[*Home*]	Line start.
[*End*]	Line end.
[*Ctrl Home*]	Top window.
[*Ctrl End*]	Bottom window.
[*PgUp*]	Previous window.
[*PgDn*]	Next window.

[*Ctrl PgUp*]	Document start.
[*Ctrl PgDn*]	End document.
[*Del*]	Delete character (right).
[*Backspace*]	Delete character Left.
[*Tab*]	Insert tab.
[*Ins*]	Toggle insert/overwrite.
[*Enter*]	Next line.
[*Ctrl Y*]	Delete line.
[*Ctrl T*]	Delete word right.
[*F10*] / [*Alt M*]	DOC() menu.

The help text file

nB provides a basic hypertext system to build simple help files. A source text file with "indexes" and "pointers" to indexes is translated into a "help text file" (a '`.DBF`' file); then, this file is browsed by nB.

The source file can have a maximum line width of 80 characters; each line can terminate with CR or CR+LF.

"Indexes" are string delimited by index delimiters (default "##"); "pointers" are string delimited by pointer delimiters (default "<" and ">") and refers to indexes.

Inside a text, indexes must be unique; pointers can be repeated anywhere. A text can contain a maximum of 4000 indexes.

Inside this manual, titles are delimited with "##" as they are indexes; strings delimited with "<" and ">" identify a reference to a title with the same string.

To browse a previously created Help Text File, use the following keys:

[*Esc*]	Exit.
[*UpArrow*]	Move cursor up.
[*DownArrow*]	Move cursor down.
[*PgUp*]	Move cursor PageUp.
[*PgDn*]	Move cursor Pagedown.
[*Ctrl PgUp*]	Move cursor Top.
[*Ctrl PgDn*]	Move cursor Bottom.
[*Enter*]	Select a reference (pointer).
[<−]	Go to previous selected reference (pointer).
[−>]	Go to next selected reference (pointer).
[*Shift F3*]	Search for a new pattern.
[*F3*]	Repeat previous search.

Macro

«

nB can execute (run) macro files. There may be three kind of macro files: ASCII (usually with .& extention); "compiled" (usually with .NB extention); EXE files (compiled with Clipper and linked).

"Compiled" macro files are executed faster then the ASCII source files.

EXE macro files are the fastest.

Macro statements

«

The statements recognised from nB are very similar to Clipper, with some restrictions.

Note that: the FOR statement is not included; there is no function declaration; procedure calls cannot transfer variables; only public variables are allowed.

PROCEDURE

Procedures are the basic building blocks of a nB macro.

Procedures are visible only inside the current macro file.

The procedure structure is as follows:

```
PROCEDURE procedure_name
        statements...
        [RETURN]
        statements...
ENDPROCEDURE
```

A procedure definition begins with a PROCEDURE declaration followed with the ***procedure_name*** and ends with ENDPROCEDURE.

Inside the PROCEDURE - ENDPROCEDURE declaration are placed the executable ***statements*** which are executed when the procedure is called.

Inside the PROCEDURE - ENDPROCEDURE declaration, the RETURN statement may appear. In this case, encountering this RETURN statement, the procedure execution is immediately terminated and control is passed to the statement following the calling one.

The procedure definition do not permit to receive parameters from the calling statement.

DO PROCEDURE

There is only one way to call a procedure:

```
DO PROCEDURE procedure_name
```

When the statement DO PROCEDURE is encountered, the control is passed to the begin of the called PROCEDURE. After the PROCEDURE execution, the control is returned to the statement following DO PROCEDURE.

The procedure call do not permit to send parameters to the procedure.

BEGIN SEQUENCE

The BEGIN SEQUENCE - END structure permits to define a sequence of operation that may be broken.

Inside nB, this control structure is useful only because there is the possibility to break the execution and pass control over the end of it.

This way, encountering BREAK means: "go to end".

```
BEGIN SEQUENCE
      statements...
      [BREAK]
      statements...
END
```

Inside nB, error exception handling is not supported.

DO CASE

This is a control structure where only the statements following a True CASE condition are executed.

When the DO CASE statement is encountered, the following CASE statements are tested. The first time that a condition returns True, the CASE's statements are executed and then control is passed over the END case.

That is: only one CASE is taken into consideration.

If no condition is True, the statements following OTHERWISE are executed.

```
DO CASE
CASE lCondition1
      statements...
[CASE lCondition2]
      statements...
[OTHERWISE]
      statements...
END
```

WHILE

The structure WHILE - END defines a loop based on a condition: the loop is repeated until the condition is True.

The loop execution may be broken with the EXIT statement: it transfer control after the END while.

The LOOP statement may be use to repeat the loop: it transfer the control to the beginning of the loop.

```
WHILE lCondition
      statements...
      [EXIT]
      statements...
      [LOOP]
      statements...
END
```

IF

The IF - END control structure executes a section of code if a specified condition is True. The structure can also specify alternative code to execute if the condition is False.

```
IF lCondition1
      statements...
[ELSE]
      statements...
END
```

Variable declaration

Inside nB, variables are created using a specific function:

```
MEMPUBLIC( "cVarName" )
```

For example,

```
MEMPUBLIC( "Name" )
```

creates the variable Name.

The scope of the created variable is global and there is no way to restrict the visibility of it.

When a variable is no more needed or desired, it can be released:

```
MEMRELEASE( "cVarName" )
```

The variable declaration do not defines the variable type. Every variable may receive any kind of data; that is that the type depends on the type of data contained.

Macro structure

A nB macro must be organised as follow. There may be two situations: Macros with procedures and macros without procedures.

Macro with procedures:

```
PROCEDURE procedure_name_1
        statements...
        [RETURN]
        statements...
ENDPROCEDURE
PROCEDURE procedure_name_2
        statements...
        [RETURN]
        statements...
ENDPROCEDURE
...
...
DO PROCEDURE procedure_name_n
```

Macro without procedures:

```
statements...
statements...
statements...
statements...
statements...
```

nB Macros may be compiled with Clipper. To do so, the first structure example must be changed as follows:

```
#INCLUDE MACRO.CH

DO PROCEDURE procedure_name_nth
…
PROCEDURE procedure_name_1
      statements…
      [RETURN]
      statements…
ENDPROCEDURE
PROCEDURE procedure_name_2
      statements…
      [RETURN]
      statements…
ENDPROCEDURE
…
…
```

To compile a macro with Clipper, the macro file name can be changed into ‘MACRO.PRG’ and

```
RTLINK MACRO.RMK [Enter]
```

should be started.

Macro comments

A nB Macro source file can contain comments. only the "//" comment is recognised! This way: * and /*...*/ will generate errors!

ATTENTION: to simplify the macro interpretation, lines such as this:

```
qqout( "You can’t do that // you can’t do that!" )
```

will generate an error as the interpreter will read only:

```
qqout( "You can’t do that
```

Sorry!

Macro long lines split

Inside a nB macro, long lines may be splitted using ";" (semicolon). Please note that: lines can only be splitted and not joined; a resulting command line cannot be longer then 254 characters.

The macro recorder

Inside the functions ASSIST() and DOC() is available the Macro recorder menu.

When a macro recording is started, a "&" appears on the left side of the status bar. It it blinks, the recording is active, if it is stable, the recording is paused.

The macro recording is not exactly a step-by-step recording of all action taken, but a translation (as good as possible) of what you have done.

The macro recorder is able to record only the menu functions that terminates with the "&" symbol and all what is inserted at the dot command line.

The macro recording can be viewed and edited during the recording. The macro recording can be saved into a text file (a macro file).

Data types

The data types supported in the nB macro language are the same as Clipper:

Array

Character

Code Block

Numeric

Date

Logical

Memo

NIL

Character

«

The character data type identifies character strings of a fixed length. The character set corresponds to: CHR(32) through CHR(255) and the null character, CHR(0).

Valid character strings consist of zero or more characters with a theoretical maximum of 65535 characters. The real maximum dimension depends on the available memory.

Character string constants are formed by enclosing a valid string of characters within a designed pair of delimiters. There are three possible delimiter pairs:

two single quotes like '***'string_constant'***';

two double quotes like '**"*string_constant*"**';

left and right square brackets like '**[***string_constant***]**'.

These three different kind of delimiters are available to resolve some possible problems:

I don't want it -> "I don't want it"

She said, "I love hin" -> 'She said, "I love hin"'

He said, "I don't want it" -> [He said, "I don't want it"]

The following table shows all operations available inside nB for

character data types. These operations act on one or more character expressions and the result is not necessarily a character data type.

+	Concatenate.
-	Concatenate without intervening spaces.
==	Compare for exact equity.
!=, <>, #	Compare for inequity.
<	Compare for sorts before
<=	Compare for sorts before or same as.
>	Compare for sorts after.
>=	Compare for sorts after or same as.
:=	In line assign.
$	Test for substring existence.
ALLTRIM()	Remove leading and trailing spaces.
ASC()	Convert to numeric ASCII code equivalent.
AT()	Locate substring position.
CTOD()	Convert to date.
DESCEND()	Convert to complemented form.
EMPTY()	Test for null or blank string.
ISALPHA()	Test for initial letter.
ISDIGIT()	Test for initial digit.
ISLOWER()	Test for initial lowercase letter.
ISUPPER()	Test for initial uppercase letter.
LEFT()	Extract substring form the left.
LEN()	Compute string length in characters.
LOWER()	Convert letters to lowercase.
LTRIM()	Remove leading spaces.
PADC()	Pad with leading and trailing spaces.
PADL()	Pad with leading spaces.
PADR()	Pad with trailing spaces.
RAT()	Locate substring position starting from the right.

RIGHT()	Extract substring form the right.
RTRIM()	Remove trailing spaces.
SOUNDEX()	Convert to soundex equivalent.
SPACE()	Create a blank string of a defined length.
STRTRAN()	Search and replace substring.
STUFF()	Replace substring.
SUBSTR()	Extract substring.
TRANSFORM()	Convert to formatted string.
UPPER()	Convert letters to uppercase
VAL()	Convert to numeric.
VALTYPE()	Evaluates data type directly.

Memo

«

The memo data type is used to represent variable length character data that can only exist in the form of a database field.

Memo fields are not stored inside the main database file (.DBF) but inside a separate file (.DBT).

A memo field can contain up to 65535 characters, that is the same maximum dimension of character fields. In fact, originally xBases, couldn't have character string longer than 254 characters.

As here memo fields are very similar to long character strings, you may forget that there is a difference.

All the operations that may be applied to character strings, may be used with memo fields; the following functions may be use especially for memo fields or long character strings.

HARDCR()	Replace soft with hard carriage returns.
MEMOEDIT()	Edit contents.

MEMOLINE()	Extract a line of a text.
MEMOREAD()	Read form a disk text file.
MEMOTRAN()	Replace soft and hard carriage returns.
MEMOWRIT()	Write to disk text file.
MLCOUNT()	Count lines.
MLPOS()	Compute position.

Date

«

The date data type is used to represent calendar dates.

Supported dates are from 0100.01.01 to 2999.12.31 and null or blank date.

The appearance of a date is controlled from SETVERB("DATEFORMAT"). The default is "dd/mm/yyyy" and it may easily changed for example with SETVERB("DATEFORMAT", "MM/DD/YYYY") to the US standard.

There is no way to represent date constants; these must be replaced with the CTOD() function. For example if the date 11/11/1995 is to be written, the right way is:

```
CTOD( "11/11/1995" )
```

The character string "11/11/1995" must respect the date format defined as before explained.

The function CTOD() will accept only valid dates, and null dates:

```
CTOD( "" )
```

A null date is ever less than any other valid date.

The following table shows all operations available inside nB for date data types. These operations act on one or more date expressions and the result is not necessarily a character data type.

+	Add a number of days to a date.
-	Subtract days to a date.
==	Compare for equity.
!=, <>, #	Compare for inequity.
<	Compare for earlier
<=	Compare for earlier or same as.
>	Compare for later.
>=	Compare for later or same as.
:=	In line assign.
CDOW()	Compute day of week name.
CMONTH()	Compute month name.
DAY()	Extract day number.
DESCEND()	Convert to complemented form.
DOW()	Compute day of week.
DTOC()	Convert to character string with the format defined with SETVERB("DATEFORMAT").
DOTOS()	Convert to character string in sorting format (YYYYMMDD).
EMPTY()	Test for null date.
MONTH()	Extract month number.
VALTYPE()	Evaluates data type directly.
YEAR()	Extract entire year number, including century.

The numeric data type identifies real number. The theoretical range is form 10^-308 to 10^308 but the numeric precision is guaranteed up to 16 significant digits, and formatting a numeric value for display is guaranteed up to a length of 32 (30 digits, a sign, and a decimal point). That is: numbers longer than 32 bytes may be displayed as asterisks, and digits other then most 16 significant ones are displayed as zeroes.

Numeric constants are written without delimiters. The following are valid constant numbers:

12345

12345.678

-156

+1256.789

-.789

If a numeric constant is delimited like character strings, it becomes a character string.

The following table shows all operations available inside nB for numeric data types. These operations act on one or more numeric expressions and the result is not necessarily a numeric data type.

+	Add or Unary Positive.
-	Subtract or Unary Negative.
*	Multiply.
/	Divide.
%	Modulus.
^, **	Exponentiate.

==	Compare for equity.
!=, <>, #	Compare for inequity.
<	Compare for less than.
>=	Compare for less than or equal.
>	Compare for greater than.
>=	Compare for greater than or equal.
:=	In line assign.
ABS()	Compute absolute value.
CHR()	Convert to ASCII character equivalent.
DESCEND()	Convert to complemented form.
EMPTY()	Test for zero.
EXP()	Exponentiate with e as the base.
INT()	Convert to integer.
LOG()	Compute natural logarithm.
MAX()	Compute maximum.
MIN()	Compute minimum.
ROUND()	Round up or down()
SQRT()	Compute square root.
STR()	Convert to character.
TRANSFORM()	Convert to formatted string.
VALTYPE()	Evaluates data type directly.

Number appearence may be affected by SETVERB("FIXED") and consequently by SETVERB("DECIMALS"). If SETVERB("FIXED") is True, numbers are displayed with a fixed decimal position. The number of decimal positions is defined by SETVERB("DECIMALS"). For that reason, the default is SETVERB("FIXED", .F.) and SETVERB("DECIMALS", 2), that is, no fixed decimal position, but if they will be activated, the default is two decimal digits.

Logical

The logical data type identifies Boolean values.

Logical constants are:

'**.T.**'	True.
'**.F.**'	False.

When editing a logical field, inputs may be:

y, Y, t, T	for True
n, N, f, F	for False

The following table shows all operations available inside nB for logical data types. These operations act on one or more logical expressions and the result is not necessarily a logical data type.

.AND.	And.
.OR.	Or.
.NOT. or !	Negate.
==	Compare for equity.
!=, <>, or #	Compare for inequity.

Comparing two logical values, False ('**.F.**') is always less than True ('**.T.**').

NIL

NIL is not properly a data type, it represent the value of an uninitialised variable.

Inside nB (like what it happens inside Clipper), variables are not declared with the data type that they will contain. This means that a variable can contain any kind of data. In fact, nB variables are

pointer to data and a pointer to "nothing" is NIL.

NIL may be used as constant for assignment or comparing purpose:

```
NIL
```

Fields (database fields) cannot contain NIL.

The following table shows all operations available inside nB for the NIL data type. Except for these operations, attempting to operate on a NIL results in a runtime error.

==	Compare for equity.
!=, <>, #	Compare for inequity.
<	Compare for less than.
<=	Compare for less than or equal.
>	Compare for greater than.
>=	Compare for greater than or equal.
:=	In line assign.
EMPTY()	Test for NIL.
VALTYPE()	Evaluates data type directly.

For the purpose of comparison, NIL is the only value that is equal to NIL. All other values are greater than NIL.

Variables are created inside nB with MEMPUBLIC(). This function creates variables which will be automatically initialised to NIL.

Array

«

The array data type identifies a collection of related data items that share the same name. Each value in an array is referred to as an element.

Array elements can be of any data type except memo (memo is available only inside database fields). For example the first element can

be a character string, the second a number, the third a date and so on. Arrays can contain other arrays and code blocks as elements.

The variable containing the array does not contains the entire array, but the reference to it.

When the NIL type was described, it was cleared that variables doesn't contains real data, but pointer to data. But this happens in a transparent way, that is that when the a variable is assigned to another (for example A := B) the variable receiving the assignment will receive a pointer to a new copy of the source data. This is not the same with arrays: assigning to a variable an array, will assign to that variable a pointer to the same source array and not to a new copy of it.

If arrays are to be duplicated, the ACLONE() function is to be used.

An array constant may be expressed using curly brackets {}. See the examples below.

```
A := { "first_element", "second_element", "third_element" }
```

With this example, the variable A contain the reference to an array with three element containing character string.

```
A[1] == "first_element"

A[2] == "second_element"

A[3] == "third_element"
```

Arrays may contain also no element: empty array and may be expressed as:

```
{}
```

The array element is identified by a number enclosed with square brackets, following the variable name containing the reference to the array. The first array element is one.

If an array contains arrays, we obtain a multidimensional array. For example:

```
A := { { 1, 2 }, { 3, 4 }, { 5, 6 } }
```

is equivalent to the following table.

```
1          2

3          4

5          6
```

With this example, the variable A contain the reference to a bidimensional array containing numbers.

A[1,1] or A[1][1] contains 1

A[1,2] or A[1][2] contains 2

A[2,1] or A[2][1] contains 3

and so on.

As arrays may contain mixed data, it is the user who have to handle correctly the element numbers. For example:

```
A := { "hello", { 3, 4 }, 1234 }

A[1] == "hello"

A[2] == reference to { 3, 4 }

A[3] == 1234
```

A[2,1] or A[2][1] contains 3

A[2,2] or A[2][2] contains 4

A[1,1] is an error!

The following table shows all operations available inside nB for arrays.

:=	In line assign.
AADD()	Add dynamically an element to an array.
ACLONE()	Create a copy of an array.
ACOPY()	Copy element by element an array to another.
ADEL()	Delete one element inside an array.
AFILL()	Fill all array elements with a value.
AINS()	Insert an element inside an array.
ARRAY()	Creates an array of empty elements.
ASCAN()	Scan the array elements.
ASIZE()	Resize an array.
ASORT()	Sort the array elements.
EMPTY()	Test for no elements.
VALTYPE()	Evaluates data type directly.

Code block

«

The code block data type identifies a small piece of executable program code.

A code block is something like a little user defined function where only a sequence of functions or assignments may appear: no loops, no IF ELSE END.

A code block may receive argument and return a value after execution, just like a function.

The syntax is:

```
{ | [argument_list] | exp_list }
```

That is: the ***argument_list*** is optional; the ***exp_list*** may contain one or more expressions separated with a comma.

For example, calling the following code block will give the string "hello world" as result.

```
{ || "hello world" }
```

The following code block require a numeric argument an returns the number passed as argument incremented:

```
{ | n | n+1 }
```

The following code block requires two numeric arguments and returns the sum of the two square radix:

```
{ | nFirst, nSecond | SQRT(nFirst) + SQRT(nSecond) }
```

But code blocks may contains more expressions and the result of the execution of the code block is the result of the last expression.

The following code block executes in sequence some functions and give ever "hello world" as a result.

```
{ | a, b | functionOne(a), functionTwo(b), "hello world" }
```

To start the execution of a code block a function is used: EVAL()

For example, a code block is assigned to a variable and then executed.

```
B := { || "hello world" }
```

EVAL(B) == "hello world"

Another example with a parameter.

```
B := { | n | n+1 }
```

EVAL(B, 1) == 2

Another example with two parameters.

```
B := { | nFirst, nSecond | SQRT(nFirst) + SQRT(nSecond) }
```

EVAL(B, 2, 4) == 20

And so on.

The following table shows some operations available inside nB for code blocks: many functions use code blocks as argument.

:=	In line assign.
AEVAL()	Evaluate (execute) a code block for each element in an array.
BCOMPILE()	Convert (compile) a character string into a code block.
DBEVAL()	Evaluate (execute) a code block for each record in the active alias.
EVAL()	Evaluate a code block once.
VALTYPE()	Evaluates data type directly.

Operators

«

Here is a list with a brief description of the operators available inside nB.

```
cString1 $ cString2
```

Substring comparison.

If ***cString1*** is contained inside ***cString2*** the result is true (‘**.T.**’).

```
nNumber1 % nNumber2
```

Modulus.

The result is the remainder of ***nNumber1*** divided by ***nNuber2***.

```
()
```

Function or grouping indicator.

```
nNumber1 * nNumber2
```

Multiplication.

```
nNumber1 ** nNumber2
nNumber1 ^ nNumber2
```

Exponentiation.

nNumber1 + ***nNumber2***
dDate + ***nNumber***

Addition, unary positive.

cString1 + ***cString2***

String concatenation.

The result is a string beginning with the content of ***cString1*** and following with the content of ***cString2***.

nNumber1 - ***nNumber2***
dDate1 - ***dDate2***
dDate - ***nNumber***

Subtraction, unary negative.

cString1 - ***cString2***

String concatenation.

The result is a string containing ***cString1*** after trimming trailing blanks and ***cString2***.

idAlias->***idField***
`FIELD->`***idVar***
`MEMVAR->`***idVar***

Alias assignment.

The alias operator implicitly SELECTs the ***idAlias*** before evaluating ***idField***. When the evaluation is complete, the original work area is SELECTed again.

```
lCondition1 .AND. lCondition2
```

Logical AND.

```
.NOT. lCondition
```

Logical NOT.

```
lCondition1 .OR. lCondition2
```

Logical OR.

```
nNumber1 / nNumber2
```

Division.

```
object:message[(argument list)]
```

Send.

```
idVar := exp
```

Inline assign.

```
exp1 <= exp2
```

Less than or equal.

```
exp1 <> exp2
```

Not equal.

```
exp1 = exp2
```

Equal.

```
exp1 == exp2
```

Exactly equal.

```
exp1 > exp2
```

Greater than.

```
exp1 >= exp2
```

Greater than or equal.

```
@idVar
```

Pass-by-reference.

```
[]
aArray[nSubscript, ...]
aArray[nSubscript1][nSubscript2] ...
```

Array element indicator.

Delimiters

Here is the delimiter list recognised from nB.

```
{  exp_list  }
```

Literal array delimiters.

```
{  |param_list|  exp_list  }
```

Code block delimiters.

```
"cString"
'cString'
[cString]
```

String delimiters.

Code blocks

A code block is a sequence of function, assignments and constant like the following:

```
sqrt(10)
nResult := 10 * nIndex
```

Suppose that the above sequence of operations has a meaning for you. We want to create a box containing this sequence of operation. This box is contained inside a variable:

```
bBlackBox := { || sqrt(10), nResult := 10 * nIndex }
```

Note the comma used as separator.

Now ***bBlackBox*** contains the small sequence seen before. To execute this sequence, the function EVAL() is used:

```
EVAL(bBlackBox)
```

The execution of the code block gives a result: the value of the last operation contained inside the code block. In this case it is the result of 10****nIndex***. For that reason, if the execution of the code block must give a fixed result, it can terminate with a constant.

A code block may receive parameters working like a function. Try to imagine that we need to do the following.

```
function multiply( nVar1, nVar2 )
     return nVar * nVar2
endfunction
```

A code block that does the same is:

```
bMultiply := { | nVar1, nVar2 | nVar1 * nVar2 }
```

To evaluate it, for example trying to multiply 10 * 5:

```
nResult := EVAL( bMultiply, 10, 5 )
```

and ***nResult*** will contain 50.

Standard functions

With nB all Clipper standard functions may be used. Here follows a short description.

AADD()

Array add

AADD(***aTarget***, ***expValue***) ⇒ ***Value***

aTarget	is the array to add a new element to.
expValue	is the value assigned to the new element.

It increases the actual length of the target array by one. The newly created array element is assigned the value specified by ***expValue***.

ABS()

«

Absolute

```
ABS(nExp) ⇒ nPositive
```

nExp	is the numeric expression to evaluate.

ABS() returns a number representing the absolute value of its argument.

ACLONE()

«

Array clone

```
ACLONE(aSource) ⇒ aDuplicate
```

aSource	is the array to duplicate.

ACLONE() returns a duplicate of ***aSource***.

ACOPY()

Array copy

```
ACOPY(aSource, aTarget,
      [nStart], [nCount], [nTargetPos]) ⇒ aTarget
```

aSource	is the array to copy elements from.
aTarget	is the array to copy elements to.
nStart	is the starting element position in the ***aSource*** array. If not specified, the default value is one.
nCount	is the number of elements to copy from the ***aSource*** array beginning at the ***nStart*** position. If ***nCount*** is not specified, all elements in ***aSource*** beginning with the starting element are copied.
nTargetPos	is the starting element position in the ***aTarget*** array to receive elements from ***aSource***. If not specified, the default value is one.

ACOPY() is an array function that copies elements from the ***aSource*** array to the ***aTarget*** array. The ***aTarget*** array must already exist and be large enough to hold the copied elements.

ADEL()

Array delete

```
ADEL(aTarget, nPosition) ⇒ aTarget
```

aTarget	is the array to delete an element from.
nPosition	is the position of the target array element to delete.

ADEL() is an array function that deletes an element from an array. The contents of the specified array element is lost, and all elements from that position to the end of the array are shifted up one element. The last element in the array becomes NIL.

AEVAL()

«

Array evaluation

```
AEVAL(aArray, bBlock,
      [nStart], [nCount]) ⇒ aArray
```

aArray	is the array to be evaluated.
bBlock	is a code block to execute for each element encountered.
nStart	is the starting element. If not specified, the default is element one.
nCount	is the number of elements to process from ***nStart***. If not specified, the default is all elements to the end of the array.

AEVAL() is an array function that evaluates a code block once for each element of an array, passing the element value and the element index as block parameters. The return value of the block is ignored. All elements in ***aArray*** are processed unless either the ***nStart*** or the ***nCount*** argument is specified.

AFILL()

«

Array fill

```
AFILL(aTarget, expValue,
      [nStart], [nCount]) ⇒ aTarget
```

aTarget	is the array to fill.
expValue	is the value to place in each array element. It can be an expression of any valid data type.
nStart	is the position of the first element to fill. If this argument is omitted, the default value is one.
nCount	is the number of elements to fill starting with element ***nStart***. If this argument is omitted, elements are filled from the starting element position to the end of the array.

AFILL() is an array function that fills the specified array with a single value of any data type (including an array, code block, or NIL) by assigning ***expValue*** to each array element in the specified range.

AINS()

«

Array insert

```
AINS(aTarget, nPosition) ⇒ aTarget
```

aTarget	is the array into which a new element will be inserted.
nPosition	is the position at which the new element will be inserted.

AINS() is an array function that inserts a new element into a specified array. The newly inserted element is NIL data type until a new value is assigned to it. After the insertion, the last element in the array is discarded, and all elements after the new element are shifted down one position.

ALERT()

«

```
ALERT( cMessage , [aOptions] ) ⇒ nChoice
```

cMessage	is the message text displayed, centered, in the alert box. If the message contains one or more semicolons, the text after the semicolons is centered on succeeding lines in the dialog box.
aOptions	defines a list of up to 4 possible responses to the dialog box.

ALERT() returns a numeric value indicating which option was chosen. If the Esc key is pressed, the value returned is zero. The ALERT() function creates a simple modal dialog. The user can respond by moving a highlight bar and pressing the Return or Space-Bar keys, or by pressing the key corresponding to the first letter of the option. If ***aOptions*** is not supplied, a single "Ok" option is presented.

ALIAS()

«

```
ALIAS( [nWorkArea] ) ⇒ cAlias
```

nWorkArea	is any work area number.

ALIAS() returns the alias of the specified work area as a character string. If ***nWorkArea*** is not specified, the alias of the current work area is returned. If there is no database file in USE for the specified work area, ALIAS() returns a null string ("").

ALLTRIM()

«

```
ALLTRIM(cString) ⇒ cTrimmedString
```

cString	is the character expression to trim.

ALLTRIM() returns a character string with leading and trailing spaces removed.

ARRAY()

«

```
ARRAY(nElements [, nElements...]) ⇒ aArray
```

nElements	is the number of elements in the specified dimension.

ARRAY() is an array function that returns an uninitialized array with the specified number of elements and dimensions.

ASC()

«

ASCII

```
ASC(cExp) ⇒ nCode
```

cExp	is the character expression to convert to a number.

ASC() returns an integer numeric value in the range of zero to 255 , representing the ASCII value of ***cExp***.

ASCAN()

«

Array scan

```
ASCAN(aTarget, expSearch,
      [nStart], [nCount]) ⇒ nStoppedAt
```

aTarget	is the array to scan.
expSearch	is either a simple value to scan for, or a code block. If ***expSearch*** is a simple value it can be character, date, logical, or numeric type.
nStart	is the starting element of the scan. If this argument is not specified, the default starting position is one.
nCount	is the number of elements to scan from the starting position. If this argument is not specified, all elements from the starting element to the end of the array are scanned.

ASCAN() returns a numeric value representing the array position of the last element scanned. If ***expSearch*** is a simple value, ASCAN() returns the position of the first matching element, or zero if a match is not found. If ***expSearch*** is a code block, ASCAN() returns the position of the element where the block returned true (‘`.T.`’).

ASIZE()

Array size

```
ASIZE(aTarget, nLength) ⇒ aTarget
```

aTarget	is the array to grow or shrink.
nLength	is the new size of the array.

ASIZE() is an array function that changes the actual length of the ***aTarget*** array. The array is shortened or lengthened to match the specified length. If the array is shortened, elements at the end of the array are lost. If the array is lengthened, new elements are added to the end of the array and assigned NIL.

ASORT()

Array sort

```
ASORT(aTarget, [nStart],
      [nCount], [bOrder]) ⇒ aTarget
```

aTarget	is the array to sort.
nStart	is the first element of the sort. If not specified, the default starting position is one.

nCount	is the number of elements to sort. If not specified, all elements in the array beginning with the starting element are sorted.
bOrder	is an optional code block used to determine sorting order. If not specified, the default order is ascending.

ASORT() is an array function that sorts all or part of an array containing elements of a single data type. Data types that can be sorted include character, date, logical, and numeric. If the ***bOrder*** argument is not specified, the default order is ascending. Each time the block is evaluated, two elements from the target array are passed as block parameters. The block must return true (‘`.T.`’) if the elements are in sorted order.

AT()

«

```
AT(cSearch, cTarget) ⇒ nPosition
```

cSearch	is the character substring for which to search.
cTarget	is the character string to search.

AT() returns the position of the first instance of ***cSearch*** within ***cTarget*** as an integer numeric value. If ***cSearch*** is not found, AT() returns zero.

AT() is a character function used to determine the position of the first occurrence of a character substring within another string.

ATAIL()

Array TAIL

```
ATAIL(aArray) ⇒ Element
```

aArray	is the array.

ATAIL() is an array function that returns the highest numbered element of an array. It can be used in applications as shorthand for ***aArray***[LEN(***aArray***)] when you need to obtain the last element of an array.

BIN2I()

Binary to integer

```
BIN2I(cSignedInt) ⇒ nNumber
```

cSignedInt	is a character string in the form of a 16-bit signed integer number--least significant byte first.

BIN2I() returns an integer obtained converting the first two byte contained inside ***cSignedInt***.

BIN2L()

Binary to long

```
BIN2L(cSignedInt) ⇒ nNumber
```

cSignedInt	is a character string in the form of a 32-bit signed integer number--least significant byte first.

