# The Data Base System (TDBS)

# Version 1.2

# **User Manual**

An Option module for TBBS Version 2.x

by Philip L. Becker and J. P. McMillan

Copyright 1992 by Philip L. Becker, Ltd. All Rights Reserved

#### WARRANTY

TDBS is manufactured by eSoft, Inc. We have made every effort to provide the user with error-free data diskettes. Should there be difficulty in transferring the data from these diskettes to your operating system diskette, contact us at (303) 699-6565 (voice) for replacement.

TDBS is distributed on an "AS IS" basis only, without warranty. Neither eSoft, Inc. nor its authorized dealers shall have liability or responsibility to any person or entity with respect to liability, loss, or damage caused or alleged to be caused by this software. This includes, but is not limited to, any interruption of service, loss of business or anticipatory profits, or consequential damage resulting from the use of this software.

#### eSoft SOFTWARE SUPPORT

eSoft, Inc., provides a 24-hour-a-day multi-line support BBS service to registered TBBS/TDBS system owners at (303) 699-8222 (eSoft Software Support BBS -- 300/1200/2400/9600 baud modem). We strongly urge that this telecommunications system be used for error reporting and troubleshooting assistance. Because all system corrections will be distributed first via the eSoft Software Support BBS, we recommend that all registered licensees log on to the Support BBS at least once a month to receive latest system update information. (At your first logon, you will need to register the serial number of your **TBBS** master disk(s) and supply a password, which will be approved for full support access during the next working day. Other information is given online during the registration process.)

The Data Base System software is distributed without copy prevention protection, since we feel that such protection drastically reduces the utility of a program and punishes most the honest purchaser. We do, however, expect purchasers of TDBS to honor their license agreement. This software is copyrighted, and licensed to the purchaser for your individual and exclusive use on a single CPU at a time. Any reproduction for use by other persons is a violation of our copyright and your license agreement. eSoft, Inc. does not "sell" TDBS. It sells only the media it is contained on. It licenses you the use of the software only under the following license terms and conditions.

# License Agreement

Carefully read the following terms and conditions. Use of this product constitutes your acceptance of the terms and conditions, and your agreement to abide by them subject to paragraph 7 below.

- 1. This is an end-user license. You, the original purchaser, are granted this license for the use of the TDBS software under the terms stated in this agreement. You may not assign or transfer the software or this license to any other person without the express written consent of eSoft, Inc. Any attempt to sublicense, assign, or transfer any of the rights, duties, or obligations hereunder is void. eSoft, Inc. does allow you to transfer this license under the conditions outlined on your registration form. This procedure constitutes express written consent under this provision if it is followed properly.
- 2. The TDBS software is copyrighted material. Once you have paid the required single copy license fee, you may use the software as long as you like provided you do not violate the copyright or any of the following conditions.
- 3. Single CPU License. You may use the software on any computer for which it is designed so long as it is not in use on more than one computer at the same time. You must pay for additional licenses if you want to use this software on more than one computer at the same time. You may use the utilities on one computer while the run time system is in use on another computer, as long as the same program is not in use at the same time on more than one computer.
- 4. **Backup Copies.** You may make as many backup copies of the software as you require to avoid loss. You are responsible for all backup copies you make, and must assure they do not result in any use of the software which would conflict with the provisions of paragraph 2 above.

- 5. Software Modification. You may not make any changes or modifications to the Licensed software not expressly authorized by eSoft, Inc., Philip L. Becker, Ltd. or their agents. This includes but is not limited to disassembly and reverse engineering the software. The single exception granted under this license is the changing of text strings in the programs for customized presentation.
- 6. Federal Government. This Software is Commercial Computer Software under the Federal Government Acquisition Regulations and agency supplements to them. The Software is provided to the Federal Government and its agencies only under the Restricted Rights Provisions of the Federal Acquisition Regulations applicable to commercial computer software developed at private expense and not in the public domain.
- 7. You may refuse to abide by this license by returning all materials within 30 days, along with a written statement that you have kept no copies of the software or documentation. This statment must be signed by you and becomes a legally binding statement that you have indeed destroyed any backup copies you may have made in those thirty days. If you keep the materials beyond the 30 day period, or refuse to assure that you have not kept any copies of the software or its documentation, then you are fully bound by this agreement.
- 8. Limitation of Liability. In no case shall the Liability of eSoft, Inc. or Philip L. Becker, Ltd. exceed the license fees paid for the right to use this software or One Hundred Dollars (\$100.00), whichever is greater.
- 9. This agreement may not be modified except by a written instrument signed by eSoft, Inc. This license constitutes the entire agreement and understanding between you and eSoft, Inc., and supersedes any prior agreement or understanding whether oral or written relating to the subject of this License.

# Table of Contents

# **Chapter 1: Installation**

| HOW TO USE THIS MANUAL            | .1-1   |
|-----------------------------------|--------|
| What's new in TDBS 1.2?           | .1-2   |
| Installing TDBS on a TBBS system  | .1-3   |
| Installing the TDBS compiler      | .1-3   |
| Installing the TDBS Option Module | .1-3   |
| TDBSOM Memory Requirements        | . 1-5  |
| Adding TDBS programs to a Menu    | .1-6   |
| The HOMEPATH directory            | . 1-7  |
| TDBS CONFIG.SYS considerations    |        |
| Compiling TDBS Source Files       | .1-9   |
| Multiple Procedure Control File   | .1-12  |
| The TDBS Autocompile feature      | . 1-13 |
|                                   |        |

# **Chapter 2: Overview**

| Introduction                              | 2-1  |
|---|------|
| TDBS structure                            | 2-1  |
| TDBS compatibility level                  | 2-3  |
| Introduction to the TDBS Language         |      |
| Source Syntax Compatibility               | 2-4  |
| Conditional TDBS commands                 | 2-5  |
| Database Files                            | 2-5  |
| Index Files                               | 2-6  |
| Commands and Functions                    | 2-6  |
| Data Types                                | 2-7  |
| Memory Variable Arrays                    | 2-8  |
| Operators and Precedence                  | 2-10 |
| Program Structure and Control             | 2-12 |
| Subroutine Calling                        | 2-13 |
| Screen and Keyboard I/O                   | 2-14 |
| I/O Commands requiring ANSI terminals     | 2-14 |
| I/O commands not requiring ANSI terminals | 2-14 |
| Keyboard mapping                          |      |
| Memory Usage                              |      |
| Understanding Work Pool Allocation        |      |
| TDBS Maximum Limits                       |      |
|   |      |

| Command Differences in TDBS     | 2-21 |
|---------------------------------|------|
| Commands not supported          | 2-21 |
| Command Extensions              | 2-23 |
| Extended Functions              |      |
| Extended File Sharing Support   |      |
| DOS File Limits and FCB Sharing |      |
| Macro Compatibility             | 2-30 |
| Memory Variable Domains         |      |
| Private Memory Variables        | 2-31 |
| Public Memory Variables         |      |
| Hidden Variables                | 2-32 |
| Parameter Passing               | 2-33 |
| Memo field support              | 2-34 |
| The TDBS Memo Editor            | 2-34 |

# Chapter 3: Multiuser Programming

| Introduction                            |      |
|---|------|
| Multiuser Overview                      |      |
| TDBS Multiuser Features                 | 3-4  |
| Exclusive or Shared files               | 3-6  |
| Explicit Record and File Locking        | 3-7  |
| FLOCK()                                 | 3-7  |
| RLOCK()                                 | 3-7  |
| UNLOCK                                  | 3-7  |
| WAIT4RLOCK([n])                         | 3-8  |
| WAIT4FLOCK([n])                         | 3-9  |
| Fielding Locking Conflicts              | 3-9  |
| Sample File Locking Handler             | 3-9  |
| Sample Record Locking Handler           | 3-10 |
| Transparent File Sharing                | 3-12 |
| Screen Update and Rollback on Collision | 3-13 |
| SET UPDATE BELL                         | 3-13 |
| Hybrid Automatic Record Locking         | 3-14 |
| TDBS Mailboxes                          | 3-15 |
| Establishing a Mailbox                  | 3-15 |
| Forcing a Mailbox Checkpoint            | 3-16 |
| Sending Mail                            | 3-16 |
| Receiving Mail                          | 3-17 |
| NEWMAIL([wa])                           | 3-17 |
| WAIT4MAIL([n])                          | 3-17 |
| ON NEWMAIL                              |      |
| USING_BOX field                         | 3-18 |
|   |      |

| Printer Support      |  |
|----------------------|--|
| WAIT4LPT(n[,s])      |  |
| Printer Control      |  |
| SET ALTERNATE        |  |
| Flat File I/O        |  |
| Flat File I/O Basics |  |
| Line Mode I/O        |  |
| Binary Mode I/O      |  |
|                      |  |

# **Chapter 4: TDBS Commands**

| Command Notation Conventions              |      |
|---|------|
| Element Types used in syntax descriptions |      |
| Summary of TDBS Commands                  |      |
| ? / ??                                    |      |
| @ CLEAR                                   |      |
| @ SAY GET                                 |      |
| @ TO                                      |      |
| ACCEPT                                    |      |
| APPEND BLANK                              |      |
| APPEND FROM                               | 4-24 |
| AVERAGE                                   |      |
| CLEAR                                     |      |
| CLEAR ALL                                 |      |
| CLEAR GETS                                |      |
| CLEAR MEMORY                              |      |
| CLEAR TYPEAHEAD                           | 4-31 |
| CLOSE                                     | 4-32 |
| CONTINUE                                  | 4-33 |
| COPY TO                                   |      |
| COPY FILE                                 |      |
| COPY STRUCTURE                            |      |
| COPY STRUCTURE EXTENDED                   |      |
| COUNT                                     |      |
| CREATE                                    |      |
| CREATE FROM                               |      |
| DECLARE                                   |      |
| DELETE                                    |      |
| DIR                                       |      |
| DO  |      |
| DO CASE                                   |      |
| DO WHILE                                  |      |
| DOTBBS                                    |      |
|   |      |

| EJECT  | 4-51   |   |
|--|--|---|
| ERASE  | 4-52   |   |
| FBREAD   | 4-53   |   |
| FBWRITE  | 4-55   |   |
| FCLOSE   | 4-57   |   |
| FCREATE  |  |   |
| FIND   |  |   |
| FLFIND   |  |   |
| FLREAD   |  |   |
| FLWRITE  |  |   |
| FOPEN  |  |   |
| FSEEK  |  |   |
| GO/GOTO  |  |   |
| HALT   |  |   |
| IF   |  |   |
| INDEX ON   | 4-73   |   |
| INPUT  |  |   |
| LOCATE   |  |   |
| NOTE/*/&&  |  |   |
| ON DISCONNECT                                    |  |   |
| ON ERROR   |  |   |
| ON ESCAPE  |  |   |
| ON KEY   |  | 1 |
| ON NEWMAIL                                       |  | 1 |
| PARAMETERS                                       |  |   |
| PRIVATE  |  |   |
| PROCEDURE  |  |   |
| PUBLIC   |  |   |
| QUIT   | 4-91   |   |
| READ   | 4-92   |   |
| RECALL   |  |   |
| RELEASE  | 4-97   |   |
|  |  |   |
| RENAME   | 4-98   |   |
| RENAME   |  |   |
| REPLACE  | 4-99   |   |
| REPLACE  | 4-99<br>4-101  |   |
| REPLACE<br>RESTORE                               | 4-99<br>4-101<br>4-102   |   |
| REPLACE<br>RESTORE<br>RETURN<br>RETURN TO MASTER | 4-99<br>4-101<br>4-102<br>4-103  |   |
| REPLACE<br>RESTORE                               | 4-99<br>4-101<br>4-102<br>4-103<br>4-104   |   |
| REPLACE  | 4-99<br>4-101<br>4-102<br>4-103<br>4-104<br>4-105  |   |
| REPLACE  | 4-99<br>4-101<br>4-102<br>4-103<br>4-104<br>4-105<br>4-106                                     | ( |
| REPLACE  | 4-99<br>4-101<br>4-102<br>4-103<br>4-104<br>4-105<br>4-106<br>4-107                            | ( |
| REPLACE  | 4-99<br>4-101<br>4-102<br>4-103<br>4-104<br>4-105<br>4-105<br>4-106<br>4-107<br>4-108<br>4-109 |   |

| SET CONFIRM4        |      |
|---------------------|------|
| SET CONSOLE4        |      |
| SET DATE4           |      |
| SET DECIMALS4       |      |
| SET DELETED4        |      |
| SET DELIMITERS      | -117 |
| SET DEVICE          | -118 |
| SET DISCONNECT      | -119 |
| SET DISPLAY RULES4  | -120 |
| SET DIVIDE BY ZERO4 | -121 |
| SET EDITOR4         | -122 |
| SET ESCAPE          | -123 |
| SET EXACT4          | -124 |
| SET EXCLUSIVE4      | -125 |
| SET FILTER4         | -126 |
| SET FIXED4          | -127 |
| SET FORMAT4         |      |
| SET FUNCTION4       | -129 |
| SET INDEX4          |      |
| SET INTENSITY4      | -131 |
| SET MARGIN4         |      |
| SET MEMOWIDTH4      | -133 |
| SET ORDER4          | -134 |
| SET PRINT4          | -135 |
| SET PRINTER TO4     | -136 |
| SET PROCEDURE       |      |
| SET RELATION        | -138 |
| SET SOFTSEEK        | -140 |
| SET TYPEAHEAD4      | -141 |
| SET UNIQUE          |      |
| SET UPDATE BELL     | -143 |
| SKIP                | -144 |
| STORE               | -145 |
| SUM                 | -146 |
| TEXT                |      |
| TYPE                | -148 |
| UNLOCK4             | -149 |
| USE                 |      |
| USE MAILBOX         |      |
| WAIT                |      |
| ZAP4                |      |
|                     |      |

# **Chapter 5: TDBS Functions**

| Function Overview         |
|---------------------------|
| Summary of TDBS Functions |
| ABS()                     |
| ACOPY()                   |
| ADEL()                    |
| AFIELDS()                 |
| AFILL()                   |
| AINS()                    |
| ALIAS()                   |
| ASCAN()                   |
| ASORT()/ADSORT()          |
| ASC()                     |
| AT()                      |
| BOF()                     |
| CAPFIRST()                |
| CDOW()                    |
| CEILING()                 |
| CHR()                     |
| CMONTH()                  |
| COL()                     |
| CRTRIM()                  |
| CTOD()                    |
| DATE()                    |
| DAY()                     |
| DBF()                     |
| DEC2HEX()                 |
| DELETED()                 |
| DESCEND()                 |
| DISKSPACE()               |
| DOTBBS()                  |
| DOW()                     |
| DTOC()                    |
| DTOS()                    |
| EMPTY()                   |
| EOF()                     |
| ERROR()                   |
| EXP()                     |
| FBEXTRACT()               |
| FBFILL()                  |
| FBINSERT()                |

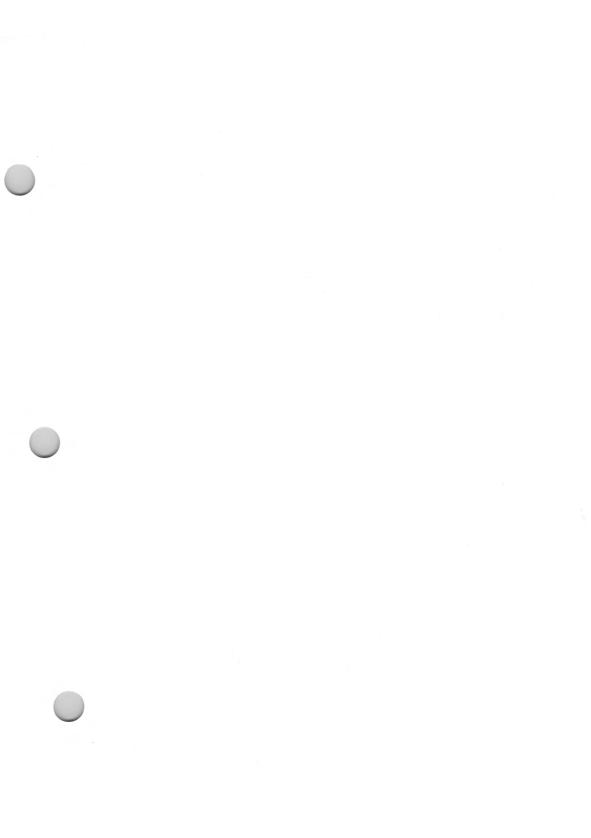
| FBMOVE()          | 5-52 |
|-------------------|------|
| FCOUNT()          |      |
| FDATE()           |      |
| FERROR()          |      |
| FIELD()           |      |
| FILE()            |      |
| FINDFIRST()       |      |
| FINDNEXT()        |      |
| FKLABEL()/FKMAX() |      |
| FLEN()            |      |
| FLOCK()           |      |
| FLOOR()           |      |
| FMAXLEN()         |      |
| FOUND()           |      |
| FSIZE()           |      |
| FTIME()           |      |
| GETENV()          |      |
| GETLPT()          |      |
| HARDCR()          |      |
| U U               |      |
| HEX2DEC()         |      |
|                   |      |
|                   |      |
|                   |      |
| INDEXKEY()        |      |
| INDEXORD()        |      |
| INKEY()           |      |
| INT()             |      |
| ISALPHA()         |      |
| ISINT()           |      |
| ISLASTDAY()       | 5-83 |
| ISLEAP()          |      |
| ISLOWER()         |      |
| ISSHARE()         |      |
| ISSTATE()         |      |
| ISUPPER()         |      |
| LASTDAY()         |      |
| LASTKEY()         |      |
| LEFT()            | 5-91 |
| LEN()             | 5-92 |
| LJUST()           |      |
| LOG()             | 5-94 |
| LOWER()           | 5-95 |
| LTRIM()           |      |
| LUPDATE()         | 5-97 |
|                   |      |

| MAX()                | . 5-98  |
|----------------------|---------|
| MESSAGE()            | . 5-99  |
| MIN()                | . 5-100 |
| MOD()                |         |
| MONTH()              | .5-102  |
| NDX()                | . 5-103 |
| NEWMAIL()            | . 5-104 |
| NEXTKEY()            |         |
| NMYUSERS()           |         |
| NUSERS()             |         |
| OPTDATA()            |         |
| OS()                 |         |
| PCOL()               |         |
| PROCLINE()           |         |
| PROCNAME()           |         |
| PROW()               |         |
| RAT()                |         |
| READKEY()            |         |
| RECCOUNT()/LASTREC() | .5-117  |
| RECNO()              |         |
| RECSIZE()            |         |
| REPLICATE()          |         |
| RIGHT()              |         |
| RJUST()              |         |
| RLOCK()/LOCK()       |         |
| ROUND()              | . 5-124 |
| ROW()                |         |
| RTRIM()/TRIM()       |         |
| SECONDS()            |         |
| SELECT()             |         |
| SETPRC()             | . 5-129 |
| SOUNDEX()            | . 5-130 |
| SPACE()              | . 5-132 |
| SQRT()               |         |
| STATENAME()          | . 5-134 |
| STR()                | . 5-135 |
| STUFF()              |         |
| SUBSTR()             |         |
| TIME()               | . 5-138 |
| TRANSFORM()          |         |
| TYPE()               | . 5-140 |
| UANSI()              | . 5-141 |
| UAUTH()              |         |
| UIBM()               | . 5-143 |

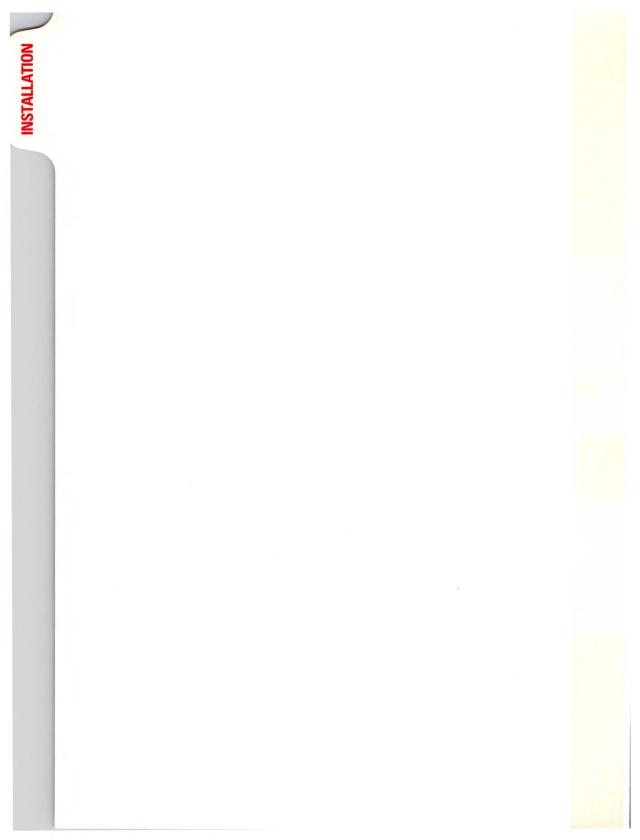
| ULINE()      | 5-144 |
|--------------|-------|
| ULOCATION()  | 5-145 |
| ULPEEK()     |       |
| ULPOKE()     |       |
| ULREPLACE()  |       |
| UMORE()      |       |
| UNAME()      |       |
| UNOTES()     |       |
| UPDATED()    |       |
| UPPER()      |       |
| UPRIV()      | 5-155 |
| USING()      |       |
| UWIDTH()     | 5-157 |
| VAL()        |       |
| VERSION()    |       |
| WAIT4FLOCK() |       |
| WAIT4LPT()   |       |
| WAIT4MAIL()  |       |
| WAIT4RLOCK() |       |
| YEAR()       |       |
|              |       |

# **Chapter 6: Technical Information**

| 6-1 |
|-----|
| 6-2 |
| 6-3 |
| 6-5 |
| 6-6 |
| 6-7 |
| 6-8 |
|     |



# INSTALLATION



# HOW TO USE THIS MANUAL

## READ THE NEXT TWO PAGES, EVEN IF YOU DON'T READ ANYTHING ELSE.

|                             | Many people only read their user's manual as a last resort. If you are one of those, the next two pages tell you where to find information when you need it.  |
|-----------------------------|---|
| Installation                | To learn how to install The Data Base System on your computer<br>for the first time, read the Installation section of this chapter.   |
| How to Start                | To learn how to compile programs and run them on your TDBS<br>system, read the sections of this chapter on Compiling TDBS<br>source files and Adding TDBS programs to a Menu.   |
| Introduction                | Chapter 2 is an introduction to The Data Base System. This chapter<br>will give you a frame of reference and introduce you to the concepts<br>behind TDBS. If you are already familiar with dBASE III +<br>programming, this chapter is all you will need to read to begin using<br>TDBS. |
| Multiuser<br>Considerations | Chapter 3 explains the special cases you must consider when<br>programming applications for multiuser access. You will learn here<br>about how TDBS automatically shares files without damage, and<br>how you can exercise more control in special cases if you need to.                  |
| Command<br>Reference        | Chapter 4 is a reference guide for TDBS commands. Each com-<br>mand is listed here (in alphabetical order) so you can find the<br>details of its operation easily. This is a reference, not a tutorial<br>section.  |
| Function<br>Reference       | Chapter 5 is a reference guide for TDBS functions. Each function<br>is listed here (in alphabetical order) so you can find the details of<br>its operation easily.  |
| Technical                   | Chapter 6 contains the Technical specifications of interest to<br>programmers who wish to write custom utilities or programs which<br>require more technical details. Included is a bit map and record  |
|                             |   |

layout of all data structures with which you might need to interface if you write specialized utility programs etc.

You should begin by reading the overview section of this manual to learn the concepts of TDBS in a general way. Then you may begin to write TDBS programs of your own or compiling dBASE III + programs to run under TDBS.

# What's new in TDBS 1.2?

TDBS 1.2 has several enhancements over TDBS 1.1. However, all .TPG programs which were compiled and running in TDBS 1.1 will continue to execute properly under TDBS 1.2. The following is a summary of the most important TDBS 1.2 enhancements.

- Flat File I/O has been implemented to allow direct access to any type of file. Both text mode (line at a time) and binary mode are available.
- Rapid text searching is now possible using the text mode Flat File I/O command FLFIND.
- The SOUNDEX function is now available to allow "sounds like" keyword operations.
- The boolean functions .AND., .OR. and .NOT. now operate on numeric variables. This allows true bit operations on up to 32 bit numeric values.
- The SELECT option on READ allows direct entry into the memo editor without user intervention.
- READONLY access allows use of protected .DBF files.
- FINDFIRST, FINDNEXT, FDATE, FTIME, and FSIZE functions provide robust file handling capability.
- SET ALTERNATE now allows APPEND operation for journaling.
- Faster operation and smoother scheduling for better overall system operation.

# Installing TDBS on a TBBS system

TDBS is comprised of two portions.

- 1. The TDBS compiler which translates ASCII source code into compiled token program (.TPG) files.
- 2. The TDBSOM option module which allows these compiled token program files to be executed online on a TBBS system. This option module manages all multiuser actions, and contains the library routines which allow a .TPG program to interface to TBBS.

Note: TDBS is designed to be installed on a working TBBS system. If you have not yet installed your TBBS system, then that is the first step. Do not attempt to install TDBS until TBBS has been installed and is operating correctly.

## Installing the TDBS compiler

The TDBS compiler is contained in a single file named TDBS.EXE and is a stand alone DOS program. Installation consists of copying this file to your hard drive. To allow its use from any directory during program development, the directory you place the TDBS.EXE file in should be listed in the DOS PATH directory list.

## Installing the TDBS Option Module

The TDBS option module is installed by copying the files TDBSOM.EXE and TDBSEMSG.TXT to your hard disk. These files should be placed in the same directory as your TBBS run time program MLTBBS.EXE. You then must modify the RUNBBS.BAT command which invokes TBBS by adding the following command line switch:

### /O:TDBSOM

As an example if the calling line before installing TDBS was:

#### MLTBBS /U /F

After installing TDBS this calling line would read:

#### MLTBBS /U /F /O:TDBSOM

If you must place the TDBSOM.EXE file in another directory, you must list the entire specification of where you placed it. As an example, if you placed the TDBSOM.EXE file in the D:\TDBS directory, then the calling line would be:

#### MLTBBS /U /F /O:D:\TDBS\TDBSOM

If you already have another option module installed, then separate the module names with commas on the /O: specification. As an example, if you already have the SYSOM option module installed the specification would be:

#### MLTBBS /U /F /O:SYSOM,TDBSOM

The position of TDBSOM in the option module list is not significant.

### **TDBSOM Memory Requirements**

Once the TDBS option module is installed, your system will require more memory to operate than it did before you installed TDBS. The extra memory required for this option module is:

#### OM CODE memory = 123,666 bytes OM UDATA memory = 49,152 bytes/user conventional memory or 48k/user EMS memory

The OM CODE memory must fit in the 640k conventional memory on your computer, while part or all of the OM UDATA memory may go in either conventional or EMS memory.

Note: Because TDBS uses the maximum OM UDATA memory allowed by TBBS, the OM UDATA memory used by any additional Option Modules doesn't need to be considered when calculating required memory size. The only additional memory any other option modules will use is the OM CODE memory they require. Add the amount of OM CODE memory required by each of the other Option Modules you are running to that required by TDBS and use that total value as the OM CODE memory requirements in the following formulas to obtain the required memory to run all of the option modules simultaneously.

The total extra memory required may be calculated as 123,666 + (49,152\*users). For example if you have 5 lines configured in CEDIT, there are really 6 users (the console is a user too) and thus the memory required to install TDBSOM on a 5 line 2.1M system is:

123,666 + (49,152\*6) = 418,578 bytes.

Note: The 123,666 bytes of OM CODE memory must be in conventional memory. The 294,912 bytes of OM UDATA memory may be EMS memory, or divided (in 48k "chunks") between EMS and conventional memory. Thus the maximum EMS memory required by the TDBSOM (on a 32 line system) would be 1,584k if all users were placed in EMS memory.

## Adding TDBS programs to a Menu

When the TDBSOM option module is installed, a new command type becomes available for use in menus. This command becomes like any other TBBS menu command and may be used in any menu as you wish. The new command is:

 $\mathbf{TYPE} = 200$ 

This command means "execute a TDBS program" and the Opt Data field syntax is:

OPT DATA = [d:][\path\]filename[.ext] [/Q][/U:n][/OU][/HP]

*d*: specifies the drive with the compiled TDBS program file. If not specified, the default drive is assumed.

\path\ specifies the subdirectory which contains the compiled TDBS program file. If not specified, the current path for the specified drive (or default drive if d: not specified) is assumed.

filename is the name of the TDBS compiled program.

ext is the extension of the TDBS compiled program. If no extension is given, then the normal one of .TPG is assumed.

/Q is an optional switch which will cause the TDBS copyright and normal exit messages to be suppressed when this program is run.

/U:n is an optional switch which limits the number of simultaneous users of TDBS. If "n" or more users are currently in TDBS when this menu item is invoked, TDBS will exit immediately with an error. Note: this option cannot be used to increase the maximum number of users above your licensed limit.

/HP is an optional switch which restricts all file accesses by this program to the HOMEPATH() directory (see below). Any attempt to access a file in another directory will generate an error if this switch is specified.

/OU is an optional switch which changes the operation of the ULINE() function to the format it had prior to TBBS 2.2. Some older programs may require this switch to operate correctly.

Example:

Entry: <R>un sample TDBS program KEY = R TYPE = 200 OPT DATA = D:\TDBS\SAMPLE /Q

This will run the compiled TDBS program named **SAMPLE.TPG** which resides in the **TDBS** subdirectory on drive **D**: and will suppress the TDBS copyright message.

Because all TDBS programs are invoked by using the normal TBBS menu command structure, all security, authorization, and privilege options are available. Any number of menu entries may run TDBS programs, just as any other TBBS command.

#### The HOMEPATH directory

The drive and path specified (or defaulted) in the Opt Data specification when a TDBS program is loaded becomes known as the HOMEPATH directory for that program run. This is the drive and directory where the .TPG program resides, and all database files are assumed by default to be there also while the TDBS program is running.

Unless overridden by an explicit path specification, TDBS will assume that all files referenced by this run of the TDBS program reside in the HOMEPATH directory. Thus the HOMEPATH directory in a TDBS program takes on the same characteristics as the default logged on directory does when a stand alone dBASE program is run under DOS.

The HOMEPATH directory may be overridden by explicitly specifying a drive and/or path in any file specification in the TDBS source code (unless the /HP switch was specified in Opt Data). The HOMEPATH default allows you to easily group a program and its files in a single sub-directory without ever specifying that drive or subdirectory explicitly in the TDBS source. This allows easy portability of applications.

The TDBS program may learn what its HOMEPATH directory and drive are while it is running by using the TDBS extended function HOMEPATH() which returns a string with the drive and path designator.

# **TDBS CONFIG.SYS considerations**

Since TDBS can run programs which open many files, you may need to increase the number of FILES = and BUFFERS = in your DOS CONFIG.SYS file. To determine if this is so, you need to know the following about how TDBS uses DOS file resources.

FILES =

TDBS will only use one FILE = block for a file, even if it is opened by many users at the same time. Thus if you are running many copies of the same program, you only need to count the maximum number of files it may have open at once, since all users of the program will use the same DOS FCB entry for each file.

If, on the other hand, several users are running different programs that access many different files, then you will need to consider this and adjust the FILES = specification accordingly. The maximum files that TBBS will allow all users to have open at once is 150. The normal TBBS installation sets the FILE = to a value which allows two open files per user. You may need to adjust this higher if you anticipate a lot of different files open at once under TDBS. The notification you receive if you don't have FILES = set high enough, is that TDBS will report the error message:

#### **TOO MANY FILES OPEN**

You add more files by modifying the FILES = value in the file CONFIG.SYS which resides in the root directory of the DOS boot drive (usually C:). See also **DOS File Limits and FCB Sharing** in Chapter 2 for more information.

#### **BUFFERS** =

This parameter is less important in that TBBS/TDBS will not malfunction if it is set improperly, it will just suffer performance problems when the system is heavily used. This setting is a tradeoff of conventional memory usage against performance. Usually you do not need to increase this setting for TDBS.

# Compiling TDBS Source Files

|                 | The TDBS compiler reads one or more source files and optionally<br>generates a compiled token program file and listing file. The TDBS<br>command line syntax is:   |
|-----------------|--|
|                 | <b>TDBS</b> [@]sourcefile [tpgfile [listingfile]] [option switches]  |
| sourcefile      | [d:][path]filename[.ext] of the main .PRG source file to be com-<br>piled.   |
| @sourcefile     | [d:][path]filename[.ext] of the .TDB multiple file specification file.<br>This is a file which lists each file to be compiled, and allows optional<br>switches on each file. It is used either in the rare case where the<br>autocompile feature does not find all required files, or where<br>different options are desired on different portions of the compila-<br>tion. (see description below). |
| tpgfile         | [d:][path]filename[.ext] of the .TPG token program output file. This is the file that TDBSOM can execute online.   |
| listingfile     | [d:][path]filename[.ext] of the .LST file which contains the com-<br>piled listing with line numbers and all included files.   |
| option switches | The following Command Line Option Switches are supported by the TDBS 1.2 compiler:   |
| /L              | Send a listing to the console.   |
| /DB             | Insert special debug information in the .TPG file. This will allow<br>the full source line to be listed at run time in case of an error.<br>Normally, only the source line number, along with the procedure<br>or subroutine name is listed. This option enhances debug, but<br>creates a much larger .TPG file than would otherwise be required.  |
| /XDB            | Don't insert special debug information (default)   |
| /S              | Search .PRG directory for missing procedures. This is the autocompile feature, and is usually wanted. (default)  |
| /XS             | Don't search for undefined procedures. This option disables the TDBS autocompile feature.  |

- /XC Comment lines which begin with "\*@" are not comments. TDBS will logically strip the \*@ and compile these lines. This allows TDBS Specific commands to be embedded in a program which can still be run correctly under other dBASE compilers or interpreters.
- /GLOSS bytes This compiler switch is used to change the allocation of the run time global glossary from the default. The global glossary is the TDBS symbol table, and consists of two portions. The static portion is generated at compile time and consists of all symbols to which direct reference is made in the source. The run time portion consists of symbols which are not defined until the program executes. Such symbols include field names that are not referenced, and variable names which are only referenced through macros or RESTORE operations.

By default the TDBS 1.2 compiler will assign a default value for the run time portion of the global glossary of either 10% of the static glossary size or 1024 bytes whichever is larger. In order to assure total compatibility for any programs which were developed using TDBS 1.0 or TDBS 1.1, the total glossary size (static plus run time) will never be less than 4300 bytes.

These defaults will almost always be satisfactory, but you may use the /GLOSS compiler switch to change this run time allocation. The maximum total global glossary (static + run time) allowed is 12k. When the total grows beyond 4300 bytes, memory is allocated from the disk I/O buffers so asking for extra space can slow down performance of the program.

If TDBS cannot allocate the static glossary space at run time, the program will abort when it is selected. If the requested run time glossary space is not available, the program will still begin execution as long as at least a 128 byte run time glossary and/or a 1024 byte I/O buffer space is available.

If a program runs out of run time glossary space while executing, the usual error message you will receive is "Unable to create variable". If you request too large a run time glossary resulting in not enough I/O buffer space, you will receive out of memory error messages.

Example: /GLOSS 2000

This switch requests that the run time glossary have 2000 bytes.

- /DIBUF Because of the limited memory available (see TDBS run time memory usage) TDBS only reserves memory for three index buffers. This switch tells the TDBS run-time module to dynamically allocate extra index buffers from any free space in the global glossary. The compiler statistics will display the maximum number of DIBUFs the program can allocate at run time. The /GLOSS command can be used to increase the run time glossary to allow more DIBUFs to be allocated at the expense of memory in the work pool. DIBUFs can dramatically speed up many programs.
- /FGLOSS bytes When /DIBUF is used, this switch indicates the minimum number of run-time glossary bytes to keep free for a single instruction. By default /FGLOSS 256 is used. This is sufficient for most programs, but if a program gives any form of "out of memory error" when you add /DIBUF, but runs normally without /DIBUF, then it requires a /FGLOSS with a value larger than 256.
- /GETPOOL bytes This switch allows you to set the size of the memory area used by TDBS to hold GET command information for READ. By default /GETPOOL 2248 is used for compatibility with previous versions. You may set the GETPOOL to any size from 0 to 16,535 bytes. Note: The memory used by the GETPOOL is removed from the work pool. /GETPOOL with a value less than 2248 will increase work pool space when only a small number of GET commands are used.
  - /REL11 This switch causes the TDBS 1.2 compiler to generate .TPG files in the format of the TDBS 1.1 compiler. It must be used if you are compiling a program that is to be run on any TDBSOM prior to TDBS 1.2. The compiler will also disallow any use of commands or functions that would not work on TDBS versions prior to 1.2.

Note: The format of the .TPG file changed with version 1.2 to allow performance improvement. TDBSOM 1.2 will detect prior version .TPG formats and execute them correctly, however this run-time format conversion does cause a program to run slower than it will if it is re-compiled in the TDBS 1.2 format.

## **Multiple Procedure Control File**

Most TDBS programs will have more than one source file. This allows modular development, and keeps any individual portion of the program to a manageable size. However, the TDBS compiler needs to know about all of the source files it must compile and link together to produce a complete program file. The .**TDB** multiple procedure control file allows you to specify explicitly which files are to be included in the compile. A .**TDB** file is invoked by using it as the source file name in the TDBS compiler command line, but preceding it with an "@". Control files have the following syntax:

- 1. The general syntax is identical to .PRG files. "\*", "NOTE", and "&&" are comments, blank lines are ignored, and ";" indicates continuation.
- 2. Each non comment line specifies the next file to compile. This line has the format:

[d:\path\]filename[.ext] [switches] [&& comment]

If [d:\path] is not specified, then the file is assumed to be in the same directory as the .TDB control file itself.