BIN2L() returns an integer obtained from the first tour characters contained in ***cSignedInt***.

BIN2W()

«

Binary to word

```
BIN2W(cUnsignedInt) ⇒ nNumber
```

cUnsignedInt	is a character string in the form of a 16-bit unsigned integer number--least significant byte first.

BIN2W() returns an integer obtained from the first two characters contained in ***cSignedInt***.

BOF()

«

Begin of file

```
BOF() ⇒ lBoundary
```

BOF() returns true (‘**.T.**’) after an attempt to SKIP backward beyond the first logical record in a database file; otherwise, it returns false (‘**.F.**’). If there is no database file open in the current work area, BOF() returns false (‘**.F.**’). If the current database file contains no records, BOF() returns true (‘**.T.**’).

CDOW()

Character day of week

```
CDOW(dExp) ⇒ cDayName
```

dExp	is the date value to convert.

CDOW() returns the name of the day of the week as a character string. The first letter is uppercase and the rest of the string is lowercase. For a null date value, CDOW() returns a null string ("").

CHR()

Character

```
CHR(nCode) ⇒ cChar
```

nCode	is an ASCII code in the range of zero to 255.

CHR() returns a single character value whose ASCII code is specified by ***nCode***.

CMONTH()

Character month

```
CMONTH(dDate) ⇒ cMonth
```

dDate	is the date value to convert.

CMONTH() returns the name of the month as a character string from a date value with the first letter uppercase and the rest of the string lowercase. For a null date value, CMONTH() returns a null string ("").

COL()

«

Column

```
COL() ⇒ nCol
```

COL() is a screen function that returns the current column position of the cursor. The value of COL() changes whenever the cursor position changes on the screen.

COLORSELECT()

«

```
COLORSELECT(nColorIndex) ⇒ NIL
```

nColorIndex	is a number corresponding to the ordinal positions in the current list of color attributes, as set by SETCOLOR().

COLORSELECT() activates the specified color pair from the current list of color attributes (established by SETCOLOR()).

CTOD()

«

Character to date

```
CTOD(cDate) ⇒ dDate
```

cDate	is a character string consisting of numbers representing the month, day, and year separated by any character other than a number. The month, day, and year digits must be specified in accordance with the SET DATE format. If the century digits are not specified, the century is determined by the rules of SET EPOCH.

CTOD() returns a date value. If ***cDate*** is not a valid date, CTOD() returns an empty date.

CURDIR()

Current directory

```
CURDIR( [cDrivespec] ) ⇒ cDirectory
```

cDrivespec	specifies the letter of the disk drive to query. If not specified, the default is the current DOS drive.

CURDIR() returns the current DOS directory of the drive specified by ***cDrivespec*** as a character string without either leading or trailing backslash (\) characters.

DATE()

```
DATE() ⇒ dSystemDate
```

DATE() returns the system date as a date value.

DAY()

«

```
DAY(dDate) ⇒ nDay
```

dDate	is a date value to convert.

DAY() returns the day number from ***dDate***.

DBAPPEND()

«

```
DBAPPEND( [lReleaseRecLocks] ) ⇒ NIL
```

lReleaseRecLocks	is a logical data type that if true (‘**.T.**’), clears all pending record locks, then appends the next record. If ***lReleaseRecLocks*** is false (‘**.F.**’), all pending record locks are maintained and the new record is added to the end of the Lock List. The default value of ***lReleaseRecLocks*** is true (‘**.T.**’).

DBAPPEND() adds a new empty record to the active alias.

DBCLEARFILTER()

«

```
DBCLEARFILTER() ⇒ NIL
```

DBCLEARFILTER() clears the logical filter condition, if any, for the current work area.

DBCLEARINDEX()

«

```
DBCLEARINDEX() ⇒ NIL
```

DBCLEARINDEX() closes any active indexes for the active alias.

DBCLEARRELATION()

«

```
DBCLEARRELATION() ⇒ NIL
```

DBCLEARRELATION() clears any active relations for the active alias.

DBCLOSEALL()

«

```
DBCLOSEALL() ⇒ NIL
```

DBCLOSEALL() releases all occupied work areas from use. It is equivalent to calling DBCLOSEAREA() on every occupied work area.

Attention: DBCLOSEALL() cannot be used inside a "compiled" macro as this will stop the macro execution. In substitution, DBCLOSE() should be used.

DBCLOSEAREA()

«

```
DBCLOSEAREA() ⇒ NIL
```

DBCLOSEAREA() releases the current work area from use.

DBCOMMIT()

«

```
DBCOMMIT() ⇒ NIL
```

DBCOMMIT() causes all updates to the current work area to be written to disk. All updated database and index buffers are written to DOS and a DOS COMMIT request is issued for the database (.dbf) file and any index files associated with the work area. Inside a network environment, DBCOMMIT() makes database updates visible to other processes. To insure data integrity, issue DBCOMMIT() before an UNLOCK operation.

DBCOMMITALL()

«

```
DBCOMMITALL() ⇒ NIL
```

DBCOMMITALL() causes all pending updates to all work areas to be written to disk. It is equivalent to calling DBCOMMIT() for every occupied work area.

DBCREATE()

«

DBCREATE(*cDatabase*, *aStruct*, [*cDriver*]) ⇒ *NIL*

cDatabase	is the name of the new database file, with an optional drive and directory, specified as a character string. If specified without an extension (.dbf) is assumed.
aStruct	is an array that contains the structure of ***cDatabase*** as a series of subarrays, one per field. Each subarray contains the definition of each field's attributes and has the following structure: ***aStruct[n][1] == cName*** ***aStruct[n][2] == cType*** ***aStruct[n][3] == nLength*** ***aStruct[n][4] == nDecimals***
cDriver	specifies the replaceable database driver (RDD) to use to process the current work area. ***cDriver*** is name of the RDD specified as a character expression.

DBCREATE() is a database function that creates a database file from an array containing the structure of the file.

DBCREATEINDEX()

«

DBCREATEINDEX(*cIndexName*, *cKeyExpr*, *bKeyExpr*, [*lUnique*])
⇒ *NIL*

cIndexName	is a character value that specifies the file-name of the index file (order bag) to be created.
cKeyExpr	is a character value that expresses the index key expression in textual form.
bKeyExpr	is a code block that expresses the index key expression in executable form.
lUnique	is an optional logical value that specifies whether a unique index is to be created. If ***lUnique*** is omitted, the current global _SET_UNIQUE setting is used.

DBCREATEINDEX() creates an index for the active alias. If the alias has active indexes, they are closed.

DBDELETE()

«

```
DBDELETE() ⇒ NIL
```

DBDELETE() marks the current record as deleted (*). Records marked for deletion can be filtered using SET DELETED or removed from the file using the PACK command.

DBEVAL()

«

DB evaluate

```
DBEVAL(bBlock,
       [bForCondition],
       [bWhileCondition],
       [nNextRecords],
       [nRecord],
       [lRest]) => NIL
```

bBlock	is a code block to execute for each record processed.
bForCondition	the FOR condition expressed as code block.
bWhileCondition	the WHILE condition expressed as code block.
nNextRecords	is an optional number that specifies the number of records to process starting with the current record. It is the same as the NEXT clause.
nRecord	is an optional record number to process. If this argument is specified, ***bBlock*** will be evaluated for the specified record. This argument is the same as the RECORD clause.
lRest	is an optional logical value that determines whether the scope of DBEVAL() is all records, or, starting with the current record, all records to the end of file.

DBEVAL() is a database function that evaluates a single block for each record within the active alias.

DBFILTER()

«

```
DBFILTER() ⇒ cFilter
```

BFILTER() returns the filter condition defined in the current work area as a character string. If no FILTER has been SET, DBFILTER() returns a null string ("").

DBGOBOTTOM()

«

```
DBGOBOTTOM() ⇒ NIL
```

DBGOBOTTOM() moves to last logical record in the active alias.

DBGOTO()

«

```
DBGOTO(nRecordNumber) ⇒ NIL
```

nRecordNumber	is a numeric value that specifies the record number of the desired record.

DBGOTO() moves to the record whose record number is equal to ***nRecordNumber***. If no such record exists, the work area is positioned to LASTREC() + 1 and both EOF() and BOF() return true (‘**.T.**’).

DBGOTOP()

«

```
DBGOTOP() ⇒ NIL
```

DBGOTOP() moves to the first logical record in the current work area.

DBRECALL()

«

```
DBRECALL() ⇒ NIL
```

DBRECALL() causes the current record to be reinstated if it is marked for deletion.

DBREINDEX()

«

```
DBREINDEX() ⇒ NIL
```

DBREINDEX() rebuilds all active indexes associated with the active alias.

DBRELATION()

«

```
DBRELATION(nRelation) ⇒ cLinkExp
```

nRelation	is the position of the desired relation in the list of active alias relations.

DBRELATION() returns a character string containing the linking

expression of the relation specified by ***nRelation***. If there is no RELATION SET for ***nRelation***, DBRELATION() returns a null string ("").

DBRLOCK()

«

DB record lock

```
DBRLOCK( [nRecNo] ) ⇒ lSuccess
```

nRecNo	is the record number to be locked. The default is the current record.

DBRLOCK() is a database function that locks the record identified by ***nRecNo*** or the current record.

DBRLOCKLIST()

«

```
DBRLOCKLIST() ⇒ aRecordLocks
```

DBRLOCKLIST() returns a one-dimensional array of the locked records in the active alias.

DBRSELECT()

«

DB relation select

```
DBRSELECT(nRelation) ⇒ nWorkArea
```

nRelation	is the position of the desired relation in the list of current work area relations.

DBRSELECT() returns the work area number of the relation specified by ***nRelation*** as an integer numeric value. If there is no RELATION SET for ***nRelation***, DBRSELECT() returns zero.

DBRUNLOCK()

DB relation unlock

```
DBRUNLOCK( [nRecNo] ) ⇒ NIL
```

nRecNo	is the record number to be unlocked. The default is all previously locked records.

DBRUNLOCK() is a database function that unlocks the record identified by ***nRecNo*** or all locked records.

DBSEEK()

```
DBSEEK(expKey, [lSoftSeek]) ⇒ lFound
```

expKey	is a value of any type that specifies the key value associated with the desired record.
lSoftSeek	is an optional logical value that specifies whether a soft seek is to be performed. This determines how the work area is positioned if the specified key value is not found. If ***lSoftSeek*** is omitted, the current global _SET_SOFTSEEK setting is used.

DBSEEK() returns true (‘**.T.**’) if the specified key value was found; otherwise, it returns false (‘**.F.**’).

DBSELECTAREA()

«

```
DBSELECTAREA(nArea | cAlias) ⇒ NIL
```

nArea	is a numeric value between zero and 250, inclusive, that specifies the work area being selected.
cAlias	is a character value that specifies the alias of a currently occupied work area being selected.

DBSELECTAREA() causes the specified work area to become the current work area. All subsequent database operations will apply to this work area unless another work area is explicitly specified for an operation.

DBSETDRIVER()

«

```
DBSETDRIVER([cDriver]) ⇒ cCurrentDriver
```

cDriver	is an optional character value that specifies the name of the database driver that should be used to activate and manage new work areas when no driver is explicitly specified.

DBSETDRIVER() returns the name of the current default driver.

DBSETFILTER()

```
DBSETFILTER(bCondition, [cCondition]) ⇒ NIL
```

bCondition	is a code block that expresses the filter condition in executable form.
cCondition	is a character value that expresses the filter condition in textual form. If ***cCondition*** is omitted, the DBSETFILTER() function will return an empty string for the work area.

DBSETFILTER() sets a logical filter condition for the current work area. When a filter is set, records which do not meet the filter condition are not logically visible. That is, database operations which act on logical records will not consider these records. The filter expression supplied to DBSETFILTER() evaluates to true ('**.T.**') if the current record meets the filter condition; otherwise, it should evaluate to false ('**.F.**').

DBSETINDEX()

```
DBSETINDEX(cOrderBagName) ⇒ NIL
```

cOrderBagName	is a character value that specifies the filename of the index file (index bag) to be opened.

DBSETINDEX() is a database function that adds the contents of an Order Bag into the Order List of the current work area. Any Orders

already associated with the work area continue to be active. If the newly opened Order Bag is the only Order associated with the work area, it becomes the controlling Order; otherwise, the controlling Order remains unchanged. If the Order Bag contains more than one Order, and there are no other Orders associated with the work area, the first Order in the new Order Bag becomes the controlling Order.

DBSETORDER()

«

```
DBSETORDER(nOrderNum) ⇒ NIL
```

nOrderNum	is a numeric value that specifies which of the active indexes is to be the controlling index.

DBSETORDER() controls which of the active alias' active indexes is the controlling index.

DBSETRELATION()

«

```
DBSETRELATION(nArea | cAlias, bExpr, [cExpr]) ⇒ NIL
```

nArea	is a numeric value that specifies the work area number of the child work area.
cAlias	is a character value that specifies the alias of the child work area.
bExpr	is a code block that expresses the relational expression in executable form.

cExpr	is an optional character value that expresses the relational expression in textual form. If ***cExpr*** is omitted, the DBRELATION() function returns an empty string for the relation.

DBSETRELATION() relates the work area specified by ***nArea*** or ***cAlias*** (the child work area), to the current work area (the parent work area). Any existing relations remain active.

DBSKIP()

«

```
DBSKIP( [nRecords] ) ⇒ NIL
```

nRecords	is the number of logical records to move, relative to the current record. A positive value means to skip forward, and a negative value means to skip backward. If ***nRecords*** is omitted, a value of 1 is assumed.

DBSKIP() moves either forward or backward relative to the current record. Attempting to skip forward beyond the last record positions the work area to LASTREC() + 1 and EOF() returns true (‘**.T.**’). Attempting to skip backward beyond the first record positions the work area to the first record and BOF() returns true (‘**.T.**’).

DBSTRUCT()

«

```
DBSTRUCT() ⇒ aStruct
```

DBSTRUCT() returns the structure of the current database file in an array whose length is equal to the number of fields in the database file. Each element of the array is a subarray containing information for one field. The subarrays have the following format:

```
aStruct[n][1] == cName
aStruct[n][2] == cType
aStruct[n][3] == nLength
aStruct[n][4] == nDecimals
```

If there is no database file in USE in the current work area, DBSTRUCT() returns an empty array ({}).

DBUNLOCK()

«

```
DBUNLOCK() ⇒ NIL
```

DBUNLOCK() releases any record or file locks obtained by the current process for the current work area. DBUNLOCK() is only meaningful on a shared database in a network environment.

DBUNLOCKALL()

«

```
DBUNLOCKALL() ⇒ NIL
```

DBUNLOCKALL() releases any record or file locks obtained by the current process for any work area. DBUNLOCKALL() is only meaningful on a shared database in a network environment.

DBUSEAREA()

«

```
DBUSEAREA( [lNewArea], [cDriver], cName, [xcAlias],
      [lShared], [lReadonly] ) ⇒ NIL
```

lNewArea	is an optional logical value. A value of true (‘**.T.**’) selects the lowest numbered unoccupied work area as the current work area before the use operation. If ***lNewArea*** is false (‘**.F.**’) or omitted, the current work area is used; if the work area is occupied, it is closed first.
cDriver	is an optional character value. If present, it specifies the name of the database driver which will service the work area. If ***cDriver*** is omitted, the current default driver is used.
cName	specifies the name of the database (.dbf) file to be opened.
xcAlias	is an optional character value. If present, it specifies the alias to be associated with the work area. The alias must constitute a valid identifier. A valid ***xcAlias*** may be any legal identifier (i.e., it must begin with an alphabetic character and may contain numeric or alphabetic characters and the underscore). If ***xcAlias*** is omitted, a default alias is constructed from ***cName***.

lShared	is an optional logical value. If present, it specifies whether the database (.dbf) file should be accessible to other processes on a network. A value of true (‘**.T.**’) specifies that other processes should be allowed access; a value of false (‘**.F.**’) specifies that the current process is to have exclusive access. If ***lShared*** is omitted, the current global _SET_EXCLUSIVE setting determines whether shared access is allowed.
lReadonly	is an optional logical value that specifies whether updates to the work area are prohibited. A value of true (‘**.T.**’) prohibits updates; a value of false (‘**.F.**’) permits updates. A value of true (‘**.T.**’) also permits read-only access to the specified database (.dbf) file. If ***lReadonly*** is omitted, the default value is false (‘**.F.**’).

DBUSEAREA() opens the specified database (.DBF).

DBDELETE()

«

```
DELETED() ⇒ lDeleted
```

DELETED() returns true (‘**.T.**’) if the current record is marked for deletion; otherwise, it returns false (‘**.F.**’). If there is no database file in USE in the current work area, DELETED() returns false (‘**.F.**’).

DESCEND()

«

```
DESCEND(exp) ⇒ ValueInverted
```

exp	is any valid expression of character, date, logical, or numeric type.

DESCEND() returns an inverted expression of the same data type as the ***exp***, except for dates which return a numeric value. A DESCEND() of CHR(0) always returns CHR(0).

DEVOUT()

«

Device output

```
DEVOUT(exp, [cColorString]) ⇒ NIL
```

exp	is the value to display.
cColorString	is an optional argument that defines the display color of ***exp***.

DEVOUT() is a full-screen display function that writes the value of a single expression to the current device at the current cursor or printhead position.

DEVOUTPICT()

«

Device output picture

```
DEVOUTPICT(exp, cPicture, [cColorString]) ⇒ NIL
```

exp	is the value to display.
cPicture	defines the formatting control for the display of ***exp***.
cColorString	is an optional argument that defines the display color of ***exp***.

DEVOUTPICT() is a full-screen display function that writes the value of a single expression to the current device at the current cursor or printhead position.

DEVPOS()

«

Device position

```
DEVPOS(nRow, nCol) ⇒ NIL
```

nRow, ***nCol***	are the new row and column positions of the cursor or printhead.

DEVPOS() is an environment function that moves the screen or printhead depending on the current DEVICE.

DIRECTORY()

«

```
DIRECTORY(cDirSpec, [cAttributes]) ⇒ aDirectory
```

cDirSpec	identifies the drive, directory and file specification for the directory search. Wildcards are allowed in the file specification. If ***cDirSpec*** is omitted, the default value is *.*.

cAttributes	specifies inclusion of files with special attributes in the returned information. ***cAttributes*** is a string containing one or more of the following characters: H Include hidden files S Include system files D Include directories V Search for the DOS volume label only Normal files are always included in the search, unless you specify V.

DIRECTORY() returns an array of subarrays, with each subarray containing information about each file matching ***cDirSpec***. The subarray has the following structure:

```
aDirectory[n][1] == cName
aDirectory[n][2] == cSize
aDirectory[n][3] == dDate
aDirectory[n][4] == cTime
aDirectory[n][5] == cAttributes
```

If no files are found matching ***cDirSpec*** or if ***cDirSpec*** is an illegal path or file specification, DIRECTORY() returns an empty ({}) array.

DISKSPACE()

«

```
DISKSPACE( [nDrive] ) ⇒ nBytes
```

nDrive	is the number of the drive to query, where one is drive A, two is B, three is C, etc. The default is the current DOS drive if ***nDrive*** is omitted or specified as zero.

DISKSPACE() returns the number of bytes of empty space on the specified disk drive as an integer numeric value.

DISPBOX()

«

Display box

```
DISPBOX(nTop, nLeft, nBottom, nRight,
      [cnBoxString], [cColorString]) ⇒ NIL
```

nTop, ***nLeft***, ***nBottom***, ***nRight***	define the coordinates of the box.
cnBoxString	is a numeric or character expression that defines the border characters of the box. If specified as a numeric expression, a value of 1 displays a single-line box and a value of 2 displays a double-line box. All other numeric values display a single-line box. If ***cnBoxString*** is a character expression, it specifies the characters to be used in drawing the box. This is a string of eight border characters and a fill character.
cColorString	defines the display color of the box that is drawn.

DISPBOX() is a screen function that draws a box at the specified display coordinates in the specified color.

DISPOUT()

«

Display out

```
DISPOUT(exp, [cColorString]) ⇒ NIL
```

exp	is the value to display.
cColorString	is an optional argument that defines the display color of ***exp***.
cColorString	is a character expression containing the standard color setting.

DISPOUT() is a simple output function that writes the value of a single expression to the display at the current cursor position. This function ignores the SET DEVICE setting; output always goes to the screen.

DOW()

«

Day of week

```
DOW(dDate) ⇒ nDay
```

dDate	is a date value to convert.

DOW() returns the day of the week as a number between zero and seven. The first day of the week is one (Sunday) and the last day is seven (Saturday). If ***dDate*** is empty, DOW() returns zero.

DTOC()

«

Date to character

```
DTOC(dDate) ⇒ cDate
```

dDate	is the date value to convert.

DTOC() returns a character string representation of a date value. The return value is formatted in the current date format. A null date returns a string of spaces equal in length to the current date format.

DTOS()

«

Date to sort

```
DTOS(dDate) ⇒ cDate
```

dDate	is the date value to convert.

DTOS() returns a character string eight characters long in the form, yyyymmdd. When ***dDate*** is a null date (CTOD("")), DTOS() returns a string of eight spaces.

EMPTY()

«

```
EMPTY(exp) ⇒ lEmpty
```

exp	is an expression of any data type.

EMPTY() returns true (‘**.T.**’) if the expression results in an empty value; otherwise, it returns false (‘**.F.**’):

Array	{}
Character/Memo	Spaces, tabs, CR/LF, or ""
Numeric	0
Date	CTOD("")
Logical	‘**.F.**’
NIL	NIL

EOF()

End of file

```
EOF() ⇒ lBoundary
```

EOF() returns true (‘**.T.**’) when an attempt is made to move the record pointer beyond the last logical record in a database file; otherwise, it returns false (‘**.F.**’). If there is no database file open in the current work area, EOF() returns false (‘**.F.**’). If the current database file contains no records, EOF() returns true (‘**.T.**’).

EVAL()

Code block evaluation

```
EVAL(bBlock, [BlockArg_list]) ⇒ LastBlockValue
```

bBlock	is the code block to evaluate.
BlockArg_list	is a list of arguments to send to the code block before it is evaluated.

To execute or evaluate a code block, call EVAL() with the block value and any parameters. The parameters are supplied to the block when it is executed. Code blocks may be a series of expressions separated by commas. When a code block is evaluated, the returned value is the value of the last expression in the block.

EXP()

«

Exponent

```
EXP(nExponent) ⇒ nAntilogarithm
```

nExponent	is the natural logarithm for which a numeric value is to be calculated.

EXP() returns a numeric value that is equivalent to the value e raised to the specified power.

FCLOSE()

«

File close

```
FCLOSE(nHandle) ⇒ lError
```

nHandle	is the file handle obtained previously from FOPEN() or FCREATE().

FCLOSE() is a low-level file function that closes binary files and forces the associated DOS buffers to be written to disk. If the operation fails, FCLOSE() returns false (‘**.F.**’). FERROR() can then be used to determine the reason for the failure.

FCOUNT()

Field count

```
FCOUNT() ⇒ nFields
```

FCOUNT() returns the number of fields in the database file in the active alias as an integer numeric value. If there is no database file open, FCOUNT() returns zero.

FCREATE()

Field create

```
FCREATE(cFile, [nAttribute]) ⇒ nHandle
```

cFile	is the name of the file to create. If the file already exists, its length is truncated to zero without warning.
nAttribute	is the binary file attribute, the default value is zero. ***nAttribute*** = 0 Normal (default) ***nAttribute*** = 1 Read-only ***nAttribute*** = 2 Hidden ***nAttribute*** = 4 System

FCREATE() returns the DOS file handle number of the new binary file in the range of zero to 65,535. If an error occurs, FCREATE() returns -1 and FERROR() is set to indicate an error code.

FERASE()

«

File erase

```
FERASE(cFile) ⇒ nSuccess
```

cFile	is the name (with or without path) of the file to be deleted from disk.

FERASE() is a file function that deletes a specified file from disk. FERASE() returns -1 if the operation fails and zero if it succeeds.

FERROR()

«

File error

```
FERROR() ⇒ nErrorCode
```

FERROR() returns the DOS error from the last file operation as an integer numeric value. If there is no error, FERROR() returns zero.

nErrorCode value	Meaning
0	Successful
2	File not found
3	Path not found
4	Too many files open
5	Access denied
6	Invalid handle
8	Insufficient memory
15	Invalid drive specified
19	Attempted to write to a write-protected disk
21	Drive not ready

nErrorCode value	Meaning
23	Data CRC error
29	Write fault
30	Read fault
32	Sharing violation
33	Lock Violation

FERROR() is a low-level file function that indicates a DOS error after a file function is used.

FIELDBLOCK()

«

```
FIELDBLOCK(cFieldName) ⇒ bFieldBlock
```

cFieldName	is the name of the field to which the set-get block will refer.

FIELDBLOCK() returns a code block that, when evaluated, sets (assigns) or gets (retrieves) the value of the given field. If ***cFieldName*** does not exist in the current work area, FIELDBLOCK() returns NIL.

FIELDGET()

«

```
FIELDGET(nField) ⇒ ValueField
```

nField	is the ordinal position of the field in the record structure for the current work area.

FIELDGET() returns the value of the specified field. If ***nField*** does

not correspond to the position of any field in the current database file, FIELDGET() returns NIL.

FIELDNAME()

«

```
FIELDNAME(nPosition) ⇒ cFieldName
```

nPosition	is the position of a field in the database file structure.

FIELDNAME() returns the name of the specified field as a character string. If ***nPosition*** does not correspond to an existing field in the current database file or if no database file is open in the current work area, FIELDNAME() returns a null string ("").

FIELDPOS()

«

Field position

```
FIELDPOS(cFieldName) ⇒ nFieldPos
```

cFieldName	is the name of a field in the current or specified work area.

FIELDPOS() returns the position of the specified field within the list of fields associated with the current or specified work area. If the current work area has no field with the specified name, FIELDPOS() returns zero.

FIELDPUT()

«

```
FIELDPUT(nField, expAssign) ⇒ ValueAssigned
```

nField	is the ordinal position of the field in the current database file.
expAssign	is the value to assign to the given field. The data type of this expression must match the data type of the designated field variable.

FIELDPUT() is a database function that assigns ***expAssign*** to the field at ordinal position ***nField*** in the current work area. This function allows you to set the value of a field using its position within the database file structure rather than its field name.

FIELDWBLOCK()

«

Field work area block

```
FIELDWBLOCK(cFieldName, nWorkArea) ⇒ bFieldWBlock
```

cFieldName	is the name of the field specified as a character string.
nWorkArea	is the work area number where the field resides specified as a numeric value.

FIELDWBLOCK() returns a code block that, when evaluated, sets (assigns) or gets (retrieves) the value of ***cFieldName*** in the work area designated by ***nWorkArea***. If ***cFieldName*** does not exist in the specified work area, FIELDWBLOCK() returns NIL.

FILE()

«

```
FILE(cFilespec) ⇒ lExists
```

cFilespec	is in the current default directory and path. It is a standard file specification that can include the wildcard characters * and ? as well as a drive and path reference.

FILE() returns true (‘**.T.**’) if there is a match for any file matching the ***cFilespec*** pattern; otherwise, it returns false (‘**.F.**’).

FLOCK()

«

File lock

```
FLOCK() ⇒ lSuccess
```

FLOCK() tries to lock the active alias and returns true (‘**.T.**’) if it succeeds; otherwise, it returns false (‘**.F.**’).

FOPEN()

«

File open

```
FOPEN(cFile, [nMode]) ⇒ nHandle
```

cFile	is the name of the file to open including the path if there is one.

nMode	is the requested DOS open mode indicating how the opened file is to be accessed. The open mode is composed of the sum of two elements: the Open mode and the Sharing mode. Open mode: 0 Open for reading (default) 1 Open for writing 2 Open for reading or writing Sharing mode: 0 Compatibility mode (default) 16 Exclusive use 32 Prevent others from writing 48 Prevent others from reading 64 Allow others to read or write

FOPEN() returns the file handle of the opened file in the range of zero to 65,535. If an error occurs, FOPEN() returns -1.

FOUND()

```
FOUND() ⇒ lSuccess
```

FOUND() returns true (‘**.T.**’) if the last search command was successful; otherwise, it returns false (‘**.F.**’).

FREAD()

File read

```
FREAD(nHandle, @cBufferVar, nBytes) ⇒ nBytes
```

nHandle	is the file handle obtained from FOPEN(), FCREATE(), or predefined by DOS.
cBufferVar	is the name of an existing and initialized character variable used to store data read from the specified file. The length of this variable must be greater than or equal to ***nBytes***. ***cBufferVar*** must be passed by reference and, therefore, must be prefaced by the pass-by-reference operator (@).
nBytes	is the number of bytes to read into the buffer.

FREAD() tries to read ***nBytes*** of the binary file ***nHandle*** inside ***cBufferVar***. It returns the number of bytes successfully read as an integer numeric value. A return value less than ***nBytes*** or zero indicates end of file or some other read error.

FREADSTR()

«

File read string

```
FREADSTR(nHandle, nBytes) ⇒ cString
```

nHandle	is the file handle obtained from FOPEN(), FCREATE(), or predefined by DOS.
nBytes	is the number of bytes to read, beginning at the current DOS file pointer position.

FREADSTR() returns a character string up to 65,535 (64K) bytes. A null return value ("") indicates an error or end of file. FREADSTR() is a low-level file function that reads characters from an open binary

file beginning with the current DOS file pointer position. Characters are read up to ***nBytes*** or until a null character (CHR(0)) is encountered. All characters are read including control characters except for CHR(0). The file pointer is then moved forward ***nBytes***. If ***nBytes*** is greater than the number of bytes from the pointer position to the end of the file, the file pointer is positioned to the last byte in the file.

FRENAME()

File rename

```
FRENAME(cOldFile, cNewFile) ⇒ nSuccess
```

cOldFile	is the name of the file to rename, including the file extension. A drive letter and/or path name may also be included as part of the filename.
cNewFile	is the new name of the file, including the file extension. A drive letter and/or path name may also be included as part of the name.

FRENAME() returns -1 if the operation fails and zero if it succeeds.

FSEEK()

File seek

```
FSEEK(nHandle, nOffset, [nOrigin]) ⇒ nPosition
```

nHandle	is the file handle obtained from FOPEN(), FCREATE(), or predefined by DOS.

nOffset	is the number of bytes to move the file pointer from the position defined by ***nOrigin***. It can be a positive or negative number. A positive number moves the pointer forward, and a negative number moves the pointer backward in the file.
nOrigin	defines the starting location of the file pointer before FSEEK() is executed. The default value is zero, representing the beginning of file. If ***nOrigin*** is the end of file, ***nOffset*** must be zero or negative.
nOrigin == 0	Seek from beginning of file
nOrigin == 1	Seek from the current pointer position
nOrigin == 2	Seek from end of file

FSEEK() returns the new position of the file pointer relative to the beginning of file (position 0) as an integer numeric value. This value is without regard to the original position of the file pointer. FSEEK() is a low-level file function that moves the file pointer forward or backward in an open binary file without actually reading the contents of the specified file. The beginning position and offset are specified as function arguments, and the new file position is returned.

FWRITE()

«

File write

`FWRITE(`***nHandle***`,` ***cBuffer***`,` `[`***nBytes***`]``)` ⇒ ***nBytesWritten***

nHandle	is the file handle obtained from FOPEN(), FCREATE(), or predefined by DOS.

cBuffer	is the character string to write to the specified file.
nBytes	indicates the number of bytes to write beginning at the current file pointer position. If omitted, the entire content of ***cBuffer*** is written.

FWRITE() returns the number of bytes written as an integer numeric value. If the value returned is equal to ***nBytes***, the operation was successful. If the return value is less than ***nBytes*** or zero, either the disk is full or another error has occurred.

GETENV()

Get environment

```
GETENV(cEnvironmentVariable) ⇒ cString
```

cEnvironmentVariable	is the name of the DOS environment variable. When specifying this argument, you can use any combination of upper and lowercase letters; GETENV() is not case- sensitive.

GETENV() returns the contents of the specified DOS environment variable as a character string. If the variable cannot be found, GETENV() returns a null string ("").

HARDCR()

«

Hard carriage return

```
HARDCR(cString) ⇒ cConvertedString
```

cString	is the character string or memo field to convert.

HARDCR() is a memo function that replaces all soft carriage returns (CHR(141)) with hard carriage returns (CHR(13)). It is used to display long character strings and memo fields containing soft carriage returns with console commands.

HEADER()

«

```
HEADER() ⇒ nBytes
```

HEADER() returns the number of bytes in the header of the current database file as an integer numeric value. If no database file is in use, HEADER() returns a zero (0).

I2BIN()

«

Integer to binary

```
I2BIN(nInteger) ⇒ cBinaryInteger
```

nInteger	is an integer numeric value to convert. Decimal digits are truncated.

I2BIN() returns a two-byte character string containing a 16-bit binary integer.

IF()

«

```
[I] IF(lCondition, expTrue, expFalse) ⇒ Value
```

lCondition	is a logical expression to be evaluated.
expTrue	is the value, a condition-expression, of any data type, returned if ***lCondition*** is true (‘**.T.**’).
expFalse	is the value, of any date type, returned if ***lCondition*** is false (‘**.F.**’). This argument need not be the same data type as ***expTrue***.

IF() returns the evaluation of ***expTrue*** if ***lCondition*** evaluates to true (‘**.T.**’) and ***expFalse*** if it evaluates to false (‘**.F.**’).

INDEXEXT()

«

Index extention

```
INDEXEXT() ⇒ cExtension
```

INDEXEXT() returns the default index file extension by determining which database driver is currently linked.

INDEXKEY()

«

```
INDEXKEY(nOrder) ⇒ cKeyExp
```

nOrder	is the ordinal position of the index in the list of index files opened by the last USE...INDEX or SET INDEX TO command for the current work area. A zero value specifies the controlling index, without regard to its actual position in the list.

INDEXKEY() returns the key expression of the specified index as a character string. If there is no corresponding index or if no database file is open, INDEXKEY() returns a null string ("").

INDEXORD()

«

Index order

```
INDEXORD() ⇒ nOrder
```

INDEXORD() returns an integer numeric value. The value returned is equal to the position of the controlling index in the list of open indexes for the current work area. A value of zero indicates that there is no controlling index and records are being accessed in natural order. If no database file is open, INDEXORD() will also return a zero.

INKEY()

Input key

```
INKEY( [nSeconds] ) ⇒ nInkeyCode
```

nSeconds	specifies the number of seconds INKEY() waits for a keypress. You can specify the value in increments as small as one-tenth of a second. Specifying zero halts the program until a key is pressed. If ***nSeconds*** is omitted, INKEY() does not wait for a keypress.

INKEY() returns an integer numeric value from -39 to 386, identifying the key extracted from the keyboard buffer. If the keyboard buffer is empty, INKEY() returns zero. INKEY() returns values for all ASCII characters, function, Alt+function, Ctrl+function, Alt+letter, and Ctrl+letter key combinations.

nInkeyCode value	Key or key combination
5	[*Up arrow*], [*Ctrl*]+[*E*]
24	[*Down arrow*], [*Ctrl*]+[*X*]
19	[*Left arrow*], [*Ctrl*]+[*S*]
4	[*Right arrow*], [*Ctrl*]+[*D*]
1	[*Home*], [*Ctrl*]+[*A*]
6	[*End*], [*Ctrl*]+[*F*]
18	[*PgUp*], [*Ctrl*]+[*R*]
3	[*PgDn*], [*Ctrl*]+[*C*]
397	[*Ctrl*]+[*Up arrow*]
401	[*Ctrl*]+[*Down arrow*]

nInkeyCode value	Key or key combination
26	[*Ctrl*]+[*Left arrow*], [*Ctrl*]+[*Z*]
2	[*Ctrl*]+[*Right arrow*], [*Ctrl*]+[*B*]
29	[*Ctrl*]+[*Home*]
23	[*Ctrl*]+[*End*], [*Ctrl*]+[*W*]
31	[*Ctrl*]+[*PgUp*], [*Ctrl*]+[*Hyphen*]
30	[*Ctrl*]+[*PgDn*], [*Ctrl*]+[^]
408	[*Alt*]+[*Up arrow*]
416	[*Alt*]+[*Down arrow*]
411	[*Alt*]+[*Left arrow*]
413	[*Alt*]+[*Right arrow*]
407	[*Alt*]+[*Home*]
415	[*Alt*]+[*End*]
409	[*Alt*]+[*PgUp*]
417	[*Alt*]+[*PgDn*]
13	[*Enter*], [*Ctrl*]+[*M*]
32	[*Space bar*]
27	[*Esc*]
10	[*Ctrl*]+[*Enter*]
379	[*Ctrl*]+[*Print Screen*]
309	[*Ctrl*]+[*?*]
284	[*Alt*]+[*Enter*]
387	[*Alt*]+[*Equals*]
257	[*Alt*]+[*Esc*]
422	Keypad [*Alt*]+[*Enter*]
399	Keypad [*Ctrl*]+[*5*]
405	Keypad [*Ctrl*]+[*/*]
406	Keypad [*Ctrl*]+[*]
398	Keypad [*Ctrl*]+[−]
400	Keypad [*Ctrl*]+[+]

nInkeyCode value	Key or key combination
5	Keypad [*Alt*]+[*5*]
420	Keypad [*Alt*]+[*/*]
311	Keypad [*Alt*]+[*]
330	Keypad [*Alt*]+[−]
334	Keypad [*Alt*]+[+]
22	[*Ins*], [*Ctrl*]+[*V*]
7	[*Del*], [*Ctrl*]+[*G*]
8	[*Backspace*], [*Ctrl*]+[*H*]
9	[*Tab*], [*Ctrl*]+[*I*]
271	[*Shift*]+[*Tab*]
402	[*Ctrl*]+[*Ins*]
403	[*Ctrl*]+[*Del*]
127	[*Ctrl*]+[*Backspace*]
404	[*Ctrl*]+[*Tab*]
418	[*Alt*]+[*Ins*]
419	[*Alt*]+[*Del*]
270	[*Alt*]+[*Backspace*]
421	[*Alt*]+[*Tab*]
1	[*Ctrl*]+[*A*], [*Home*]
2	[*Ctrl*]+[*B*], [*Ctrl*]+[*Right arrow*]
3	[*Ctrl*]+[*C*], [*PgDn*], [*Ctrl*]+[*ScrollLock*]
4	[*Ctrl*]+[*D*], [*Right arrow*]
5	[*Ctrl*]+[*E*], [*Up arrow*]
6	[*Ctrl*]+[*F*], [*End*]
7	[*Ctrl*]+[*G*], [*Del*]
8	[*Ctrl*]+[*H*], [*Backspace*]
9	[*Ctrl*]+[*I*], [*Tab*]
10	[*Ctrl*]+[*J*]
11	[*Ctrl*]+[*K*]

nInkeyCode value	Key or key combination
12	[*Ctrl*]+[*L*]
13	[*Ctrl*]+[*M*], [*Return*]
14	[*Ctrl*]+[*N*]
15	[*Ctrl*]+[*O*]
16	[*Ctrl*]+[*P*]
17	[*Ctrl*]+[*Q*]
18	[*Ctrl*]+[*R*], [*PgUp*]
19	[*Ctrl*]+[*S*], [*Left arrow*]
20	[*Ctrl*]+[*T*]
21	[*Ctrl*]+[*U*]
22	[*Ctrl*]+[*V*], [*Ins*]
23	[*Ctrl*]+[*W*], [*Ctrl*]+[*End*]
24	[*Ctrl*]+[*X*], [*Down arrow*]
25	[*Ctrl*]+[*Y*]
26	[*Ctrl*]+[*Z*], [*Ctrl*]+[*Left arrow*]
286	[*Alt*]+[*A*]
304	[*Alt*]+[*B*]
302	[*Alt*]+[*C*]
288	[*Alt*]+[*D*]
274	[*Alt*]+[*E*]
289	[*Alt*]+[*F*]
290	[*Alt*]+[*G*]
291	[*Alt*]+[*H*]
279	[*Alt*]+[*I*]
292	[*Alt*]+[*J*]
293	[*Alt*]+[*K*]
294	[*Alt*]+[*L*]
306	[*Alt*]+[*M*]
305	[*Alt*]+[*N*]

nInkeyCode value	Key or key combination
280	[*Alt*]+[*O*]
281	[*Alt*]+[*P*]
272	[*Alt*]+[*Q*]
275	[*Alt*]+[*R*]
287	[*Alt*]+[*S*]
276	[*Alt*]+[*T*]
278	[*Alt*]+[*U*]
303	[*Alt*]+[*V*]
273	[*Alt*]+[*W*]
301	[*Alt*]+[*X*]
277	[*Alt*]+[*Y*]
300	[*Alt*]+[*Z*]
376	[*Alt*]+[*1*]
377	[*Alt*]+[*2*]
378	[*Alt*]+[*3*]
379	[*Alt*]+[*4*]
380	[*Alt*]+[*5*]
381	[*Alt*]+[*6*]
382	[*Alt*]+[*7*]
383	[*Alt*]+[*8*]
384	[*Alt*]+[*9*]
385	[*Alt*]+[*0*]
28	[*F1*], [*Ctrl*]+[*Backslash*]
-1	[*F2*]
-2	[*F3*]
-3	[*F4*]
-4	[*F5*]
-5	[*F6*]
-6	[*F7*]

nInkeyCode value	Key or key combination
-7	[*F8*]
-8	[*F9*]
-9	[*F10*]
-40	[*F11*]
-41	[*F12*]
-20	[*Ctrl*]+[*F1*]
-21	[*Ctrl*]+[*F2*]
-22	[*Ctrl*]+[*F4*]
-23	[*Ctrl*]+[*F3*]
-24	[*Ctrl*]+[*F5*]
-25	[*Ctrl*]+[*F6*]
-26	[*Ctrl*]+[*F7*]
-27	[*Ctrl*]+[*F8*]
-28	[*Ctrl*]+[*F9*]
-29	[*Ctrl*]+[*F10*]
-44	[*Ctrl*]+[*F11*]
-45	[*Ctrl*]+[*F12*]
-30	[*Alt*]+[*F1*]
-31	[*Alt*]+[*F2*]
-32	[*Alt*]+[*F3*]
-33	[*Alt*]+[*F4*]
-34	[*Alt*]+[*F5*]
-35	[*Alt*]+[*F6*]
-36	[*Alt*]+[*F7*]
-37	[*Alt*]+[*F8*]
-38	[*Alt*]+[*F9*]
-39	[*Alt*]+[*F10*]
-46	[*Alt*]+[*F11*]
-47	[*Alt*]+[*F12*]

nInkeyCode value	Key or key combination
-10	[*Shift*]+[*F1*]
-11	[*Shift*]+[*F2*]
-12	[*Shift*]+[*F3*]
-13	[*Shift*]+[*F4*]
-14	[*Shift*]+[*F5*]
-15	[*Shift*]+[*F6*]
-16	[*Shift*]+[*F7*]
-17	[*Shift*]+[*F8*]
-18	[*Shift*]+[*F9*]
-19	[*Shift*]+[*F10*]
-42	[*Shift*]+[*F11*]
-43	[*Shift*]+[*F12*]

INT()

Integer

```
INT(nExp) ⇒ nInteger
```

nExp	is a numeric expression to convert to an integer.

INT() is a numeric function that converts a numeric value to an integer by truncating all digits to the right of the decimal point. INT() is useful in operations where the decimal portion of a number is not needed.

ISALPHA()

«

```
ISALPHA(cString) ⇒ lBoolean
```

cString	is the character string to examine.

ISALPHA() returns true (‘**.T.**’) if the first character in ***cString*** is alphabetic; otherwise, it returns false (‘**.F.**’).

ISCOLOR()

«

```
ISCOLOR() | ISCOLOUR() ⇒ lBoolean
```

ISCOLOR() returns true (‘**.T.**’) if there is a color graphics card installed; otherwise, it returns false (‘**.F.**’).

ISDIGIT()

«

```
ISDIGIT(cString) ⇒ lBoolean
```

cString	is the character string to examine.

ISDIGIT() returns true (‘**.T.**’) if the first character of the character string is a digit between zero and nine; otherwise, it returns false (‘**.F.**’).

ISLOWER()

«

```
ISLOWER(cString) ⇒ lBoolean
```

cString	is the character string to examine.

ISLOWER() returns true (‘**.T.**’) if the first character of the character string is a lowercase letter; otherwise, it returns false (‘**.F.**’).

ISPRINTER()

«

```
ISPRINTER() ⇒ lReady
```

ISPRINTER() returns true (‘**.T.**’) if ‘LPT1:’ is ready; otherwise, it returns false (‘**.F.**’).

ISUPPER()

«

```
ISUPPER(cString) ⇒ lBoolean
```

cString	is the character string to examine.

ISUPPER() returns true (‘**.T.**’) if the first character is an uppercase letter; otherwise, it returns false (‘**.F.**’).

L2BIN()

«

Long to binary

```
L2BIN(nExp) ⇒ cBinaryInteger
```

nExp	is the numeric value to convert. Decimal digits are truncated.

L2BIN() returns a four-byte character string formatted as a 32- bit binary integer.

LASTKEY()

«

```
LASTKEY() ⇒ nInkeyCode
```

LASTKEY() is a keyboard function that reports the INKEY() value of the last key fetched from the keyboard buffer by the INKEY() function, or a wait state. LASTKEY() retains its current value until another key is fetched from the keyboard buffer.

LASTREC()

«

Last record

```
LASTREC() ⇒ nRecords
```

LASTREC() returns the number of physical records in the active alias as an integer numeric value.

LEFT()

«

```
LEFT(cString, nCount) ⇒ cSubString
```

cString	is a character string from which to extract characters.
nCount	is the number of characters to extract.

LEFT() returns the leftmost ***nCount*** characters of ***cString*** as a character string. If ***nCount*** is negative or zero, LEFT() returns a null string (""). If ***nCount*** is larger than the length of the character string, LEFT() returns the entire string.

LEN()

«

Length

```
LEN(cString | aTarget) ⇒ nCount
```

cString	is the character string to count.
aTarget	is the array to count.

LEN() returns the length of a character string or the number of elements in an array as an integer numeric value.

LOG()

«

```
LOG(nExp) ⇒ nNaturalLog
```

nExp	is a numeric value greater than zero to convert to its natural logarithm.

LOG() returns the natural logarithm as a numeric value. If ***nExp*** is less than or equal to zero, LOG() returns a numeric overflow (displayed as a row of asterisks).

LOWER()

«

```
LOWER(cString) ⇒ cLowerString
```

cString	is a character string to convert to lowercase.

LOWER() returns a copy of ***cString*** with all alphabetic characters converted to lowercase.

LTRIM()

«

Left trim

```
LTRIM(cString) ⇒ cTrimString
```

cString	is the character string to copy without leading spaces.

LTRIM() returns a copy of ***cString*** with the leading spaces removed.

LUPDATE()

Last update

```
LUPDATE() ⇒ dModification
```

LUPDATE() returns the date of last change to the open database file in the current work area.

MAX()

```
MAX(nExp1, nExp2) ⇒ nLarger
```

```
MAX(dExp1, dExp2) ⇒ dLarger
```

nExp1*, *nExp2	are the numeric values to compare.
dExp1*, *dExp2	are the date values to compare.

MAX() returns the larger of the two arguments. The value returned is the same type as the arguments.

MAXCOL()

Max column

```
MAXCOL() ⇒ nColumn
```

MAXCOL() returns the column number of the rightmost visible column for display purposes.

MAXROW()

«

```
MAXROW() ⇒ nRow
```

MAXROW() returns the row number of the bottommost visible row for display purposes.

MEMOEDIT()

«

```
MEMOEDIT( [cString] ,
        [nTop] , [nLeft] ,
        [nBottom] , [nRight] ,
        [lEditMode] ,
        [cUserFunction] ,
        [nLineLength] ,
        [nTabSize] ,
        [nTextBufferRow] ,
        [nTextBufferColumn] ,
        [nWindowRow] ,
        [nWindowColumn] ) ⇒ cTextBuffer
```

cString	is the character string or memo field to copy to the MEMOEDIT() text buffer.
nTop, ***nLeft***, ***nBottom***, ***nRight***	are window coordinates. The default coordinates are 0, 0, MAXROW(), and MAXCOL().
lEditMode	determines whether the text buffer can be edited or merely displayed. If not specified, the default value is true (‘**.T.**’).

cUserFunction	is the name of a user-defined function that executes when the user presses a key not recognized by MEMOEDIT() and when no keys are pending in the keyboard buffer.
nLineLength	determines the length of lines displayed in the MEMOEDIT() window. If a line is greater than ***nLineLength***, it is word wrapped to the next line in the MEMOEDIT() window. The default line length is (***nRight*** - ***nLeft***).
nTabSize	determines the size of a tab character to insert when the user presses Tab. The default is four.
nTextBufferRow, ***nTextBufferColumn***	define the display position of the cursor within the text buffer when MEMOEDIT() is invoked. ***nTextBufferRow*** begins with one and ***nTextBufferColumn*** begins with zero. Default is the beginning of MEMOEDIT() window.
nWindowRow, ***nWindowColumn***	define the initial position of the cursor within the MEMOEDIT() window. Row and column positions begin with zero. If these arguments are not specified, the initial window position is row zero and the current cursor column position.

MEMOEDIT() is a user interface and general purpose text editing function that edits memo fields and long character strings. Editing occurs within a specified window region placed anywhere on the screen.

[*Uparrow*]/[*Ctrl*]+E	Move up one line

[*Dnarrow*]/[*Ctrl*]+X	Move down one line
[*Leftarrow*]/[*Ctrl*]+S	Move left one character
[*Rightarrow*]/[*Ctrl*]+D	Move right one character
[*Ctrl*]- [*Leftarrow*]/[*Ctrl*]+A	Move left one word
[*Ctrl*]- [*Rightarrow*]/[*Ctrl*]+F	Move right one word
[*Home*]	Move to beginning of current line
[*End*]	Move to end of current line
[*Ctrl*]+[*Home*]	Move to beginning of current window
[*Ctrl*]+[*End*]	Move to end of current window
[*PgUp*]	Move to previous edit window
[*PgDn*]	Move to next edit window
[*Ctrl*]+[*PgUp*]	Move to beginning of memo
[*Ctrl*]+[*PgDn*]	Move to end of memo
[*Return*]	Move to beginning of next line
[*Delete*]	Delete character at cursor
[*Backspace*]	Delete character to left of cursor
[*Tab*]	Insert tab character or spaces
Printable characters	Insert character
[*Ctrl*]+Y	Delete the current line
[*Ctrl*]+T	Delete word right
[*Ctrl*]+B	Reform paragraph
[*Ctrl*]+V/[*Ins*]	Toggle insert mode
[*Ctrl*]+W	Finish editing with save
[*Esc*]	Abort edit and return original

```
MEMOLINE(cString,
        [nLineLength],
        [nLineNumber],
        [nTabSize],
        [lWrap]) ⇒ cLine
```

cString	is the memo field or character string from which to extract a line of text.
nLineLength	specifies the number of characters per line and can be between four and 254 . If not specified, the default line length is 79.
nLineNumber	is the line number to extract. If not specified, the default value is one.
nTabSize	defines the tab size. If not specified, the default value is four.
lWrap	toggles word wrap on and off. Specifying true (‘**.T.**’) toggles word wrap on; false (‘**.F.**’) toggles it off. If not specified, the default value is true (‘**.T.**’).

MEMOLINE() returns the line of text specified by ***nLineNumber*** in ***cString*** as a character string. If the line has fewer characters than the indicated length, the return value is padded with blanks. If the line number is greater than the total number of lines in ***cString***, MEMOLINE() returns a null string (""). If ***lWrap*** is true (‘**.T.**’) and the indicated line length breaks the line in the middle of a word, that word is not included as part of the return value but shows up at the beginning of the next line extracted with MEMOLINE(). If ***lWrap*** is false (‘**.F.**’), MEMOLINE() returns only the number of characters

specified by the line length. The next line extracted by MEMOLINE() begins with the character following the next hard carriage return, and all intervening characters are not processed.

MEMOREAD()

«

```
MEMOREAD(cFile) ⇒ cString
```

cFile	is the name of the file to read from disk. It must include an extension if there is one, and can optionally include a path.

MEMOREAD() returns the contents of a text file as a character string.

MEMORY()

«

```
MEMORY(nExp) ⇒ nKbytes
```

nExp	is a numeric value that determines the type of value MEMORY() returns.

MEMORY() returns an integer numeric value representing the amount of memory available.

MEMORY(0)	Estimated total space available for character values
MEMORY(1)	Largest contiguous block available for character values
MEMORY(2)	Area available for RUN commands

MEMOTRAN()

Memo translate

```
MEMOTRAN(cString,
        [cReplaceHardCR],
        [cReplaceSoftCR]) ⇒ cNewString
```

cString	is the character string or memo field to search.
cReplaceHardCR	is the character to replace a hard carriage return/linefeed pair with. If not specified, the default value is a semicolon (;).
cReplaceSoftCR	is the character to replace a soft carriage return/linefeed pair with. If not specified, the default value is a space.

MEMOTRAN() returns a copy of ***cString*** with the specified carriage return/linefeed pairs replaced.

MEMOWRIT()

Memo write

```
MEMOWRIT(cFile, cString) ⇒ lSuccess
```

cFile	is the name of the target disk file including the file extension and optional path and drive designator.
cString	is the character string or memo field to write to ***cFile***.

MEMOWRIT() is a memo function that writes a character string or memo field to a disk file. If a path is not specified, MEMOWRIT() writes ***cFile*** to the current DOS directory and not the current DEFAULT directory. If ***cFile*** already exists, it is overwritten. MEMOWRIT() returns true (‘`.T.`’) if the writing operation is successful; otherwise, it returns false (‘`.F.`’).

MEMVARBLOCK()

«

```
MEMVARBLOCK(cMemvarName) ⇒ bMemvarBlock
```

cMemvarName	is the name of the variable referred to by the set-get block, specified as a character string.

MEMVARBLOCK() returns a code block that when evaluated sets (assigns) or gets (retrieves) the value of the given memory variable. If ***cMemvarName*** does not exist, MEMVARBLOCK() returns NIL.

MIN()

«

```
MIN(nExp1, nExp2) ⇒ nSmaller
```

```
MIN(dExp1, dExp2) ⇒ dSmaller
```

nExp1, ***nExp2***	are the numeric values to compare.
dExp1, ***dExp2***	are the date values to compare.

MIN() returns the smaller of the two arguments. The value returned

is the same data type as the arguments.

MLCOUNT()

Memo line count

```
MLCOUNT(cString, [nLineLength],
        [nTabSize], [lWrap]) ⇒ nLines
```

cString	is the character string or memo field to count.
nLineLength	specifies the number of characters per line and can range from four to 254 . If not specified, the default line length is 79.
nTabSize	defines the tab size. If not specified, the default value is four.
lWrap	toggles word wrap on and off. Specifying true (‘**.T.**’) toggles word wrap on; false (‘**.F.**’) toggles it off. If not specified, the default value is true (‘**.T.**’).

MLCOUNT() returns the number of lines in ***cString*** depending on the ***nLineLength***, the ***nTabSize***, and whether word wrapping is on or off.

MLCTOPOS()

Memo line column to position

```
MLCTOPOS(cText, nWidth, nLine,
         nCol, [nTabSize], [lWrap]) ⇒ nPosition
```

cText	is the text string to scan.
nWidth	is the line length formatting width.
nLine	is the line number counting from 1.
nCol	is the column number counting from 0.
nTabSize	is the number of columns between tab stops. If not specified, the default is 4.
lWrap	is the word wrap flag. If not specified, the default is true (‘`.T.`’).

MLCTOPOS() returns the byte position within ***cText*** counting from 1.

MLPOS()

«

Memo line position

```
MLPOS(cString, nLineLength,
      nLine, [nTabSize], [lWrap]) ⇒ nPosition
```

cString	is a character string or memo field.
nLineLength	specifies the number of characters per line.
nLine	specifies the line number.
nTabSize	defines the tab size. The default is four.
lWrap	toggles word wrap on and off. Specifying true (‘`.T.`’) toggles word wrap on, and false (‘`.F.`’) toggles it off. The default is true (‘`.T.`’).

MLPOS() returns the character position of ***nLine*** in ***cString*** as an integer numeric value. If ***nLine*** is greater than the number of lines in ***cString***, MLPOS() returns the length of ***cString***.

MONTH()

«

```
MONTH(dDate) ⇒ nMonth
```

dDate	is the date value to convert.

MONTH() returns an integer numeric value in the range of zero to 12. Specifying a null date (CTOD("")) returns zero.

MPOSTOLC()

«

Memo position to line column

```
MPOSTOLC(cText, nWidth, nPos,
      [nTabSize], [lWrap]) ⇒ aLineColumn
```

cText	is a text string.
nWidth	is the length of the formatted line.
nPos	is the byte position within text counting from one.
nTabSize	is the number of columns between tab stops. If not specified, the default is four.
lWrap	is the word wrap flag. If not specified, the default is true (‘`.T.`’).

MPOSTOLC() returns an array containing the line and the column values for the specified byte position, ***nPos***. MPOSTOLC() is a memo function that determines the formatted line and column corresponding to a particular byte position within ***cText***. Note that the line number returned is one-relative, the column number is zero-relative. This is compatible with MEMOEDIT(). ***nPos*** is one-relative, com-

patible with AT(), RAT(), and other string functions.

NETERR()

«

Net error

```
NETERR( [lNewError] ) ⇒ lError
```

lNewError	if specified sets the value returned by NETERR() to the specified status. ***lNewError*** can be either true (‘**.T.**’) or false (‘**.F.**’). Setting NETERR() to a specified value allows the runtime error handler to control the way certain file errors are handled.

NETERR() returns true (‘**.T.**’) if a USE or APPEND BLANK fails. The initial value of NETERR() is false (‘**.F.**’). If the current process is not running under a network operating system, NETERR() always returns false (‘**.F.**’).

NETNAME()

«

```
NETNAME() ⇒ cWorkstationName
```

NETNAME() returns the workstation identification as a character string up to 15 characters in length. If the workstation identification was never set or the application is not operating under the IBM PC Network, it returns a null string ("").

NEXTKEY()

«

```
NEXTKEY() ⇒ nInkeyCode
```

NEXTKEY() returns an integer numeric value ranging from -39 to 386. If the keyboard buffer is empty, NEXTKEY() returns zero. If SET TYPEAHEAD is zero, NEXTKEY() always returns zero. NEXTKEY() is like the INKEY() function, but differs in one fundamental respect. INKEY() removes the pending key from the keyboard buffer and updates LASTKEY() with the value of the key. NEXTKEY(), by contrast, reads, but does not remove the key from the keyboard buffer and does not update LASTKEY().

NOSNOW()

«

```
NOSNOW(lToggle) ⇒ NIL
```

lToggle	is a logical value that toggles the current state of snow suppression. A value of true (‘**.T.**’) enables the snow suppression on, while a value of false (‘**.F.**’) disables snow suppression.

NOSNOW() is used to suppress snow on old CGA monitors.

ORDBAGEXT()

«

```
ORDBAGEXT() ⇒ cBagExt
```

ORDBAGEXT() returns a character expression that is the default Order Bag extension of the current work area. cBagExt is determined by the RDD active in the current work area.

ORDBAGNAME()

«

```
ORDBAGNAME(nOrder | cOrderName) ⇒ cOrderBagName
```

nOrder	is an integer that identifies the position in the Order List of the target Order whose Order Bag name is sought.
cOrderName	is a character string that represents the name of the target Order whose Order Bag name is sought.

ORDBAGNAME() returns a character string, the Order Bag name of the specific Order.

ORDCREATE()

«

```
ORDCREATE(cOrderBagName, [cOrderName], cExpKey, [bExpKey],
      [lUnique]) ⇒ NIL
```

cOrderBagName	is the name of a disk file containing one or more Orders.
cOrderName	is the name of the Order to be created.
cExpKey	is an expression that returns the key value to place in the Order for each record in the current work area. The maximum length of the index key expression is determined by the database driver.

bExpKey	is a code block that evaluates to a key value that is placed in the Order for each record in the current work area.
lUnique	specifies whether a unique Order is to be created. Default is the current global _SET_UNIQUE setting.

ORDCREATE() is an Order management function that creates an Order in the current work area. It works like DBCREATEINDEX() except that it lets you create Orders in RDDs that recognize multiple Order Bags.

ORDDESTROY()

«

```
ORDDESTROY(cOrderName [, cOrderBagName ]) ⇒ NIL
```

cOrderName	is the name of the Order to be removed from the current or specified work area.
cOrderBagName	is the name of a disk file containing one or more Orders.

ORDDESTROY() is an Order management function that removes a specified Order from multiple-Order Bags. ORDDESTROY() is not supported for DBFNDX and DBFNTX.

ORDFOR()

«

```
ORDFOR(cOrderName | nOrder [, cOrderBagName]) ⇒ cForExp
```

cOrderName	is the name of the target Order, whose cForExp is sought.
nOrder	is an integer that identifies the position in the Order List of the target Order whose cForExp is sought.
cOrderBagName	is the name of an Order Bag containing one or more Orders.

ORDFOR() returns a character expression, cForExp, that represents the FOR condition of the specified Order. If the Order was not created using the FOR clause the return value will be an empty string (""). If the database driver does not support the FOR condition, it may either return an empty string ("") or raise an "unsupported function" error, depending on the driver.

ORDKEY()

«

```
ORDKEY(cOrderName | nOrder [, cOrderBagName]) ⇒ cExpKey
```

cOrderName	is the name of an Order, a logical ordering of a database.
nOrder	is an integer that identifies the position in the Order List of the target Order whose cExpKey is sought.
cOrderBagName	is the name of a disk file containing one or more Orders.

ORDKEY() is an Order management function that returns a character expression, cExpKey, that represents the key expression of the specified Order.

ORDLISTADD()

«

```
ORDLISTADD(cOrderBagName [, cOrderName]) ⇒ NIL
```

cOrderBagName	is the name of a disk file containing one or more Orders.
cOrderName	the name of the specific Order from the Order Bag to be added to the Order List of the current work area. If you do not specify ***cOrderName***, all orders in the Order Bag are added to the Order List of the current work area.

ORDLISTADD() is an Order management function that adds the contents of an Order Bag , or a single Order in an Order Bag, to the Order List. Any Orders already associated with the work area continue to be active. If the newly opened Order Bag contains the only Order associated with the work area, it becomes the controlling Order; otherwise, the controlling Order remains unchanged.

ORDLISTCLEAR()

«

```
ORDLISTCLEAR() ⇒ NIL
```

ORDLISTCLEAR() is an Order management function that removes all Orders from the Order List for the current work area.