If [.ext] is not specified, .PRG is assumed.

[switches] /DB or /XDB may be specified for each file to limit which files compile with the special debug information. Note: at least one space must be placed between the file name and any switches. If no switch is given, the DB option defaults to the setting given (or defaulted) on the command line.

Leading blanks are allowed on .TDB file lines.

3. The first file specified is the MAIN program. Execution will begin with the first statement of this program.

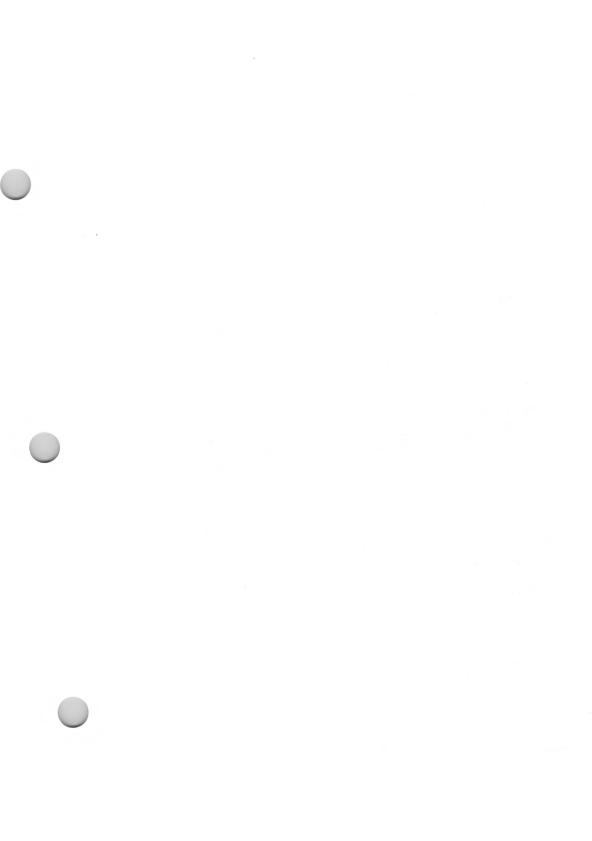
#### The TDBS Autocompile feature

When a .TDB control file is not used, TDBS is given the name of the MAIN program file. If the /XS option switch is specified, then only the named file will be compiled. However, by default, TDBS assumes the /S switch and uses SET PROCEDURE TO, SET FORMAT TO and DO commands to automatically find and compile all of the procedures required to build the program. This mode usually allows a main TDBS program and its associated procedure and subroutine files to be compiled into a single .TPG file with no modifications, and without the trouble of building a control file.

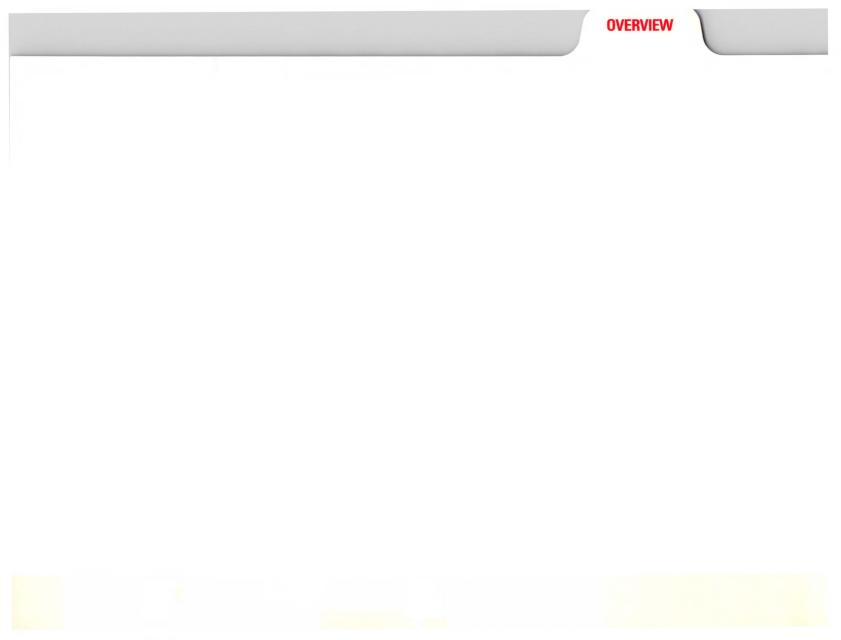
When TDBS is in autocompile mode the main program (the one specified on the command line) is compiled first. Any files named by DO, SET PROCEDURE TO, and SET FORMAT TO commands which are undefined at the end of the main program are put on a list. The first file on this list is compiled next, and the procedure is repeated until all external references are resolved.

#### Special considerations when using autocompile

- 1. You may not have procedure files and subroutines which have the same name.
- 2. All program files must have the extension .PRG. This means that files referenced on a SET FORMAT TO must be renamed from .FMT to .PRG in order to be found correctly by autocompile.
- 3. All program files must reside in the same disk directory as the main program which is referenced on the TDBS command line.



# **OVERVIEW**



### Introduction

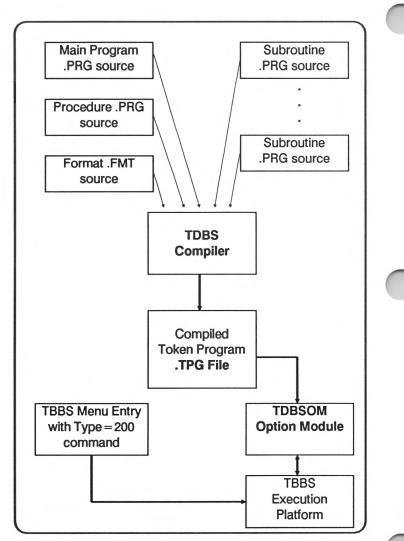
The Data Base System (TDBS) provides the capability to write dBASE (xBASE) language programs which can be run under TBBS. With multiline TBBS this means that up to 64 users can run the same program at the same time. Because each program becomes a TBBS menu entry, an unlimited (except by disk space) number of programs may be placed online. TDBS handles file locking and other multiuser considerations transparently, making writing multi-user applications as easy as possible. It also fully supports the dBASE III Plus standard explicit file and record locking capabilities, but these are not required for file protection, only for explicit program control in special circumstances.

In this chapter we will introduce you to TDBS, and the overall structure and operation of it. Issues of compatibility with dBASE III Plus, dBXL, Quicksilver, FoxBASE, Clipper, etc. are discussed as well as TDBS extensions to the language and use in the TBBS multiuser environment. Finally some of the more technical considerations of TDBS (e.g. maximum memory and program sizes) will be introduced. It is assumed you are already familiar with the dBASE language and only need to know about the TDBS dialect.

### **TDBS** structure

A TDBS program begins with one or more ASCII source files containing dBASE language instructions. These source files are compiled by the TDBS compiler and become a single "token program" (.TPG) file. The TDBSOM option module directly executes these .TPG files as specified by TBBS menu entries.

TDBS is technically known as a P-code compiler. This means that the compiler translates the source code into a "pseudo-code" form which is executed by the TDBSOM option module. This implementation allows faster execution and smaller online program sizes than the dBASE III Plus interpretive execution of the source file. It also allows the TDBSOM option module to retain enough execution control to fully utilize the TBBS scheduler for system efficiency, and prevent errant programs from damaging other TBBS users or crashing the system. Finally, it allows programs to be distributed without releasing the source code. The following diagram shows the relationships between the TDBS program source, .TPG program file, and the TDBS compiler, TDBSOM execution module, and the TBBS platform on which it runs.



Note that to execute the program only the .TPG file and the TDBSOM option module are required. The program source and TDBS compiler are not required at run time.

# **TDBS compatibility level**

The target level of compatibility for TDBS is the dBASE III Plus program language. TDBS is not 100% compatible with dBASE III Plus and these differences arise for the following reasons:

- TDBS is a compiler while dBASE III Plus is an interpreter. Thus some dBASE commands make no sense in a compiled environment and are not present in TDBS.
- TDBS data memory is limited to 48k per user, while dBASE is free to use as much of the 640k memory as is needed. TDBS thus has some memory limit constraints which dBASE does not have.
- The full screen interactive built in commands (such as EDIT, BROWSE etc.) are not implemented. The full screen @ SAY ... GET and READ commands must be used to explicitly code such constructs.
- Because TDBS runs under TBBS, extensions have been added to the language to take advantage of that environment. These enhancements are not backward compatible with other dBASE language dialects.
- TDBS has implemented **Transparent Locking** and **Automatic Locking** in addition to the dBASE standard **Explicit File and Record Locking** for shared file access. TDBS also provides **Screen Update and Rollback on Collision** allowing transparent *visual* record sharing as well. These capabilities allow TDBS to handle multiuser file accesses smoothly in ways which are impossible in most dBASE language dialects. See Chapter 3 for a full discussion of transparent, automatic, and explicit file and record locking multiuser programming considerations, as well as transparent shared screen updates.

## Introduction to the TDBS Language

TDBS allows you to write programs in the dBASE language which can then be run by users on a TBBS system. This chapter will give an overview of that language and how it is used to access files and write program control structures. It will also discuss the compatibility of TDBS to the *de facto* standard xBASE language. This chapter is not intended to teach you the language if you have never programmed before. Supplementary materials on the dBASE language are widely available in your local bookstore, to teach you programming at both beginning and advanced levels.

# Source Syntax Compatibility

TDBS will directly compile both **.PRG** source files and **.FMT** format files which were written to be used by dBASE.

TDBS follows 100% of the syntax rules of dBASE III Plus. Source lines are limited to 254 characters in length, the comment conventions are all identical, Upper and Lower case are treated the same, and the legal character set is the same. In this regard, all source coding conventions and methods which work with dBASE III Plus will work identically with TDBS.

While dBASE III Plus exhibits some "quirkiness" when a variable or field name is longer than 10 characters, TDBS will allow any number of characters to be used, but the name is only significant to the first 10 characters.

All commands may be abbreviated to four characters, and function names may be abbreviated to their shortest significant length, which is also usually four characters.

# **Conditional TDBS commands**

TDBS has conditional compilation capability so that lines may be marked as comments for other xBASE dialects, and still contain valid TDBS commands. The syntax of this extension is:

\*@<command line>

Because this line begins with an asterisk (\*) it is normally treated as a comment. While it will always be treated as a comment by other xBASE dialects, if you set the /XC switch on the TDBS compiler, the \*@ is logically removed from the front of the line and the command will be compiled as part of the program source. This allows you to have TDBS specific commands in source files which will be ignored by other xBASE language dialects.

Additionally the variable name **TDBS** may be used to determine if a program is running under TDBS. If this variable is declared **PUBLIC**, all other xBASE language dialects will assign it a value of .F. by default. Under TDBS, this variable will be assigned the value .T. and thus may be used to determine if a program is running under TDBS or another xBASE dialect.

# **Database Files**

Database files store data which most TDBS commands can operate on. A database file is made up of records which in turn are comprised of fields. Each field is given a name (up to 10 characters long) in TDBS, and has a data type and length. This fully defines to the TDBS language what sort of data is stored in each field. Up to 10 database files may be open at the same time in TDBS. A file is opened by the USE command which positions it to the first record. The fields of the current record are immediately available for use in any TDBS expression, as part of any command which can reference data. The SKIP and GOTO commands change the record number and perform the read function from the database file. Reading is implied rather than explicitly specified in TDBS. When a database file is positioned, the data "appears" in each named field for that record immediately when the field is referenced.

#### **Index Files**

Index files are used to allow rapid access and data ordering in a database file. Each index file is a sorted list of key expressions calculated from each record in the corresponding database file. This file is in a special format which allows TDBS to rapidly look up the indexed keys to locate the record they are part of. When a USE command specifies a master index file associated with a database file, then the order of the database file appears to be the sorted order of the keys in the index file. This is a powerful capability for organizing access to database files

More than one index file may be associated with a database, but only one is "in control" at any time. The remainder of the index files must be listed, however, so that TDBS will know to keep them updated if any of the fields they index are updated, or if new records are appended to the database file.

#### **Commands and Functions**

The xBASE language which TDBS uses is implemented as a series of commands and functions. Each source line begins with a command which indicates the overall action that line will perform. The remainder of the line is comprised of either keywords or expressions which control the specific action of that instruction.

The xBASE language expression implementation is responsible for much of its flexibility and power. A robust and complete expression evaluator allows much work to be done in a single expression. While memory variables, literal values, and database fields may be part of an expression, a function may also be used as any argument. A function returns a value (which has one of the four allowed types) and behaves identically as a variable of the same type would in an expression.

Functions can perform data transformations, return status values, do keyboard input, and perform control functions as they are evaluated in an expression. TDBS has extended the already large dBASE library of functions to give you more capabilities. See Chapter 5 for a complete list of TDBS functions.

# Data Types

When you store information in databases or memory variables, the data must be one of four data types. These types are character, numeric, date, and logical. TDBS is 100% compatible with these data types. The following are the characteristics of each data type:

**Character:** This data type can store "strings" of characters up to 254 characters in length. It is used for such things as names and addresses etc. Any valid character may be stored in a string.

Numeric: This data type can store numbers up to 19 digits long (including the plus or minus sign and decimal point). Numeric accuracy is 15.9 digits, excluding the decimal point, which means that the 15 most significant digits in the number will be reliable. When non-zero numbers are compared, numeric accuracy is reduced to 13 digits. While there are only 15.9 significant digits, the range of values is much larger. The largest allowable number is  $10^{308}$ . The smallest positive number is  $10^{-307}$ .

**Date:** This data type stores dates in MM/DD/YY format. Date arithmetic is supported, and dates may be in the range from 01/01/100 through 12/31/2099.

Logical: This data type stores one of two conditions. These are referred to as true (.T.) or false (.F.). Though always stored as true or false, they may be optionally entered or displayed as Yes (.Y.) and No (.N.) if desired.

When these data types are stored in the MEMVAR area of your program they take the following amount of memory each:

Character: 2 + number of characters Numeric: 9 bytes Date: 9 bytes Logical: 2 bytes

# Memory Variable Arrays

|                 | TDBS 1.2 has also extended the dBASE language to allow the definition and use of one-dimensional memory variable arrays. The limit on the number of arrays you declare and the size of these arrays is determined by the MEMVAR memory limit of 6k.   |  |
|-----------------|---|--|
|                 | Arrays may be passed as parameters to procedures and they have<br>the same domain rules as memory variables (see Parameter Passing<br>and Memory Variable Domains later in this chapter). Additionally<br>there are a number of extended functions to allow you to insert,<br>delete, sort, scan and fill an array with values.   |  |
|                 | In any expression, an array element may be used just as any memory<br>variable. Array elements are referenced by placing their subscript<br>in square brackets following the array name. An entire array is<br>referenced when the array name is used without a subscript.  |  |
| Example:        | <pre>DECLARE sample[20] STORE 1 TO sample[1], sample[5] sample[2] = sample[1] + sample[5] ? sample[2] &amp;&amp; Result: 2</pre>  |  |
|                 | This example shows how array elements may be used as normal memory variables. The only exception is that the SAVE and RE-STORE commands do not save and restore array elements.   |  |
| Array Creation: | Arrays may be created by the DECLARE, PRIVATE, and<br>PUBLIC commands. DECLARE and PRIVATE operate identi-<br>cally in defining an array in the current procedure domain.<br>PUBLIC defines and array with global domain. If an array has<br>been defined already in the current procedure domain and a<br>DECLARE or PRIVATE is issued with the same array name, then<br>the current array is released and a new, empty array with the<br>specified dimension is defined in the current procedure domain. If<br>an array was defined as PUBLIC, a new PUBLIC command issued<br>on the same array name will release the current array and define a<br>new empty array with the new dimension. |  |
|                 | At the time of an array's creation, all of its elements are undefined.<br>Each element may be defined as a separate data type, and an array<br>can have a mixture of defined and undefined elements.  |  |

|                                      | •  |
|--------------------------------------|--|
| RELEASEing<br>arrays and<br>elements | An array may be undefined by using the RELEASE command. A<br>RELEASE ALL [LIKE/EXCEPT < skeleton >] will select entire<br>arrays in the same manner as it selects normal memory variables to<br>release. An array selected in this manner has all of its elements<br>released and the array structure itself is also released. An entire<br>array is also released if only the array name is specified.  |
|                                      | A RELEASE of one or more specific array elements will only<br>release that element. The array will remain defined, and any other<br>defined elements not specified in the RELEASE command will<br>retain their values. Even if every individual element of an array is<br>specified explicitly on a RELEASE command, the array itself will<br>remain defined.  |
| Implicit<br>Release                  | An array will also be released if either the PUBLIC or PRIVATE<br>command are issued with the array name and no dimension. In that<br>case, the current array is released and the normal PRIVATE or<br>PUBLIC action is taken which turns that name into a scalar (non-<br>array) memory variable. An array is also implicitly released if a<br>STORE TO or assignment command (=) is issued on the same<br>name without a subscript. As an example:                                       |
|                                      | <pre>DECLARE var1[20] var1[1] = 5 var1[2] = "ABC" var1 = 1</pre>   |
|                                      | In this example, var1 is defined as a 20 element array and two of its elements are defined. However, when the "var1 = 1" command is executed, the array var1 is released and a scalar memory variable is created in its place. Note: This only happens when the array is defined in the current domain. If an array has been hidden using a PRIVATE command, then using the array name as a normal memory variable at the current level will not release the hidden array or its elements. |
| Array Memory<br>Requirements:        | When an array is declared, two bytes of MEMVAR space are<br>immediately used for each potential element in the array. An<br>additional three bytes are used for a fixed header. Each element<br>takes the same space it would as a normal memory variable and only<br>when it is defined. Thus DECLARE array[10] takes 23 bytes plus<br>the size of each element as it is defined.   |
|                                      |  |

# **Operators and Precedence**

|                        | TDBS is 100% compatible with dBASE III Plus for all expression<br>operators and their precedence. All operators and data type<br>operations are 100% supported. Operators fall into three<br>categories as follows: |
|------------------------|---|
| Mathematical operators | Perform basic arithmetic on numeric values. Additionally the + and - operators may be used to concatenate character strings.  |
| Relational operators   | Compare two values and return a logical value of .T. or .F. depend-<br>ing on the outcome of the comparison.  |
| Logical operators      | Produce a .T. or .F. result after comparing two or more expressions<br>that use mathematical or relational operators. Extended in TDBS<br>1.2 to also allow 32 bit operations on numeric values.                    |

#### Mathematical operators:

- + Adds two numbers, or adds a number and a date giving a date, or concatenates two character strings.
- Subtracts two numbers, subtracts two dates giving a number, or concatenates two character strings.
- \* Multiplies two numbers.
- / Divides two numbers.
- ^ or \*\* Exponentiation of one number by another (requires math coprocessor or 486 with internal math processor).
  - () Parenthesis are used for grouping and overriding the standard order of operator precedence.

When TDBS evaluates expressions, it follows the standard order of precedence (as opposed to a strict left-to-right order). The order of precedence for mathematical operators is:

- 1. Unary + and signs
- 2. Exponentiation
- 3. Multiplication and Division
- 4. Addition and Subtraction

#### **Relational Operators**

- Less than. For Example: 1 < 10 is true (.T.), "A" < "B" is true, and 12/31/88 < 01/25/89 is true.
- > Greater than.
- = Equal.

<

- < > or # Not Equal.
  - < = Less than or Equal to.
  - > = Greater than or Equal to.
    - \$ Is first string embedded in the second string. Example:
       "dog" \$ "Hot dog and a beer" is true.
       Note: Case is significant in this comparison.

There is no order of precedence in relational operators. All relational operations are performed left to right.

#### **Logical Operators**

| AND. | Returns .T. if both arguments are true, otherwise .F. |
|------|---|
|------|---|

- .OR. Returns .T. if either argument is true, otherwise .F.
- .NOT. Inverts the value of the following argument.

NumericFor numeric arguments the numbers are internally converted to 32Booleanbit integers, a logical bitwise AND, OR, or NOT is performed, and<br/>the 32 bit result is returned as a number.

The order of precedence for logical operators is:

1. .NOT. 2. .AND. 3. .OR.

When operators of different types are mixed, the order of precedence is: Mathematical, then Relational, then Logical. Parenthesis may always be used to override the standard order of precedence.

# **Program Structure and Control**

TDBS is a block structured programming language. This means that it does not have labels and branching within modules. It provides instead all of the block structured capabilities needed to do full programming flow control. Each function is written in modular form, and the modules are called as needed to perform the functions. TDBS is 100% compatible with all standard language control structures.

The three program control structures available in TDBS are:

```
IF < condition >

...

[ELSE ...]

ENDIF

DO WHILE < condition >

...

ENDDO

DO CASE

CASE < condition >

...

CASE < condition >

...

[OTHERWISE

...]

ENDCASE
```

Since the < condition > qualifiers may be complex expressions, these control structures allow you to generate any program flow which is required. In addition, the DO WHILE structure has two "non structured" branching commands which can be quite useful. These are **EXIT** which branches to the next instruction after the ENDDO and **LOOP** which branches back to the DO WHILE command.

# Subroutine Calling

TDBS also provides a complete subroutine calling and return structure with optional parameter passing. In the dBASE language subroutines come in two "flavors", sub programs and procedures. In TDBS there is no difference between these two types of subroutines. They exist as different entities in dBASE III Plus for performance and memory reasons which are not present in a compiled language. The TDBS compiler recognizes subroutines in both forms and properly compiles them, however there is no execution difference between them in TDBS.

Subroutines are called using the command:

**DO** subroutine [WITH < parameter list > ]

Program control passes to the program (or procedure) named subroutine and the return address is remembered. The subroutine exits back to the calling program with the command:

#### RETURN

The subroutine is named either by having its source placed in a separate .PRG file, or by having the command:

**PROCEDURE** subroutine

as its first line. TDBS allows inline procedures in any source file. That is you may end one program and begin another simply by placing a new PROCEDURE command in the source file. The TDBS compiler will recognize this and compile the procedures as separate subroutines.

To pass parameters to a subroutine the calling command uses the optional WITH < paramlist> capability, and the receiving subroutine must have as the first active statement in it the command:

**PARAMETERS** < paramlist >

and <paramlist> must have the same number of arguments on both the PARAMETERS and the WITH command.

# Screen and Keyboard I/O

The dBASE language provides extensive screen and keyboard formatted I/O commands. TDBS implements 100% of these commands. However, some of them require that the user be configured for ANSI and be on a terminal which is either ANSI or VT-100 compatible to operate. Other commands may be used on either ANSI or non-ANSI terminals as follows:

#### I/O Commands requiring ANSI terminals

@ x,y @ x,y SAY @ x,y GET @ x,y CLEAR @ x,y TO x,y

READ

# I/O commands not requiring ANSI terminals

INPUT

 $? < \exp \text{list} >$ 

?? < exp list >

WAIT

CLEAR

TEXT ... ENDTEXT

There are also several functions for fine control of keyboard input. None of these functions require ANSI or VT-100 compatible terminals.

TDBS requires that a terminal (or terminal emulator) be ANSI or VT-100 compatible to use the full screen formatted display and input commands. The terminal should support all of the terminal control functions listed on pages 2-23 and 2-24 of your TBBS manual. If you use the SET COLOR TO command to implement color then the terminal must support all of the functions listed on page 2-24 of the TBBS manual under the Set Graphics Rendition ANSI sequence. If you do not use SET COLOR TO, then only normal and reverse video, and high and low intensity need to be supported.

If a screen command is executed which requires ANSI terminal codes, and the user is not configured for ANSI support in his TBBS profile, then an error message will result. In order to avoid this, the program may use the TDBS function UANSI() to test the user's profile to see if ANSI support is turned on. Alternate screen routines may also be written keyed by the user's profile setting.

Keyboard input may be either the normal VT-100 or ANSI mapping (shown in the next section) or IBM PC scan-code mapping. TDBS 1.2 supports IBM PC scan code mapping (also sometimes called "doorway mode") transparently so no special programming is required for either mode. IBM PC scan code keyboard mapping sends a single null character followed by the IBM PC scan code of the function key pressed. With scan code keyboard mapping all function keys work as they do from the local console keyboard.

### Keyboard mapping

TDBS supports all of the normal single character dBASE language input control function characters. In addition, to allow ANSI, VT-52 and VT-100 terminal keyboards to use extended keys directly, the following escape key sequences are mapped to control keys:

| Key(s)          | Maps to                      | Function Performed        |
|-----------------|------------------------------|---------------------------|
| <esc>[A</esc>   | ^E                           | Cursor Up                 |
| < esc > B       | ^X                           | Cursor Down               |
| < esc > [C]     | ^D                           | Cursor Left               |
| < esc > [D]     | ^S                           | Cursor Right              |
| <esc>[H</esc>   | ^A                           | Word Left                 |
| <esc>[K</esc>   | ^F                           | Word Right                |
| ^O `            | ^S                           | Cursor Right              |
| $< \csc > OP o$ | r < esc > P                  | 0                         |
|                 | F1                           | Function Key 1            |
| < esc > OQ      | or < esc > Q                 |                           |
|                 | F2                           | Function Key 2            |
| < esc > OR c    | or < esc > O w o             | r < esc > ? w             |
|                 | F3                           | Function Key 3            |
| < esc > OSo     | r < esc > O x or             | $\langle esc \rangle ? x$ |
|                 | F4                           | Function Key 4            |
| < esc > Ot or   | r < esc > ?t                 |                           |
|                 | F5                           | Function Key 5            |
| < esc > Ou o    | r < esc ? u                  |                           |
|                 | F6                           | Function Key 6            |
| < esc > Oq o    | r < esc > ?q                 |                           |
| _               | F7                           | Function Key 7            |
| < esc > Or or   | r < esc > ? R                |                           |
|                 | F8                           | Function Key 8            |
| < esc > O p o   | r < esc > ? p                |                           |
|                 | F9                           | Function Key 9            |
| < esc > O M     | or $\langle esc \rangle ? M$ |                           |
|                 | F10                          | Function Key 10           |
| NT . A O 1      |                              |                           |

Note: ^O key mapping is because TBBS always intercepts ^S as flow control. Thus ^O must be used to input the ^S TDBS function key from the keyboard as it cannot be entered directly.

Internally these key sequences appear identical to the standard single key functions and the program isn't aware of them. They allow programming terminal emulators to use their arrow and function keys as marked.

#### Memory Usage

TDBS is constrained to a maximum of 48k per user of data memory for all variables, buffers, tables etc. This is a limit which is placed on all TBBS option modules. Because of this, the following differences between TDBS and dBASE III Plus arise with regard to memory usage:

Memvar Memory The TDBS MEMVAR area is 6k. This is the same size as the default dBASE MEMVAR area, however it cannot be enlarged in TDBS. This memory only contains the data for all current memory variables, not the text of their names. None of this memory is used by fields. dBASE III Plus is limited to a maximum of 256 active variables, TDBS does not have this limit. As many variables as will fit in the 6k of Memvar memory may be used.

Field and Record Memory TDBS has a total of 12k (called the work pool) to hold all field data and buffer records. TDBS will attempt to swap record buffers and reuse this memory as long as possible, but if you open all ten work areas to very large files, and access many fields from these records in a single operation you can run out of memory in TDBS where you would not in dBASE. While it is rare that you will actually run out of memory, it is a performance consideration, since you can write programs which will cause TDBS to "thrash" in its use of file buffer memory if they are poorly structured. TDBS has the same 4k per record and 128 fields per record limit as dBASE III does.

Global Glossary Memory TDBS keeps all symbols in a table called the Global Glossary. The amount of memory this table requires is larger if you use longer variable names (up to 10 characters). It also is larger if you use many different variable names rather than declare variables private and reuse them. The Global Glossary has two portions, the static and the run-time glossary. The **static glossary** is produces by the compiler and contains all symbol names used in the program. The **run-time glossary** is allocated by the /GLOSS switch on the compiler and reserves memory for symbol names added by the USE and RESTORE commands (or macros which create new symbol names). The run-time glossary memory may also be used for dynamic index buffering if the /DIBUF compiler switch was used. Dynamic index buffering can return dramatic performance improvements if the program can tolerate a large run-time glossary allocation.

| Get     | Each GET command requires an entry in the Get Pool. In dBASE        |
|---------|---|
| Pool    | there is a separate pool for PICTURE information and GET            |
| Memory  | entries (called the BUCKET and maximum number of GETS               |
|         | respectively). By default in dBASE you may have 2k of BUCKET        |
|         | space and 128 GET commands pending. In TDBS, all of this            |
|         | information is put into a single pool, which is set by the /GETPOOL |
|         | compiler switch. There is no maximum limit on the number of         |
|         | GETS pending in TDBS, the entire limit is the size of the Get Pool  |
|         | area.   |
| Program | TDBS programs are executed in a buffered manner from disk.          |

Memory Thus there is absolutely no limit on the total size of a TDBS program beyond the amount of disk space available. TDBS caches the program code from disk so that performance remains good even with very large program files.

# **Understanding Work Pool Allocation**

Because it must operate in a very small amount of memory, TDBS has a dynamic memory area that it "slices up" as a program runs to provide many different buffer and memory segments. If you understand how each portion of this memory interacts, you have a lot of control within your program and can often make programs fit that won't otherwise. The total size of this dynamic memory area is about 14k and it includes the following memory structures:

- WAD Work Area Descriptor: Each open work area has a Work Area Descriptor which is a minimum of 80 bytes in size. Each open index file adds to this size and a maximum of nearly 1k is possible if several complex indexes are used. The WAD must be resident in the work pool as long as a work area is open, it may not be swapped out.
- FIRPOOL Field Intermediate Results Pool: This area contains the values of any field in any open .DBF file which has been accessed by the program in internal form (i.e. converted from ASCII as it is stored in the file to 9 bytes for numeric, etc.). TDBS can purge this pool between instructions to swap use of this space with other functions.
- WARBPOOL Work Area Buffer Pool: This memory contains the buffers that hold the current DBF records for the active work area(s). If space is available the current record is cached for each active work area. The WARBPOOL must always be large enough to hold to largest

DBF record from all work areas. This area can be swapped with some other uses of the work pool memory.

FIOBPOOL Flat file I/O Buffer Pool: This memory area is defined by the program when the FOPEN or FCREATE commands are used. This memory is removed from any other work pool use until the FCLOSE command is issued.

ONDSV ON DISCONNECT Save Area: This area contains a "shadow stack" whenever an ON DISCONNECT command is active. It is a fixed size of 400 bytes and is removed from any other work pool use until the ON DISCONNECT is cancelled.

- INSTPOOL Instruction Pool: This memory contains any "canned" instructions (in compiled form) that are used when SET RELATION or SET FILTER commands are active. This memory is removed from other work pool use until each SET command is cancelled.
- FKPOOL Function Key Pool: This memory contains any text currently assigned to function keys via the SET FUNCTION command. The memory is removed from other work pool use until the SET FUNC-TION command is cancelled.

You can control the size of the total memory available for the work pool to some degree. The work pool is the memory "left over" after the Global Glossary (both static and run-time) and the Get Pool have been allocated. You can use the compiler switches /GLOSS and /GETPOOL to control the size of those memory areas. If you minimize the Get Pool and the Global Glossary memory, you will return more memory to the work pool.

Obtaining optimum performance from a TDBS program is a process of trial and error. If you understand the memory allocation methods used, it can help you to "tune" your usage more rapidly. In addition, understanding how TDBS uses memory can help you know what action to take when your program gets a "Not enough room for ..." error.

# **TDBS Maximum Limits**

There are several other areas where maximum limits exist. These limits vary greatly in other xBASE dialects, and are often hard to determine. The following are their values in TDBS:

| Maximum number of nested functions  |
|---|
| Maximum function calls in one instruction line100                             |
| Maximum expressions in a single instruction line                              |
| Maximum DO procedure nesting depth 100 levels                                 |
| Maximum depth of IF nesting   |
| Maximum depth of DO WHILE nesting   |
| Maximum depth of DO CASE nesting  |
| Maximum number of CASES per DO CASE Unlimited                                 |
| Maximum total number of ELSE, ENDIF, CASE,<br>ENDCASE, and ENDDOs per program |
| Maximum parameters passed by a DO WITH100                                     |
| Maximum number of procedures per program                                      |
| Maximum records per file1 billion   |
| Maximum bytes per file2 billion   |
| Maximum fields per record128  |
| Maximum bytes per record  |
| Maximum length of Field or Memvar names                                       |
| Maximum length of procedure names   |

# **Command Differences in TDBS**

Command differences in TDBS are of two types. First, there are some dBASE III Plus commands which are not supported by TDBS. Secondly, there are some extended commands and command options which TDBS adds to the xBASE language.

#### **Commands not supported**

The following standard dBASE III Plus language commands are not supported in TDBS version 1.2.

ASSIST and HELP Interactive help facility

SUSPEND and RESUME Interpreter control commands

APPEND, CREATE, INSERT, BROWSE, EDIT, CHANGE, MODIFY, DISPLAY, and LIST full screen commands

EXPORT and IMPORT Data transfer to PFS files

RUN or !CALL andLOADDOS Shell and Assembly language programs

LABEL and REPORT Pre-formatted print commands

REINDEX and PACK file maintenance commands

SORT TO Sort a file

JOIN Merge two databases into a third

TOTAL TO Create DBF file with totals of specified fields

UPDATE ON Change one database based on another

#### Unimplemented commands (cont.)

SET commands which apply to the interactive environment which doesn't exist in TDBS:

SET CARRY CATALOG DEBUG HELP DOHISTORY HISTORY STEP ECHO TITLE MENU MESSAGE ODOMETER

Set commands which apply to screen features which are only available in TDBS is one form. These commands may be entered with the specified setting only for compatibility. If they are not present in a program, then the given default settings apply. Attempts to set them differently will give an error message:

SET TALK OFF SET SAFETY OFF SET STATUS OFF SET HEADING OFF SET SCOREBOARD OFF

Set commands which apply to features not implemented in TDBS version 1.2.

SET ENCRYPTION SET VIEW

#### **Command Extensions**

The following commands have extensions in TDBS which add to the dBASE III Plus functionality:

- USE Has *MAILBOX* option added. This option is used to give an efficient method of communicating in real-time between programs being run by different users. Also has *READONLY* added to allow access to restricted files.
- HALT This extended command allows exiting a program with a program generated error message and pauses before returning to the calling menu.
- SET FORMAT TO The NOCLEAR option has been added to this command to allow you to make a .FMT file operate the same as in Clipper. By default TDBS treats .FMT operation compatibly with dBASE and clears the screen before each embedded READ command. Additionally, TDBS has enhanced .FMT operation to allow any commands to be used instead of only @ and READ commands as in dBASE.
  - SET DISPLAY RULES Allows adjusting certain screen display boundary conditions to operate as they do in various dBASE dialects where they behave differently. Default is dBASE III + with bugs removed.
- SET DIVIDE BY ZERO TO Allows the program to choose the response to a divide by zero condition. Default is to return the largest possible number as dBASE III + does. Optionally an error may be given.
- SET UPDATE BELL When using the extended TDBS "Transparent file locking" feature, it is possible for multiple users to be doing safe real-time screen updates of the same record at the same time. TDBS will immediately propagate any fields which are changed by another user to your display. This command allows you to select the alert bell function you want in this case independently from the normal SET BELL command. You may select no alert, alert only if a field you have edited but not committed was rolled back, or alert if any field on the screen was changed, even if you haven't edited it.