ORDLISTREBUILD()

«

```
ORDLISTREBUILD() ⇒ NIL
```

ORDLISTREBUILD() is an Order management function that rebuilds all the orders in the current Order List.

ORDNAME()

«

```
ORDNAME(nOrder [,cOrderBagName]) ⇒ cOrderName
```

nOrder	is an integer that identifies the position in the Order List of the target Order whose database name is sought.
cOrderBagName	is the name of a disk file containing one or more Orders.

ORDNAME() returns the name of the specified Order in the current Order List or the specified Order Bag if opened in the Current Order list.

ORDNUMBER()

«

```
ORDNUMBER(cOrderName [, cOrderBagName]) ⇒ nOrderNo
```

cOrderName	the name of the specific Order whose position in the Order List is sought.
cOrderBagName	is the name of a disk file containing one or more Orders.

ORDNUMBER() returns nOrderNo, an integer that represents the position of the specified Order in the Order List.

ORDSETFOCUS()

«

```
ORDSETFOCUS( [cOrderName | nOrder] [, cOrderBagName] )
      ⇒ cPrevOrderNameInFocus
```

cOrderName	is the name of the selected Order, a logical ordering of a database.
nOrder	is a number representing the position in the Order List of the selected Order.
cOrderBagName	is the name of a disk file containing one or more Orders.

ORDSETFOCUS() is an Order management function that returns the Order Name of the previous controlling Order and optionally sets the focus to an new Order.

OS()

«

```
OS() ⇒ cOsName
```

OS() returns the operating system name as a character string.

OUTERR()

«

Output error

```
OUTERR(exp_list) ⇒ NIL
```

exp_list	is a list of values to display and can consist of any combination of data types including memo.

OUTERR() is identical to OUTSTD() except that it writes to the standard error device rather than the standard output device. Output sent to the standard error device bypasses the console and output devices as well as any DOS redirection. It is typically used to log error messages in a manner that will not interfere with the standard screen or printer output.

OUTSTD()

«

Output standard

```
OUTSTD(exp_list) ⇒ NIL
```

exp_list	is a list of values to display and can consist of any combination of data types including memo.

OUTSTD() is a simple output function similar to QOUT(), except that it writes to the STDOUT device (instead of to the console output stream).

PAD?()

«

```
PADL(exp, nLength, [cFillChar]) ⇒ cPaddedString
```

```
PADC(exp, nLength, [cFillChar]) ⇒ cPaddedString
```

```
PADR(exp, nLength, [cFillChar]) ⇒ cPaddedString
```

exp	is a character, numeric, or date value to pad with a fill character.
nLength	is the length of the character string to return.
cFillChar	is the character to pad ***exp*** with. If not specified, the default is a space character.

PADC(), PADL(), and PADR() are character functions that pad character, date, and numeric values with a fill character to create a new character string of a specified length. PADC() centers ***exp*** within ***nLength*** adding fill characters to the left and right sides; PADL() adds fill characters on the left side; and PADR() adds fill characters on the right side.

PCOL()

Printed column

```
PCOL() ⇒ nColumn
```

PCOL() returns an integer numeric value representing the last printed column position, plus one. The beginning column position is zero.

PROW()

«

Printed row

```
PROW() ⇒ nRow
```

PROW() returns an integer numeric value that represents the number of the current line sent to the printer. The beginning row position is zero.

QOUT()

«

```
QOUT( [exp_list] ) ⇒ NIL
```

```
QQOUT( [exp_list] ) ⇒ NIL
```

exp_list	is a comma-separated list of expressions (of any data type other than array or block) to display to the console. If no argument is specified and QOUT() is specified, a carriage return/linefeed pair is displayed. If QQOUT() is specified without arguments, nothing displays.

QOUT() and QQOUT() are console functions. They display the results of one or more expressions to the console. QOUT() outputs carriage return and linefeed characters before displaying the results of ***exp_list***. QQOUT() displays the results of ***exp_list*** at the current ROW() and COL() position. When QOUT() and QQOUT() display

to the console, ROW() and COL() are updated.

RAT()

Right at

```
RAT(cSearch, cTarget) ⇒ nPosition
```

cSearch	is the character string to locate.
cTarget	is the character string to search.

RAT() returns the position of ***cSearch*** within ***cTarget*** as an integer numeric value, starting the search from the right. If ***cSearch*** is not found, RAT() returns zero.

RDDLIST()

```
RDDLIST([nRDDType]) ⇒ aRDDList
```

nRDDType	is an integer that represents the type of the RDD you wish to list. ***nRDDType*** = 1 Full RDD implementation ***nRDDType*** = 2 Import/Export only driver.

RDDLIST() returns a one-dimensional array of the RDD names registered with the application as ***nRDDType***.

RDDNAME()

«

```
RDDNAME() ⇒ cRDDName
```

RDDNAME() returns a character string, cRDDName, the registered name of the active RDD in the current or specified work area.

RDDSETDEFAULT()

«

```
RDDSETDEFAULT( [cNewDefaultRDD] ) ⇒ cPreviousDefaultRDD
```

cNewDefaultRDD	is a character string, the name of the RDD that is to be made the new default RDD in the application.

RDDSETDEFAULT() is an RDD function that sets or returns the name of the previous default RDD driver and, optionally, sets the current driver to the new RDD driver specified by ***cNewDefaultRDD***.

READINSERT()

«

```
READINSERT( [lToggle] ) ⇒ lCurrentMode
```

lToggle	toggles the insert mode on or off. True (‘**.T.**’) turns insert on, while false (‘**.F.**’) turns insert off. The default is false (‘**.F.**’) or the last user-selected mode in READ or MEMOEDIT().

READINSERT() returns the current insert mode state as a logical value.

READMODAL()

«

```
READMODAL(aGetList) ⇒ NIL
```

aGetList	is an array containing a list of Get objects to edit.

READMODAL() is like the READ command, but takes a GetList array as an argument and does not reinitialize the GetList array when it terminates. The GET system is implemented using a public array called GetList. Each time an @...GET command executes, it creates a Get object and adds to the currently visible GetList array. The standard READ command is preprocessed into a call to READMODAL() using the GetList array as its argument.

READVAR()

«

```
READVAR() ⇒ cVarName
```

READVAR() returns the name of the variable associated with the current Get object or the variable being assigned by the current MENU TO command as an uppercase character string.

RECNO()

«

Record number

```
RECNO() ⇒ nRecord
```

RECNO() returns the current record number as an integer numeric value. If the work area contains a database file with zero records, RECNO() returns one, BOF() and EOF() both return true (‘.T.’), and LASTREC() returns zero. If the record pointer is moved past the last record, RECNO() returns LASTREC() + 1 and EOF() returns true (‘.T.’). If an attempt is made to move before the first record, RECNO() returns the record number of the first logical record in the database file and BOF() returns true (‘.T.’). If no database file is open, RECNO() will return a zero.

RECSIZE()

«

Record size

```
RECSIZE() ⇒ nBytes
```

RECSIZE() returns, as a numeric value, the record length, in bytes, of the database file open in the current work area. RECSIZE() returns zero if no database file is open.

REPLICATE()

«

```
REPLICATE(cString, nCount) ⇒ cRepeatedString
```

cString	is the character string to repeat.
nCount	is the number of times to repeat ***cString***.

REPLICATE() returns a character string. Specifying a zero as the ***nCount*** argument returns a null string ("").

RESTSCREEN()

Restore screen

```
RESTSCREEN( [nTop], [nLeft],
      [nBottom], [nRight], cScreen) ⇒ NIL
```

nTop, ***nLeft***, ***nBottom***, ***nRight***	define the coordinates of the screen information contained in ***cScreen***. If the ***cScreen*** was saved without coordinates to preserve the entire screen, no screen coordinates are necessary with RESTSCREEN().
cScreen	is a character string containing the saved screen region.

RESTSCREEN() is a screen function that redisplays a screen region saved with SAVESCREEN(). The target screen location may be the same as or different than the original location when the screen region was saved.

RIGHT()

«

```
RIGHT(cString, nCount) ⇒ cSubString
```

cString	is the character string from which to extract characters.
nCount	is the number of characters to extract.

RIGHT() returns the rightmost ***nCount*** characters of ***cString***. If ***nCount*** is zero, RIGHT() returns a null string (""). If ***nCount*** is negative or larger than the length of the character string, RIGHT() returns ***cString***.

RLOCK()

«

Record lock

```
RLOCK() ⇒ lSuccess
```

RLOCK() is a network function that locks the current record, preventing other users from updating the record until the lock is released. RLOCK() provides a shared lock, allowing other users read-only access to the locked record while allowing only the current user to modify it. A record lock remains until another record is locked, an UNLOCK is executed, the current database file is closed, or an FLOCK() is obtained on the current database file.

ROUND()

«

```
ROUND(nNumber, nDecimals) ⇒ nRounded
```

nNumber	is the numeric value to round.
nDecimals	defines the number of decimal places to retain. Specifying a negative ***nDecimals*** value rounds whole number digits.

ROUND() is a numeric function that rounds ***nNumber*** to the number of places specified by ***nDecimals***. Specifying a zero or negative value for ***nDecimals*** allows rounding of whole numbers. A negative ***nDecimals*** indicates the number of digits to the left of the decimal point to round. Digits between five to nine, inclusive, are rounded up. Digits below five are rounded down.

ROW()

«

```
ROW() ⇒ nRow
```

ROW() returns the cursor row position as an integer numeric value. The range of the return value is zero to MAXROW().

RTRIM()

«

Right trim

```
[R]TRIM(cString) ⇒ cTrimString
```

cString	is the character string to copy without trailing spaces.

RTRIM() returns a copy of ***cString*** with the trailing spaces removed. If ***cString*** is a null string ("") or all spaces, RTRIM() returns a null string ("").

SAVESCREEN()

«

```
SAVESCREEN( [nTop] , [nLeft] ,
      [nBottom] , [nRight] ) ⇒ cScreen
```

nTop, ***nLeft***, ***nBottom***, ***nRight***	define the coordinates of the screen region to save. Default is the entire screen.

SAVESCREEN() returns the specified screen region as a character string.

SCROLL()

«

```
SCROLL( [nTop] , [nLeft] ,
      [nBottom] , [nRight] , [nVert] [nHoriz] ) ⇒ NIL
```

nTop, ***nLeft***, ***nBottom***, ***nRight*** define the scroll region coordinates.

nVert	defines the number of rows to scroll, vertically. A positive value scrolls up the specified number of rows. A negative value scrolls down the specified number of rows. A value of zero disables vertical scrolling. If ***nVert*** is not specified, zero is assumed.

nHoriz	defines the number of rows to scroll horizontally. A positive value scrolls left the specified number of columns. A negative value scrolls right the specified number of columns. A value of zero disables horizontal scrolling. If ***nHoriz*** is not specified, zero is assumed. If you supply neither ***nVert*** or ***nHoriz*** parameters to SCROLL(), the area specified by the first four parameters will be blanked.

SCROLL() is a screen function that scrolls a screen region up or down a specified number of rows. When a screen scrolls up, the first line of the region is erased, all other lines are moved up, and a blank line is displayed in the current standard color on the bottom line of the specified region. If the region scrolls down, the operation is reversed. If the screen region is scrolled more than one line, this process is repeated.

SECONDS()

```
SECONDS() ⇒ nSeconds
```

SECONDS() returns the system time as a numeric value in the form seconds.hundredths. The numeric value returned is the number of seconds elapsed since midnight, and is based on a twenty-four hour clock in a range from zero to 86399.

SELECT()

«

```
SELECT( [cAlias] ) ⇒ nWorkArea
```

cAlias	is the target work area alias name.

SELECT() returns the work area of the specified alias as a integer numeric value.

SET()

«

```
SET(nSpecifier, [expNewSetting], [lOpenMode])
      ⇒ CurrentSetting
```

nSpecifier	is a numeric value that identifies the setting to be inspected or changed.
expNewSetting	is an optional argument that specifies a new value for the ***nSpecifier***. The type of ***expNewSetting*** depends on ***nSpecifier***.
lOpenMode	is a logical value that indicates whether or not files are opened for some settings. A value of false (‘**.F.**’) means the file should be truncated. A value of true (‘**.T.**’) means the file should be opened in append mode. In either case, if the file does not exist, it is created. If this argument is not specified, the default is append mode.

SET() returns the current value of the specified setting.

Inside nB, the function SET() is not so easy to use as inside the Clip-

per environment. This because nB cannot support manifest constants and a numeric specifier ***nSpecifier*** is not easy to manage. Instead of SET() you can use SETVERB().

SETBLINK()

«

```
SETBLINK( [lToggle] ) ⇒ lCurrentSetting
```

lToggle	changes the meaning of the asterisk (*) character when it is encountered in a SETCOLOR() string. Specifying true (‘**.T.**’) sets character blinking on and false (‘**.F.**’) sets background intensity. The default is true (‘**.T.**’).

SETBLINK() returns the current setting as a logical value.

SETCANCEL()

«

```
SETCANCEL( [lToggle] ) ⇒ lCurrentSetting
```

lToggle	changes the availability of Alt-C and Ctrl-Break as termination keys. Specifying true (‘**.T.**’) allows either of these keys to terminate an application and false (‘**.F.**’) disables both keys. The default is true (‘**.T.**’).

SETCANCEL() returns the current setting as a logical value.

SETCOLOR()

«

SETCOLOR([*cColorString*]) ⇒ *cColorString*

cColorString	is a character string containing a list of color attribute settings for subsequent screen painting.

SETCURSOR()

«

SETCURSOR([*nCursorShape*]) ⇒ *nCurrentSetting*

nCursorShape	is a number indicating the shape of the cursor. ***nCursorShape*** == 0 None ***nCursorShape*** == 1 Underline ***nCursorShape*** == 2 Lower half block ***nCursorShape*** == 3 Full block ***nCursorShape*** == 4 Upper half block

SETCURSOR() returns the current cursor shape as a numeric value.

SETKEY()

«

SETKEY(*nInkeyCode*, [*bAction*]) ⇒ *bCurrentAction*

nInkeyCode	is the INKEY() value of the key to be associated or queried.

bAction	specifies a code block that is automatically executed whenever the specified key is pressed during a wait state.

SETKEY() returns the action block currently associated with the specified key, or NIL if the specified key is not currently associated with a block.

SETMODE()

«

```
SETMODE(nRows, nCols) ⇒ lSuccess
```

nRows	is the number of rows in the desired display mode.
nCols	is the number of columns in the desired display mode.

SETMODE() is an environment function that attempts to change the mode of the display hardware to match the number of rows and columns specified. The change in screen size is reflected in the values returned by MAXROW() and MAXCOL().

SETPOS()

«

Set position

```
SETPOS(nRow, nCol) ⇒ NIL
```

nRow*, *nCol	define the new screen position of the cursor. These values may range from 0, 0 to MAXROW(), MAXCOL().

SETPOS() is an environment function that moves the cursor to a new position on the screen. After the cursor is positioned, ROW() and COL() are updated accordingly.

SETPRC()

«

Set printer row column

```
SETPRC(nRow, nCol) ⇒ NIL
```

nRow	is the new PROW() value.
nCol	is the new PCOL() value.

SETPRC() is a printer function that sends control codes to the printer without changing the tracking of the printhead position.

SOUNDEX()

«

```
SOUNDEX(cString) ⇒ cSoundexString
```

cString	is the character string to convert.

SOUNDEX() returns a four-digit character string in the form A999.

SPACE()

«

```
SPACE(nCount) ⇒ cSpaces
```

nCount	is the number of spaces to return.

SPACE() returns a character string. If ***nCount*** is zero, SPACE() returns a null string ("").

SQRT()

«

```
SQRT(nNumber) ⇒ nRoot
```

nNumber	is a positive number to take the square root of.

SQRT() returns a numeric value calculated to double precision. The number of decimal places displayed is determined solely by SET DECIMALS regardless of SET FIXED. A negative ***nNumber*** returns zero.

STR()

«

String

```
STR(nNumber, [nLength], [nDecimals]) ⇒ cNumber
```

nNumber	is the numeric expression to convert to a character string.

nLength	is the length of the character string to return, including decimal digits, decimal point, and sign.
nDecimals	is the number of decimal places to return.

STR() returns ***nNumber*** formatted as a character string.

STRTRAN()

«

```
STRTRAN(cString, cSearch,
      [cReplace], [nStart], [nCount]) ⇒ cNewString
```

cString	is the character string or memo field to search.
cSearch	is the sequence of characters to locate.
cReplace	is the sequence of characters with which to replace ***cSearch***. If this argument is not specified, the specified instances of the search argument are replaced with a null string ("").
nStart	is the first occurrence that will be replaced. If this argument is omitted, the default is one.
nCount	is the number of occurrences to replace. If this argument is not specified, the default is all.

STRTRAN() returns a new character string with the specified instances of ***cSearch*** replaced with ***cReplace***.

STUFF()

«

```
STUFF(cString, nStart,
      nDelete, cInsert) ⇒ cNewString
```

cString	is the target character string into which characters are inserted and deleted.
nStart	is the starting position in the target string where the insertion/deletion occurs.
nDelete	is the number of characters to delete.
cInsert	is the string to insert.

STUFF() returns a copy of ***cString*** with the specified characters deleted and with ***cInsert*** inserted.

SUBSTR()

«

Sub string

```
SUBSTR(cString, nStart, [nCount]) ⇒ cSubstring
```

cString	is the character string from which to extract a substring.
nStart	is the starting position in ***cString***. If ***nStart*** is positive, it is relative to the leftmost character in ***cString***. If ***nStart*** is negative, it is relative to the rightmost character in the ***cString***.

nCount	is the number of characters to extract. If omitted, the substring begins at ***nStart*** and continues to the end of the string. If ***nCount*** is greater than the number of characters from ***nStart*** to the end of ***cString***, the extra is ignored.

SUBSTR() is a character function that extracts a substring from another character string or memo field.

TIME()

«

```
TIME() ⇒ cTimeString
```

TIME() returns the system time as a character string in the form hh:mm:ss. hh is hours in 24-hour format, mm is minutes, and ss is seconds.

TIME() is a time function that displays the system time on the screen or prints it on a report.

TONE()

«

```
TONE(nFrequency, nDuration) ⇒ NIL
```

nFrequency	is a positive numeric value indicating the frequency of the tone to sound.
nDuration	is a positive numeric value indicating the duration of the tone measured in increments of 1/18 of a second. For example, an ***nDuration*** value of 18 represents one second.

For both arguments, noninteger values are truncated (not rounded) to their integer portion.

TRANSFORM()

«

```
TRANSFORM(exp, cSayPicture) ⇒ cFormatString
```

exp	is the value to format. This expression can be any valid data type except array, code block, and NIL.
cSayPicture	is a string of picture and template characters that describes the format of the returned haracter string.

TRANSFORM() converts ***exp*** to a formatted character string as defined by ***cSayPicture***.

TYPE()

«

```
TYPE(cExp) ⇒ cType
```

cExp	is a character expression whose type is to be determined. ***cExp*** can be a field, with or without the alias, a private or public variable, or an expression of any type.

TYPE() returns one of the following characters:

A	Array
B	Block
C	Character

D	Date
L	Logical
M	Memo
N	Numeric
O	Object
U	NIL, local, or static
UE	Error syntactical
UI	Error indeterminate

TYPE() is a system function that returns the type of the specified expression. TYPE() is like VALTYPE() but uses the macro operator (&) to determine the type of the argument. VALTYPE(), by contrast, evaluates an expression and determines the data type of the return value.

UPDATED()

«

```
UPDATED() ⇒ lChange
```

UPDATED() returns true (‘**.T.**’) if data in a GET is added or changed; otherwise, it returns false (‘**.F.**’).

UPPER()

«

```
UPPER(cString) ⇒ cUpperString
```

cString	is the character string to convert.

UPPER() returns a copy of ***cString*** with all alphabetical characters

converted to uppercase. All other characters remain the same as in the original string.

USED()

«

```
USED() ⇒ lDbfOpen
```

USED() returns true (‘**.T.**’) if there is a database file in USE in the current work area; otherwise, it returns false (‘**.F.**’).

VAL()

«

Value

```
VAL(cNumber) ⇒ nNumber
```

cNumber	is the character expression to convert.

VAL() is a character conversion function that converts a character string containing numeric digits to a numeric value. When VAL() is executed, it evaluates ***cNumber*** until a second decimal point, the first non-numeric character, or the end of the expression is encountered.

VALTYPE()

«

Value type

```
VALTYPE(exp) ⇒ cType
```

exp	is an expression of any type.

VALTYPE() returns a single character representing the data type returned by ***exp***. VALTYPE() returns one of the following characters:

A	Array
B	Block
C	Character
D	Date
L	Logical
M	Memo
N	Numeric
O	Object
U	NIL

VALTYPE() is a system function that takes a single argument, evaluates it, and returns a one character string describing the data type of the return value.

YEAR()

«

```
YEAR(dDate) ⇒ nYear
```

dDate	is the date value to convert.

YEAR() returns the year of the specified date value including the century digits as a four-digit numeric value. The value returned is not affected by the current DATE or CENTURY format. Specifying a null date (CTOD("")) returns zero.

nB functions

Some functions made into nB are available for macro use. Not all available functions are here documented.

ACCEPT()

```
ACCEPT( Field, [cMessage], [cHeader] ) ⇒ updatedField|NIL
```

It is a prompt function that shows ***cMessage*** asking to type something into ***Field***. It returns the updated data or NIL if [*Esc*] was pressed. The string ***cHeader*** is showed centered at the top window.

ACHOICE()

```
ACHOICE(nTop, nLeft, nBottom, nRight,
        acMenuItems,
        [alSelectableItems],
        [nInitialItem],
        [lButtons | aButtons]) ⇒ nPosition
```

nTop, ***nLeft***, ***nBottom***, ***nRight***	are the window coordinates.
acMenuItems	is an array of character strings to display as the menu items.

alSelectableItems	is a parallel array of logical values (one element for each item in ***acMenuItems***) that specify the selectable menu items. Elements can be logical values or character strings. If the element is a character string, it is evaluated as a macro expression which should evaluate to a logical data type. A value of false (‘`.F.`’) means that the corresponding menu item is not available, and a value of true (‘`.T.`’) means that it is available. By default, all menu items are available for selection.
nInitialItem	is the position in the ***acMenuItems*** array of the item that will be highlighted when the menu is initially displayed.
lButtons	if True means that default buttons will appear.
aButtons	is an array of buttons.
aButtons[***n***][1] == N	the nth button row position;
aButtons[***n***][2] == N	the nth button column position;
aButtons[***n***][3] == C	the nth button text;
aButtons[***n***][4] == B	the nth button code block.

ACHOICE() returns the numeric position in the ***acMenuItems*** array of the menu item selected. If no choice is made, ACHOICE() returns zero.

ACHOICEWINDOW()

«

```
ACHOICEWINDOW( acMenuItems, [cDescription],
      nTop, nLeft, nBottom, nRight,
      [alSelectableItems],
      [nInitialItem] ) ⇒ nPosition
```

acMenuItems	is an array of character strings to display as the menu items.
cDescription	is a header to be shown at the top of window.
nTop, ***nLeft***, ***nBottom***, ***nRight***	are the window coordinates.
alSelectableItems	is a parallel array of logical values (one element for each item in ***acMenuItems***) that specify the selectable menu items. Elements can be logical values or character strings. If the element is a character string, it is evaluated as a macro expression which should evaluate to a logical data type. A value of false (‘**.F.**’) means that the corresponding menu item is not available, and a value of true (‘**.T.**’) means that it is available. By default, all menu items are available for selection.
nInitialItem	is the position in the ***acMenuItems*** array of the item that will be highlighted when the menu is initially displayed.

ACHOICEWINDOW() calls ACHOICE() with a window border around the ACHOICE() screen area.

ALERTBOX()

«

ALERTBOX(***cMessage*** , [***aOptions***]) ⇒ ***nChoice***

cMessage	is the message text displayed, centered, in the alert box. If the message contains one or more semicolons, the text after the semicolons is centered on succeeding lines in the dialog box.
aOptions	defines a list of up to 4 possible responses to the dialog box.

ALERTBOX() returns a numeric value indicating which option was chosen. If the [*Esc*] key is pressed, the value returned is zero. The ALERTBOX() function creates a simple modal dialog. The user can respond by moving a highlight bar and pressing the Return or SpaceBar keys, or by pressing the key corresponding to the first letter of the option. If ***aOptions*** is not supplied, a single "Ok" option is presented.

ALERTBOX() is similar to ALERT() but it accept mouse input.

ATB()

«

```
ATB( [nTop], [nLeft], [nBottom], [nRight],
     aArray, [nSubscript],
     [acColSayPic],
     [acColTopSep], [acColBodySep], [acColBotSep],
     [acColHead], [acColFoot],
     [abColValid],
     [abColMsg],
     [cColor], [abColColors],
     [lModify],
     [lButtons | aButtons] ) => NIL
```

nTop, ***nLeft***, ***nBottom***, ***nRight***	defines the screen area where browse have to take place.
aArray	bidimensional array to be browsed.
nSubscript	starting array position.
acColSayPic	is the picture array.
acColTopSep	is the top separation array: default is chr(194)+chr(196).
acColBodySep	is the body separation array: default is chr(179).
acColBotSep	is the bottom separation array: default is chr(193)+chr(196).
acColHead	is the header array for every column.
acColFoot	is the footer array for every column.
abColValid	is the validation array that specify when a field is properly filled. The condition must be specified in code block format.

abColMsg	is the message array that permits to show information at the bottom of browse area. The array must be composed with code blocks which result with a character string.
cColor	is the color string: it may be longer than the usual 5 elements.
abColColors	is the color code block array. The code block receive as parameter the value contained inside the field and must return an array containing two numbers: they correspond to the two color couple from ***cColor***.
lModify	indicates whether the browse can modify data.
lButtons	if True, default buttons are displayed.
aButtons	array of buttons.
aButtons[***n***][1] N	the nth button row position;
aButtons[***n***][2] N	the nth button column position;
aButtons[***n***][3] C	the nth button text;
aButtons[***n***][4] B	the nth button code block.

This function starts the browse of a bidimensional array. Only arrays containing monodimensional array containing the same kind of editable data are allowed. The function can handle a maximum of 61 columns.

BCOMPILE()

«

```
BCOMPILE( cString ) ⇒ bBlock
```

Compiles the string ***cString*** and returns the code block ***bBlock***

BUTTON()

«

```
BUTTON( @aButtons,
      [nRow], [nCol], [cText], [cColor],
      [bAction] ) ⇒ NIL
```

aButtons	the array of buttons to be increased with a new button array.
nRow and ***nCol***	is the row and column starting position for the button string.
cText	is the text that make up the button.
cColor	is the color string.
bAction	is the code block associated to the button.

This function adds to ***aButtons*** a new button array. Please note that the button array added is compatible only with the READ() function and not the other function using array of buttons: the others do not have a color string.

COLORARRAY()

«

```
COLORARRAY( cColor ) ⇒ aColors
```

cColors	a color string to be translated into a color array.

This function transform a color string into a color array. The array has as many elements as the colors contained inside ***cColor*** string.

COORDINATE()

«

```
COORDINATE( [@nTop, @nLeft], @nBottom, @nRight,
      [cHorizontal], [cVertical] ) ⇒ NIL
```

nTop, ***nLeft***, ***nBottom*** and ***nRight***	are the starting position of a window that is to be differently aligned.
cHorozontal	determinates the horizontal alignment: "L" all left; "l" middle left; "C" center; "c" center; "R" all right; "r" middle right.
cVertical	determinate the vertical alignment: "T" top; "t" up; "C" center; "c" center; "B" bottom; "b" down.

This function helps with the windows alignment recalculating and modifying ***nTop***, ***nLeft***, ***nBottom*** and ***nRight*** in the way to obtain the desired alignment.

COPYFILE()

«

```
COPYFILE( cSourceFile, cTargetFile | cDevice ) ⇒ NIL
```

cSourceFile	the source filename.
cTargetFile	the target filename.
cDevice	the target devicename.

This function copies the ***cSourceFile*** to ***cTargetFile*** or to ***cDevice***.

DBAPP()

```
DBAPP( cFileName, [acFields],
       [bForCondition], [bWhileCondition],
       [nNextRecords],
       [nRecord],
       [lRest],
       [cDriver] ) ⇒ NIL
```

cFileName	the filename containing data to append to the active alias.
acFields	array of fieldnames indicating the fields that should be updated on the active alias (default is all).
bForCondition	a code block containing the FOR condition to respect for the data append. Will be appended data that makes the evaluation of this code block True.
bWhileCondition	a code block containing the WHILE condition to respect for the data append. Will be appended data as long as the evaluation of this code block is True: the first time it becomes False, the data appending is terminated.
nNextRecord	if used, means that only the first ***nNextRecords*** will be appended.

nRecord	if used, means that that only the record ***nRecord*** will be appended.
lRest	this option is not available here also if the function saves a place for it.
cDriver	is the optional driver name to use to open the ***cFileName*** file.

This function is used to append data to the active alias using data from the ***cFileName*** file, that in this case is a '`.DBF`' file.

DBCLOSE()

«

```
DBCLOSE() ⇒ NIL
```

It is a substitution function of DBCLOSEALL() to use inside "compiled" macros, as a true DBCLOSEALL() will close the macro file too.

DBCONTINUE()

«

```
DBCONTINUE() ⇒ NIL
```

This function resumes a pending DBLOCATE().

DBCOPY()

«

```
DBCOPY( cFileName , [acFields] ,
        [bForCondition] , [bWhileCondition] ,
        [nNextRecords] ,
        [nRecord] ,
        [lRest] ,
        [cDriver] ) ⇒ NIL
```

cFileName	the target filename for the data contained inside the active alias.
acFields	array of fieldnames indicating the fields that should be used from the active alias (default is all).
bForCondition	a code block containing the FOR condition to respect for the data copy. Will be copied the data that makes the evaluation of this code block True.
bWhileCondition	a code block containing the WHILE condition to respect for the data copy. Will be copied data as long as the evaluation of this code block is True: the first time it becomes False, the data copying is terminated.
nNextRecord	if used, means that only the first ***nNextRecords*** will be copied.
nRecord	if used, means that that only the record ***nRecord*** will be copied.
lRest	if used means that only the remaining records inside the active alias are copied.
cDriver	is the optional driver name to use to open the ***cFileName*** file.

This function is used to copy data to ***cFileName*** form the active alias.

DBCOPYSTRUCT()

«

```
DBCOPYSTRUCT( cDatabase, [acFields] ) ⇒ NIL
```

cDatabase	is a structure '.DBF' file that will be filled with structure information about the active alias.
acFields	is an array of fieldnames that should be taken into consideration.

This function creates a structure '.DBF' file copying the structure of the active alias.

DBCOPYXSTRUCT()

«

```
DBCOPYXSTRUCT( cExtendedDatabase ) ⇒ NIL
```

cExtendedDatabase	is a structure '.DBF' file that will be filled with structure information about the active alias, accepting extended structure informations.

This function creates a structure '.DBF' file copying the structure of the active alias. This function accept non-standard structure, that is, the extended structure available inside Clipper.

DBDELIM()

«

```
DBDELIM( lCopyTo , cFileName , [cDelimiter] , [acFields] ,
       [bForCondition] , [bWhileCondition] ,
       [nNextRecords] , [nRecord] , [lRest] ) ⇒ NIL
```

lCopyTo	if True the function work copying data to ***cFileName*** from the active alias, if False the function work appending data from ***cFileName*** to the active alias.
cFileName	the filename containing data to append to the active alias or to use as the target of the data copy from the active alias.
cDelimiter	the delimiter string (or character) used to separate fields inside ***cFileName***.
acFields	array of fieldnames indicating the fields of the active alias that should be taken into consideration (default is all).
bForCondition	a code block containing the FOR condition to respect. The operation will be made for all records that respect the condition.
bWhileCondition	a code block containing the WHILE condition to respect. The first time it becomes False, the operation is terminated.
nNextRecord	if used, means that only the first ***nNextRecords*** will be appended/copied.
nRecord	if used, means that that only the record ***nRecord*** will be appended/copied.
lRest	if used means that only the remaining records will be taken into consideration.

This function is used to append data to the active alias using data

from the ***cFileName*** file or to copy data into ***cFileName*** using the active alias as the source. ***cFileName*** is a delimited ASCII file.

DBISTATUS()

«

```
DBISTATUS() ⇒ cDBInformations
```

This function returns the informations on the active alias in a text form.

DBISTRUCTURE()

«

```
DBISTRUCTURE() ⇒ cTextStructure | NIL
```

This function returns the structure information on the active alias in a text form.

DBJOIN()

«

```
DBJOIN( cAlias, cDatabase,
      [acFields], [bForCondition] ) ⇒ NIL
```

cAlias	the name of the alias to use to merge with records from the active alias.
cDatabase	the target '.DBF' filename.
acFields	the array of fieldnames which represent the projection of fields form both Aliases into the new '.DBF' file. If not specified, all fields from the primary work area are included in the target '.DBF' file.

This function creates a new database file by merging selected records and fields form two work areas (Aliases) based on a general condition. It works by making a complete pass through the secondary work area ***cAlias*** for each record in the primary work area (the active alias), evaluating the condition for each record in the secondary work area. When ***bForCondition*** is evaluated True, a new record is created in the target database file ***cDatabase*** using the fields specified from both work areas inside ***acFields***.

DBLABELFORM()

```
DBLABELFORM( cLabel, [lToPrinter], [cFile],
      [lNoConsole], [bForCondition], [bWhileCondition],
      [nNextRecords], [nRecord], [lRest], [lSample] )
      ⇒ NIL
```

cLabel	is the name of the label file (.LBL) that contains the label format definition.
lToPrinter	if True, the output is copied to printer (‘LPT1:’).
cFile	if present, it is the name of a ASCII file where the output is copied.
lNoConsole	if True, the output is not sent to the console.
bForCondition	a code block containing the FOR condition to respect for label print. Only the records contained inside the active alias that respect the condition will be used for labels.
bWhileCondition	a code block containing the WHILE condition to respect for the label print. The first time that the condition is False, the label print terminates.

nNextRecord	if used, means that only the first ***nNextRecords*** will be used.
nRecord	if used, means that that only the record ***nRecord*** will be used.
lRest	if used means that only the remaining records inside the active alias will be used.
lSample	if True displays test labels as rows of asterisks.

This function prints labels to the console.

DBLIST()

«

```
DBLIST( [lToDisplay], abListColumns,
        [lAll],
        [bForCondition], [bWhileCondition],
        [nNextRecords], [nRecord], [lRest],
        [lToPrinter], [cFileName] )
```

lToDisplay	if True the printout is sent to the console screen.
abListColumns	is an array of columns expressions to list.
lAll	if True prints all the records contained inside the active alias.
bForCondition	a code block containing the FOR condition to respect. Only the records contained inside the active alias that respect the condition will be used for list.
bWhileCondition	a code block containing the WHILE condition to respect. The first time that the condition is False, the list terminates.
nNextRecord	if used, means that only the first ***nNextRecords*** will be used.

nRecord	if used, means that that only the record ***nRecord*** will be used.
lRest	if used means that only the remaining records inside the active alias will be used.
lToPrinter	if True, the output is copied to printer (‘LPT1:’).
cFileName	if present, it is the name of a ASCII file where the output is copied.

This function prints a list of records to the console.

DBLOCATE()

«

```
DBLOCATE( [bForCondition], [bWhileCondition],
      [nNextRecords], [nRecord], [lRest] ) ⇒ NIL
```

bForCondition	a code block containing the FOR condition to respect. Only the records contained inside the active alias that respect the condition will be taken into consideration.
bWhileCondition	a code block containing the WHILE condition to respect. The first time that the condition is False, the locate terminates.
nNextRecord	if used, means that only the first ***nNextRecords*** will be used.
nRecord	if used, means that that only the record ***nRecord*** will be used.
lRest	if used means that only the remaining records inside the active alias will be used.

This function searches sequentially for the first record matching the FOR and WHILE conditions. Once a DBLOCATE() has been issued

you can resume the search from the current record pointer position with DBCONTINUE().

The WHILE condition and the scope (***nNextRecord***, ***nRecord*** and ***lRest***) apply only to the initial DBLOCATE() and are not operational for any subsequent DBCONTINUE() call.

DBOLDCREATE()

«

```
DBOLDCREATE( cDatabase, cExtendedDatabase,
      [cDriver], [lNew], [cAlias] ) ⇒ NIL
```

cDatabase	is the name of the new database file, with an optional drive and directory, specified as a character string. If specified without an extension (.dbf) is assumed.
cExtendedDatabase	is a '.DBF' file containing the structure information of the file to create.
cDriver	specifies the replaceable database driver (RDD) to use to process the current work area. ***cDriver*** is the name of the RDD specified as a character expression.
lNew	if True the newly created '.DBF' file is opened using the next available work area making it the current work area (the active alias).
cAlias	if ***lNew*** is set to True, this is the alias name to use to open the file.

This function is a old database function (superseded form DBCREATE()) that creates a database file from the structure information contained inside a structure file.

DBPACK()

```
DBPACK() ⇒ NIL
```

This function eliminates definitively the active alias records previously signed for deletion. It works only if the active alias is opened in exclusive mode.

DBSDF()

```
DBSDF( lCopyTo, cFileName, [acFields],
       [bForCondition], [bWhileCondition],
       [nNextRecords], [nRecord], [lRest] ) ⇒ NIL
```

lCopyTo	if True the function works copying data to ***cFileName*** from the active alias, if False the function work appending data from ***cFileName*** to the active alias.
cFileName	the filename containing data to append to the active alias or to use as the target of the data copy from the active alias.
acFields	array of fieldnames indicating the fields of the active alias that should be taken into consideration (default is all).
bForCondition	a code block containing the FOR condition to respect. The operation will be made for all records that respect the condition.
bWhileCondition	a code block containing the WHILE condition to respect. The first time it becomes False, the operation is terminated.

nNextRecord	if used, means that only the first ***nNextRecords*** will be appended/copied.
nRercord	if used, means that that only the record ***nRecord*** will be appended/copied.
lReset	if used means that only the remaining records will be taken into consideration.

This function is used to append data to the active alias using data from the ***cFileName*** file or to copy data into ***cFileName*** using the active alias as the source. ***cFileName*** is a SDF ASCII file.

DBSORT()

«

```
DBSORT( cDatabase , [acFields] ,
        [bForCondition] , [bWhileCondition] ,
        [nNextRecords] , [nRecord] , [lRest] ) ⇒ NIL
```

cDatabase	the '.DBF' file to create.
acFields	the array of fields to be used to create the new sorted ***cDatabase*** file.
bForCondition	a code block containing the FOR condition to respect. Only the records contained inside the active alias that respect the condition will be taken into consideration.
bWhileCondition	a code block containing the WHILE condition to respect. The first time that the condition is False, the sort terminates.
nNextRecord	if used, means that only the first ***nNextRecords*** inside the active alias will be used.
nRecord	if used, means that that only the record ***nRecord*** will be used.

lRest	if used means that only the remaining records inside the active alias will be used.

Copy the active alias to a '.DBF' file in sorted order.

DBTOTAL()

«

```
DBTOTAL( cDatabase, bKey, [acFields],
      [bForCondition], [bWhileCondition],
      [nNextRecords], [nRecord], [lRest] ) ⇒ NIL
```

cDatabase	the '.DBF' file to create that will contain the copy of summarised records.
bKey	the code block key expression that should correspond to the key expression of the active index of the active alias.
acFields	the array of fields to be used to create the new ***cDatabase*** file.
bForCondition	a code block containing the FOR condition to respect. Only the records contained inside the active alias that respect the condition will be taken into consideration.
bWhileCondition	a code block containing the WHILE condition to respect. The first time that the condition is False, the sort terminates.
nNextRecords	if used, means that only the first ***nNextRecords*** inside the active alias will be used.
nRecord	if used, means that that only the record ***nRecord*** will be used.
lRest	if used means that only the remaining records inside the active alias will be used.

This function summarises records by key value to a '.DBF' file. It sequentially process the active alias scanning the specified scope of records. Records with the same key will be summarised inside the destination '.DBF' file. The value of numeric fields of records with the same key are added.

DBUPDATE()

«

```
DBUPDATE( cAlias , bKey , [lRandom] , [bReplacement] )
```

cAlias	is the alias containing data to be used to update the active alias.
bKey	is a code block expression using information form the ***cAlias*** to obtain a key to refer to the active alias.
lRandom	if True, allows record in the ***cAlias*** to be in any order. In this case, the active alias must be indexed with the same key as ***bKey***.
bReplacement	is the code block that will be executed when records matches: it should contains the criteria for data update.

This function updates the active alias with data from another .DBF file.

Example:

```
dbUpdate( "INVOICE", {|| LAST}, .T.,;
     {|| FIELD->TOTAL1 := INVOICE->SUM1,;
     FIELD->TOTAL2 := INVOICE->SUM2 } )
```

DBZAP()

```
DBZAP() ⇒ NIL
```

This function erases immediately all the records contained inside the active alias.

DISPBOXCOLOR()

```
DISPBOXCOLOR( [nColorNumber], [cBaseColor] ) ⇒ cColor
```

nColorNumber	may be 1 or 2 and are the two color used to create shadowed borders. 1 is usually used for the left and top line; 2 is used for the right and bottom line.
cBaseColor	is the starting color string. The default is the actual color.

This function return a color string used for DISPBOXSHADOW() the function that create a shadowed border around a screen window.

DISPBOXSHADOW()

```
DISPBOXSHADOW( nTop, nLeft, nBottom, nRight,
      [cBoxString], [cColor1], [cColor2] ) ⇒ NIL
```

nTop, ***nLeft***, ***nBottom*** and ***nRight***	are the screen coordinate where the box is to be displayed.
cBoxString	is the box string containing the character to use to build the box. Default is a single line box.

cColor1	is the color string to use for the left and top side of the box.
cColor2	is the color string to use for the right and bottom side of the box.

This function draws a screen box like DISPBOX() but allowing the variation of colors around the border to simulate a sort of shadow.

DIR()

«

```
DIR( [cFileSpec], [lDrives], [lDirs], [lFiles],
     [lNoDirReturn], [nSortColumn] ) ⇒ cPathname
```

cFileSpec	the filename or Pathname, also with wild-cards, to be searched.
lDrives	true (‘`.T.`’) means: include drives letters.
lDirs	true (‘`.T.`’) means: include directory names.
lFiles	true (‘`.T.`’) means: include file names.
lNoRirReturn	true (‘`.T.`’) means: do not return the shown directory if [*Esc*] is used to exit.
nSortColumn	the column number to use to sort the list. The columns are: Name = 1, Size = 2, Date = 3, Time = 4, Attribute = 5. It is not possible to sort for extention.

It is a window function useful to search a file or a directory. The

complete pathname of the selected file is returned.

DOC()

«

```
DOC( [cTextFileName] ) ⇒ NIL
```

cTextFileName	can contain the text file to open and edit; if empty, the editing of 'UNTITLED.TXT' will start.

It is the nB Text editor useful for small text files (less then 64K) and contains a complete menu that can be started with [F10].

Attention: doc() should not be used inside macros.

DOTLINE()

«

```
DOTLINE() ⇒ NIL
```

This function is a "dot" command line useful for calculations resolution. The dot-line content may be passed to the keyboard buffer.

DTEMONTH()

«

Date of month

```
DTEMONTH( nMonth , cLanguage ) ⇒ cMonth
```

nMonth	the month number.
cLanguage	the language name.

This function translates the ***nMonth*** number into the month name translated using the ***cLanguage*** language.

DTEWEEK()

«

Date of week

```
DTEWEEK( nWeek, cLanguage ) ⇒ cWeek
```

nWeek	is the week number (1 is Sunday, 7 is Saturday) to be translated into text.
cLanguage	is the language name into which the week must be expressed. At the moment it works only for Italian, so ***cLanguage*** can only contain "ITALIANO".

This function translates the week number into the week name translated using the ***cLanguage*** language.

EX()

«

Execute

```
EX( cFileMacro ) ⇒ nExitCode
```

Executes the macro file ***cFileName***. The extention must be specified.

cFileMacro may be the name of a "compiled" macro or a text macro file.

```
GET( @aGetList,
     [nTop], [nLeft],
     { |x| iif( pcount() > 0, Var := x, Var ) }
     [cGetPicture], [cColorString],
     [bPreExpression], [bValid] )
```

aGetList	is the get list array that will be increased with this get().
nTop and ***nLeft***	define the starting position of this get object on the screen.
Var	is the variable that is to be edited with this get. ***Var*** is in fact sent to the GET() function using a code block.
cGetPicture	is the get picture to use for ***Var***.
cColorString	is the color string to use for the get.
bPreExpression	is a code block that will be evaluated before the get object will became active. It must result True to obtain that the get object became active.
bValid	is a code block that will be evaluated after the get object is edited. It must result True to obtain that the get object may become inactive.

Create screen editing masks.

GVADD()

«

Get validation add

```
GVADD( @cField, cAdd ) ⇒ .T.
```

cField	the field to fill with more data.
cAdd	is the string to be added to the content of ***cField***.

This function is to be used inside GETs for pre/post validation, when a the content of a field should be added with more data.

cField is returned with the same length as before to avoid troubles with current and future GETs.

GVDEFAULT()

«

Get validation default

```
GVDEFAULT( @cField, cDefault ) ⇒ .T.
```

@cField	the field to check and if empty correct with ***cDefault***.
cDefault	is the default value to be used to replace ***cField***.

This function is to be used inside GETs for pre/post validation, when a field should have a default value.

cField is returned with the same length as before to avoid troubles with current and future GETs.

GVFILEDIR()

«

Get validation file directory

```
GVFILEDIR( @cWildName ) ⇒ .T.
```

cWildName	is the file name taken from the current get to be used for search with DIR().

This function is to be used inside GETs for pre validation: the ***cWildName*** is a file name with wild cards that can be searched with the DIR() function after that a specific key is pressed.

cWildName is returned with the same length as before to avoid troubles with current and future GETs.

GVFILEEXIST()

«

```
GVFILEEXIST( @cNameToTest, [cExtention] ) ⇒ lSuccess
```

@cNameToTest	is the file name taken from the current get to test for existence.
cExtention	is the normal extention of the file.

This function is to be used inside GETs for post validation: the file name have to exist.

cNameToTest is returned with the same length as before to avoid troubles with current and future GETs.

GVFILEEXTENTION()

«

```
GVFILEEXTENTION( @cName, cExt ) ⇒ .T.
```

@cName	the file name to be eventually corrected with file extention.
cExt	the file extention to use as default.

This function is to use inside GETs for pre/post validation, when the content of a field should contain a file name that should be corrected adding a default extention if not given from the user.

GVSUBST()

«

```
GVSUBST( @cField, cSubst ) ⇒ .T.
```

@cField	the field to be replaced with ***cSubst***.
cSubst	is the string to be used to replace the content of ***cField***.

This function is to use inside GETs for pre/post validation, when the content of a field should be replaced with other data.

cField is returned with the same length as before to avoid troubles with current and future GETs.

HTF()

«

```
HTF( [nInitialRecord] ) ⇒ NIL
```

nInitialRecord	is the record number where to start the Help Text File browse. Default is the actual record pointer.

This function browse a Help Text File that must be already opened and be the active alias.

ISFILE()

```
ISFILE( cName ) ⇒ lFileExists
```

cName	is the file name (with or without path) to be checked for existence.

This function returns true (‘**.T.**’) if the file ***cName*** exists. The difference between this function and the standard FILE() function is that ISFILE() checks for wildcards before. If ***cName*** contains wildcards, the result is false (‘**.F.**’).

ISWILD()

```
ISWILD( cName ) ⇒ lIsWild
```

cName	is the file name (with or without path) to be checked for wildcards presence.

This function returns true (‘**.T.**’) if ***cName*** contains wildcards.

ISMEMVAR()

«

```
ISMEMVAR( cName ) ⇒ lIsMemvar
```

cName	is the name of a possible memvar.

This function returns true (‘**.T.**’) if the ***cName*** is a declared Memvar.

ISCONSOLEON()

«

```
ISCONSOLEON() ⇒ lConsoleIsOn
```

This function returns true (‘**.T.**’) if the console will show the result of QOUT() and QQOUT().

ISPRINTERON()

«

```
ISPRINTERON() ⇒ lPrinterIsOn
```

This function returns true (‘**.T.**’) if the default printer will report the the result of QOUT() and QQOUT().

The default printer is ‘`PRN:`’ or ‘`LPT1:`’. If SET ALTERNATE TO is configured to send outputs to ‘`LPT2:`’ or another printer, the function will report false (‘**.F.**’).

KEYBOARD()

«

```
KEYBOARD( [cString] ) ⇒ NIL
```

This function stuff a string into the keyboard buffer.

LISTWINDOW()

«

```
LISTWINDOW( acMenuItem, [cDescription],
      [nTop], [nLeft], [nBottom], [nRight],
      [cColorTop], [cColorBody] ) ⇒ nPosition
```

acMenuItem	is the character array containing the list of choices.
cDescription	is the header to be shown at the top window.
nTop, ***nLeft***, ***nBottom***, ***nRight***	are the window coordinates.
cColorTop	is the color to use for window header and footer.
cColorBody	is the color to use for the window body that is the space where the text appears.

This function is an similar to achoice(), but it shows a header and footer, and it saves the screen, acting like a window.

MEMOWINDOW()

«

```
MEMOWINDOW( cVar, [cDescription], [nTop], [nLeft],
      [nBottom], [nRight], [cColorTop], [cColorBody],
      [lEditMode], [nLineLength], [nTabSize] ) ⇒ cVar
```

cVar	is the character field (variable) to be edited.
cDescription	is the header to be shown at the top window.
nTop, ***nLeft***, ***nBottom***, ***nRight***	are the window coordinates.
cColorTop	is the color to use for window header and footer.
cColorBody	is the color to use for the window body that is the space where the text appears.
lEditMode	is equivalent to memoedit().
nLineLength	is equivalent to memoedit().
nTabSize	is equivalent to memoedit().

This function lets you easily edit a long character field (memo) defining automatically a simple window and providing a simple help.

MEMPUBLIC()

«

```
MEMPUBLIC( cMemvarName | acMemvarNames ) ⇒ NIL
```

cMemvarName	is the name of the PUBLIC variable to create (max 10 characters).
acMemvarNames	is an array of PUBLIC variable names to create (max 10 characters).

Creates a PUBLIC variables or a group of variables.

MEMRELEASE()

«

```
MEMRELEASE( cMemvarName | acMemvarNames ) ⇒ NIL
```

cMemvarName	is the name of the PUBLIC variable to be released.
acMemvarNames	is an array of PUBLIC variable names to be released.

This function releases a previously created PUBLIC variables or a group of variables.

MEMRESTORE()

«

```
MEMRESTORE( cMemFileName, [lAdditive] ) ⇒ NIL
```

cMemFileName	the memory file (.MEM) to load from disk.
lAdditive	if True causes memory variables loaded from the memory file to be added to the existing pool of memory variables. If False, the existing memory variables are automatically released.

Retrieve memory variables form a memory file (.MEM).

MEMSAVE()

«

```
MEMSAVE( cMemFileName, [cSkeleton], [lLike] ) ⇒ NIL
```

cMemFileName	the memory file (.MEM) where public variables should be saved.
cSkeleton	the skeleton mask for defining a group of variables. Wildcard characters may be used: _*_ and _?_.
lLike	if True, the variables grouped with ***cSkeleton*** are saved, else only the other variables are saved.

Saves memory variables to a memory file (.MEM).

MENUPROMPT()

«

```
MENUPROMPT( @aoGet,
      [nRow], [nCol],
      [cPrompt], [bBlock] ) ⇒ NIL
```

aoGet	is an array of get objects where a new get is added by MENUPROMPT(). These gets are read only.
nRow and ***nCol***	are the screen coordinates where the menu prompt will appear.
cPrompt	is the menu prompt string.
bBlock	is the code block to execute when the cursor is on the current menu prompt. It is usually a code block that shows a message somewhere on the screen.

This function should substitute the @...PROMPT command and handle the mouse.

MENUTO()

«

```
MENUTO( aoGet, nPos ) ⇒ nChoice
```

aoGet	array of get objects.
nPos	starting position to be edited.

Like MENU TO. It returns the selected menu item created with MENUPROMPT(). It supports the mouse.

MESSAGELINE()

«

```
MESSAGELINE( [cMessage], [cColor], [nPosTop], [nPosLeft] )
      ⇒ NIL
```

aMessage	the message to be displayed.
cColor	the color string.
nPosTop and ***nPosLeft***	the starting position where the string message would appear on the screen. Default values are respectively ROW() and COL().

MESSAGELINE() is a function that display a message on the screen on the selected position. If ***cMessage*** is NIL, the message is eliminated from screen restoring the previous screen content.

MOUSESCRSAVE()

«

```
MOUSESCRSAVE( [nTop], [nLeft], [nBottom], [nRight] )
      ⇒ cSavedScreen
```

nTop, ***nLeft***, ***nBottom*** and ***nRight***	are the screen coordinates that will be to save the screen.

This function works line SAVESCREEN() but it hide the mouse cursor before a screen save is made.

MOUSESCRRESTORE()

«

```
MOUSESCRRESTORE( [nTop], [nLeft], [nBottom], [nRight],
      [cScreen] ) ⇒ cSavedScreen
```

nTop, ***nLeft***, ***nBottom*** and ***nRight***	are the screen coordinates where the saved screen will be restored.
cScreen	is the previously saved screen to restore.

This function works line RESTSCREEN() but it hide the mouse cursor before a screen restore is made.

PICCHRMAX()

«

```
PICCHRMAX( [nCol], [nMaxCol] ) ⇒ cPictureString
```

nCol	is the starting position on the screen for the get field.
nMaxCol	is the end position on the screen of the get field.

This function is useful when a character field is to be used on a get object. The generated picture will be the of the maximum possible extention, eventually with scroll.

QUIT()

«

```
QUIT() ⇒ NIL
```

Terminates program execution.

READ()

«

```
READ( aoGet, [nPos], [aButtons], [lReadOnly] )
      ⇒ lUpdated
```

aoGet	is the array of get objects.
nPos	is the starting position.
aButtons	is the array of buttons.
lReadOnly	if True, get fields cannot be modified; the default value is False.

This function is made to substitute the READMODAL() allowing the use of the mouse. The array ***aButtons*** is made with the help of the function BUTTON().

RF()

«

```
RF( cFRMName,
      [bForCondition], [bWhileCondition],
      [nNext], [nRecord], [lRest], [lPlain],
      [cbHeading], [lBeforeEject], [lSummary],
      [lDate], [acExtra] ) ⇒ NIL
```

cFRMName	the form (.FRM) file to use to print the active alias.
bForCondition	code block for the FOR condition.
bWhileCondition	code block for the WHILE condition.
nNext	see REPORT FORM.
nRecord	see REPORT FORM
lRest	see REPORT FORM
lPlain	if true (‘`.T.`’), force the print in a simple way.
cbHeading	additional header in character or code block form. If a code block is sent, the final result must be a character string.
lBeforeEject	if true (‘`.T.`’), force a form feed before the print.
lSummary	if true (‘`.T.`’), force a summary print only.
lDate	if false (‘`.F.`’), force the print without date at the top of page.
acExtra	a character array that may be used for translating standard printed report form words and to add vertical and horizontal separations. The default value of acExtra is: ***acExtra***[1] "Page No." ***acExtra***[2] "** Subtotal **" ***acExtra***[3] "* Subsubtotal *" ***acExtra***[4] "*** Total ***" ***acExtra***[5] " " vertical column separation ***axExtra***[6] "" horizontal separation: no separation.

This function does the same work of REPORT FORM or __Report-Form or dbReportForm, but it prints where qout() and qqout() print.

RPT()

«

```
RPT( cText ) ⇒ NIL
```

This function prints the text contained into ***cText*** using print commands. This function accepts other parameters here not described, as they are not to be used for macro purpose. The printing is made using QOUT() and QQOUT(), this way it is sensible to the "alternate" file definition.

RPTMANY()

«

```
RPTMANY( cText, [bWhileCondition], [bForCondition] )
      ⇒ NIL
```

cText	is the text to be printed.
bWhileCondition	is a code block for a WHILE condition to respect for the records to print.
bForCondition	is a code block for a FOR condition to respect for the records to print.

This function prints the text contained into ***cText*** many times: one for every record contained into the active alias.

RPTTRANSLATE()

«

```
RPTTRANSLATE( cText ) ⇒ cTranslatedText
```

This function translates once ***cText*** replacing variables with memvars or Fields.

RUN()

«

```
RUN( cCommand ) ⇒ NIL
```

This function start execution of ***cCommand*** in a DOS session. It works only if there is enough available memory.

SAY()

«

```
SAY( nTop , nLeft , Expr ,
     [ cSayPicture ] , [ cColorString ] ) ⇒ NIL
```

nTop and ***nLeft***	define the starting position on the screen where the ***Expr*** should be displayed.
nLeft	is an expression that will be solved and displayed.
cSayPicture	is the picture to use to display ***Expr***.
cColorString	is the color string to use.

This function displays the result of ***Expr*** on the screen on the desired position.

SETCOLORSTANDARD()

«

```
SETCOLORSTANDARD( [ nColor ] , [ cColor | acColor ] )
      ⇒ cPreviousColor | acPreviousColor
```

nColor	is the color number to take into consideration: 0 All colors 1 Base 2 Menu 3 Head 4 Body (Say - Get) 5 Button (Mouse buttons) 6 Message 7 Alert
cColor	the color string to be associated with ***nColor***.
acColor	it the color array

This function is a way to handle colors inside the application. The functions that display something use a default color depending on what they does. These colors may be changed with SETCOLOR-STANDARD(), all together or only one.

SETFUNCTION()

«

```
SETFUNCTION( nFunctionKey, cString ) ⇒ NIL
```

nFunctionKey	the number of the function key (1=F1, 12=F12) to be assigned.
cString	the character string.

This function assigns a character string to a function key (obsolete).

SETMOUSE()

«

```
SETMOUSE( [lShow] ) ⇒ lPrevious
```

lShow	True shows the mouse cursor, False hide the mouse cursor, NIL reports only the status.

This function is made to show, hide or report only the mouse cursor status.

SETOUTPUT()

«

```
SETOUTPUT( [cPeriperal | aPeripheral] )
      ⇒ aPrevious_Output_Peripherals
```

cPeripheral	is the new output peripheral for qout() and qqout() functions.
aPeripheral	are the new output peripherals configurations for qout() and qqout() functions.

nB is organised in the way to have only one output peripheral at the time. This function help to make order inside SET CONSOLE, SET PRINTER and SET ALTERNATE.

If ***cPeripheral*** contains:

"CON"
SET CONSOLE is set to ON,
SET PRINTER is set to OFF,
SET ALTERNATE is set to OFF;

"PRN"
SET CONSOLE is set to OFF,
SET PRINTER is set to ON,
SET ALTERNATE is set to OFF;

"LPT1"
same as "PRN";

otherwise
SET CONSOLE is set to OFF,
SET PRINTER is set to OFF,
SET ALTERNATE is set to ON,
SET ALTERNATE TO is set to ***cPeripheral***.

aPeripheral is organised this way:

aPeripheral[1] = _SET_CONSOLE

aPeripheral[2] = _SET_PRINTER

aPeripheral[3] = _SET_ALTERNATE

aPeripheral[4] = _SET_ALTFILE

aPeripheral[5] = _SET_EXTRA

aPeripheral[6] = _SET_EXTRAFILE

This function is necessary because SET ALTERNATE alone is not enough to print on the screen when the peripheral name is "CON" or to print on the printer when the peripheral name is "PRN" or "LPT1". In fact, in the first case, ROW() and COL() will not be updated, in the second case, PROW() and PCOL() will not be updated.

This function returns an array organised in the same way as ***aPeripheral*** is, that shows the active output configuration.

SETRPTEJECT()

«

```
SETRPTEJECT( [lbEject] ) ⇒ lPreviousEjectMode
```

This function is used to set the eject mode after every page print for RPT(). If single sheet paper is used, then SETRPTEJECT(.T.) must be set; for continuous paper, SETRPTEJECT(.F.) is correct. The default value is .F..

lbEject	logical or code block, is the eject mode to set. Default is no change, the starting value is '**.F.**'

SETRPTLINES()

«

```
SETRPTLINES() ⇒ nRemainingLines
```

This function is used to report the number of lines available before the completion of the page print for RPT().

SETVERB()

«

Set verbose

```
SETVERB( cSpecifier, [xNewSetting], [lOpenMode] )
      ⇒ xPreviousValueSet
```

cSpecifier	a word that defines the kind of set is going to be considered.
xNewSetting	is the new value to set up.
lOpenMode	used only for some kind of set.

This function is analogue to SET() but it uses a character string (with ***cSpecifier***) and not a number to select the set. This is made to make easier the work with macros.

cSpecifier may contain:

"EXACT"
"FIXED"
"DECIMALS"
"DATEFORMAT"
"EPOCH"
"PATH"
"DEFAULT"
"EXCLUSIVE"
"SOFTSEEK"
"UNIQUE"
"DELETED"
"CANCEL"
"TYPEAHEAD"
"COLOR"
"CURSOR"
"CONSOLE"
"ALTERNATE"
"ALTFILE"
"DEVICE"
"EXTRA"
"EXTRAFILE"
"PRINTER"
"PRINTFILE"
"MARGIN"

"BELL"
"CONFIRM"
"ESCAPE"
"INSERT"
"EXIT"
"INTENSITY"
"SCOREBOARD"
"DELIMITERS"
"DELIMCHARS"
"WRAP"
"MESSAGE"
"MCENTER"

SETVERB("EXACT") (obsolete)

«

```
SETVERB( "EXACT", [lExact] ) ⇒ lPrevious
```

If ***lExact*** is True, it forces exact comparison of character strings, including length. If it is False, character strings are compared until the left string length is exhausted; that is that "" (the null string) is equal to any other string.

Please note that the == operator is a comparison operator for exact match and using it, SETVERB("EXACT", '**.F.**') will not work.

The starting value is True; the recommended value is True.

SETVERB("FIXED")

«

```
SETVERB( "FIXED", [lFixed] ) ⇒ lPrevious
```

If ***lFixed*** contains True, numeric values are displayed ever with a fixed number of decimal digits, depending on the value set by SETVERB("DECIMALS").

The starting value is False.

The recommended value is False: if you have to display a fixed number of decimal digits it is better to define a good display picture.