# SET SOFTSEEK Allows "relative" seeking. If a specified record is not found, the record pointer is positioned at the record with the next highest key.

| DECLARE,<br>PRIVATE,<br>PUBLIC | These commands have been extended to allow the definition of one dimensional arrays. All other commands have been extended to handle arrays as describe in "arrays" in this chapter                 |  |
|--------------------------------|---|--|
| DOTBBS                         | This command has been added to allow a TDBS program to "shell"<br>to a TBBS internal command. Note: TDBS requires TBBS 2.2 or<br>newer to support this command.                                     |  |
| SET EDITOR TO                  | This command allows tailoring the operation of the TDBS memo<br>editor. It allows read only access, or the ability to allow or disallow<br>access to text files outside of the TDBS program itself. |  |
| @SAY/GET                       | Keywords have been added to thee commands to allow fine control<br>over the memo editor as well as passive monitoring of shared record<br>fields.   |  |
| READ                           | The NOEDIT keyword has been added to the READ command to make any memo editor accesses read only for this READ.   |  |
| ON DISCONNECT                  | Allows a "cleanup" procedure to be executed if an accidental disconnect occurs while a TDBS program is running.   |  |
| SET DISCONNECT                 | Controls limits of TDBS during an ON DISCONNECT procedure.  |  |
| ON NEWMAIL                     | Allows reception of data via a MAILBOX to "interrupt" an execut-<br>ing TDBS program.   |  |
| FOPEN<br>FCREATE<br>FCLOSE     | Allow direct access to non-database files.  |  |
| FLREAD<br>FLWRITE<br>FLFIND    | Allow line-by-line access to ASCII text files. Also allows rapid file searches for key words or text fragments.   |  |
| FBREAD<br>FBWRITE              | Allow direct access to binary files.  |  |

# **Extended Functions**

TDBS has added several extended functions as follows:

| ALIAS(n)     | Return a character string with the Alias name for work area "n".                   |
|--------------|--|
| ACOPY()      | Copy array elements.   |
| ADEL()       | Delete array elements  |
| AFIELDS()    | Move database definition info into arrays  |
| AFILL()      | Fill an array with a value   |
| AINS()       | Insert an element into an array  |
| ASCAN()      | Find a matching array element  |
| ASORT()      | Do an ascending sort of array elements   |
| ADSORT()     | Do a descending sort of array elements   |
| FCOUNT()     | Count the number of fields in a work area.   |
| SELECT()     | Return number of currently selected work area.                                     |
| DTOS(d)      | Return string with date "d" in yyyymmdd format.                                    |
| EMPTY(exp)   | Return logical .T. if "exp" expression is empty.                                   |
| PROCLINE()   | Return number of current procedure line.   |
| PROCNAME()   | Return string with current procedure name.   |
| SECONDS()    | Return time as number of decimal seconds and hundredths of seconds since midnight. |
| FLOOR(n)     | Return integer number equal to or next lower than n.                               |
| CEILING(n)   | Return integer number equal to or next higher than n                               |
| ISLASTDAY(d) | Return .T. if this is last day in month, else .F.                                  |

|              | TDBS Extended Functions (cont.)   |  |
|--------------|---|--|
| ISLEAP(d)    | Return .T. if date is in a leap year, else .F.  |  |
| LASTDAY(d)   | Return date value = last day of month of "d".   |  |
| HEX2DEC(c)   | Convert string "c" (treated as hex digits) to a number.   |  |
| DEC2HEX(n)   | Convert number "n" to hex digits, return as string.   |  |
| CAPFIRST(c)  | Convert only first character to capital letter.   |  |
| LJUST(c)     | Left Justify string "c".  |  |
| RJUST(c)     | Right Justify string "c".   |  |
| RAT()        | Find rightmost occurrence of a string within another string.  |  |
| ISINT(n)     | Return .T. if "n" is integer, else .F.  |  |
| ISSTATE(c)   | If "c" is two characters in length and a valid post office US State abbreviation return .T.   |  |
| STATENAME(c) | Convert two character state abbreviation "c" to a string with the full state name. Return a null string if "c" is not a state abbreviation. |  |
| LASTKEY()    | Returns the "n" value of the last key processed as input by TDBS.   |  |
| NEXTKEY()    | Returns the "n" value of the next key typed ahead, but does not remove it from the buffer. If no key typed ahead, returns 0.                |  |
| INDEXKEY()   | Find the key expression for a given index file.   |  |
| INDEXORD()   | Find the current index file list order.   |  |
| HOMEPATH()   | Return string with the home path (from Opt Data on menu entry).   |  |
| NMYUSERS()   | Return "n" indicating the number of current users running this TDBS program.  |  |
| NUSERS()     | Return "n" indicating the number of current users running any TDBS program.   |  |
| USING()      | Determine which users are sharing a database or mailbox.  |  |

|               | · · · · · · · · · · · · · · · · · · ·   |
|---------------|---|
|               | TDBS Extended Functions (cont.)   |
| ISSHARE()     | Determine if any user is sharing a file or mailbox.   |
| UNAME()       | Return string with user name or id.   |
| ULOCATION()   | Return string with user's location field.   |
| UNOTES()      | Return string with userlog NOTES field.   |
| UPRIV()       | Return numeric value of user PRIV 0-255.  |
| UAUTH(n)      | Return string of form ".XX." corresponding to the users A(n) flags. If INT(n) is not 1, 2, 3, or 4 then return a null string. |
| UANSI()       | Return .T. if user has ANSI set on, else .F.  |
| UIBM()        | Return .T. if user has IBM Graphics set on, else .F.  |
| UMORE()       | Return "n" with the value of the user's -more- setting.   |
| UWIDTH()      | Return "n" with the number of characters per line the caller is configured for.   |
| ULREPLACE()   | Update selected userlog record fields.  |
| ULPEEK()      | Read any userlog record field.  |
| ULPOKE()      | Update any userlog record field.  |
| ULINE()       | Return single character string with line indicator (0 - W).   |
| NEWMAIL()     | Return .T. if a mailbox has new mail waiting.   |
|               | TDPS Extended Eurotions (cont.)   |
| GETLPT(n)     | <b>TDBS Extended Functions (cont.)</b><br>Return .T. if LPTn can be assigned to this program for use.                         |
| WAIT4RLOCK(n) | Allows a programmed wait for record locking.  |
| WAIT4FLOCK(n) | Allows a programmed wait for file locking.  |
| WAIT4LPT(n)   | Allows a programmed wait for a printer resource.  |
|               |   |

#### **Chapter 2: Overview**

| keyboard input.FINDFIRST(c)Allows searching for files which match a DOS wildcard specifica<br>tion when the names are not known in advance.FDATE(c)Allow determination of file attributes.FTIME(c)FSIZE(c)SOUNDEX(c)Produces a "sounds like" code for a text keyword.FERROR(h)These functions allow manipulation of data and detection of error<br>conditions associated with direct file I/O commands and buffers.FLEN(h)FBEXTRACT(h) |                                      |   |
|--|--------------------------------------|---|
| keyboard input.FINDFIRST(c)Allows searching for files which match a DOS wildcard specifica<br>tion when the names are not known in advance.FDATE(c)Allow determination of file attributes.FTIME(c)FSIZE(c)SOUNDEX(c)Produces a "sounds like" code for a text keyword.FERROR(h)These functions allow manipulation of data and detection of error<br>conditions associated with direct file I/O commands and buffers.FLEN(h)FBEXTRACT(h) | WAIT4MAIL(n)                         | Allows a programmed wait for a received mail  |
| FINDNEXT(c)tion when the names are not known in advance.FDATE(c)Allow determination of file attributes.FTIME(c)FSIZE(c)SOUNDEX(c)Produces a "sounds like" code for a text keyword.FERROR(h)These functions allow manipulation of data and detection of error<br>conditions associated with direct file I/O commands and buffers.FLEN(h)FBEXTRACT(h)FBINSERT(h)FILEN(h)   | INKEY(n)                             | This function has been extended to allow a programmed wait for keyboard input.  |
| FTIME(c)         FSIZE(c)         SOUNDEX(c)       Produces a "sounds like" code for a text keyword.         FERROR(h)       These functions allow manipulation of data and detection of error conditions associated with direct file I/O commands and buffers.         FLEN(h)       FBEXTRACT(h)   | • •                                  | Allows searching for files which match a DOS wildcard specifica-<br>tion when the names are not known in advance.                     |
| FERROR(h)These functions allow manipulation of data and detection of error<br>conditions associated with direct file I/O commands and buffers.FLEN(h)FBEXTRACT(h)FBINSERT(h)FBINSERT(h)  | FTIME(c)                             | Allow determination of file attributes.   |
| FMAXLEN() conditions associated with direct file I/O commands and buffers.<br>FLEN(h)<br>FBEXTRACT(h)<br>FBINSERT(h)   | SOUNDEX(c)                           | Produces a "sounds like" code for a text keyword.   |
|  | FMAXLEN()<br>FLEN(h)<br>FBEXTRACT(h) | These functions allow manipulation of data and detection of error<br>conditions associated with direct file I/O commands and buffers. |

# **Extended File Sharing Support**

In addition, TDBS offers several transparent multiuser file sharing features which are not normally available. These allow a program to share files much more easily than is usual with dBASE language dialects.

TDBS will even automatically update all user's screens when a shared database file field which is part of a READ is updated by another user! This means that in many applications records never need to be restricted from use by all users for even a small amount of time.

See Chapter 3 for a full discussion of both the dBASE standard, and TDBS extended multiuser features.

# **DOS File Limits and FCB Sharing**

Since all users of TDBS share the same TBBS and DOS resource pool, the absolute limit on different files open at the same time by all TDBS users is 128 files. This is because TBBS is limited to 150 simultaneous files open, and reserves 22 DOS *FILES* = entries to assure it can meet its internal needs at all times. Note: The TDBS limit is smaller than 128 unique files open if the DOS CONFIG.SYS has a *FILES* = value smaller than *FILES* = 150.

In order to make this limited resource stretch as far as possible, TDBS will share the same FILES = entry (FCB) for all TDBS users which are currently using the same file. Thus if a single file is opened by one TDBS program, or by 64 TDBS programs that file will still require only a single FILES = entry from DOS to support its operation. Note: Each program also takes one FILES = entryto open the .TPG file which remains open for the duration of the program execution. Here again, if more than one user is running the same program, TDBS will share one FILES = entry to service all users of that .TPG file.

TBBS is normally configured so that there are enough FILES = entries for each user to have two of them in use at the same time without running out. In configuring for TDBS, you need to increase the FILES = setting in the DOS CONFIG.SYS file if you expect to run TDBS programs which will result in more than two unique files per user being open simultaneously.

For example, if you are only running a single TDBS program, and that program can open a maximum of 15 files at the same time (the dBASE III + limit) then you would need to add 13 FILES = entries to the CONFIG.SYS setting. Why 13? Because if all users are running the same program, they will be accessing the same data, index, and .TPG files. TDBS will share the FILES = resource for each file when multiple users have it open. Thus for a 32 user system instead of TDBS requiring 480 FILES = entries (as would be required without FCB sharing) only 15 entries are needed. So you can see that FCB sharing makes the difference between an application which fits easily, and one which would not fit at all.

Note: If the *FILES* = limit is exceeded, the TDBS program will give the error message *Too Many Files Open*.

# Macro Compatibility

|                       | Because dBASE III Plus is an interpreter, macros may be used for<br>any character string substitution in any portion of any command<br>line. Since TDBS is a compiled language there are some restric-<br>tions on macro usage which are not present in dBASE. To sum-<br>marize these differences, we must first understand that there are<br>really three different types of macros, based on where they occur<br>in an instruction line. TDBS does not allow a "mixed mode" macro<br>which crosses the boundary between types. The categories are: |  |
|-----------------------|---|--|
| Text String<br>Macros | This category includes any macro which is part of a text string which<br>is enclosed in quotes, but does not include the quotes themselves.   |  |
| Literal<br>Macros     | This category includes any macro which expands to exactly replace<br>one argument of a command line such as a file name or variable<br>name.  |  |
| Expression<br>Macros  | This category includes any macro which contains an expression<br>which must be evaluated at run time as part of the substitution.   |  |
|                       | TDBS 100% supports all macros of any of these types, but does not<br>support a macro of a combined type. This is commonly referred to<br>as "no commas in a macro" but that is not strictly true. Commas are<br>allowed in macros as argument separators in a function in an<br>expression macro for example. Commas are not allowed in a TDBS<br>macro however, if they indicate a separation between two of the<br>above macro types, even the same type.   |  |
|                       | TDBS also 100% supports combined macros, for example where macros are used to build variable names in the following example:  |  |
|                       | &mac.VAR⊂   |  |
|                       | This would result in a single variable or field name and thus is a legal macro construct in TDBS.   |  |
| Empty Macros          | Unlike many other dBASE compilers, TDBS will handle empty macros in a command as in the following example:  |  |
|                       | <pre>var1 = "" SET DELIMITED &amp;var1 &amp;&amp; set/reset delims</pre>  |  |

# **Memory Variable Domains**

As a programmer you need to have a thorough understanding of the way TDBS variables are handled. This process is 100% compatible with the xBASE language standards. Memory variables have three possible domains: *public*, *private*, and *hidden*. These domains are defined at each program level, so let's first define what a program level is.

#### **Program Levels**

Initial TDBS program execution begins with the first instruction of the main program. The main program is the "top level" of the program's execution. Any time a program calls a procedure (or sub-program) by using the DO command, the called procedure is one level "lower" than the calling program. If this procedure calls another procedure before it returns, then that procedure is considered one level lower. TDBS allows a program to "nest" procedure calls in this fashion up to 100 levels deep. When a lower level procedure executes a RETURN command then the level of execution returns to one higher until at last all procedures return and the program is again at the top level in the main program. These levels are important because they affect the way that memory variables are created and accessed as the program is executed. Note: because TDBS is a compiled language, there is no difference between individual programs and procedures in a procedure file as there is in dBASE III +. TDBS is even extended to allow inline procedures in any program file it compiles.

#### **Private Memory Variables**

By default all TDBS variables are private. This means that they are only accessible to the procedure that created them and any procedures which are executed at a lower level. A RETURN command erases all private variables that were created at the current program level. The RELEASE command returns the memory of all private variables at the current program level, but does not erase the variable itself. This is a subtle but important difference which will be explored after we define PUBLIC memory variables.

#### **Public Memory Variables**

Public memory variables are available to procedures at all levels. Public variables must be declared by using the PUBLIC command. The placement of the PUBLIC command in the program is also important. It must come before the variable is used in any way, because if a variable is currently defined, it is by default PRIVATE at the level it was defined. The PUBLIC command cannot override a currently defined variable. Once a variable is declared PUBLIC, the RELEASE command will not erase it. Only the CLEAR MEMORY or CLEAR ALL command will erase the PUBLIC variable definition. Public variables can be hidden, however, and their names temporarily used as a private variable as is described next.

#### **Hidden Variables**

Hidden memory variables are those that have the same names as variables in lower level procedures, but are temporarily set aside or hidden while the lower level procedures are running. To hide a variable which was created by a higher level procedure, a lower level procedure must use the PRIVATE command. When the PRIVATE command is issued on a variable name, the higher level value of the variable is hidden. All references to this variable will now access a new *local* version of this variable until the procedure level at which the PRIVATE command was issued executes a RETURN. When this RETURN occurs, the PRIVATE version of the variable is erased from memory and the original value of the variable comes out of hiding and is again accessible.

A variable can be hidden repeatedly at different levels, and as the nested procedures return past each PRIVATE command for the variable the previous version of the variable value will return. This operation may be considered a stacking operation as variables are hidden at various levels. The purpose of all of this is to allow you to create general purpose procedures which can use local variables and not worry about affecting variables which may have the same names in programs which call them.

# Parameter Passing

By use of the DO WITH and the PARAMETERS commands you can pass variables from a calling procedure to a sub-procedure. Variables in TDBS may be passed in two ways. These two ways are known as *Call by value* and *Call by reference*. This is 100% compatible with the dBASE language standard.

Normally a parameter is passed to the subprogram using the *Call* by value method. In this case, a **private variable** with the receiving PARAMETER name is created and the value of the calling expression from the DO WITH line is copied into the variable. The new variable has the original calling value, but if it is changed by the called procedure the original variable is unaffected. Thus the parameter passing is a one way operation. The value is passed down to the called procedure and is erased when that procedure returns and any original variables in the calling expression are unchanged.

There are two ways to allow a called program to return a parameter value. One is to declare a variable PUBLIC, and put the return value in this variable. In this case, the return variable is not referenced in the DO WITH or PARAMETERS list, so the *Call by value* problem is worked around.

However there is a special case where the PARAMETER and DO WITH commands can be used to do *Call by reference* passing of a specified variable, which will allow a lower level procedure to have direct access to the higher level program's variable. In this case a value may be returned in either a private or public variable.

This special case occurs when the DO WITH and PARAMETERS line both specify a simple variable name (or array name) as a parameter. In this case TDBS does not create a private receiving variable for the parameter, but instead makes all references to the passed variable point to the calling program's variable instead. Thus the value is passed to the calling program by *reference* to the calling variable directly, and any modifications the called program makes go directly into the calling program's variable and are returned to the calling program in that variable which is not erased by the RETURN.

#### Memo field support

TDBS provides 100% compatible dBASE III Plus memo field support. Memo fields allow text of any size to be carried as a logical field in a database. This type of field cannot be used in expressions, but can be displayed or edited under program control. Note: dBASE IV users must convert memo files (.DBT) to dBASE III Plus format for use with TDBS.

A memo field can be displayed using the ? or ?? commands. It will be displayed as word wrapped text, and the width used will be either the user's TBBS terminal profile width, or a width set by the SET MEMOWIDTH command. These two commands can also be routed to the printer or an alternate file to send memo field text there as well.

When memo fields are displayed using @ ... SAY or @ ... GET, the word "memo" will indicate the presence of a memo field. Memo fields can be viewed and edited by using a special form of the command @ ... GET coupled with the READ command. What is special is that this command must be placed in a FORMAT file and invoked through the use of the SET FORMAT TO command.

In this case, if the user presses  $^HOME$  or  $^]$  the memo field will be opened. When a memo field is opened, the memo editor is entered and the memo text may be viewed and optionally edited.

When a blank record is added to a database which has one or more memo fields, those fields begin as empty. The memo editor must be used to add text to each memo field.

## The TDBS Memo Editor

The TDBS memo editor is compatible in operation with the dBASE standard memo editor. It uses many of the Wordstar standard editing commands as well and thus should be easy to learn. It provides full screen editing since it is only available if the user's terminal supports ANSI or VT-100 emulation. Memos up to 16 megabytes in size may be edited with this editor.

In addition, TDBS extends its shared screen update with rollback on collision (see Multiuser considerations in chapter 3) to the memo editor. If more than one user are editing the same memo field, when one commits changed text to the database, the other user will immediately be editing the updated text.

The memo editor also allows importing text files into memo fields and exporting memo fields to text files.

The SET EDITOR command allows you to tailor the memo editor features to control what the user can do when in the editor. you can make the editor read-only (the user can view but not modify the text). You can also restrict or eliminate the ability of the user to import or export text files. This allows you to enhance the overall security of your system. You can also control the initial presence or absence of the editor help menu when the user enters it.

In addition, through the use of keywords on the @...GET command and the READ command you can temporarily override the global editor settings to allow each editor entry to have only the features you wish it to have, and you can also use the SELECT keyword to directly enter the memo editor if you wish.

In general, memo fields should be restricted to 5k or less in order to be compatible with dBASE III Plus itself. However, TDBS has no limit below 32 megabytes on the size of a memo field. Entry to the editor may take a few seconds if the memo field is very large.

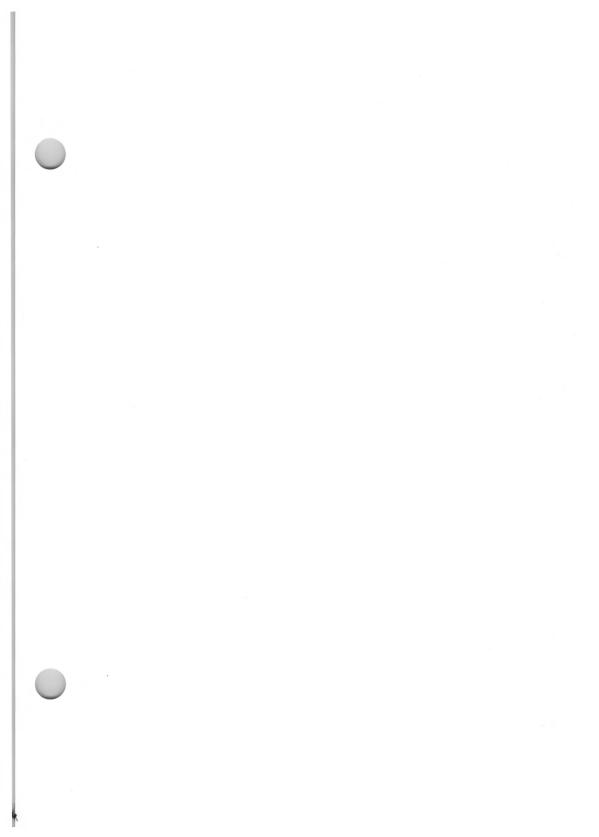
**Example:** To allow a user to enter a memo field, the @...GET command must exist in a FORMAT file. Assume the file MEMO.PRG contains the following lines:

@ 1,1 SAY "Press ^] to edit the memo field" @ 10,1 GET memo field

Then the main program would contain:

SET FORMAT TO MEMO READ SET FORMAT TO

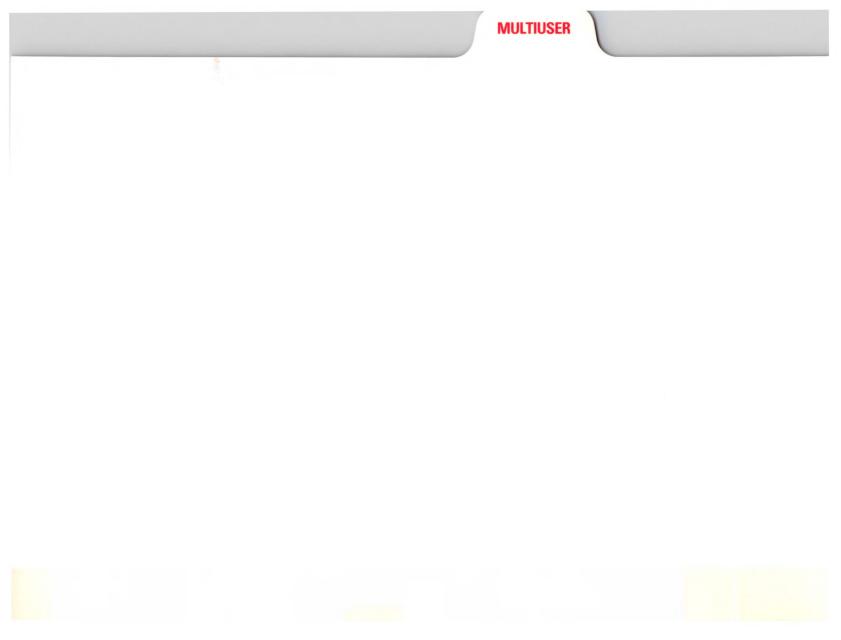
The trailing SET FORMAT TO clears out the FORMAT file mode of the READ command returning READ to its normal operation.