SETVERB("DECIMALS")

«

```
SETVERB( "DECIMALS", [nDecimals] ) ⇒ nPrevious
```

nDecimals is the number of digits to display after the decimal position. This set is enabled of disabled with SETVERB("FIXED").

The starting value is 8.

SETVERB("DATEFORMAT")

«

```
SETVERB( "DATEFORMAT", [cDateFormat] ) ⇒ cPrevious
```

cDateFormat is a character expression that specifies the date format.

The starting value is "dd/mm/yyyy".

Some date format examples:

AMERICAN	"mm/dd/yyyy"

ANSI	"yyyy.mm.dd"
BRITISH	"dd/mm/yyyy"
FRENCH	"dd/mm/yyyy"
GERMAN	"dd.mm.yyyy"
ITALIAN	"dd-mm-yyyy"
JAPAN	"yyyy/mm/dd"
USA	"mm-dd-yyyy"

SETVERB("EPOCH")

«

```
SETVERB( "EPOCH", [nYear] ) ⇒ nPrevious
```

nYear specifies the base year of 100-year period in which all dates containing only two year digits are assumed to fall.

The starting value is 1900.

SETVERB("PATH")

«

```
SETVERB( "PATH", [cPath] ) ⇒ cPrevious
```

cPath identifies the paths that nB uses when searching for a file not found in the current directory. The list of paths can be separated by commas or semicolons.

The starting value is "".

SETVERB("DEFAULT")

«

```
SETVERB( "DEFAULT", [cPath] ) ⇒ cPrevious
```

cPath identifies the default disk drive and directory.

The starting value is "".

SETVERB("EXCLUSIVE")

«

```
SETVERB( "EXCLUSIVE", [lExclusive] ) ⇒ lPrevious
```

If ***lPath*** is True, the default database (.DBF) file open is made in exclusive mode; in the other case, in shared mode.

The starting value is True.

SETVERB("SOFTSEEK")

«

```
SETVERB( "SOFTSEEK", [lSoftSeek] ) ⇒ lPrevious
```

If ***lSoftSeek*** is True, if a DBSEEK() index search fails, the record pointer is moved to the next record with a higher key. If it is False, in case of a DBSEEK() index search failure, the record pointer is moved at EOF().

The starting value is False.

SETVERB("UNIQUE") (obsolete)

«

```
SETVERB( "UNIQUE", [lUnique] ) ⇒ lPrevious
```

If ***lUnique*** is True, during creation or update of '.DBF' indexes, if two or more records are found with the same key, only the first record will be included inside the index.

If ***lUnique*** is False, duplicated record keys are allowed.

The starting value is False.

SETVERB("DELETED")

«

```
SETVERB( "DELETED", [lDeleted] ) ⇒ lPrevious
```

If ***lDeleted*** is True, record signed for deletion are not filtered, that is, these are still normally visible as they were not deleted. In the other case, they hare (in most cases) hidden to the user.

The starting value is False.

SETVERB("CANCEL")

«

```
SETVERB( "CANCEL", [lCancel] ) ⇒ lPrevious
```

If ***lCancel*** is True, enables [*Alt c*] and [*Ctrl Break*] as termination keys. In the other case, not.

The starting value is True.

SETVERB("TYPEAHEAD")

```
SETVERB( "TYPEAHEAD", [nTypeAhead] ) ⇒ nPrevious
```

nTypeAhead is the number of keystrokes the keyboard buffer can hold from a minimum of zero to a maximum of 4096.

The starting value is 15.

SETVERB("COLOR")

```
SETVERB( "COLOR", [cColorString] ) ⇒ cPrevious
```

nColorString defines the normal screen colors. There are five couple of colors, but only three are really operative:

standard	This is the standard color used for screen output.
enhanced	This is the color used for highlighted screen output.
border	Normally unused.
background	Normally unused.
unselected	This is the color used for GET fields without focus.

The default color string is "BG+/B,N/W,N/N,N/N,W/N" that is:

standard	bright Cyan on Blue
enhanced	Black on White
border	Black on Black
background	Black on Black
unselected	White on Black

The following table explains the use of letters inside the color string. Note that the plus sign (+) means high intensity, the star (*) means blink and that + and * can be allowed only to the first letter inside a couple.

Color	Letter	Monochrome
Black	N, Space	Black
Blue	B	Underline
Green	G	White
Cyan	BG	White
Red	R	White
Magenta	RB	White
Brown	GR	White
White	W	White
Gray	N+	Black
Bright Blue	B+	Bright Underline
Bright Green	G+	Bright White
Bright Cyan	BG+	Bright White
Bright Red	R+	Bright White
Bright Magenta	RB+	Bright White
Bright Brown	GR+	Bright White
Bright White	W+	Bright White
Black	U	Underline
Inverse Video	I	Inverse Video
Blank	X	Blank

SETVERB("CURSOR")

```
SETVERB( "CURSOR", [lCursor] ) ⇒ lPrevious
```

If ***lCursor*** is True, the cursor is showed, else it is hidden.

The starting value is True.

SETVERB("CONSOLE")

```
SETVERB( "CONSOLE", [lConsole] ) ⇒ lPrevious
```

If ***lConsole*** is True, the output of console commands is displayed on the screen, else it is not.

The starting value is True.

SETVERB("ALTERNATE")

```
SETVERB( "ALTERNATE", [lAlternate] ) ⇒ lPrevious
```

If ***lAlternate*** is True, the output of console commands is send also to a standard ASCII text file.

The starting value is False.

SETVERB("ALTFILE")

```
SETVERB( "ALTFILE", [cAlternateFilename], [lAdditive] )
      ⇒ cPrevious
```

If SETVERB("ALTERNATE") is True, the output of the console is send also to ***cAlternateFilename***, a standard ASCII file.

If ***lAdditive*** is True, the output is appended to the ASCII file if it already exists, else it is erased first.

SETVERB("DEVICE")

«

```
SETVERB( "DEVICE", [cDevice] ) ⇒ cPrevious
```

cDevice is the name of the device where SAY() will display its output.

The starting value is "SCREEN", the alternative is "PRINTER".

The recommended value is "SCREEN".

SETVERB("EXTRA")

«

```
SETVERB( "EXTRA", [lExtra] ) ⇒ lPrevious
```

If ***lExtra*** is True, the output of console commands is send also to a standard ASCII text file.

The starting value is False.

SETVERB("EXTRAFILE")

«

```
SETVERB( "EXTRAFILE", [cExtraFilename], [lAdditive] )
      ⇒ cPrevious
```

If SETVERB("EXTRA") is True, the output of the console is send also to ***cExtraFilename***, a standard ASCII file.

If ***lAdditive*** is True, the output is appended to the ASCII file if it already exists, else it is erased first.

SETVERB("PRINTER")

«

```
SETVERB( "PRINTER", [lPrinter] ) ⇒ lPrevious
```

If ***lPrinter*** is True, the output of console commands is also printed, else it is not.

The starting value is False.

SETVERB("PRINTFILE")

«

```
SETVERB( "PRINTFILE", [cPrintFileName] ) ⇒ cPrevious
```

cPrintFileName is the name of the printer peripheral name.

The starting value is "" (null string).

SETVERB("MARGIN")

«

```
SETVERB( "MARGIN", [nPageOffset] ) ⇒ nPrevious
```

nPageOffset is the positive number of column to be used as a left margin for all printer output.

The starting value is 0.

SETVERB("BELL")

«

```
SETVERB( "BELL", [lBell] ) ⇒ lPrevious
```

If ***lBell*** is True, the sound of the bell is used to get the attention of the user when some wrong actions are made.

The starting value is False.

SETVERB("CONFIRM")

«

```
SETVERB( "CONFIRM", [lConfirm] ) ⇒ lPrevious
```

If ***lConfirm*** is False, the GET is simply terminated typing over the end of the get field; in the other case (True), the GET is terminated only pressing an "exit key". The starting value is True.

SETVERB("ESCAPE")

«

```
SETVERB( "ESCAPE", [lEscape] ) ⇒ lPrevious
```

If ***lEscape*** is True, the [*Esc*] key is enabled to be a READ exit key, in the other case not.

The starting value is True.

The recommended value is True.

SETVERB("INSERT")

«

```
SETVERB( "INSERT", [lInsert] ) ⇒ lPrevious
```

If ***lInsert*** is True, the data editing is in INSERT mode, in the other case, it is in OVERWRITE mode.

The starting value is True.

SETVERB("EXIT")

«

```
SETVERB( "EXIT", [lExit] ) ⇒ lPrevious
```

If ***lExit*** is True, [Up] and [Down] key may be used as exit key when the cursor is (respectively) on the first or on the last GET field. In the other case not.

The starting value is False.

The recommended value is False.

SETVERB("INTENSITY")

«

```
SETVERB( "INTENSITY", [lIntensity] ) ⇒ lPrevious
```

If ***lIntensitiy*** is True, the display of standard and enhanced display colors are enabled. In the other case, only standard colors are enabled.

The starting value is True.

The recommended value is True.

SETVERB("SCOREBOARD")

«

```
SETVERB( "SCOREBOARD", [lScoreboard] ) ⇒ lPrevious
```

If ***lScoreboard*** is True, the display of messages from READ() and MEMOREAD() is allowed; in the order case not.

The starting value is False.

The recommended value is False: nB do not support scoreboard.

SETVERB("DELIMITERS")

«

```
SETVERB( "DELIMITERS", [lDelimiters] ) ⇒ lPrevious
```

If ***lDelimiters*** is True, GET variables appear on the screen delimited with the delimiter symbols. In the other case, GET variables are not delimited this way, but only with the use of different colors.

The starting value is False.

The recommended value is False: the use of delimiters creates one more trouble when designing a screen mask.

SETVERB("DELIMCHARS")

«

```
SETVERB( "DELIMCHARS", [cDelimterCharacters] ) ⇒ cPrevious
```

cDelimterCharacters are the delimiter characters used to delimit a GET field when SETVERB("DELIMITERS") is True.

The starting value is "::".

SETVERB("WRAP")

«

```
SETVERB( "WRAP", [lWrap] ) ⇒ lPrevious
```

If ***lWrap*** is True, the wrapping of the highlight in MENUs should be active, but this option is actually not active and all works as it is False.

The starting value is False.

SETVERB("MESSAGE")

«

```
SETVERB( "MESSAGE", [nMessageRow] ) ⇒ nPrevious
```

nMessageRow is the row number where the @..PROMPT message line should appear on the screen. This option is not supported.

The starting value is 0.

SETVERB("MCENTER")

«

```
SETVERB( "MCENTER", [lMessageCenter] ) ⇒ lPrevious
```

If ***lMessageCenter*** is True, the @..PROMPT message line should appear centered on the screen. This option is not supported.

The starting value is False.

STRADDEXTENTION()

«

```
STRADDEXTENTION( cName , cExt ) ⇒ cCompleteName
```

cName	the file name (with or without path) that is probably without extention.
cExt	the extention that must be added to ***cName*** if it has not one.

This function check ***cName*** for the presence of an extention. It it has not one, ***cExt*** will be added.

STRCUTEXTENTION()

«

```
STRCUTEXTENTION( cName ) ⇒ cName
```

cName	the file name (with or without path) that is probably with extention.

This function check ***cName*** for the presence of an extention. It it has one, the extention is removed.

STRDRIVE()

«

```
STRDRIVE( cName ) ⇒ cDrive
```

cName	the file name (with or without path) that contains the drive letter.

This function tries to extract the drive letter information from

cName.

STREXTENTION()

«

```
STREXTENTION( cName ) ⇒ cExtention
```

cName	the file name (with or without path) that contains an extention.

This function tries to extract the extention information from ***cName***.

STRFILE()

«

```
STRFILE( cName ) ⇒ cFileName
```

cName	the file name with or without path.

This function tries to extract the file name without path from ***cName***.

STRFILEFIND()

«

```
STRFILEFIND( cName, cPath ) ⇒ cFileName
```

cName	the file name or pathname containing the file name to search inside the ***cPath*** list.
cPath	a list of paths separated with semicolon (just like Dos does), where ***cFile*** should be searched.

If your file is to be found on different possible positions, this func-

tion search the first place where the file is found and returns a valid pathname to that file.

STRGETLEN()

«

```
STRGETLEN( xExpr, cPicture ) ⇒ nFieldLength
```

xExpr	a generic expression.
cPicture	the picture string.

This function returns the length of field when using ***xExpr*** with ***cPicture***.

STRLISTASARRAY()

«

```
STRLISTASARRAY( cList, [cDelimiter] ) ⇒ aList
```

cList	a character string containing a list separated with ***cDelimiter***.
cDelimiter	the delimiter used to separate the elements contained inside the list.

This function transform a character string list into an array.

STROCCURS()

«

```
STROCCURS( cSearch, cTarget ) ⇒ nOccurrence
```

cSearch	the search string to find inside ***cTarget***.
cTarget	the string to be searched for the presence of ***cSearch***.

This function returns the number of occurrence that ***cSearch*** is contained inside ***cTarget***.

STRPARENT()

«

```
STRPARENT( cName ) ⇒ cParentPath
```

cName	the pathname.

This function tries to return a parent path from ***cName***.

STRPATH()

«

```
STRPATH( cName ) ⇒ cPath
```

cName	the pathname.

This function tries to extract the path from ***cName***.

STRTEMPPATH()

«

```
STRTEMPPATH() ⇒ cTempPath
```

This function returns a temporary path searching for possible definitions inside the environmental variables.

STRXTOSTRING()

«

```
STRXTOSTRING( xVar, [cType] ) ⇒ cTrasformed_to_string
```

xVar	is the data of any type to be converted into string.
cType	is the type of the data contained inside ***xVar***.

This function returns ***xVar*** transformed into a character string.

TB()

«

```
TB( [nTop], [nLeft], [nBottom], [nRight],
    [acCol], [acColSayPic],
    [acColTopSep], [acColBodySep], [acColBotSep],
    [acColHead], [acColFoot],
    [alColCalc],
    [abColValid],
    [abColMsg],
    [cColor], [abColColors],
    [nFreeze],
    [lModify],
    [lAppend],
    [lDelete],
    [lButtons | aButtons] ) ⇒ NIL
```

nTop, ***nLeft***, ***nBottom***, ***nRight*** defines the screen area where browse have to take place.

acCol	is the columns array to be included into the browse.
acColSayPic	is the picture array.
acColTopSep	is the top separation array: default is chr(194)+chr(196).
acColBodySep	is the body separation array: default is chr(179).
acColBotSep	is the bottom separation array: default is chr(193)+chr(196).
acColHead	is the header array for every column.
acColFoot	is the footer array for every column.
alColCalc	is the array that identify the calculated column (not editable). True (‘**.T.**’) means calculated.
abColValid	is the validation array that specify when a field is properly filled. The condition must be specified in code block format.
abColMsg	is the message array that permits to show information at the bottom of browse area. The array must be composed with code blocks which result with a character string.
cColor	is the color string: it may be longer than the usual 5 elements.
abColColors	is the color code block array. The code block receive as parameter the value contained inside the field and must return an array containing two numbers: they correspond to the two color couple from ***cColor***.
nFreeze	indicates the number of columns to be left frozen on the left side.
lModify	indicates whether the browse can modify data.

lDelete	indicates whether the browse can delete and recall records.
lButtons	if True, default buttons are displayed.
aButtons	array of buttons.
aButtons[***n***][1] N	the ***n***th button row position;
aButtons[***n***][2] N	the ***n***th button column position;
aButtons[***n***][3] C	the ***n***th button text;
aButtons[***n***][4] B	the ***n***th button code block.

This function, called without parameters, starts the browse of the active alias, and if relations are established, the browse includes also related data.

Please note that due to an unresolved problem, the field names contained inside ***acCol*** should better contain also the alias (ALIAS->FIELD_NAME). See also the examples.

TEXT()

«

```
TEXT( cText ) ⇒ NIL
```

Shows the text contained into ***cText***.

TGLINSERT()

«

```
TGLINSERT() ⇒ NIL
```

Toggle the global insert mode and the cursor shape.

TIMEX2N()

«

```
TIMEX2N( [nHH], [nMM], [nSS] ) ⇒ nTime
```

nHH	is the number of hours.
nMM	is the number of minutes.
nSS	is the number of seconds.

This function calculate the "time number" that is a number representing days and/or portion of a day: 1 is 1 day or 24 hours, 0.5 is 12 hours, and so on.

TIMEN2H()

«

```
TIMEN2H( nTime ) ⇒ nHours
```

nTime	is the "time number" that is a number representing days and/or portion of a day: 1 is 1 day or 24 hours, 0.5 is 12 hours, and so on.

This function returns the integer number of hours contained inside ***nTime***.

TIMEN2M()

«

```
TIMEN2M( nTime ) ⇒ nMinutes
```

nTime	is the "time number" that is a number representing days and/or portion of a day: 1 is 1 day or 24 hours, 0.5 is 12 hours, and so on.

This function returns the integer number of minutes contained inside ***nTime*** after subtracting the hours.

TIMEN2S()

«

```
TIMEN2S( nTime ) ⇒ nSeconds
```

nTime	is the "time number" that is a number representing days and/or portion of a day: 1 is 1 day or 24 hours, 0.5 is 12 hours, and so on.

This function returns the number of seconds (with eventual decimals) contained inside ***nTime*** after subtracting the hours and the minutes.

TRUESETKEY()

«

```
TRUESETLEY( nInkeyCode, bAction ) ⇒ .T.
```

This function is equivalent to SETKEY() but it returns always '**.T.**'

WAITFILEEVAL()

«

```
WAITFILEEVAL( lClose ) ⇒ .T.
```

Shows a wait bar calling WAITPROGRESS() for operation on records of a database.

If there is no index active, it is equivalent to WAITPROGRES(RECNO()/LASTREC()).

if an index is active, this cannot work, so an increment for each call is made: WAITPROGRES((nIncrement++)/LASTREC()).

This function must be closed calling it with the ***lClose*** parameter to true (‘**.T.**’). This way, internal counters are closed and WAITPROGRESS() is closed too.

WAITFOR()

«

```
WAITFOR( [cMessage] ) ⇒ NIL
```

Shows ***cMessage*** until it is called again. The wait window is closed when called without parameter or with NIL.

WAITPROGRESS()

«

```
WAITPROGRESS( [nPercent] ) ⇒ .T.
```

Shows a wait bar on the screen top depending on the value contained into ***nPercent***. ***nPercent*** starts form 0 and ends to 1 (100%). If a value of one or more, or NIL is passed, the wait window is closed.

Normal command substitution

«

Clipper works only with functions and commands that are converted into function using the ‘STD.CH’. Here are described some command replacement that can be used also with nB macros.

?

```
? [exp_list]
```

```
qout([exp_list])
```

```
?? [exp_list]
```

```
qqout([exp_list])
```

@BOX

```
@ nTop, nLeft, nBottom, nRight BOX cnBoxString [COLOR cColorString]
```

```
dispbox(nTop, nLeft, nBottom, nRight, [cnBoxString], [cColorString])
```

@TO

```
@ nTop, nLeft TO nBottom, nRight DOUBLE [COLOR cColorString]
```

```
dispbox(nTop, nLeft, nBottom, nRight, 2 [,cColorString])
```

```
@ nTop, nLeft TO nBottom, nRight [COLOR cColorString]
```

```
dispbox(nTop, nLeft, nBottom, nRight, 1 [,cColorString])
```

```
@ nTop, nLeft CLEAR [TO nBottom, nRight]
```

```
scroll([nTop], [nLeft], [nBottom, nRight])
```

```
setpos(nRow, nCol)
```

@GET

```
@ nTop, nLeft GET Var [PICTURE cGetPicture] [COLOR cColorString]
[WHEN lPreExpression] ↩
↪[VALID lPostExpression]
```

```
setpos(nTop, nLeft)
```

```
aadd( GetList, _GET_( Var, "Var", cGetPicture, [
{|| lPostExpression}], ↩
↪[{|| lPreExpression}] ):display() ) atail(GetList):colorDisp(cCo
```

@SAY

```
@ nTop, nLeft SAY exp [COLOR cColorString]
```

```
devpos(nTop, nLeft)
```

```
devout(exp [, cColorString])
```

```
@ nTop, nLeft SAY exp PICTURE cSayPicture [COLOR cColorString]
```

```
devpos(nTop, nLeft)
```

```
devoutpic(exp, cSayPicture, [cColorString])
```

APPEND

```
APPEND BLANK
```

```
dbappend()
```

CLEAR

```
CLEAR
```

```
Scroll()
```

```
SetPos(0,0)
```

```
ReadKill(.T.)
```

```
GetList := {}
```

```
CLEAR GETS
```

```
ReadKill(.T.)
```

```
GetList := {}
```

```
CLEAR SCREEN | CLS
```

```
Scroll()
```

```
SetPos(0,0)
```

CLOSE

```
CLOSE
```

```
dbCloseArea()
```

```
CLOSE idAlias
```

```
idAlias->( dbCloseArea() )
```

```
CLOSE ALTERNATE
```

```
Set(19, "")
```

```
CLOSE DATABASES
```

```
dbCloseAll()
```

```
CLOSE INDEXES
```

```
dbClearIndex()
```

COMMIT

```
COMMIT
```

```
dbCommitAll()
```

COUNT

```
COUNT TO idVar [FOR lForCondition] [WHILE lWhileCondition] [
NEXT nNextRecords] ↩
↪ [RECORD nRecord] [REST] [ALL]
```

```
dbeval( {||idVar:=idVar+1}, {||lForCondition}, {||lWhileCondition}, ↩
↪nNextRecords, nRecord, lRest )
```

DEFAULT

```
DEFAULT xVar TO xDefaultValue
```

```
DEFAULT( @xVar, xDefaultValue ) ⇒ xVar
```

DELETE

```
DELETE
```

```
dbDelete()
```

```
DELETE [FOR lForCondition] [WHILE lWhileCondition] [
NEXT nNextRecords] ↩
↪[RECORD nRecord] [REST] [ALL]
```

```
dbeval( {||dbDelete()}, {||lForCondition}, {||lWhileCondition}, ↩
↪nNextRecords, nRecord, lRest )
```

```
DELETE FILE xcFile
```

```
ferase( cFile )
```

EJECT

```
EJECT
```

```
qqout( chr(13) )
```

ERASE

```
ERASE xcFile
```

```
ferase( cFile )
```

FIND

```
FIND xcSearchString
```

```
dbSeek( cSearchString )
```

GO

```
GO [TO] nRecord
```

```
dbgoto(nRecord)
```

```
GO [TO] BOTTOM
```

```
dbGoBottom()
```

```
GO [TO] TOP
```

```
dbgotop()
```

INDEX ON

```
INDEX ON expKey TO xcIndexName [UNIQUE] [FOR lForCondition] ↩
↪[WHILE lWhileCondition] [[EVAL lEvalCondition] [EVERY nRecords
]] [ASCENDING|DESCENDING]
```

```
ordCondSet( [cForCondition], [bForCondition], , [bWhileCondition],↩
↪[bEvalCondition], [nRecords], RECNO(), , , , lDescending )
```

```
ordCreate( cIndexName, , cExpKey, bExpKey, lUnique )
```

READ

```
READ
```

```
ReadModal(GetList)
```

```
GetList := {}
```

```
READ SAVE
```

```
ReadModal(GetList)
```

RECALL

```
RECALL
```

```
dbRecall()
```

```
RECALL [FOR lForCondition] [WHILE lWhileCondition] [
NEXT nNextRecords] ↩
↪[RECORD nRecord] [REST] [ALL]
```

```
dbeval( {||dbRecall()}, {||lForCondition}, {||lWhileCondition},↩
↪nNextRecords, nRecord, lRest )
```

REINDEX

```
REINDEX [EVAL lEvalCondition] [EVERY nRecords]
```

```
ordCondSet(, , , , [bEvalCondition], [nRecords]
, , , , , , , , )
```

```
ordListRebuild()
```

RENAME

```
RENAME xcOldFile TO xcNewFile
```

```
frename( cOldFile, cNewFile )
```

REPLACE

```
REPLACE idField1 WITH exp1 [, idField2 WITH exp2...] ↩
↪[FOR lForCondition] [WHILE lWhileCondition] [NEXT nNextRecords]
↩
↪[RECORD nRecord] [REST] [ALL]
```

```
dbeval( {|| idField1 := exp1 [, idField2 := exp2...] }, ↩
↪{||lForCondition}, {||lWhileCondition}, nNextRecords, ↩
↪nRecord, lRest )
```

```
REPLACE idField1 WITH exp1
```

```
idField1 := exp1
```

RESTORE

```
RESTORE SCREEN FROM cScreen
```

```
restscreen( 0, 0, Maxrow(), Maxcol(), cScreen )
```

SAVE

```
SAVE SCREEN TO cScreen
```

```
cScreen := savescreen( 0, 0, maxrow(), maxcol() )
```

SEEK

```
SEEK expSearch [SOFTSEEK]
```

```
dbSeek( expSearch [, lSoftSeek] )
```

SELECT

```
SELECT xnWorkArea | idAlias
```

```
dbSelectArea( nWorkArea | cIdAlias )
```

SET

```
SET ALTERNATE TO xcFile [ADDITIVE]
```

```
Set( 19, cFile, lAdditive )
```

```
SET ALTERNATE ON | OFF | xlToggle
```

```
Set( 18, "ON" | "OFF" | lToggle )
```

```
SET BELL ON | OFF | xlToggle
```

```
Set( 26, "ON" | "OFF" | lToggle )
```

```
SET COLOR | COLOUR TO (cColorString)
```

```
SetColor( cColorString )
```

```
SET CONFIRM ON | OFF | xlToggle
```

```
Set( 27, "ON" | "OFF" | lToggle )
```

```
SET CONSOLE ON | OFF | xlToggle
```

```
Set( 17, "ON" | "OFF" | lToggle )
```

```
SET CURSOR ON | OFF | xlToggle
```

```
SetCursor( 1 | 0 | iif( lToggle, 1, 0 ) )
```

```
SET DATE FORMAT [TO] cDateFormat
```

```
Set( 4, cDateFormat )
```

```
SET DECIMALS TO
```

```
Set( 3, 0 )
```

```
SET DECIMALS TO nDecimals
```

```
Set( 3, nDecimals )
```

```
SET DEFAULT TO
```

```
Set( 7, "" )
```

```
SET DEFAULT TO xcPathspec
```

```
Set( 7, cPathspec )
```

```
SET DELETED ON | OFF | xlToggle
```

```
Set( 11, "ON" | "OFF" | lToggle )
```

```
SET DELIMITERS ON | OFF | xlToggle
```

```
Set( 33, "ON" | "OFF" | lToggle )
```

```
SET DELIMITERS TO [DEFAULT]
```

```
Set( 34, "::" )
```

```
SET DELIMITERS TO cDelimiters
```

```
Set( 34, cDelimiters )
```

```
SET DEVICE TO SCREEN | PRINTER
```

```
Set( 20, "SCREEN" | "PRINTER" )
```

```
SET EPOCH TO nYear
```

```
Set( 5, nYear )
```

```
SET ESCAPE ON | OFF | xlToggle
```

```
Set( 28, "ON" | "OFF" | lToggle )
```

```
SET EXACT ON | OFF | xlToggle
```

```
Set( 1, "ON" | "OFF" | lToggle )
```

```
SET EXCLUSIVE ON | OFF | xlToggle
```

```
Set( 8, "ON" | "OFF" | lToggle )
```

```
SET FILTER TO
```

```
dbclearfilter()
```

```
SET FILTER TO lCondition
```

```
dbsetfilter( bCondition, cCondition )
```

```
SET FIXED ON | OFF | xlToggle
```

```
Set( 2, "ON" | "OFF" | lToggle )
```

```
SET INDEX TO [xcIndex [, xcIndex1...] ]
```

```
ordListClear()
ordListAdd( cIndex )
ordListAdd( cIndex1 )
…
```

```
SET INTENSITY ON | OFF | xlToggle
```

```
Set( 31, "ON" | "OFF" | lToggle )
```

```
SET KEY nInkeyCode [TO]
```

```
SetKey( nInkeyCode, NIL )
```

```
SET KEY nInkeyCode TO [idProcedure]
```

```
SetKey( nInkeyCode, { |p, l, v| idProcedure(p, l, v)} )
```

```
SET MARGIN TO
```

```
Set( 25, 0 )
```

```
SET MARGIN TO [nPageOffset]
```

```
Set( 25, nPageOffset )
```

```
SET MESSAGE TO
```

```
Set( 36, 0 )
```

```
Set( 37, .F. )
```

```
SET MESSAGE TO [nRow [CENTER | CENTRE]]
```

```
Set( 36, nRow )
```

```
Set( 37, lCenter )
```

```
SET ORDER TO [nIndex]
```

```
ordSetFocus( nIndex )
```

```
SET PATH TO
```

```
Set( 6, "" )
```

SET PATH TO [*xcPathspec* [, *cPathspec1*...]]

Set(6, *cPathspec* [, *cPathspec1*...])

SET PRINTER ON | OFF | xlToggle

Set(23, "ON" | "OFF" | lToggle)

SET PRINTER TO

Set(24, "")

SET PRINTER TO [*xcDevice* | *xcFile* [ADDITIVE]]

Set(24, *cDevice* | *cFile*, lAdditive)

SET RELATION TO

dbclearrelation()

```
SET RELATION TO [expKey1 INTO xcAlias1]
     [, [TO] expKey2 INTO xcAlias2...]
     [ADDITIVE]
```

```
if !lAdditive
     dbClearRel()
end
dbSetRelation( cAlias1, {|| expKey1}, ["expKey1"] )
dbSetRelation( cAlias2, {|| expKey2}, ["expKey1"] )
```

```
SET SCOREBOARD ON | OFF | xlToggle
```

```
Set( 32, "ON" | "OFF" | lToggle )
```

```
SET SOFTSEEK ON | OFF | xlToggle
```

```
Set( 9, "ON" | "OFF" | lToggle )
```

```
SET TYPEAHEAD TO nKeyboardSise
```

```
Set( 14, nKeyboardSise )
```

```
SET UNIQUE ON | OFF | xlToggle
```

```
Set( 10, "ON" | "OFF" | lToggle )
```

```
SET WRAP ON | OFF | xlToggle
```

```
Set( 35, "ON" | "OFF" | lToggle )
```

SKIP

```
SKIP [nRecords] [ALIAS idAlias | nWorkArea]
```

```
[idAlias | nWorkArea -> ] ( dbSkip( [nRecords] ) )
```

STORE

```
STORE value TO variable
```

```
variable := value
```

SUM

```
SUM nExp1 [, nExp2...] TO idVar1 [, idVar2...] [FOR lForCondition]
↩
↪[WHILE lWhileCondition] [NEXT nNextRecords] [RECORD nRecord]
[REST] [ALL]
```

```
dbeval( {||idVar1:=idVar1+nExp1 [, idVar2:=idVar2+nExp2...] },↩
↪{||lForCondition}, {||lWhileCondition}, nNextRecords, nRecord, lRest )
```

UNLOCK

```
UNLOCK
```

```
dbUnlock()
```

```
UNLOCK ALL
```

```
dbUnlockAll()
```

USE

```
USE
```

```
dbclosearea()
```

```
USE [xcDatabase] ↩
↪[INDEX xcIndex1 [, xcIndex2...] [ALIAS xcAlias] [EXCLUSIVE|
SHARED] ↩
↪[NEW] [READONLY] [VIA cDriver] ]
```

```
dbUseArea( [lNewArea], [cDriver], cDatabase, [cAlias], [lShared
], [lReadOnly] )
[dbSetIndex( cIndex1 )]
[dbSetIndex( cIndex2 )]
...
```

nB command substitution functions

«

Inside nB there are many functions made only in substitution to other Clipper commands.

GET

```
@ nTop, nLeft GET Var
      [PICTURE cGetPicture]
      [COLOR cColorString]
      [WHEN lPreExpression]
      [VALID lPostExpression]
```

```
Get( @aGetList,
      [nTop], [nLeft],
      { |x| iif( pcount() > 0, Var := x, Var ) }
      [cGetPicture], [cColorString],
      [bPreExpression], [bValid] )
```

aGetList	is the get list array that will be increased with this get().

SAY

```
@ nTop, nLeft SAY exp
      PICTURE cSayPicture
      [COLOR cColorString]
```

```
Say( nTop, nLeft, cVar, [cSayPicture], [cColorString] )
```

APPEND FROM

```
APPEND FROM xcFile
      [FIELDS idField_list]
      [scope]
      [WHILE lCondition]
      [FOR lCondition]
      [VIA xcDriver]
```

```
dbApp( cFileName, [acFields],
      [bForCondition], [bWhileCondition],
      [nNextRecords],
      [nRecord],
      [lRest],
      [cDriver] )
```

```
APPEND FROM xcFile
      [FIELDS idField_list]
      [scope]
      [WHILE lCondition]
      [FOR lCondition]
     DELIMITED xcDelimiter
```

```
dbDelim( .f., cFileName, [cDelimiter], [acFields],
      [bForCondition], [bWhileCondition],
      [nNextRecords], [nRecord], [lRest] )
```

```
APPEND FROM xcFile
      [FIELDS idField_list]
      [scope]
      [WHILE lCondition]
      [FOR lCondition]
     SDF
```

```
dbSDF( .f., cFileName, [acFields],
      [bForCondition], [bWhileCondition],
      [nNextRecords], [nRecord], [lRest] )
```

CONTINUE

```
CONTINUE
```

```
dbContinue()
```

COPY

```
COPY FILE xcSourceFile TO xcTargetFile|xcDevice
```

```
CopyFile( cSourceFile, cTargetFile|cDevice )
```

```
COPY STRUCTURE [FIELDS idField_list]
     TO xcDatabase
```

```
dbCopyStruct( cDatabase, [acFields] )
```

```
COPY STRUCTURE EXTENDED
     TO xcExtendedDatabase
```

```
dbCopyXStruct( cExtendedDatabase )
```

```
COPY TO xcFile
     [FIELDS idField_list]
     [scope]
     [WHILE lCondition]
     [FOR lCondition]
     [VIA xcDriver]
```

```
dbCopy( cFileName, [acFields],
      [bForCondition], [bWhileCondition],
      [nNextRecords],
      [nRecord],
      [lRest],
      [cDriver] )
```

```
COPY TO xcFile
      [FIELDS idField_list]
      [scope]
      [WHILE lCondition]
      [FOR lCondition]
     DELIMITED xcDelimiter
```

```
dbDelim( .t., cFileName, [cDelimiter], [acFields],
      [bForCondition], [bWhileCondition],
      [nNextRecords], [nRecord], [lRest] )
```

```
COPY TO xcFile
      [FIELDS idField_list]
      [scope]
      [WHILE lCondition]
      [FOR lCondition]
     SDF
```

```
dbSDF( .t., cFileName, [acFields],
       [bForCondition], [bWhileCondition],
       [nNextRecords], [nRecord], [lRest] )
```

CREATE

```
CREATE xcDatabase
       FROM xcExtendedDatabase
       [NEW]
       [ALIAS cAlias]
       [VIA cDriver]
```

```
dbOldCreate( cDatabase, cExtendedDatabase,
       [cDriver], [lNew], [cAlias] )
```

JOIN

```
JOIN WITH xcAlias TO xcDatabase
       [FOR lCondition] [FIELDS idField_list]
```

```
dbJoin( cAlias, cDatabase,
       [acFields], [bForCondition] )
```

KEYBOARD

```
KEYBOARD cString
```

```
Keyboard( [cString] ) ⇒ NIL
```

LABEL FORM

```
LABEL FORM xcLabel
      [TO PRINTER]
      [TO FILE xcFile]
      [NOCONSOLE]
      [scope]
      [WHILE lCondition]
      [FOR lCondition]
      [SAMPLE]
```

```
dbLabelForm( cLabel, [lToPrinter], [cFile],
      [lNoConsole], [bForCondition], [bWhileCondition],
      [nNextRecords], [nRecord], [lRest], [lSample] )
```

LIST

```
LIST exp_list
      [TO PRINTER]
      [TO FILE xcFile]
      [scope]
      [WHILE lCondition]
      [FOR lCondition]
      [OFF]
```

```
dbList( [lToDisplay], abListColumns,
      [lAll],
      [bForCondition], [bWhileCondition],
      [nNextRecords], [nRecord], [lRest],
      [lToPrinter], [cFileName] )
```

LOCATE

```
LOCATE [scope] FOR lCondition
      [WHILE lCondition]
```

```
dbLocate( [bForCondition], [bWhileCondition],
      [nNextRecords], [nRecord], [lRest] )
```

PACK

```
PACK
```

```
dbPack()
```

PUBLIC

```
PUBLIC idMemvar
```

```
MemPublic( cMemvarName|acMemvarNames )
```

QUIT

```
QUIT
```

```
Quit()
```

RELEASE

```
RELEASE idMemvar
```

```
MemRelease( cMemvarName|acMemvarNames )
```

REPORT FORM

```
REPORT FORM xcReport
      [TO PRINTER]
      [TO FILE xcFile]
      [NOCONSOLE]
      [scope]
      [WHILE lCondition]
      [FOR lCondition]
      [PLAIN | HEADING cHeading]
      [NOEJECT] [SUMMARY]
```

```
RF( cForm,
      [bForCondition], [bWhileCondition],
      [nNext], [nRecord], [lRest], [lPlain],
      [cbHeading], [lBeforeEject], [lSummary],
      [lDate], [acExtra] ) ⇒ NIL
```

RESTORE FROM

```
RESTORE FROM xcMemFile [ADDITIVE]
```

```
MemRestore( cMemFileName, [lAdditive] )
```

RUN

```
RUN xcCommandLine
```

```
Run( cCommand )
```

SAVE TO

```
SAVE TO xcMemFile
     [ALL [LIKE|EXCEPT skeleton]]
```

```
MemSave( cMemFileName, [cSkeleton], [lLike] )
```

SET FUNCTION

```
SET FUNCTION nFunctionKey TO cString
```

```
SetFunction( nFunctionKey , cString )
```

SORT

```
SORT TO xcDatabase
     ON idField1 [/[A|D] [C] ]
     [, idField2 [/[A|D] [C] ] …]
     [scope]
     [WHILE lCondition]
     [FOR lCondition]
```

```
dbSort( cDatabase , [acFields] ,
     [bForCondition] , [bWhileCondition] ,
     [nNextRecords] , [nRecord] , [lRest] )
```

TOTAL

```
TOTAL ON expKey
     [FIELDS idField_list] TO xcDatabase
     [scope]
     [WHILE lCondition]
     [FOR lCondition]
```

```
dbTotal( cDatabase , bKey , [acFields ,
     [bForCondition] , [bWhileCondition] ,
     [nNextRecords] , [nRecord] ] , [lRest] )
```

UPDATE

```
UPDATE FROM xcAlias
      ON expKey  [RANDOM]
      REPLACE idField1 WITH exp
      [, idField2 WITH exp ...]
```

```
dbUpdate( cAlias, bKey, [lRandom], [bReplacement] )
```

Example:

```
     dbUpdate( "INVOICE", {|| LAST}, .T.,;
          {|| FIELD->TOTAL1 := INVOICE->SUM1,;
          FIELD->TOTAL2 := INVOICE->SUM2 } )
```

ZAP

```
ZAP
```

```
dbZap()
```

RPT: the nB print function

The function RPT() helps to print ASCII file containing Memvars, Fields and print commands. RPT() is accessible from the DOC() menu.

Memvars and fields

As usual with standard word processors, variables are written delimited with "<" (Alt+174) and ">" (Alt+175).

Inside these delimiters can find place character Memvars, character Fields and functions giving a character result.

The RPT() function generates a public variable n_Lines that contains the available lines inside the actual sheet. Every time a line is written, this value is reduced, until a new page is reached and then it will start again from the maximum value. It is useful to read this variable to determinate if there is enough space or it is better to change page.

Commands

«

The function RPT() recognise some print commands. These commands starts with the asterisk (*) symbol. This means that "*" is a print command prefix.

It follows the command syntax.

*COMMAND

```
*COMMAND
        cStatement
        cStatement
        ...
*END
```

The lines contained inside *COMMAND - *END are executed with the nB macro interpreter.

*DBSKIP

```
*DBSKIP [nSkip]
```

It Executes a dbskip() on the active alias.

*FOOT

```
*FOOT
      cFooter
      cFooter
      ...
*END
```

The lines contained inside *FOOT - *END are printed each time at the bottom of pages.

*HEAD

```
*HEAD
      cHeader
      cHeader
      ...
*END
```

The lines contained inside *HEAD - *END are printed each time at the top of pages.

*IF

```
*IF lCondition
     ...
     ...
*END
```

If the condition ***lCondition*** is true, the lines contained inside *IF - *END are printed.

*INSERT

```
*INSERT cFileName
```

Includes the text contained into the file ***cFileName***.

*LEFT

```
*LEFT nLeftBorder
```

The ***nLeftBorder*** is the number of column to be left blank as a left border.

*LPP

```
*LPP nLinesPerPage
```

It determinates the page length expressed in lines. After printing the ***nLinesPerPage***th line, a form feed is sent.

*NEED

```
*NEED nLinesNeeded
```

If the available lines are less then ***nLinesNeeded***, the follwing text will be printed on the next page.

*PA

```
*PA
```

Jumps to a new page.

*REM

```
*REM | *COMMENT [comment_line]
```

It adds a comment that will not be printed.

*WHILE

```
*WHILE lCondition
    ...
    ...
*END
```

The lines contained inside *WHILE - *END are printed as long as ***lCondition*** is true.

Examples

It follows some example of text to be printed with the RPT() function. Example's lines are numbered. Line numbers must not be part of a real RPT text files.

PAGE DEFINITION

Margins are defined with *HEAD, *FOOT and *LEFT commands. In the following example is defined:

```
Top             2 lines;

Bottom          2 lines;

Left            10 characters.
```

The right margin is not defined as it depends on the lines length that will be printed.

The only considered page dimension is the height, *LPP (lines per page):

```
Page height    66 lines.
```

Here starts the example:

```
001  *lpp 66
002  *head
003
004
005  *end
006  *foot
007
008
009  *end
010  *left 10
011  ... text text text
012  ... test text text
...
```

At line 001 is defined the page height in lines. At line 002 is defined the header; it contains two empty lines (003 and 004) which will be printed at the top of every page. At line 006 starts the footer definition that contains two empty lines (007 and 008) that will be printed at the end of every page. At line 010 is defined the space on the left that will be added to every line printed. From line 011 starts the normal text.

HEADER AND FOOTER

The commands *HEAD and *FOOT are used to define the top and bottom border if they contains empty lines, it these lines are not empty, they became real head and foot.

The dimensions are as it follows:

```
Top             6 lines (should be one inch);

Bottom          6 lines;

Left            10 characters (should be an inch).

Page height     66 lines (should be 11 inch).
```

At position 0.5 in (after 3 lines) a one line header appears.

```
001  *lpp 66
002  *head
003
004
005
006  ------------------- MYFILE.TXT -------------------
007
008
009  *end
010  *foot
011
012
013
014
015
016
017  *end
018  *left 10
019  ... text text text
020  ... test text text
...
```

At line 006 (the fourth header line) a text appears. It will be printed on every page at the absolute fourth page line.

CODE INSERTION

Pieces of code can be inserted inside *COMMAND - *END. It can be useful to make complicated reports.

The following example declares a public variable used to number pages.

```
001  *command
002  mempublic("PageNo")
003  PageNo := 0
004  *end
005  *lpp 66
006  *head
007  *command
008  PageNo := PageNo +1
009  *end
010
011
012  *end
013  *foot
014
015                                  Page <PageNo>
016
017  *end
018  *left 10
019  ... text text text
020  ... test text text
...
```

At line 001 starts a *COMMAND definition: lines 002 and 003 will be interpreted from the function EX(), the nB interpreter. These lines define a public variable and initialize it at 0. This variable will be use to count pages.

At line 007, inside the header (nested), start another *COMMAND definition that contains an increment for the "PageNo" variable. As the header is read and "executed" for every new page, and that before the footer, the variable "PageNo" will contain the right page number.

At line 015, inside the footer, a reference to "PageNo" appears. Here will be printed the page number.

A more complicated example can be found in ‘ADDRESS.TXT’ the RPT text file used for the ADDRESS.& macro examples.

How can I...

nB is a little bit complicated as it may do many things. Here are some examples.

Create a UDF function

UDF means User Defined Function. Inside nB there isn't the possibility to create functions, but there is an alternative: code blocks.

Create a big code block

A code block cannot be longer than 254 characters, as any other instruction inside nB.

So, there is no way to make a bigger code block, but a code block can call another code block, and so on. For example:

```
mempublic( { "first", "second", "third" } )
first := {|| eval( second, "hello" ) }
second := {|x| eval( third, x ) }
third := {|x| alertbox( x ) }
eval( first )
```

This stupid example simply will show the alert box containing the word "hello".

The source files

The nB source is composed of four files:

'NB.PRG'	The main source file containing essentially the nB menu.
'REQUEST.PRG'	Contains a link to all Clipper standard functions.
'STANDARD.PRG'	Contains the most important standard functions.

‘EXTRA.PRG’	Contains some extra function not absolutely necessary during macro execution.

The file ‘REQUEST.PRG’ source file generates some warnings because not all functions listed there are directly called form nB. Don’t worry about that warning message.

Different ‘.RMK’ (rmake) files are included to compile nB differently, including/excluding some program parts, for example to obtain a runtime executor.

[1] This is the original documentation of nanoBase 1997, with minor modifications, that appeared originally at ‘http://www.geocities.com/SiliconValley/7737/nb.htm’.

A different way to program using Clipper 5.2 without commands, that is, without the file ‘`STD.CH`’.[1]

Clipper 5.2,[2] as the xBase tradition imposes, is not an ordered, clear, simple programming language. The question is: which is the right way to write a Clipper program? If the intention is not to make a xBase program, but a Clipper program, maybe it can be decided that it is better to use Clipper without commands.

Step 1: try to compile with the /P parameter

«

Supposing to compile the file '`TEST.PRG`' this way:

`C:\>`**`CLIPPER TEST.PRG /P`** [*Enter*]

It generates a preprocessed output file ('`test.PPO`'), that is a source file without comments, where commands are translated into real Clipper instructions. That is, all the '**`#COMMAND`**' substitution are

executed and the translation is sent to the '`.PPO`' file. It may be difficult to read this file the first time.

Step 2: understand well the use of code blocks

The code block is a small piece of executable program code that can be stored inside a variable, or can be used as a literal constant. The good of it, is that pieces of code may be sent to functions.

A code block is something like a little user defined function where only a sequence of expressions (functions and/or assignments) may appear: no loops, no conditional structures.

A code block may receive arguments and return a value after execution, just like a function. The syntax is the following, where curly brackets are part of the code block:

```
{ | [argument_list] | exp_list }
```

That is: the ***argument_list*** is optional; the ***exp_list*** may contain one or more expressions separated with a comma.

For example, calling the following code block will give the string ''hello world'' as result.

```
{ || "hello world" }
```

The following code block requires a numeric argument and returns the number passed as argument incremented:

```
{ | n | n+1 }
```

The following code block requires two numeric arguments and returns the sum of the two square radix:

```
{ | nFirst, nSecond | SQRT(nFirst) + SQRT(nSecond) }
```

But code blocks may contain more expressions and the result of the execution of the code block is the result of the last expression. The following code block executes in sequence some functions and gives ‘‘hello world’’ as a result.

```
{ | a, b | functionOne(a), functionTwo(b), "hello world" }
```

To start the execution of a code block a function is used: ‘**EVAL()**’. For example, a code block is assigned to a variable and then executed.

```
B := { || "hello world" }
EVAL( B ) == "hello world"
```

Another example with one parameter.

```
B := { | n | n+1 }
EVAL( B, 1 ) == 2
```

Another example with two parameters.

```
B := { | nFirst, nSecond | SQRT(nFirst) + SQRT(nSecond) }
EVAL( B, 2, 4 ) == 20
```

And so on.

Step 3: understand the object programming

Clipper 5.2 do not permit to create objects, but it gives some good objects to use: ‘**GET**’ and ‘**TBROWSE**’. Before starting to clean programming from commands, it is necessary to understand how to use well the Clipper objects.

Classes and methods

A class defines the structure of a ‘‘black box’’, that is a data container; a method is an action to make on a piece of data contained inside the black box. There is no way to reach the data contained inside the black box without a method.

The black box can be called object.

The methods may be seen as they where special functions which interact with a specific piece of data contained inside the object.

Class definition

Supposing that Clipper permits to define classes (unluckily only predefined classes can be used), the hypothetical syntax could be:

```
CLASS ClassName [FROM ParentClass]
    VAR Var1 [, Var2 [, ...VarN]]
    METHOD {method_definition_1} [, ...{method_definition_n}]
ENDCLASS
```

This way, the class defines a group of variables and a group of method to use with these variables.

Object creation

The presence of classes permits to create objects: the black boxes.

```
Variable_name := ClassName
```

This way, a variable contains (is) an object. Please note that inside Clipper, an object may be generated also from a function, that is, a

function can return an object. This way the example can be:

```
Variable_name := classfunction( … )
```

The next problem is to handle this object.

Instantiating an object

«

As already stated before, methods are used to handle data contained inside an object. This is said to be instantiating an object. Clipper permits also to handle directly (apparently without methods) some variables contained inside objects. These are called ‘‘Exported Instance Variables’’. So, an object can be instantiated this way:

```
object:exported_instance_variable := new_value
```

```
object:method()
```

An exported instance variable may be read and/or modified depending on the allowed access to it; a method, inside Clipper, is something like a function with or without parameters (if parameters are present, these are usually used to modify data inside the object), that normally returns a value.

The ``send'' symbol

«

To instantiate an object or simply to access an exported instance variable, the ‘‘send’’ symbol (colon) is used.

If this is not enough to understand objects inside Clipper, the following document should be read:

Peter M. Freese, *o:Clip - An Object Oriented Extension to Clipper 5.01* 1991, CyberSoft

ftp://ftp.simtel.net/pub/simtelnet/msdos/clipper/oclip.zip

Step 4: understand the get object

What happens with a command like the following:

```
@ nTop, nLeft GET Var
```

A get object is created containing all the necessary information for editing the variable ***Var*** at the screen position ***nTop***, ***nLeft***. After that, this get object is added to a get objects array (usually called '**GetList**'). The get objects array will contain all the get objects used during a '**READ**'.

So, what happens when a '**READ**' command is encountered. The get objects array ('**GetList**') is read and the editing of all get objects is executed. After that, the get objects array is cleared.

This method hides what Clipper really makes. The suggestion here is to create a '**GET()**' function that will substitute the '**@...GET**' command and to use the '**READMODAL()**' function to read the get objects array. Here is an example of it:

```
function GET( aoGet, nRow, nCol, bVar, cGetPicture,
              cColorString, bPreValid, bPostValid )

   // declare a local get object
```

```
    local oGet

    // create the get object using the function GETENV()
    oGet := GETENV( nRow, nCol, bVar, NIL, cGetPicture, cGetColor )

    // send to the get object the pre-validation condition code block (WHEN)
    oGet:preBlock  := bPreValid

    // send to the get object the post-validation condition code block (VALID)
    oGet:postBlock := bPostValid

    // display the get on the screen using the display() method
    oGet:display()

    // add the get object to the get objects array
    AADD( aoGet, oGet )

    return NIL
```

- '**aoGet**' is the get objects array (so here is explicitly passed). This get objects array is modified (grown) and there is no need to return it as inside Clipper, arrays are always passed by reference to functions.
- '**nRow**' and '**nCol**' are the screen coordinates where the get field should appear at, as it works with the '**@...GET**' command.
- '**bVar**' is a special code block that permits the editing of a variable. If the variable '**Var**' is to be edited, the code block is:

```
{ |x| iif( pcount() > 0, Var := x, Var }
```

- '**cGetPicture**' is the picture to use: same as the '**@...GET**' command.
- '**cColorString**' is the color string to use: same as the '**@...GET**' command.
- '**bPreValid**' is a code block containing the condition that must

be valid before the cursor can reach this get field. It is equivalent to the '**WHEN**' condition used with the '**@...GET**' command, but it must be converted into a code block. For example, if the condition is '**A > B**', the code block is '**{|| A > B}**'

- '**bPostValid**' is a code block containing the condition that must be valid before the cursor can leave this get field. It is equivalent to the '**VALID**' condition used with the '**@...GET**' command, but it must be converted into a code block. For example, if the condition is '**A > B**', the code block is '**{|| A > B}**'

If there is a get function like the above one, screen I/O may be performed like the following example:

```
function do_some_editing()

    // define a variable to use as a get objects array
    // and initialise it to the empty array
    local aoGet := {}
    ...

    ...
    // add a new get object to the get objects array
    get(;
        aoGet,;
        10, 10,;
        { |x| iif( pcount() > 0, myVariable := x, myVariable },;
        "@s30@",;
        "gb+/b,  n/w,    n,  n,  w/n",;
        { || .T. },;
        { || .T. };
    )

    ...
    // read the get objects array
    readmodal( aoGet )

    // clear the get objects array
    aoGet := {}
```

```
    ...
    return ...
```

If the function '**GET()**' is not liked, the above I/O may be done as it follows:

```
function do_some_editing()

    // define a variable to use as a get object
    local aoGet

    // define a variable to use as a get objects array
    // and initialise it to the empty array
    local aoGet := {}
    ...

    ...
    // add a new get object to the get objects array

    oGet :=;
        GETENV(;
            10, 10,;
            { |x| iif( pcount() > 0, myVariable := x, myVariable },;
            NIL,;
            "@s30@",;
            "gb+/b,  n/w,    n,  n,  w/n",;
        )
    AADD( aoGet, oGet )

    ...
    // read the get objects array
    readmodal( aoGet )

    // clear the get objects array
    aoGet := {}

    ...
    return ...
```

Step 5: trying to stop using commands

To stop using commands, it is important to understand how commands are or may be translated into functions. Sometimes Clipper uses some undocumented functions: these are functions that start with a underline.

?/??

```
? [exp_list]
```

```
qout([exp_list])
```

```
?? [exp_list]
```

```
qqout([exp_list])
```

@...BOX

```
@ nTop, nLeft, nBottom, nRight BOX cnBoxString [COLOR cColorString]
```

```
dispbox(nTop, nLeft, nBottom, nRight, [cnBoxString], [cColorString
])
```

@...GET

«

```
@ nTop, nLeft GET Var [PICTURE cGetPicture] [COLOR cColorString]
↩
↪[WHEN lPreExpression] [VALID lPostExpression]
```

```
setpos(nTop, nLeft)
```

```
aadd( GetList, _GET_( Var, "Var", cGetPicture, [
{|| lPostExpression}],↩
↪[{|| lPreExpression}] ):display() ) atail(GetList):colorDisp(cColo
```

This is the command substitution made automatically, but it shouldn't be used to make clean programs. The step 4 (u0.1) suggests to create a get function.

@...SAY

«

```
@ nTop, nLeft SAY exp [COLOR cColorString]
```

```
devpos(nTop, nLeft)
```

```
devout(exp [, cColorString])
```

```
@ nTop, nLeft SAY exp PICTURE cSayPicture [COLOR cColorString]
```

```
devpos(nTop, nLeft)
```

```
devoutpic(exp, cSayPicture, [cColorString])
```

@...TO

«

```
@ nTop, nLeft TO nBottom, nRight DOUBLE [COLOR cColorString]
```

```
dispbox(nTop, nLeft, nBottom, nRight, 2 [,cColorString])
```

```
@ nTop, nLeft TO nBottom, nRight [COLOR cColorString]
```

```
dispbox(nTop, nLeft, nBottom, nRight, 1 [,cColorString])
```

```
@ nTop, nLeft CLEAR [TO nBottom, nRight]
```

```
scroll([nTop], [nLeft], [nBottom, nRight])
```

```
setpos(nRow, nCol)
```

APPEND

«

```
APPEND BLANK
```

```
dbappend()
```

APPEND FROM

«

```
APPEND FROM xcFile [FIELDS idField_list] [scope] [WHILE lCondition
] ↩
↪[FOR lCondition] [VIA xcDriver]
```

```
__dbApp( cFileName, [acFields], [bForCondition], [bWhileCondition]
, [nNextRecords], ↩
↪[nRecord], [lRest], [cDriver] )
```

```
APPEND FROM xcFile [FIELDS idField_list] [scope] [WHILE lCondition
] [FOR lCondition] ↩
↪DELIMITED xcDelimiter
```

```
__dbDelim( .f., cFileName, [cDelimiter], [acFields], [
bForCondition], ↩
↪[bWhileCondition], [nNextRecords], [nRecord], [lRest] )
```

```
APPEND FROM xcFile [FIELDS idField_list] [scope] [WHILE lCondition
]↩
↪[FOR lCondition] SDF
```

```
__dbSDF( .f., cFileName, [acFields], [bForCondition], [
bWhileCondition], ↩
↪[nNextRecords], [nRecord], [lRest] )
```

CLEAR

«

```
CLEAR
```

```
Scroll()
```

```
SetPos(0,0)
```

```
ReadKill(.T.)
```

```
GetList := {}
```

```
CLEAR GETS
```

```
ReadKill(.T.)
```

```
GetList := {}
```

```
CLEAR SCREEN | CLS
```

```
Scroll()
```

```
SetPos(0,0)
```

CLOSE

«

```
CLOSE
```

```
dbCloseArea()
```

```
CLOSE idAlias
```

```
idAlias->( dbCloseArea() )
```

```
CLOSE ALTERNATE
```

```
Set(19, "")
```

```
CLOSE DATABASES
```

```
dbCloseAll()
```

```
CLOSE INDEXES
```

```
dbClearIndex()
```

COMMIT

«

```
COMMIT
```

```
dbCommitAll()
```

CONTINUE

«

```
CONTINUE
```

```
__dbContinue()
```

COPY

«

```
COPY FILE xcSourceFile TO xcTargetFile | xcDevice
```

```
__CopyFile( cSourceFile , cTargetFile | cDevice )
```

```
COPY STRUCTURE [FIELDS idField_list] TO xcDatabase
```

```
__dbCopyStruct( cDatabase , [acFields] )
```

```
COPY STRUCTURE EXTENDED TO xcExtendedDatabase
```

```
__dbCopyXStruct( cExtendedDatabase )
```

```
COPY TO xcFile [FIELDS idField_list] [scope] [WHILE lCondition] ↩
↪[FOR lCondition] [VIA xcDriver]
```

```
__dbCopy( cFileName, [acFields], [bForCondition], [bWhileCondition
], [nNextRecords], ↩
↪[nRecord], [lRest], [cDriver] )
```

```
COPY TO xcFile [FIELDS idField_list] [scope] [WHILE lCondition] [
FOR lCondition] ↩
↪DELIMITED xcDelimiter
```

```
__dbDelim( .t., cFileName, [cDelimiter], [acFields], [
bForCondition], ↩
↪[bWhileCondition], [nNextRecords], [nRecord], [lRest] )
```

```
COPY TO xcFile [FIELDS idField_list] [scope] [WHILE lCondition] ↩
↪[FOR lCondition] SDF
```

```
__dbSDF( .t., cFileName, [acFields], [bForCondition], [
bWhileCondition], ↩
↪[nNextRecords], [nRecord], [lRest] )
```

COUNT

«

```
COUNT TO idVar [FOR lForCondition] [WHILE lWhileCondition] [
NEXT nNextRecords] ↩
↪[RECORD nRecord] [REST] [ALL]
```

```
dbeval( {||idVar:=idVar+1}, {||lForCondition}, {||lWhileCondition},↩
↪nNextRecords, nRecord, lRest )
```

CREATE

«

```
CREATE xcDatabase FROM xcExtendedDatabase [NEW] [ALIAS cAlias] [
VIA cDriver]
```

```
__dbCreate( cDatabase, cExtendedDatabase, [cDriver], [lNew], [
cAlias] )
```

DEFAULT

«

```
DEFAULT xVar TO xDefaultValue
```

```
if xVar == NIL
   xVar := xDefaultValue
end
```

DELETE

«

```
DELETE
```

```
dbDelete()
```

```
DELETE [FOR lForCondition] [WHILE lWhileCondition] [
NEXT nNextRecords] ↩
↪[RECORD nRecord] [REST] [ALL]
```

```
dbeval( {||dbDelete()}, {||lForCondition}, {||lWhileCondition},↩
↪nNextRecords, nRecord, lRest )
```

```
DELETE FILE xcFile
```

```
ferase( cFile )
```

EJECT

«

```
EJECT
```

```
qqout( chr(13) )
```

ERASE

```
ERASE xcFile
```

```
ferase( cFile )
```

FIND

```
FIND xcSearchString
```

```
dbSeek( cSearchString )
```

GO

```
GO[TO] nRecord
```

```
dbgoto(nRecord)
```

```
GO[TO] BOTTOM
```

```
dbGoBottom()
```

```
GO[TO] TOP
```

```
dbgotop()
```

INDEX ON

«

```
INDEX ON expKey TO xcIndexName [UNIQUE] [FOR lForCondition] ↩
↪[WHILE lWhileCondition] [[EVAL lEvalCondition] [EVERY nRecords]
] ↩
↪[ASCENDING|DESCENDING]
```

```
ordCondSet( [cForCondition], [bForCondition], , [bWhileCondition]
,↩
↪[bEvalCondition], [nRecords], RECNO(), , , , lDescending )
```

```
ordCreate( cIndexName, , cExpKey, bExpKey, lUnique )
```

JOIN

«

```
JOIN WITH xcAlias TO xcDatabase [FOR lCondition] [
FIELDS idField_list]
```

```
__dbJoin( cAlias, cDatabase, [acFields], [bForCondition] )
```

KEYBOARD

«

```
KEYBOARD cString
```

```
__Keyboard( [cString] ) --> NIL
```

LABEL FORM

«

```
LABEL FORM xcLabel [TO PRINTER] [TO FILE xcFile] [NOCONSOLE]
[scope] ↩
↪ [WHILE lCondition] [FOR lCondition] [SAMPLE]
```

```
__LabelForm( cLabel, [lToPrinter], [cFile], [lNoConsole], ↩
↪ [bForCondition], [bWhileCondition], [nNextRecords], [nRecord], ↩
↪ [lRest], [lSample] )
```

LIST

«

```
LIST exp_list [TO PRINTER] [TO FILE xcFile] [scope] ↩
↪ [WHILE lCondition] [FOR lCondition] [OFF]
```

```
__dbList( [lToDisplay], abListColumns, [lAll], [bForCondition], [
bWhileCondition], ↩
↪ [nNextRecords], [nRecord], [lRest], [lToPrinter], [cFileName] )
```