# MULTIUSER



# Introduction

Programming in a multiuser environment usually requires consideration of the complex interactions which can occur when multiple users share the same database. The standard dBASE language provides the essential primitive functions to handle most of these considerations, but using them properly can be quite complex and take some experience to understand.

While TDBS offers a 100% compatible implementation of these standard functions, it also provides expanded and **much easier to** use multiuser capability. In fact, in most cases you do not need to do any special coding to have TDBS properly arbitrate all multiuser conflicts. Once you override the default setting which prohibits sharing files, TDBS does the rest. To transparently share database files simply place the following command in your main program:

#### SET EXCLUSIVE OFF

This command allows multiple programs to USE the same file at the same time. TDBS will automatically and transparently arbitrate and lock all access collisions.

The following pages will discuss the theory of multiuser arbitration in general, and all of the multiuser features TDBS provides. You should read and understand it, but you will find that multiuser file access with TDBS is much easier to program and far more powerful than it is in any other dBASE language dialect. Some features, such as *Screen Update Posting and Rollback on Collision* which are automatic in TDBS are simply not possible in other xBASE dialects, or require immense programming efforts to provide.

The other multiuser problem which arises is when several copies of the same program are running, and **require unique file names**. This occurs, for example, if a *SAVE* and *RESTORE* are used to checkpoint memory variables and you want each program copy to have its own file so that one copy of the program doesn't overwrite the other's files. To do this, use the ULINE() function to append the line number to the file name as follows:

SAVE TO "SAVE" + ULINE()

This method assures a unique file name for each copy of the program so no files are overwritten.

Now we'll list the multiuser features which TDBS version 1.2 provides. After that we'll take a look at the problems which need to be resolved in a multiuser environment and explain each TDBS feature in detail. TDBS provides the following multiuser capabilities.

- Transparent File Sharing TDBS automatically arbitrates all multiuser accesses with no special programming considerations. We know of no other dBASE dialect which can provide complete multiuser transparency as TDBS does.
- Transparent Screen Update and Rollback on Collision A long name which means that TDBS will automatically update the screens of other users who are displaying a record which was just updated. We know of no other dBASE dialect which offers this capability transparently. Most cannot provide it in any form.
- Explicit File and Record Locking Of course TDBS offers the dBASE language standard capabilities in a 100% compatible form. It also offers some extensions to these features which greatly ease the coding task when using them, and also reduces system overhead greatly.
- Automatic Record Locking TDBS also allows a hybrid access to a file where explicit record locking and transparent record locking occur at the same time. In this case TDBS provides automatic record locking where the two methods collide.
- Intraprogram Mailboxes TDBS offers an efficient method of passing messages from one program to another to allow communication between programs or users online. Functions to detect new mail easily are provided. Mail appears as a collection of fields which are fully integrated into the TDBS language for ease of access and manipulation, and which may contain any combination of the standard data types.
- Multiple Printer Assignment and Arbitration TDBS provides the ability to have up to four printers connected. It allows a program to bid for use of the printers and arbitrates access to them between multiple programs.

# Multiuser Overview

|                          | One of the strongest portions of TDBS is the way in which it handles<br>multiuser access. This chapter will explore the extra concerns<br>which arise when databases are accessed by more than one user at<br>the same time. It will also discuss the concerns which must be<br>addressed when you run more than one copy of the same program<br>at the same time. The enhancements added to TDBS to allow<br>smooth handling of these concerns will also be explored.   |
|--------------------------|--|
|                          | Multiuser TDBS programs are faced with a potential problem that<br>does not occur in a single user environment because more than one<br>user may wish to use the same file at the same time. This potential<br>problem, called collision, will occur if more than one user attempts<br>to edit or add data to a database at the same time. If this collision<br>is not arbitrated properly, data integrity can easily be compromised.  |
|                          | There are two common ways in which this arbitration may take<br>place, and which one is used depends on what the program is trying<br>to accomplish. These methods are:  |
| Exclusive<br>File<br>Use | This method prevents more than one user from opening the file at<br>the same time. When the second user attempts to open the file<br>while the first user still has it opened, an error is returned indicating<br>the file is busy. This method is simple to use and understand. Once<br>the file is open, the program has no other considerations since it<br>cannot be shared and is in essence in a single user environment<br>again. The drawback to this method is that all other users who wish<br>access to this file must wait for it until the first user is finished.  |
| Record<br>Locking        | This method allows a program to prevent other programs from<br>updating or accessing only a single record at a time. It is much less<br>likely that another program will want to actually share the same<br>record than the same file. Thus the likelihood that a program must<br>wait for access to a record in a shared file is reduced. However, a<br>wait will still occur if the record that a program wishes to change<br>or access is locked by another program. This method of preventing<br>collisions at the record level allows much better performance in a<br>shared file environment because waits for record access are much<br>less likely to happen than waits for file access. |
|                          | The standard dBASE language provides facilities for only these two<br>methods of collision arbitration. In addition, it requires that the  |

program explicitly code all collision arbitration and wait loops which can be a complex process. In some cases this level of detail is required because of the user interaction required by the application. However, in most cases automatic arbitration methods will suffice, and they are always much easier to use. TDBS features sophisticated automatic collision arbitration options which are not available in other dBASE language dialects. These are all variations on record locking except for one, which uses a third arbitration method.

This method assumes that any user can update the same record at the same time as long as data file integrity is maintained. As a result, one user may have a screen image of a record partially updated (some fields changed but not committed to the file yet as editing of the record is in progress) when another user writes an updated version of the record being edited to the database file. In this case, when a collision actually occurs, update rollback will undo the edited fields which haven't been written (since there may now be new data in them) and redisplay any fields which have values different from those now on the screen. The user can then reapply any desired updates after seeing if the changes make the updates still desirable. This method assures that no waits or retries will occur unless an actual collision takes place. It then allows the user to decide if the collision changes what he was about to do. It is the most efficient method possible when collisions are rare, as no waits will ever occur for collisions which are only potential and don't occur. In return it may require a user to occasionally re-enter an update if a collision actually does take place.

# TDBS Multiuser Features

Because TDBS operates on the TBBS multiuser platform, it can provide capabilities efficiently which either cannot be provided in other environments, or which introduce a large amount of system overhead if they can be provided. TDBS provides features which make multiuser access very efficient and smooth. These are:

Transparent Allows multiple programs to share the same file without having to File worry about multiple simultaneous updates. TDBS transparently Sharing arbitrates all shared file updates and ensures that file and record integrity are always maintained.

Screen

Allows multiple TDBS users to effectively do screen editing of the Transparent same record in the same file at the same time. It posts any changes Screen Update Posting to fields made by another user immediately to all other user's and Rollback screens. It can optionally alert a user if another user has committed an update to a field that this user has edited but not yet committed. As the name implies, this feature operates transparently, that is without specific program intervention or coding required. TDBS also 100% implements the standard dBASE language ex-**Explicit** File and Record plicit file and record locking functions and commands. These commands allow files to be opened for exclusive or shared access, Locking and for records or files to be explicitly locked from updating by any other user for extended periods under program control. **Automatic** Finally TDBS allows a hybrid of transparent record locking and explicit record or file locking. In this case one (or more) program Record Locking sharing a database file is using transparent file sharing while one (or more) other program sharing the same file is using explicit record or file locking. When the user who is using transparent file sharing attempts to update a record which is explicitly locked, TDBS will shift into automatic record locking mode to arbitrate the conflict. A program must write an ON ERROR handler to handle automatic lock retries and prevent this from causing a program abort, but automatic record locking requires no other special programming. This allows different programs to share the same file at the same time with different sharing methods and still perform all database updates correctly. Intraprogram This TDBS feature allows a program to efficiently exchange mes-Mailboxes sages with either itself or another TDBS program via a mailbox. Because mailboxes appear to the TDBS program to be single record database files, up to 4,000 characters of information may be passed in a mailbox. All of the TDBS commands operate directly on mailbox data just as they do on fields. Mailboxes are an extremely efficient method of communicating between programs in a multiuser environment. Let's now look at each of these features in detail and see how they

operate.

# **Exclusive or Shared files**

The first level of arbitration is file locking. A file is opened in exclusive mode when the file is intended for a user's private and exclusive use.

If a file is opened for Exclusive use:

- Only one user can access the file at a time.
- There is no need to lock or arbitrate any record accesses in the file.

If a file is opened in **Shared** mode, then any number of users are allowed to open and use it at the same time. If a file is shared:

- The file can be accessed by multiple users.
- The file and/or its records may require locking or other arbitration for updating.

By default all files will be opened in exclusive mode in TDBS. To allow file by file control of the sharing mode the command:

#### SET EXCLUSIVE OFF

should be issued. After this command, a normal USE command will open a file in shared mode. You can still indicate that an individual file is to be opened for exclusive use by adding the **EXCLUSIVE** modifier to the USE command.

Note: Files already open are not affected by the SET EXCLUSIVE command. This command only indicates the default access for subsequent USE commands.

If a file is opened for exclusive access, and another user currently has it open, an error will result. This error can be fielded by using the **ON ERROR** option to write a retry subroutine.

# **Explicit Record and File Locking**

When a file is opened for shared access, the standard dBASE language provides functions and commands for explicit file and record locking. In order to facilitate retry loops and error checking, the lock requests are implemented as functions, and the unlock action is a command. These functions and commands are as follows:

# FLOCK()

This function attempts to lock all records in the file open in the current work area. If it is successful, it will return .T. and all records in the file will be locked. If one or more records in the file are already locked by another user, the lock cannot be done and the function will return .F. to indicate failure.

# RLOCK()

This function attempts to lock the current record in the file in the current work area. If it is successful, it will return **.T** and the record will be locked. If this record is already locked by another user, the lock cannot be done and the function will return **.F** to indicate failure.

# UNLOCK

This command will remove any locks you now have on the current file. You may add the ALL option to remove any locks you have placed on files in any work area.

Notes: Only one RLOCK() or FLOCK() may be pending for a file. If you issue another one for this file, the first lock is removed, even if the new lock attempt fails. Closing the database file will remove any locks you have placed on it, as will a program abort or termination. Loss of carrier will also remove any locks pending.

In addition to these standard dBASE language locking functions, TDBS offers two extended functions which make coding locking loops easier. These two functions are:

## WAIT4RLOCK([n])

This function contains all of the delay and retry logic required to implement a record lock request which does the following:

- Return .T. if the record lock can be done
- Retry a failed record lock attempt every 500ms until it succeeds (in which case .T. is returned) or until one of the following occurs: 1)The user presses any key during the wait, 2) The optional timeout value of "n" seconds elapses without a successful record lock occurring, or 3) The function is attempted when there is no open file in the current work area. If any of these occur the function will return .F. to indicate failure.

This function thus allows in a single command a record locking wait which would require many instructions and nested loops without it. It also causes much less overhead to the TBBS system than a tight loop issuing RLOCK() repeatedly. Here is an example record locking handler using this function which will attempt to lock the selected record repeatedly for up to 1 minute, or until the user presses a key to abort the wait time.

```
******
  WAIT4RLOCK example
*******
DO WHILE .T.
  ACCEPT "Enter name to change" TO Mkey
  IF UPPER (Mkey) = "END"
               && Exit requested
     EXIT
  ENDIF
  SEEK MKEY
  IF .NOT. EOF()
     IF .NOT. WAIT4RLOCK(60)
        WAIT "Record Locked, can't access"
        LOOP
     ELSE
        @ 5,5 SAY "Enter new name" GET name
        READ
        UNLOCK
     ENDIF
  ENDIF
ENDDO
```

# WAIT4FLOCK([n])

This function performs identically to the WAIT4RLOCK function described on the previous page, but it will loop attempting to do the FLOCK() function instead of the RLOCK() function.

# **Fielding Locking Conflicts**

If a file locking conflict occurs on a USE command, an error is generated. To field such errors you must write an error handling procedure. The commands which aid in this procedure are:

#### ON ERROR RETRY

The ON ERROR command specifies a procedure to be entered on any error encountered. Locking errors will activate the ON ERROR routine. If the routine wishes to, it may loop for some period of time to retry the action which caused the locking error by issuing the command RETRY.

Two functions which may be used in an error routine are:

#### ERROR() MESSAGE()

The ERROR() function returns the error number which caused entry to the ON ERROR routine. This error number may be used to determine the proper action to take, and to discriminate locking errors from other error conditions. If the error handler determines the error number should result in operator intervention, it may use the MESSAGE() function to obtain the text of the error message for the error which occurred and display it.

#### Sample File Locking Handler

The following code illustrates a sample ON ERROR routine which will retry the open of a locked file up to 10 times. It also discriminates against other errors and displays any associated error message before ending the program.

```
*******
 Error Handler
*******
PROCEDURE ONERROR
DO CASE
  CASE ERROR() = 108 && Locking error?
     A=INKEY(2)
                      && 2 second delay
     IF Tries < 10
                       && Tries is PUBLIC
        Tries = Tries+1 && count a retry
        RETRY
                       && RETRY action
     ELSE
        ? "File is locked - can't proceed"
        WAIT
        QUIT
     ENDIF
  CASE ERROR() = \dots
           (other explicit errors here)
  OTHERWISE
     ? MESSAGE()
     ? "Program Aborting"
     WAIT
     QUIT
ENDCASE
```

#### Sample Record Locking Handler

The following is a sample program which illustrates handling record locking conflicts. Notice that this program uses only standard dBASE language commands, and takes many more instructions and is not as complete as the sample handler which used the WAIT4RLOCK() function shown earlier.

```
*******
*
   Record Locking Example
******
DO WHILE .T.
  ACCEPT "Enter name to change" to Mkey
  IF UPPER(Mkey) = "END"
     EXIT
                      && Exit request
  ENDIF
  SEEK Mkey
  IF .NOT. EOF()
     Try = 1
     CLEAR TYPEAHEAD
                      && clear kbd buffer
     DO WHILE .NOT. LOCK() .AND. Try < 250
        IF INKEY() <> 0
                      && bail if key press
           EXIT
        ENDIF
        Try = Try+1
     ENDDO
     IF .NOT. LOCK()
        ? "Record locked, press key = retry"
        WAIT ""
        LOOP
     ENDIF
     @5,5 SAY "Enter new name" GET name
     READ
     UNLOCK
  ENDIF
ENDDO
```

Notice that this program attempts to lock the record 250 times and then gives up. It is generally a good idea to have some method of exiting a lock wait loop. In this example an INKEY check allows the user to break out. A tight loop attempting to lock a record or file which has no termination method is not a good idea.

# **Transparent File Sharing**

As you can see, explicit file and record locking can require a great deal of programming effort. It is rarely necessary to have this level of control, and TDBS provides transparent file sharing which requires no extra programming effort from single user programming. Whenever more than one TDBS program is sharing a file, transparent file sharing is in effect.

Until a record is written, transparent file sharing does nothing but allow all users sharing the file to read the records they request. However whenever any user writes an updated record to the shared file, transparent file sharing performs the following:

- Writes the record to the shared disk file
- Instantly updates the changed fields for any user who is currently accessing the same record. This updating occurs before any other user can run again, so that total data integrity is preserved for all users.

This means that as long as multiple updates do not need to be arbitrated, shared file updates simply operate as you would expect them to. It is another case of TBBS/TDBS doing a lot of internal arbitration to make it look like sharing files is not a big deal!

Note: It is a common practice in dBASE language programming when updating record fields from the screen to move the fields to memory variables, present an update screen which updates the memory variables, and then re-write the variables to the file. If this method is used, then transparent file sharing can allow multiple users to update the same record without properly arbitrating the updates because TDBS doesn't know the updates being made to memory variables are really updates to record fields. If you use this technique, you must use explicit record locking.

However, TDBS provides a feature to handle screen updates to a shared file very well. This feature is Screen Update and Rollback on Collision, and is also totally automatic and transparent.

#### Screen Update and Rollback on Collision

This feature of TDBS is automatically invoked when the READ command directly accesses the fields in a record of a shared file. This technique is generally considered dangerous by dBASE programmers, because in most dBASE dialects there is a high possibility of file damage when the @ ... GET command directly references a database file field. However no such risk exists in TDBS, and this feature is available if you use that technique.

If two or more users have fields from the same record in a shared file displayed as part of a READ command, nothing looks different normally. As they edit fields, these changes are kept in each user's local memory area until the read is either aborted (via an <ESC>key press) or is committed to disk by another exit function key. Thus the database file integrity is preserved, as no partial or uncommitted update is ever present in the database file itself.

When one of the users commits a change to the file, any fields which were changed are immediately updated on the screens of all other users displaying those fields from the same record. This is true even if these users are running different TDBS programs, and have different subsets of the record's fields displayed.

At the time that the newly committed record's modified fields are displayed on other user's terminals, any partial update each had done is rolled back automatically. Any rolled back fields are also redisplayed with the current database record values after the update. The user's cursor is positioned to the beginning of the field it was on when the update from another user was committed to disk, and he may re-enter any updates he wishes now that he knows the new values of this record.

#### SET UPDATE BELL

The program may determine if an alert bell should be given when a screen update and/or rollback occurs. The command syntax is:

#### SET UPDATE BELL TO [OFF] [ON] [ROLLBACK]

This alert bell is totally independent of the SET BELL command. It refers only to the transparent screen update function described above. If this alert is set OFF, then no alert is given when a screen is changed due to an update from another user. If this alert is set ON, then an alert is always given when any field on the screen being displayed is updated with a new value by another user. If this alert is set to **ROLLBACK**, then an alert bell is given only when a change by another user occurs to a field that has been edited by this user during this **READ** operation. This means that one or more partially edited fields have been rolled back and must be re-edited. This is the default setting.