LOCATE

«

```
LOCATE [scope] FOR lCondition [WHILE lCondition]
```

```
__dbLocate( [bForCondition], [bWhileCondition], [nNextRecords], [nRecord], [lRest] )
```

PACK

«

```
PACK
```

```
__dbPack()
```

QUIT

«

```
QUIT
```

```
__Quit()
```

READ

«

```
READ
```

```
ReadModal(GetList)
```

```
GetList := {}
```

```
READ SAVE
```

```
ReadModal(GetList)
```

RECALL

«

```
RECALL
```

```
dbRecall()
```

```
RECALL [FOR lForCondition] [WHILE lWhileCondition] [
NEXT nNextRecords] ↩
↪[RECORD nRecord] [REST] [ALL]
```

```
dbeval( {||dbRecall()}, {||lForCondition}, {||lWhileCondition},↩
↪nNextRecords, nRecord, lRest )
```

REINDEX

«

```
REINDEX [EVAL lEvalCondition] [EVERY nRecords]
```

```
ordCondSet(, , , , [bEvalCondition], [nRecords], , , , , , , )
```

```
ordListRebuild()
```

RELEASE

«

```
RELEASE idMemvar
```

```
__MXRelease( "idMemvar" )
```

```
RELEASE ALL
```

```
__MRelease("*", .t.)
```

```
RELEASE ALL LIKE skeleton
```

```
__MRelease( "skeleton", .t. )
```

```
RELEASE ALL EXCEPT skeleton
```

```
__MRelease( "skeleton", .F. )
```

RENAME

```
RENAME xcOldFile TO xcNewFile
```

```
frename( cOldFile, cNewFile )
```

REPLACE

```
REPLACE idField1 WITH exp1 [, idField2 WITH exp2...] ↩
↪[FOR lForCondition] [WHILE lWhileCondition] [NEXT nNextRecords] ↩
↪[RECORD nRecord] [REST] [ALL]
```

```
dbeval( {|| idField1 := exp1 [, idField2 := exp2...] }, ↩
↪{||lForCondition}, {||lWhileCondition}, nNextRecords, ↩
↪nRecord, lRest )
```

```
REPLACE idField1 WITH exp1
```

```
idField1 := exp1
```

REPORT FORM

«

```
REPORT FORM xcReport [TO PRINTER] [TO FILE xcFile] [NOCONSOLE
] [scope] ↩
↪[WHILE lCondition] [FOR lCondition] [PLAIN | HEADING cHeading]
[NOEJECT] ↩
↪[SUMMARY]
```

```
__ReportForm( cForm, [lToPrinter], [cToFile], [lNoConsole], [
bForCondition], ↩
↪[bWhileCondition], [nNext], [nRecord], [lRest], [lPlain], [
cbHeading], ↩
↪[lBeforeEject], [lSummary] )
```

RESTORE

«

```
RESTORE SCREEN FROM cScreen
```

```
restscreen( 0, 0, Maxrow(), Maxcol(), cScreen )
```

RESTORE FROM

«

```
RESTORE FROM xcMemFile [ADDITIVE]
```

```
__MRestore( cMemFileName, [lAdditive] )
```

RUN

```
RUN xcCommandLine
```

```
__Run( cCommand )
```

SAVE SCREEN TO

```
SAVE SCREEN TO cScreen
```

```
cScreen := savescreen( 0, 0, maxrow(), maxcol() )
```

SAVE TO

```
SAVE TO xcMemFile [ALL [LIKE|EXCEPT skeleton]]
```

```
_MSave( cMemFileName, [cSkeleton], [lLike] )
```

SEEK

```
SEEK expSearch [SOFTSEEK]
```

```
dbSeek( expSearch [, lSoftSeek] )
```

SELECT

«

```
SELECT xnWorkArea | idAlias
```

```
dbSelectArea( nWorkArea | cIdAlias )
```

SET

«

Most of the '**SET**...' commands are translated into the '**SET()**' function that distinguishes different modes depending on a number. As this number is difficult to handle during programming (essentially because it is difficult to remember the meaning of it), Clipper offers the '**SET.CH**' include file that helps with manifest constants.

```
#define _SET_EXACT          1
#define _SET_FIXED          2
#define _SET_DECIMALS       3
#define _SET_DATEFORMAT     4
#define _SET_EPOCH          5
#define _SET_PATH           6
#define _SET_DEFAULT        7

#define _SET_EXCLUSIVE      8
#define _SET_SOFTSEEK       9
#define _SET_UNIQUE        10
#define _SET_DELETED       11

#define _SET_CANCEL        12
#define _SET_DEBUG         13
#define _SET_TYPEAHEAD     14

#define _SET_COLOR         15
#define _SET_CURSOR        16
#define _SET_CONSOLE       17
#define _SET_ALTERNATE     18
#define _SET_ALTFILE       19
#define _SET_DEVICE        20
```

```
#define _SET_EXTRA        21
#define _SET_EXTRAFILE    22
#define _SET_PRINTER      23
#define _SET_PRINTFILE    24
#define _SET_MARGIN       25

#define _SET_BELL         26
#define _SET_CONFIRM      27
#define _SET_ESCAPE       28
#define _SET_INSERT       29
#define _SET_EXIT         30
#define _SET_INTENSITY    31
#define _SET_SCOREBOARD   32
#define _SET_DELIMITERS   33
#define _SET_DELIMCHARS   34

#define _SET_WRAP         35
#define _SET_MESSAGE      36
#define _SET_MCENTER      37
#define _SET_SCROLLBREAK  38
```

```
SET ALTERNATE TO xcFile [ADDITIVE]
```

```
Set( _SET_ALTFILE, cFile, lAdditive )
```

```
SET ALTERNATE ON | OFF | xlToggle
```

```
Set( _SET_ALTERNATE, "ON" | "OFF" | lToggle )
```

```
SET BELL ON | OFF | xlToggle
```

```
Set( _SET_BELL, "ON" | "OFF" | lToggle )
```

```
SET COLOR | COLOUR TO (cColorString)
```

```
SetColor( cColorString )
```

```
SET CONFIRM ON | OFF | xlToggle
```

```
Set( _SET_CONFIRM, "ON" | "OFF" | lToggle )
```

```
SET CONSOLE ON | OFF | xlToggle
```

```
Set( _SET_CONSOLE, "ON" | "OFF" | lToggle )
```

```
SET CURSOR ON | OFF | xlToggle
```

```
SetCursor( 1 | 0 | iif( lToggle, 1, 0 ) )
```

```
SET DATE FORMAT [TO] cDateFormat
```

```
Set( _SET_DATEFORMAT, cDateFormat )
```

```
SET DECIMALS TO
```

```
Set( _SET_DECIMALS, 0 )
```

```
SET DECIMALS TO nDecimals
```

```
Set( _SET_DECIMALS, nDecimals )
```

```
SET DEFAULT TO
```

```
Set( _SET_DEFAULT, "" )
```

```
SET DEFAULT TO xcPathspec
```

```
Set( _SET_DEFAULT, cPathspec )
```

```
SET DELETED ON | OFF | xlToggle
```

```
Set( _SET_DELETED, "ON" | "OFF" | lToggle )
```

```
SET DELIMITERS ON | OFF | xlToggle
```

```
Set( _SET_DELIMITERS, "ON" | "OFF" | lToggle )
```

```
SET DELIMITERS TO [DEFAULT]
```

```
Set( _SET_DELIMCHARS, "::" )
```

```
SET DELIMITERS TO cDelimiters
```

```
Set( _SET_DELIMCHARS, cDelimiters )
```

```
SET DEVICE TO SCREEN | PRINTER
```

```
Set( _SET_DEVICE, "SCREEN" | "PRINTER" )
```

```
SET EPOCH TO nYear
```

```
Set( _SET_EPOCH, nYear )
```

```
SET ESCAPE ON | OFF | xlToggle
```

```
Set( _SET_ESCAPE, "ON" | "OFF" | lToggle )
```

```
SET EXACT ON | OFF | xlToggle
```

```
Set( _SET_EXACT, "ON" | "OFF" | lToggle )
```

```
SET EXCLUSIVE ON | OFF | xlToggle
```

```
Set( _SET_EXCLUSIVE, "ON" | "OFF" | lToggle )
```

```
SET FILTER TO
```

```
dbclearfilter()
```

```
SET FILTER TO lCondition
```

```
dbsetfilter( bCondition, cCondition )
```

```
SET FIXED ON | OFF | xlToggle
```

```
Set( _SET_FIXED, "ON" | "OFF" | lToggle )
```

```
SET FUNCTION nFunctionKey TO cString
```

```
__SetFunction( nFunctionKey, cString )
```

```
SET INDEX TO [xcIndex [, xcIndex1...] ]
```

```
ordListClear()
```

```
ordListAdd( cIndex )
ordListAdd( cIndex1 )
...
```

```
SET INTENSITY ON | OFF | xlToggle
```

```
Set( _SET_INTENSITY, "ON" | "OFF" | lToggle )
```

```
SET KEY nInkeyCode [TO]
```

```
SetKey( nInkeyCode, NIL )
```

```
SET KEY nInkeyCode TO [idProcedure]
```

```
SetKey( nInkeyCode, { |p, l, v| idProcedure(p, l, v)} )
```

```
SET MARGIN TO
```

```
Set( _SET_MARGIN, 0 )
```

```
SET MARGIN TO [nPageOffset]
```

```
Set( _SET_MARGIN, nPageOffset )
```

```
SET MESSAGE TO
```

```
Set( _SET_MESSAGE, 0 )
```

```
Set( _SET_MCENTER, .F. )
```

```
SET MESSAGE TO [nRow [CENTER | CENTRE]]
```

```
Set( _SET_MESSAGE, nRow )
```

```
Set( _SET_MCENTER, lCenter )
```

```
SET ORDER TO [nIndex]
```

```
ordSetFocus( nIndex )
```

```
SET PATH TO
```

```
Set( _SET_PATH, "" )
```

```
SET PATH TO [xcPathspec [, cPathspec1... ] ]
```

```
Set( _SET_PATH, cPathspec [, cPathspec1… ] )
```

```
SET PRINTER ON | OFF | xlToggle
```

```
Set( _SET_PRINTER, "ON" | "OFF" | lToggle )
```

```
SET PRINTER TO
```

```
Set( _SET_PRINTFILE, "" )
```

```
SET PRINTER TO [xcDevice | xcFile [ADDITIVE]]
```

```
Set( _SET_PRINTFILE, cDevice | cFile, lAdditive )
```

```
SET RELATION TO
```

```
dbclearrelation()
```

```
SET RELATION TO [expKey1 INTO xcAlias1] [, [TO]
expKey2 INTO xcAlias2…]
```

```
dbClearRel()
```

```
dbSetRelation( cAlias1, {||expKey1}, ["expKey1"] )
```

```
dbSetRelation( cAlias2, {||expKey2}, ["expKey1"] )
```

```
SET RELATION TO [expKey1 INTO xcAlias1] ↩
↪[, [TO] expKey2 INTO xcAlias2...] ADDITIVE
```

```
dbSetRelation( cAlias1, {||expKey1}, ["expKey1"] )
```

```
dbSetRelation( cAlias2, {||expKey2}, ["expKey1"] )
```

```
SET SCOREBOARD ON | OFF | xlToggle
```

```
Set( _SET_SCOREBOARD, "ON" | "OFF" | lToggle )
```

```
SET SOFTSEEK ON | OFF | xlToggle
```

```
Set( _SET_SOFTSEEK, "ON" | "OFF" | lToggle )
```

```
SET TYPEAHEAD TO nKeyboardSise
```

```
Set( _SET_TYPEAHEAD, nKeyboardSise )
```

```
SET UNIQUE ON | OFF | xlToggle
```

```
Set( _SET_UNIQUE, "ON" | "OFF" | lToggle )
```

```
SET WRAP ON | OFF | xlToggle
```

```
Set( _SET_WRAP, "ON" | "OFF" | lToggle )
```

SKIP

«

```
SKIP [nRecords] [ALIAS idAlias | nWorkArea]
```

```
[idAlias | nWorkArea -> ] ( dbSkip( [nRecords] ) )
```

SORT

«

```
SORT TO xcDatabase ON idField1 [/[A|D][C]] [, idField2 [/[A|D]
[C]] ...] ↩
↪[scope] [WHILE lCondition] [FOR lCondition]
```

```
__dbSort( cDatabase, [acFields], [bForCondition], [bWhileCondition]
, ↩
↪[nNextRecords], [nRecord], [lRest] )
```

STORE

```
STORE value TO variable
```

```
variable := value
```

SUM

```
SUM nExp1 [, nExp2...] TO idVar1 [, idVar2...] [FOR lForCondition] ↩
↪[WHILE lWhileCondition] [NEXT nNextRecords] [RECORD nRecord] [
REST] [ALL]
```

```
dbeval( {||idVar1:=idVar1+nExp1 [, idVar2:=idVar2+nExp2...] }, ↩
↪{||lForCondition}, {||lWhileCondition}, nNextRecords, nRecord, lRest )
```

TOTAL ON

«

```
TOTAL ON expKey [FIELDS idField_list] TO xcDatabase [scope] ↩
↪[WHILE lCondition] [FOR lCondition]
```

```
__dbTotal( cDatabase, bKey, [acFields], [bForCondition], [
bWhileCondition], ↩
↪[nNextRecords], [nRecord], [lRest] )
```

UNLOCK

«

```
UNLOCK
```

```
dbUnlock()
```

```
UNLOCK ALL
```

```
dbUnlockAll()
```

UPDATE FROM

«

```
UPDATE FROM xcAlias ON expKey [RANDOM] REPLACE idField1 ↩
↪WITH exp [, idField2 WITH exp ...]
```

```
__dbUpdate( cAlias, bKey, [lRandom], [bReplacement] )
```

Example:

```
__dbUpdate( "INVOICE", {|| LAST}, .T.,; {|| FIELD->TOTAL1 := ↩
↪INVOICE->SUM1,; FIELD->TOTAL2 := INVOICE->SUM2 } )
```

USE

«

```
USE
```

```
dbclosearea()
```

```
USE [xcDatabase] ↩
↪[INDEX xcIndex1 [, xcIndex2...] [ALIAS xcAlias] [EXCLUSIVE|
SHARED] [NEW] ↩
↪[READONLY] [VIA cDriver] ]
```

```
dbUseArea( [lNewArea], [cDriver], cDatabase, [cAlias], [lShared]
, [lReadOnly] )
```

```
[dbSetIndex( cIndex1 )]
```

```
[dbSetIndex( cIndex2 )]
```

```
...
```

ZAP

«

```
ZAP
```

```
dbZap()
```

Step 6: free yourself from STD.CH - /U

«

Now that no command is used, the standard include file ‘STD.CH’ is no more necessary. Clipper uses ‘STD.CH’ automatically, unless specified differently. Just compile this way:

C:>**CLIPPER TEST.PRG /U**[*Enter*]

Step 7: take control over all include files

«

Clipper comes with so many include files (‘*.CH’). To avoid confusion, a single ‘STANDARD.CH’ file containing all what is needed for the application may be prepared. At least, it is necessary the following.

```
*=============================================================
* DISPBOX()
*=============================================================

* Single-line box
#define BOX_SINGLE;
    (;
        CHR(218) +;
        CHR(196) +;
```

```
        CHR(191) +;
        CHR(179) +;
        CHR(217) +;
        CHR(196) +;
        CHR(192) +;
        CHR(179);
    )

* Double-line box
#define BOX_DOUBLE;
    (;
        CHR(201) +;
        CHR(205) +;
        CHR(187) +;
        CHR(186) +;
        CHR(188) +;
        CHR(205) +;
        CHR(200) +;
        CHR(186);
    )

* Single-line top, double-line sides
#define BOX_SINGLE_DOUBLE;
    (;
        CHR(214) +;
        CHR(196) +;
        CHR(183) +;
        CHR(186) +;
        CHR(189) +;
        CHR(196) +;
        CHR(211) +;
        CHR(186);
    )

* Double-line top, single-line sides
#define BOX_DOUBLE_SINGLE;
    (;
        CHR(213) +;
        CHR(205) +;
        CHR(184) +;
        CHR(179) +;
        CHR(190) +;
        CHR(205) +;
        CHR(212) +;
```

```
        CHR(179);
    )


*=====================================================================
* ERRORS
*=====================================================================

* Severity levels (e:severity)
#define ERROR_SEVERITY_WHOCARES        0
#define ERROR_SEVERITY_WARNING         1
#define ERROR_SEVERITY_ERROR           2
#define ERROR_SEVERITY_CATASTROPHIC    3

* Generic error codes (e:genCode)
#define ERROR_GENERIC_ARG             1
#define ERROR_GENERIC_BOUND           2
#define ERROR_GENERIC_STROVERFLOW     3
#define ERROR_GENERIC_NUMOVERFLOW     4
#define ERROR_GENERIC_ZERODIV         5
#define ERROR_GENERIC_NUMERR          6
#define ERROR_GENERIC_SYNTAX          7
#define ERROR_GENERIC_COMPLEXITY      8

#define ERROR_GENERIC_MEM            11
#define ERROR_GENERIC_NOFUNC         12
#define ERROR_GENERIC_NOMETHOD       13
#define ERROR_GENERIC_NOVAR          14
#define ERROR_GENERIC_NOALIAS        15
#define ERROR_GENERIC_NOVARMETHOD    16
#define ERROR_GENERIC_BADALIAS       17
#define ERROR_GENERIC_DUPALIAS       18

#define ERROR_GENERIC_CREATE         20
#define ERROR_GENERIC_OPEN           21
#define ERROR_GENERIC_CLOSE          22
#define ERROR_GENERIC_READ           23
#define ERROR_GENERIC_WRITE          24
#define ERROR_GENERIC_PRINT          25

#define ERROR_GENERIC_UNSUPPORTED    30
#define ERROR_GENERIC_LIMIT          31
#define ERROR_GENERIC_CORRUPTION     32
#define ERROR_GENERIC_DATATYPE       33
```

```
#define ERROR_GENERIC_DATAWIDTH      34
#define ERROR_GENERIC_NOTABLE        35
#define ERROR_GENERIC_NOORDER        36
#define ERROR_GENERIC_SHARED         37
#define ERROR_GENERIC_UNLOCKED       38
#define ERROR_GENERIC_READONLY       39

#define ERROR_GENERIC_APPENDLOCK     40
#define ERROR_GENERIC_LOCK           41


*=============================================================
* INKEY()
*=============================================================

#define K_UP                5   //   Up arrow, Ctrl-E
#define K_DOWN             24   //   Down arrow, Ctrl-X
#define K_LEFT             19   //   Left arrow, Ctrl-S
#define K_RIGHT             4   //   Right arrow, Ctrl-D
#define K_HOME              1   //   Home, Ctrl-A
#define K_END               6   //   End, Ctrl-F
#define K_PGUP             18   //   PgUp, Ctrl-R
#define K_PGDN              3   //   PgDn, Ctrl-C

#define K_CTRL_UP         397   // * Ctrl-Up arrow
#define K_CTRL_DOWN       401   // * Ctrl-Down arrow
#define K_CTRL_LEFT        26   //   Ctrl-Left arrow, Ctrl-Z
#define K_CTRL_RIGHT        2   //   Ctrl-Right arrow, Ctrl-B
#define K_CTRL_HOME        29   //   Ctrl-Home, Ctrl-</synsqb>
#define K_CTRL_END         23   //   Ctrl-End, Ctrl-W
#define K_CTRL_PGUP        31   //   Ctrl-PgUp, Ctrl-Hyphen
#define K_CTRL_PGDN        30   //   Ctrl-PgDn, Ctrl-^

#define K_ALT_UP          408   // * Alt-Up arrow
#define K_ALT_DOWN        416   // * Alt-Down arrow
#define K_ALT_LEFT        411   // * Alt-Left arrow
#define K_ALT_RIGHT       413   // * Alt-Right arrow
#define K_ALT_HOME        407   // * Alt-Home
#define K_ALT_END         415   // * Alt-End
#define K_ALT_PGUP        409   // * Alt-PgUp
#define K_ALT_PGDN        417   // * Alt-PgDn

#define K_ENTER            13   //   Enter, Ctrl-M
#define K_RETURN           13   //   Return, Ctrl-M
```

```
#define K_SPACE            32   //   Space bar
#define K_ESC              27   //   Esc, Ctrl-<synsqb>

#define K_CTRL_ENTER       10   //   Ctrl-Enter
#define K_CTRL_RETURN      10   //   Ctrl-Return
#define K_CTRL_RET         10   //   Ctrl-Return (Compat.)
#define K_CTRL_PRTSCR     379   // * Ctrl-Print Screen
#define K_CTRL_QUESTION   309   //   Ctrl-?

#define K_ALT_ENTER       284   // * Alt-Enter
#define K_ALT_RETURN      284   // * Alt-Return
#define K_ALT_EQUALS      387   // * Alt-Equals
#define K_ALT_ESC         257   // * Alt-Esc

#define KP_ALT_ENTER      422   // * Keypad Alt-Enter

#define KP_CTRL_5         399   // * Keypad Ctrl-5
#define KP_CTRL_SLASH     405   // * Keypad Ctrl-/
#define KP_CTRL_ASTERISK  406   // * Keypad Ctrl-*
#define KP_CTRL_MINUS     398   // * Keypad Ctrl--
#define KP_CTRL_PLUS      400   // * Keypad Ctrl-+

#define KP_ALT_5            5   // * Keypad Alt-5
#define KP_ALT_SLASH      420   // * Keypad Alt-/
#define KP_ALT_ASTERISK   311   // * Keypad Alt-*
#define KP_ALT_MINUS      330   // * Keypad Alt--
#define KP_ALT_PLUS       334   // * Keypad Alt-+

#define K_INS              22   //   Ins, Ctrl-V
#define K_DEL               7   //   Del, Ctrl-G
#define K_BS                8   //   Backspace, Ctrl-H
#define K_TAB               9   //   Tab, Ctrl-I
#define K_SH_TAB          271   //   Shift-Tab

#define K_CTRL_INS        402   // * Ctrl-Ins
#define K_CTRL_DEL        403   // * Ctrl-Del
#define K_CTRL_BS         127   //   Ctrl-Backspace
#define K_CTRL_TAB        404   // * Ctrl-Tab

#define K_ALT_INS         418   // * Alt-Ins
#define K_ALT_DEL         419   // * Alt-Del
#define K_ALT_BS          270   // * Alt-Backspace
#define K_ALT_TAB         421   // * Alt-Tab
```

```
#define K_CTRL_A            1   //   Ctrl-A, Home
#define K_CTRL_B            2   //   Ctrl-B, Ctrl-Right arrow
#define K_CTRL_C            3   //   Ctrl-C, PgDn, Ctrl-ScrollLock
#define K_CTRL_D            4   //   Ctrl-D, Right arrow
#define K_CTRL_E            5   //   Ctrl-E, Up arrow
#define K_CTRL_F            6   //   Ctrl-F, End
#define K_CTRL_G            7   //   Ctrl-G, Del
#define K_CTRL_H            8   //   Ctrl-H, Backspace
#define K_CTRL_I            9   //   Ctrl-I, Tab
#define K_CTRL_J           10   //   Ctrl-J
#define K_CTRL_K           11   //   Ctrl-K
#define K_CTRL_L           12   //   Ctrl-L
#define K_CTRL_M           13   //   Ctrl-M, Return
#define K_CTRL_N           14   //   Ctrl-N
#define K_CTRL_O           15   //   Ctrl-O
#define K_CTRL_P           16   //   Ctrl-P
#define K_CTRL_Q           17   //   Ctrl-Q
#define K_CTRL_R           18   //   Ctrl-R, PgUp
#define K_CTRL_S           19   //   Ctrl-S, Left arrow
#define K_CTRL_T           20   //   Ctrl-T
#define K_CTRL_U           21   //   Ctrl-U
#define K_CTRL_V           22   //   Ctrl-V, Ins
#define K_CTRL_W           23   //   Ctrl-W, Ctrl-End
#define K_CTRL_X           24   //   Ctrl-X, Down arrow
#define K_CTRL_Y           25   //   Ctrl-Y
#define K_CTRL_Z           26   //   Ctrl-Z, Ctrl-Left arrow

#define K_ALT_A           286   //   Alt-A
#define K_ALT_B           304   //   Alt-B
#define K_ALT_C           302   //   Alt-C
#define K_ALT_D           288   //   Alt-D
#define K_ALT_E           274   //   Alt-E
#define K_ALT_F           289   //   Alt-F
#define K_ALT_G           290   //   Alt-G
#define K_ALT_H           291   //   Alt-H
#define K_ALT_I           279   //   Alt-I
#define K_ALT_J           292   //   Alt-J
#define K_ALT_K           293   //   Alt-K
#define K_ALT_L           294   //   Alt-L
#define K_ALT_M           306   //   Alt-M
#define K_ALT_N           305   //   Alt-N
#define K_ALT_O           280   //   Alt-O
#define K_ALT_P           281   //   Alt-P
#define K_ALT_Q           272   //   Alt-Q
```

```
#define K_ALT_R          275   //   Alt-R
#define K_ALT_S          287   //   Alt-S
#define K_ALT_T          276   //   Alt-T
#define K_ALT_U          278   //   Alt-U
#define K_ALT_V          303   //   Alt-V
#define K_ALT_W          273   //   Alt-W
#define K_ALT_X          301   //   Alt-X
#define K_ALT_Y          277   //   Alt-Y
#define K_ALT_Z          300   //   Alt-Z
#define K_ALT_1          376   //   Alt-1
#define K_ALT_2          377   //   Alt-2
#define K_ALT_3          378   //   Alt-3
#define K_ALT_4          379   //   Alt-4
#define K_ALT_5          380   //   Alt-5
#define K_ALT_6          381   //   Alt-6
#define K_ALT_7          382   //   Alt-7
#define K_ALT_8          383   //   Alt-8
#define K_ALT_9          384   //   Alt-9
#define K_ALT_0          385   //   Alt-0

#define K_F1              28   //   F1, Ctrl-Backslash
#define K_F2              -1   //   F2
#define K_F3              -2   //   F3
#define K_F4              -3   //   F4
#define K_F5              -4   //   F5
#define K_F6              -5   //   F6
#define K_F7              -6   //   F7
#define K_F8              -7   //   F8
#define K_F9              -8   //   F9
#define K_F10             -9   //   F10
#define K_F11            -40   // * F11
#define K_F12            -41   // * F12

#define K_CTRL_F1        -20   //   Ctrl-F1
#define K_CTRL_F2        -21   //   Ctrl-F2
#define K_CTRL_F3        -22   //   Ctrl-F4
#define K_CTRL_F4        -23   //   Ctrl-F3
#define K_CTRL_F5        -24   //   Ctrl-F5
#define K_CTRL_F6        -25   //   Ctrl-F6
#define K_CTRL_F7        -26   //   Ctrl-F7
#define K_CTRL_F8        -27   //   Ctrl-F8
#define K_CTRL_F9        -28   //   Ctrl-F9
#define K_CTRL_F10       -29   //   Ctrl-F10
#define K_CTRL_F11       -44   // * Ctrl-F11
```

```
#define K_CTRL_F12        -45   // * Ctrl-F12

#define K_ALT_F1          -30   //   Alt-F1
#define K_ALT_F2          -31   //   Alt-F2
#define K_ALT_F3          -32   //   Alt-F3
#define K_ALT_F4          -33   //   Alt-F4
#define K_ALT_F5          -34   //   Alt-F5
#define K_ALT_F6          -35   //   Alt-F6
#define K_ALT_F7          -36   //   Alt-F7
#define K_ALT_F8          -37   //   Alt-F8
#define K_ALT_F9          -38   //   Alt-F9
#define K_ALT_F10         -39   //   Alt-F10
#define K_ALT_F11         -46   // * Alt-F11
#define K_ALT_F12         -47   // * Alt-F12

#define K_SH_F1           -10   //   Shift-F1
#define K_SH_F2           -11   //   Shift-F2
#define K_SH_F3           -12   //   Shift-F3
#define K_SH_F4           -13   //   Shift-F4
#define K_SH_F5           -14   //   Shift-F5
#define K_SH_F6           -15   //   Shift-F6
#define K_SH_F7           -16   //   Shift-F7
#define K_SH_F8           -17   //   Shift-F8
#define K_SH_F9           -18   //   Shift-F9
#define K_SH_F10          -19   //   Shift-F10
#define K_SH_F11          -42   // * Shift-F11
#define K_SH_F12          -43   // * Shift-F12


*====================================================================
* MEMOEDIT()
*====================================================================

* User function entry modes
#define MEMOEDIT_IDLE           0      // idle, all keys processed
#define MEMOEDIT_UNKEY          1      // unknown key, memo unaltered
#define MEMOEDIT_UNKEYX         2      // unknown key, memo altered
#define MEMOEDIT_INIT           3      // initialization mode


* User function return codes
#define MEMOEDIT_DEFAULT        0      // perform default action
#define MEMOEDIT_IGNORE        32      // ignore unknown key
#define MEMOEDIT_DATA          33      // treat unknown key as data
```

```
#define MEMOEDIT_TOGGLEWRAP    34      // toggle word-wrap mode
#define MEMOEDIT_TOGGLESCROLL  35      // toggle scrolling mode
#define MEMOEDIT_WORDRIGHT    100      // perform word-right operation
#define MEMOEDIT_BOTTOMRIGHT  101      // perform bottom-right operation


*====================================================================
* SET()
*====================================================================

#define _SET_EXACT         1
#define _SET_FIXED         2
#define _SET_DECIMALS      3
#define _SET_DATEFORMAT    4
#define _SET_EPOCH         5
#define _SET_PATH          6
#define _SET_DEFAULT       7

#define _SET_EXCLUSIVE     8
#define _SET_SOFTSEEK      9
#define _SET_UNIQUE       10
#define _SET_DELETED      11

#define _SET_CANCEL       12
#define _SET_DEBUG        13
#define _SET_TYPEAHEAD    14

#define _SET_COLOR        15
#define _SET_CURSOR       16
#define _SET_CONSOLE      17
#define _SET_ALTERNATE    18
#define _SET_ALTFILE      19
#define _SET_DEVICE       20
#define _SET_EXTRA        21
#define _SET_EXTRAFILE    22
#define _SET_PRINTER      23
#define _SET_PRINTFILE    24
#define _SET_MARGIN       25

#define _SET_BELL         26
#define _SET_CONFIRM      27
#define _SET_ESCAPE       28
#define _SET_INSERT       29
#define _SET_EXIT         30
```

```
#define _SET_INTENSITY    31
#define _SET_SCOREBOARD   32
#define _SET_DELIMITERS   33
#define _SET_DELIMCHARS   34

#define _SET_WRAP         35
#define _SET_MESSAGE      36
#define _SET_MCENTER      37
#define _SET_SCROLLBREAK  38

*=============================================================
* SETCURSOR()
*=============================================================

#define SETCURSOR_NONE     0  // No cursor
#define SETCURSOR_NORMAL   1  // Normal cursor (underline)
#define SETCURSOR_INSERT   2  // Insert cursor (lower half block)
#define SETCURSOR_SPECIAL1 3  // Special cursor (full block)
#define SETCURSOR_SPECIAL2 4  // Special cursor (upper half block)


*=============================================================
* RDD REQUESTs
*=============================================================

external dbfndx
external dbfntx // default
```

[1] This material appeared originally at ‘http://www.geocities.com/SiliconValley/7737/clipper52clean.html’, in 1996.

[2] **Clipper 5.2** Proprietary software