# Hybrid Automatic Record Locking

If two or more programs are sharing a file, and one of them is using explicit record locking while one or more of them is using transparent file sharing, then TDBS enters a hybrid mode. In this mode, it will do transparent file sharing for the programs which expect it unless one of them attempts to update a record which is explicitly locked. In this case, TDBS will automatically report a record sharing violation error. Thus to handle this condition, the program using transparent file sharing must code an ON ERROR handler which fields this error and retries the update operation if it is to successfully share a file with a program which does explicit record locking.

# **TDBS Mailboxes**

TDBS provides an efficient method of passing information between either different TDBS programs, or multiple copies of a single TDBS program. This method is known as a mailbox. In order to make this feature as easy to understand as possible, a TDBS mailbox is implemented as a single record database file.

This allows you to structure the type and size of the information passed in the mailbox as you wish, by defining the record in the file appropriately. It also allows the language to access or modify mailbox values in the same way it accesses or modifies fields. Finally, mailbox values will be checkpointed in the specified file when all programs quit using it (or on demand as will be explained below) so that a mailbox may even be used to pass information between program runs as though it were a normal database file with only a single record.

Mailboxes are significantly more efficient than simply opening a shared file and writing to it. This is because mailbox transfers are almost always memory to memory and don't actually write anything to disk. The only time a mailbox will write to disk is when it is closed, when memory requirements force a checkpoint, or when the data is explicitly checkpointed to disk by the program.

# **Establishing a Mailbox**

A TDBS mailbox is established by opening a single record database file with the USE command and specifying the MAILBOX modifier. The syntax of a mailbox USE command is:

USE <.dbf file > [ALIAS < alias >] MAILBOX [JOURNAL]

An error will result if the specified database file does not contain exactly one record. This file defines the structure of the mailbox, and the contents of the single record become the initial values in the fields of the mailbox.

The JOURNAL option forces a checkpoint of the mailbox every time data is changed by any user of the mailbox. It has no real use in current versions of TDBS, but is operational.

#### Forcing a Mailbox Checkpoint

Mailbox data is usually not checkpointed to disk during a program's operation. If the program has mailbox information it wants to assure is saved in case of a power failure or system lockup requiring a computer reset, then it must be checkpointed to disk. This may be done at any time the program wishes by executing the command:

#### **GO TO 1**

This command is interpreted by a mailbox as a request to checkpoint the current mailbox buffer to the mailbox disk file record if any data has been changed. This is not normally required, but is provided for those special cases where such checkpointing is necessary to the program's recovery process.

#### Sending Mail

Mail is sent to all other users of a mailbox automatically when you update any field in the mailbox record. If these fields are updated by being part of a READ command, then the actual update occurs when the screen changes are committed by ending the READ command with other than an <ESC> key. The TDBS Screen Update and Rollback on Collision feature is always active for mailboxes and operates as described earlier for shared files. This feature can be used to create Visual Mailboxes if you wish.

If a *REPLACE* command is used to change the mailbox fields, then each individual *REPLACE* command comprises a *mail event*. Thus if multiple REPLACE commands are used to do a single update, another user could see that new mail has been received after each *REPLACE* occurs and see a partially modified mailbox record. If the entire mailbox update cannot fit on a single *REPLACE* command, you should make one field in the mailbox be a **semaphore** which indicates that the record is valid. The first *REPLACE* in the series should set that field to **.F.** and the last *REPLACE* in the series should set that field to **.T.** again to indicate the record is fully changed. Thus anytime that new mail is received, this field may be tested by the receiver to avoid using a partially modified record. **This entire problem may be avoided if all mailbox changes are expressed on a single REPLACE command, as TDBS can properly lock mailbox accesses in that case.** 

# **Receiving Mail**

Receipt of mail is automatic. Each reference to a field in a mailbox will always obtain the most recent value placed there by any program currently sharing the mailbox. However, it is often a requirement of a program to know if any new mail has been placed in the mailbox without going to a lot of programming effort. To provide this capability, TDBS provides two special functions as follows:

# NEWMAIL([wa])

This function tests for any alterations in a mailbox since the last time it was called. The optional numeric parameter "wa" specifies which work area the mailbox to be checked is open in. If it is omitted, then the current selected work area is assumed.

If there is new mail in the specified mailbox then .T. is returned, and the new mail flag is reset.

If there has been no change to the specified mailbox since the last NEWMAIL call, or if the specified area is not a mailbox, then .F. is returned.

Note: Because the NEWMAIL function clears the new mail flag when it is reported, the next function call will return .F. unless another update to the mailbox has occurred.

# WAIT4MAIL([n])

This function allows a program to do a prolonged wait for new mail on the current selected work area. The optional parameter "n" is a maximum number of seconds to wait for new mail, if it is absent, the wait is unlimited. If the user presses a key during the wait (whether or not a time limit was specified) then the wait is aborted. This function will return .T. if the wait ended with new mail received. It will return .F. if the user presses a key, the time limit expires, or the current selected work area is not a mailbox. Note: the typeahead buffer is flushed at the beginning of this function, so only a keypress after it is issued will abort the wait.

## **ON NEWMAIL**

In some instances, a program wishes receipt of mail to be handled automatically as it is received without having the program constantly ask if mail is present. For this purpose TDBS provides the command:

ON NEWMAIL DO Procedure

When this command is used, the named procedure is entered between TDBS instructions whenever new mail is received in any open mailbox. That procedure can then process the received mail, and issue a RETURN to continue with the interrupted program.

# USING\_BOX field

Another problem which arises in using mailboxes is how the sender can determine if a receiving program is still operating. To allow this determination, MAILBOX files honor a special "magic" field name USING\_BOX. If a mailbox has a field with this name it is treated as follows:

Any time any user opens or closes the mailbox file, TDBS sends mail to all other users of the mailbox. Since mailbox files are always closed no matter how a user exits, this will notify all remaining programs of a change in the mailbox user status.

At the time of the open or close TDBS automatically updates the USING\_BOX field to pass file sharing information as follows:

If USING\_BOX is a character field, the same information that the USING() function returns is automatically posted to show which users are sharing the mailbox. If the field length is shorter than 65 characters, information about the higher lines is discarded. If the field is longer than 65 characters, it is padded with periods (.).

If USING\_BOX is a numeric field, it will contain the number of users which are now sharing the mailbox file.

If USING\_BOX is a logical field, it returns a .F. if this is the only program using the mailbox and .T. if one or more others are currently sharing the mailbox. The following is an example of a procedure which makes use of a mailbox to allow users of a program to send messages to each other. Note: **Cmsg** is a 78 character mailbox field.

```
*********************************
  Two way communications via Mailbox *
@ 0,0 SAY "Type EXIT to leave ... "
              && init display row
Drow = 1
USE COMM MAILBOX && Open the mailbox COMM
DO WHILE .T.
  IF .NOT. WAIT4MAIL()
     +--------+
*
       No New Mail, User pressed a key
*
     +-------+
     @ 23,0
                   && position & clear
     ACCEPT TO Tline && Get input line
     DO CASE
       CASE UPPER(Tline) = "EXIT"
          EXIT
                   && get out of pgm
       CASE LEN(Tline) > 0 && if input
          REPLACE Cmsg WITH Tline &&snd msg
     ENDCASE
  ELSE
     +------+
       New Mail received, put on screen |
     +------+
÷
     @ DROW, 0 SAY Cmsg && Display Rcvd Msg
     Drow = Drow+1
                 && bump row number
     IF Drow > 21
       Drow = 1
     ENDIF
  ENDIF
ENDDO
               && Close the mailbox
USE
               && done with comm pgm
RETURN
```

Notes: This program does not defend against multiple programs writing to the mailbox at exactly the same time and losing a message. Interlocking may be done either by semaphores in the mailbox or by use of the RLOCK() function which will operate correctly on mailboxes.

# **Printer Support**

TDBS version 1.2 provides the capability to use up to four printers and arbitrate them among multiple programs. TDBS version 1.2 does not provide a printer spooler, so in order to use a printer, the program must first request it. If no other program is currently using that printer, then the printer is assigned to the program, and may be used. There are two methods of requesting printer assignment. These are:

#### SET PRINTER TO [LPT1][LPT2][LPT3][LPT4]

If one of the four printers is specified, then it is requested. If it is not in use, then it is assigned to this program and the next instruction is executed. If it is not available an error is generated. This error may be fielded with an ON ERROR routine and retries or other recovery action may be taken.

#### SET PRINTER TO

If no printer is specified, as in this example, then any currently assigned printer is released for use by any other program that may want it. No error is generated if the program doesn't own any printer at this time. **Note:** Any assigned printer is automatically released if the program exits (either normally or abnormally), if carrier is lost, or if the user is aborted.

# WAIT4LPT(n[,s])

This function provides an alternate method to bid for a printer. The printer being requested is given by "n" which must be the number 1, 2, 3, or 4. The optional parameter "s" gives a maximum number of seconds during which the command will attempt to request the printer. This function will return .T. if the specified printer is assigned to the program. It will return .F. if 1) The printer requested does not exist, 2) The optional time limit was reached without being able to acquire the printer, or 3) if the user pressed a key during the wait period.

# **Printer Control**

Once a printer has been assigned, then the standard dBASE printer routing commands may be used. These commands are:

#### SET PRINT [ON][OFF] SET CONSOLE [ON][OFF] SET DEVICE [PRINT][SCREEN]

The SET PRINT ON command will route the output of all ? or ?? commands to the printer. Through the use of such functions as TRANSFORM you may have fully formatted output to the printer. You may use the SET CONSOLE OFF command to prevent the printed output from being displayed on the user's console.

The SET DEVICE PRINT routes all @ ... SAY output to the printer, and SET DEVICE SCREEN routes it back to the console. The SET DEVICE, SET PRINT, and SET CONSOLE commands operate independently of each other.

# SET ALTERNATE

TDBS also supports the alternate file routing of the ? and ?? command. While dBASE routes all output except @ ... SAY to the alternate file, TDBS only routes the ? and ?? output. An alternate file is opened using the command:

**SET ALTERNATE TO** < file >

The default file extension if none is given is .txt. Once the file is open, the command:

SET ALTERNATE [ON][OFF]

Controls the routing to the alternate file. An alternate file is closed by either of the commands:

CLOSE ALTERNATE SET ALTERNATE TO

# Flat File I/O

TDBS 1.2 adds extensive flat file I/O capability to allow you to do direct access to non-database files. Two modes exist – binary and ASCII Line – to ease handling of each type of file. In addition, a rapid text search command (FLFIND) allows searching text files for keywords or text fragments quickly.

Caution! Flat File I/O does not arbitrate multi-user access as does normal database file I/O! If you open a file for write access from more than one program or user at a time you are on your own to avoid file damage!

While the flat file I/O commands allow direct access to all files on your system, you may access TBBS control files (such as USER-LOG.BBS, MSGHDR.BBS, etc.) only in read mode. Any attempt to open these files in a write mode will result in an error.

#### Flat File I/O Basics

You access non-database files using the FOPEN or FCREATE commands. These commands establish the access modes, whether new data is appended, and the size of the file buffer you will use. Note that the file buffer is removed from the TDBS work pool, so you may need to coordinate use of flat file I/O with closing work areas in some cases (see Understanding TDBS Memory Usage in Chapter 2).

When you open a file, TDBS will return a numeric "handle". This number identifies the file you have open for all further flat file I/O commands. You may have up to five files open at once in the flat file I/O mode.

The FOPEN, or FCREATE command also establishes the access method – Binary or ASCII Line. Binary mode is direct raw access to the file, while ASCII Line mode buffers reading a line at a time into normal TDBS string variables.

Any buffers you assign for flat file I/O are released only by the FCLOSE operation. Any error termination or the QUIT function

will automatically issue an FCLOSE on all open flat file I/O handles for this program.

Since you are called upon to set a buffer size for all FOPEN or FCREATE commands, a function named FMAXLEN() is provided to allow your program to know the maximum open space which you can use as a buffer.

When you do any flat file I/O, you may encounter a DOS error. If you do, the FERROR() function may be used to determine the reason or type of error which happened.

# Line Mode I/O

In Line Mode the flat file I/O system treats a file as a series of records (lines) separated by end-of-line character sequences. In ASCII line mode an "end-of-line" is automatically determined by TDBS. It may be either "CR,LF", "LF,CR", "CR" only, or "LF" only. To allow you to determine the end-of-line sequence, these characters are left on the end of the string in your TDBS variable. The CRTRIM() function will remove them for you. Note: LF = CHR(10) and CR = CHR(13).

In other words, line mode looks at the file the same way you look at it in an editor, one line at a time. The end of the file may be signalled by either a DOS EOF mark or by the character CHR(26)  $^{2}$  in the file data stream.

To make file access of a line mode file faster, you may allocate an internal buffer for TDBS to use. This buffer size is a tradeoff in memory usage (the memory is removed from the TDBS work pool for other uses) and speed of access. A 2048 byte buffer gives good response and a 4096 byte buffer is about optimum. Buffers smaller than 2048 bytes will slow down file access noticably.

The Line Mode access commands are FLREAD, FLWRITE, and FLFIND. FLREAD will read the next ASCII line into a TDBS memvar as a normal TDBS character string. FLWRITE will write a TDBS character string to the file. FLFIND will rapidly search the file from its current position to find the specified text fragment.

The FSEEK command may be used in line mode to either determine the current file position, or to re-position the file as you wish.

# **Binary Mode I/O**

Binary mode I/O is not buffered by TDBS. The buffer size you specify is the record size TDBS will use for all DOS I/O, and all I/O occurs directly between the file and the specified buffer. No assumptions are made about the file data, and only the DOS EOF mark applies to indicate end of file.

If the file record is less than 254 bytes in length, you may do direct I/O into TDBS string variables and have no buffer at all. Records greater than 254 bytes must use a buffer. The functions FBEXTRACT(), FBFILL(), FBMOVE(), and FBINSERT() allow manipulation of the data in the buffer to and from TDBS strings.

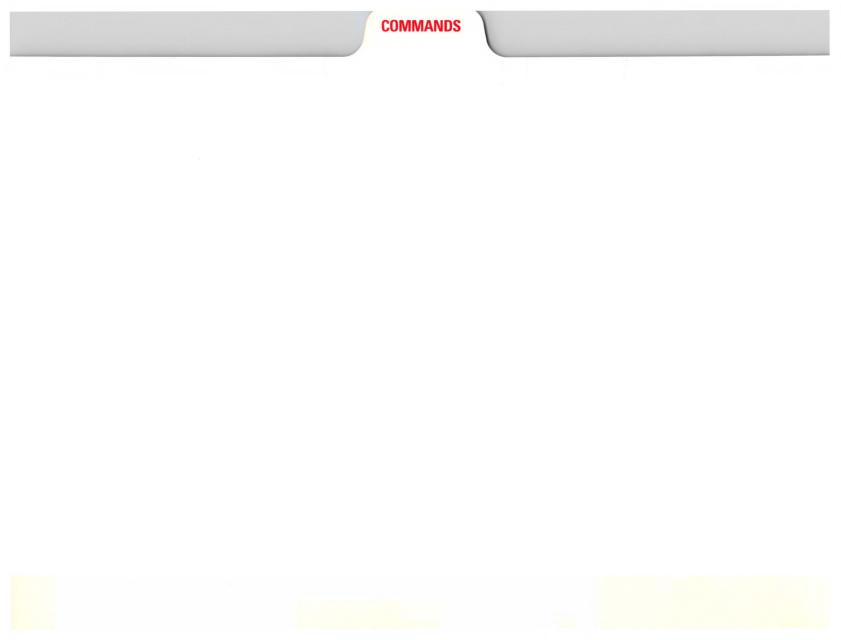
The commands FBREAD and FBWRITE are used to read and write records in a binary file. In TDBS 1.2, strings will correctly carry CHR(0) as a character (previous versions truncated strings at such a character) so that binary data may be placed in TDBS strings without loss.

The command FSEEK is used either to determine the present position within a binary file, or to move to a new position.

The function FERROR is used to determine the type of any error which occurs during binary file I/O.

# COMMANDS

# COMMANDS



# **Command Notation Conventions**

This chapter will describe each TDBS command in detail. First it will explain the notation used to describe the command syntax. Then a summary listing of all TDBS commands will be given, followed by each command described in detail.

#### **KEYWORDS**

All TDBS command names or keywords are shown in upper case. These portions of the command should be coded exactly as shown.

The following symbols are used to describe command syntax elements or their characteristics as follows:

| Symbol | Description<br>Indicates an item is optional. The brackets are<br>not typed.            |  |
|--------|---|--|
| []     |   |  |
| /      | Indicates one or the other option is specified, but not both.                           |  |
| •••    | Indicates repeating elements or intervening code.                                       |  |
| <>     | Indicates an element type as described below.<br>Note the angle brackets are not typed. |  |

In addition to these symbols, various portions of the command syntax descriptions refer to "elements" which represent a type of argument which is allowed at that point in the command. As you read the descriptions of each command, you may refer to the following section to determine the meaning of each element type, and how it relates to TDBS expressions, Keywords, and other constructs.

# Element Types used in syntax descriptions

The following element types are used to indicate items which appear in the TDBS commands. Each item will be shown in the command enclosed in angle brackets. When an actual instruction is coded, an item like this is replaced with an actual item of the described type.

| Element                  | Meaning   |                              |  |
|--------------------------|---|------------------------------|--|
| <alias></alias>          | File work area name                               |                              |  |
| <array></array>          | Array name  |                              |  |
| < col >                  | Numeric expression for screen column              |                              |  |
| < condition $>$          | Logical expression resulting in .T. or .F.        |                              |  |
| $< \exp >$               | expression of any type                            |                              |  |
| <expc></expc>            | Character expression                              |                              |  |
| <expd></expd>            | Date expression                                   |                              |  |
| <expl></expl>            | Logical expression                                |                              |  |
| <expn></expn>            | Numeric expression                                |                              |  |
| < ext >                  | File extension                                    |                              |  |
| < field >                | Database file field name                          |                              |  |
| <file></file>            | Name of a file, not including extension           |                              |  |
| <list $>$                | Items separated by commas                         |                              |  |
| <memvar></memvar>        | Memory variable of any type                       |                              |  |
| < parameter >            | Variable or expression of any type                |                              |  |
| < path >                 | Path to specified directory                       |                              |  |
| <pre>procedure&gt;</pre> | Procedure or program                              |                              |  |
| <prompt></prompt>        | Character expre                                   |                              |  |
| <row></row>              | Numeric expression for screen row                 |                              |  |
| <scope></scope>          | keyword and expression which selects the          |                              |  |
|                          | portion of a database file to process as follows: |                              |  |
|                          | RECORD n  | A single record              |  |
|                          | ALL   | All records                  |  |
|                          | NEXT n  | Group of records starting    |  |
|                          |   | with the current record      |  |
|                          | REST  | All records from the current |  |
|                          |   | record to the end of file    |  |
| < skeleton >             | Wildcard selection pattern including ? and *      |                              |  |
| <text></text>            | A string of characters                            |                              |  |
| <variable></variable>    | Memory variable or database Field name            |                              |  |
| <work area=""></work>    | Database file work area number                    |                              |  |

# Summary of TDBS Commands

#### ?/?? < exp list >

Displays the results of one or more expressions separated by a space.

#### @ <row>,<col>

Positions cursor and clears to end of line.

#### @ <row>,<col> CLEAR [TO <row>,<col>]

Clears a rectangular area of the screen.

# @ <row>,<col> [SAY <exp> [PICTURE <expC>]] [GET <variable> [PICTURE <expC>][RANGE <expN>, <expN>]]

Displays and/or inputs formatted data at specified row and column positions on screen. the @ ... SAY portion may also be optionally routed to the printer.

#### @ <row>,<col> TO <row>,<col> [DOUBLE]

Draws a single or double line box on the screen.

#### ACCEPT [<prompt>] TO <memvar>

Reads character string from the keyboard into specified memory variable.

#### APPEND BLANK

Adds a blank record to the end of the current database file.

#### APPEND FROM <file> [<scope>] [FOR <condition>] [WHILE <condition>] [TYPE] [DELIMITED [WITH BLANK/<char>]]

Adds records to the current database file from an ASCII text file or another database file.

#### AVERAGE < exp list> [<scope>] TO < memvar list> [FOR < condition>] [WHILE < condition>]

Averages a series of numeric expressions to memory variables for a range of records in the current work area.

#### CLEAR

Clears the screen, homes the cursor, and clears any pending GETs from the GET buffer.

#### **CLEAR GETS**

Releases any pending GETs from the GET buffer.

#### **CLEAR MEMORY**

Releases all PUBLIC and PRIVATE memory variables.

#### **CLEAR TYPEAHEAD**

Deletes any unread characters in the keyboard input buffer.

#### CLOSE ALL/ALTERNATE/DATABASES/FORMAT/INDEX

Closes the specified type of file, or all files.

#### CONTINUE

Resumes the most recent LOCATE search.

#### COPY TO <file> [<scope>] [FIELDS <field list>] [FOR <condition>] [WHILE <condition>] [TYPE] [DELIMITED [WITH <char>]]

Copies all or part of the current database file to a new file.

#### COPY FILE <file>.<ext> TO <file>.<ext>

Duplicates a file of any kind to the second file named.

#### COPY STRUCTURE [FIELDS < field list>] TO < file>

Creates an empty database file with field definitions from the current database file.

#### COPY TO < file > STRUCTURE EXTENDED

Creates a database file with four fields: FIELD\_NAME, FIELD\_TYPE, FIELD\_LEN, and FIELD\_DEC. The records of this new database file are the field definitions of the current database file.

#### COUNT [<scope>] [FOR <condition>] [WHILE <condition>] TO <memvar>

Tallies the number of records in the current work area for the specified scope and condition.

#### CREATE < file >

Creates an empty structure extended file.

#### CREATE < file1 > FROM < file2 >

Creates a new database file from a structure extended file.

#### DECLARE <array>[<expN>][...]

Creates one or more PRIVATE memory variable arrays.

#### DELETE [<scope>] [FOR <condition>] [WHILE <condition>]

Marks records in the current work area for deletion.

#### DIR [<path>] [<skeleton>]

Displays the names of files in the specified directory.

#### DO < procedure > [WITH < parameter list >]

Executes the specified procedure with an optional list of parameters passed to the procedure.

#### DO CASE ... CASE ... [OTHERWISE ...] ENDCASE

Selects a path of program execution from a set of conditions and branches on the first true evaluation.

#### DO WHILE < condition > ... [EXIT] ... [LOOP] ... ENDDO

Executes a looping structure while the condition is true (.T.).

#### DOTBBS TYPE < expN > OPTDATA < expC >

Execute an internal TBBS function as a subroutine.

#### EJECT

Sends a Top-of-form to the currently selected printer and resets PROW() and PCOL() to zero.

#### ERASE < filename >. < ext >

Deletes the specified file from the disk.

# FBREAD < expN1> < memvar> [<memvarC/expN2> [<expN3> [expN4]]]

Read from a binary mode file.

```
FBWRITE <expN1> <memvar> [<expC/expN2> [<expN3>
      [expN4]]]
```

Write to a binary mode file.

#### FCLOSE [<expN>]

Closes a file opened by FCREATE or FOPEN.

#### FCREATE < memvar > <file > < expN1 > [< expN2 > [expN3 > ]]

Creates a DOS file and opens it for flat file I/O.

#### FIND < character string >

Positions the record pointer to the first record with an index key that matches the specified character string.

#### FLFIND <expN1> <memvar> <expC> [<expN2>]

Locate next line in Line mode flat file which contains a target string.

#### FLREAD < expN> < memvar1> < memvar2>

Read a line from an ASCII text file.

#### FLWRITE < expN> < memvar> < expC>

Write a line to an ASCII text file.

#### FOPEN < memvar > < file > < expN1 > [< expN2 >]

Open a DOS file for flat file I/O.

#### FSEEK <expN1> <memvar> <expN2> [<expN3>]

Position a flat file (either binary or text mode).

#### GO/GOTO < expN >/BOTTOM/TOP

Moves the record pointer to the specified record in the database file open in the current work area.

#### HALT [<exp list>]

Terminates program processing after displaying the optional expression list if present. Program displays "Press any key" after exit and pressing any key returns to the calling TBBS menu. All files are closed on exit.

#### IF < condition > ... [ELSE ...] ENDIF

Permits conditional execution of commands with an optional alternative group of commands if the condition is false (.F.).

#### INDEX ON <key exp> TO <file>

Create index file for the current database.

#### INPUT [<prompt>] TO <memvar>

Takes a numeric entry from the keyboard and stores it in memvar.

#### LOCATE [<scope>] FOR <condition> [WHILE <condition>]

Positions record pointer to the first record matching the specified condition within the given scope for the current work area.

#### NOTE/\* [<text>]/&& [<text>]

Allows non-executing comments on a new line within a program source file. Comments following a && may be placed after any command to add comments at the end of a line.

#### ON DISCONNECT [< command >]

Enables or disables an accidental disconnect handler.

#### ON ERROR [<command>]

Enables or disables an error code handler.

#### ON ESCAPE [<command>]

Establishes or removes an escape key handler.

#### ON KEY [<command>]

Establishes or removes a "hot key" handler.

#### ON NEWMAIL [< command >]

Allows "interrupt" handling of received mailbox data.

#### PARAMETERS < memvar list >

Specifies memory variables to receive passed values or references to a called procedure. Matches WITH < parameter list > on DO.

#### 

Hides the specified memory variables allowing you to have new memory variables by the same name in a procedure and lower level procedures without disturbing any values stored in the originals.

#### PROCEDURE < procedure name > ... [RETURN]

Identifies the beginning of a procedure.

#### PUBLIC < memvar list >/< array list >

Declares memory variables or memory variable arrays to be global.

#### QUIT

Ends program execution, closes any open files, and returns to the calling menu in TBBS.

#### READ [SAVE] [FKEY] [SELECT]

Enters full screen editing mode using the current set of pending GETs or the pending .FMT file instructions.

#### RECALL [<scope>] [FOR <condition>] [WHILE <condition>]

Removes the DELETED mark from database records.

#### RELEASE [ALL[LIKE/EXCEPT < skeleton > ]] / < memvar list >

Erases memory variables.

#### RENAME < file1>.<ext1> TO < file2>.<ext2>

Changes the name of a file.

#### REPLACE [<scope>] [<alias>->]<field1> WITH <exp1> [,<field2> WITH <exp2>,...] [FOR <condition>] [WHILE <condition>]

Changes the contents of fields to new values.

#### **RESTORE FROM < file > [ADDITIVE]**

Retrieves memory variables from a .mem file.

#### **RETURN [TO MASTER]**

Exit a procedure and return control to either the calling procedure or the main program. Erases private variables created at this level.

#### SAVE TO <file> [ALL [LIKE/EXCEPT < skeleton>]]

Saves memory variables to a .mem file.

#### SEEK < exp>

Searches an index for the first key matching the expression.

#### SELECT < work area >/<alias>

Switches from the current work area to the specified work area.

#### SET ALTERNATE ON/OFF

Determines whether ? and ?? are echoed to the currently open alternate disk file.

#### SET ALTERNATE TO [<file>] [APPEND]

Creates a file to capture ? and ?? ASCII output.

#### SET BELL ON/OFF

Determines whether the bell rings during data entry.

#### SET CENTURY ON/OFF

Determines whether dates display century.

#### SET COLOR TO [< standard > [, < enhanced > ]]

Set screen display attributes.

#### SET CONFIRM ON/OFF

Determines if a return key is required to end GET field input.

#### SET CONSOLE ON/OFF

Determines whether the ? and ?? display to the screen.

#### SET DATE AMERICAN/ANSI/BRITISH/FRENCH/GERMAN/ITALIAN

Set the format for date type display and input.

#### SET DECIMALS TO < expN>

Sets the number of decimal places displayed for numeric values.

#### SET DELETED ON/OFF

Hides or makes visible records marked for deletion.

#### SET DELIMITERS ON/OFF

Determines whether or not delimiters display for GET input.

#### SET DELIMITERS TO [<expC>/DEFAULT]

Specifies the characters used to delimit GET input fields.

#### SET DEVICE TO SCREEN/PRINT

Sends output of @ ... SAY displays to screen or printer.

#### SET DISCONNECT [MAXINST < expN >] [MAXREPS < expN >]

Controls limits during an ON DISCONNECT procedure.

#### SET DISPLAY RULES TO STD1/STD2/TBBS

Allows adjusting certain unusual border conditions for screen displays to more closely match other dBASE language dialects.

#### SET DIVIDE BY ZERO TO ERROR/INFINITY

Set action to take on a divide by zero condition.

#### SET EDITOR

Controls features of the MEMO editor.

#### SET ESCAPE ON/OFF

Enables or disables a pending ON ESCAPE command.

#### SET EXACT ON/OFF

Determined whether exact matches are required for character comparisons, or whether short string complete matches are enough to qualify.

#### SET EXCLUSIVE ON/OFF

Determines whether a database file and its associated files are by default opened for shared or exclusive USE.

#### SET FILTER TO [<condition>]

Makes a database appear as if it contains only the records meeting the specified condition.

#### SET FIXED ON/OFF

Determines if decimal displays are or are not fixed point at the current SET DECIMALS length.

#### SET FORMAT TO [<.fmt file>] [NOCLEAR]

Activates or deactivates a format which executes whenever a READ is encountered.

#### SET FUNCTION < key> TO [<expN>]

Allows each function key to be programmed with a character string.

#### SET INDEX TO [<file list>]

Opens the specified index file(s) or closes any opened index files.

#### SET INTENSITY ON/OFF

Sets on or off the display of GET fields in enhanced video mode.

#### SET MARGIN TO < expN>

Sets the left margin of the printer for output.

#### SET MEMOWIDTH TO < expN>

Determines the column width of memo field display.

#### SET ORDER TO [<expN>]

Sets specified open index file as the master index.

#### **SET PRINT ON/OFF**

Determines if output from ? and ?? is sent to the printer.

#### SET PRINTER TO [LPT1/LPT2/LPT3/LPT4]

Requests specified printer, or releases printer if none specified.

#### SET PROCEDURE TO [<file>]

Opens named file at compile time and includes all procedures contained in it.

#### SET RELATION TO [<key exp>/RECNO()/<expN> INTO <alias>][ADDITIVE]

Relates open work areas according to key expressions.

#### SET TYPEAHEAD TO < expN >

Turns keyboard typeahead on or off.

#### SET SOFTSEEK ON/OFF

Allows relative key seeking. If the key is not found the record pointer is positioned to the next highest existing key.

#### **SET UNIQUE ON/OFF**

Toggles inclusion of non-unique keys when creating a new index.

#### SET UPDATE BELL TO ON/OFF/ROLLBACK

Sets the alert condition for a shared record display update change when using *Transparent File Locking* and the *Transparent Screen Update and Rollback on Collision* features of TDBS. This is independent of the SET BELL command.

#### SKIP < expN >

Moves the record pointer in the current work area either forward or backward the specified number of records.

#### STORE <exp> TO <memvar list> / <memvar> = <exp>

Stores the result of an expression to one or more memory variables.

#### SUM <expN list> [<scope>] TO <memvar list> [FOR <condition>] [WHILE <condition>]

Sums a series of numeric expressions to memory variables for records within the specified scope and qualifying conditions in current work area.

#### **TEXT ... ENDTEXT**

Displays a block of text to the screen or printer.

#### TYPE < file> [TO PRINT]

Displays the contents of a text file.

#### UNLOCK [ALL]

Releases file and record locks previously established by the functions RLOCK(), FLOCK(), WAIT4RLOCK(), and WAIT4FLOCK().

#### USE [<.dbf file>] [INDEX < file list>][ALIAS <alias>] [EXCLUSIVE] [READONLY]

Opens an existing database (.dbf) file, its associated memo (.dbt) file if any, and optionally associated index (.ndx) files in the current work area.

#### USE [<.dbf file>] [ALIAS <alias>] MAILBOX [JOURNAL]

Establishes an intraprogram mailbox with the structure and initial data of the specified single record .dbf file.

#### WAIT [<prompt>] [TO <memvarC>]

Suspends processing until a key is pressed.

#### ZAP

Removes all records from the active database file.

# ? / ??

|           | ? / ??   |
|-----------|--|
| Syntax:   | ?/?? <exp list=""></exp>   |
| Purpose:  | To display the results of one or more expressions separated by a space.  |
| Argument: | <exp list=""> is the list of values of any data type to display. The list may consist of any combination of expressions of any data types.</exp>   |
| Usage:    | There are two forms of the command. ? by itself displays a carriage<br>return/line feed before displaying the results of the expression list.<br>The ?? sends only the results of the expression list, without any<br>leading carriage return, line feed thus allowing successive ?? com-<br>mands to place output on the same line. |
|           | Note that the result of each expression in the list is separated by a space.   |
| Examples: | ? "Hello "<br>? "there"  |
|           | Results: Hello<br>there  |
|           | This example shows use of the ?? to display on the same line.  |
|           | ? "Hello "<br>?? "there"   |
|           | Results: Hello there   |
| See Also: | @ SAY<br>TEXT ENDTEXT  |

| @ | <b>CLEAR</b> |
|---|--------------|
|---|--------------|

| Syntax:    | @ <row>,<col/> CLEAR [TO <row2>,<col2></col2></row2></row>   |
|------------|--|
| Purpose:   | To clear a rectangular area on the screen  |
| Arguments: | <row $>$ , $<$ col $>$ define the coordinates of the upper left corner of the area to clear.   |
| Options:   | TO: The TO $< row2>, < col2>$ clause defines the lower right corner coordinates of the area to clear. If this option is absent, the then the lower left corner defaults to 24, 79. |
| Examples:  | 0 15, 15 CLEAR TO 20, 30<br>0 6, 0 CLEAR   |
| See Also:  | CLEAR  |

|            | @ SAY GET  |
|------------|--|
| Syntax:    | @ <row>,<col/> [SAY <exp> [PICTURE <expc]]<br>[GET <variable> [PICTURE <expc]<br>[RANGE <expn1>,<expn2>]]<br/>[NOEDIT][NOENHANCE][READONLY][SHARED][EDIT]</expn2></expn1></expc]<br></variable></expc]]<br></exp></row>  |
| Purpose:   | To display and input formatted data at specified row and column<br>positions. Note: Use of this command requires the user to be<br>configured for ANSI mode in the TBBS caller profile.  |
| Arguments: | <row> is the row of the start of the display.<br/><col/> is the column of the start of the display.</row>  |
| Options:   | SAY: The SAY option displays the result of the following expres-<br>sion (which may be of any type) at the specified coordinates on the<br>current DEVICE. TDBS supports two devices for the SAY option,<br>the console screen and the printer. Normally this output is directed<br>to the screen, but the SET DEVICE TO command may be used to<br>redirect the output to the printer or the screen. |
|            | @ SAYs to the printer behave slightly differently than they do<br>to the screen. If the row and column address given are less than the<br>current printer position, then an eject to a new page is done before<br>the text is output. So printing must proceed in order line by line<br>and left to right.   |
|            | On the screen, SAYs display using the standard color setting of the SET COLOR TO command. This is White on Black by default.   |
|            | <b>GET:</b> The GET option displays a field or memory variable at a specified screen coordinate and adds it to the list of pending GETs in the GET buffer. A subsequent READ invokes the full screen editing mode and allows the user to edit the contents of the variables in the pending GETs. See the READ command for a complete list of the editing keys and functions available.               |
|            | TDBS supports GETs which reference fields from other work areas if the fields are referenced using the alias: For Example:   |
|            | 0 25,36 GET alias->field   |

GETs display in the ENHANCED color setting of the SET COLOR TO command. The output of GETs is only to the screen, it cannot be redirected to the printer.

**PICTURE:** The PICTURE option defines the format options for display for both GET and SAY, and the input format for a GET when a READ is executed. There are two portions to a PICTURE format; Functions and the Template. Functions apply to the entire SAY or GET, while templates affect characters position by position.

Functions: A PICTURE function is a symbol preceded by an @ character inside the character string argument. If a template follows the function, it must be preceded by a space. More than one function may be specified, but only the first is preceded by the @ character. The functions available are:

| Func    | Туре | Action  |
|---------|------|---|
| A       | С    | Allows only alphabetic characters into a GET                    |
| В       | Ν    | Displays numbers left justified                                 |
| С       | Ν    | Displays CR after positive numbers                              |
| D       | D,N  | Displays dates in SET DATE format                               |
| Ε       | D,N  | Displays dates in European format                               |
| Κ       | All  | Clears GET if first key is not a cursor key                     |
| R       | С    | Inserts non-template characters                                 |
| S < n > | С    | Allows horizontal scrolling within a GET                        |
| Х       | Ν    | Displays DB after negative numbers                              |
| Ζ       | Ν    | Displays zero as blanks   |
| (       | N    | Encloses negative numbers in parenthesis with leading spaces    |
| )       | N    | Encloses negative numbers in parenthesis without leading spaces |
| !       | С    | Converts alphabetic characters to upper case                    |

**Templates:** Template symbols follow any functions specified in the PICTURE string. Either functions, or template characters or both may be specified in a PICTURE string. Each position in the output or input stream is mapped to the symbol in the same position in the template. The template symbols provided and their actions are:

#### **Chapter 4: TDBS Commands**

| Template | Action  |
|----------|---|
| A        | Displays only alphabetic characters                                     |
| N        | Displays only alphabetic and numeric characters                         |
| Х        | Displays any character  |
| 9        | Displays digits for any data type including sign for numeric data type. |
| #        | Displays digits, signs, and spaces for any data type                    |
| L        | Displays logicals as T or F   |
| Y        | Allows only Y or N  |
| 1        | Converts an alphabetic character to upper case                          |
| \$       | Displays dollar sign in place of leading space in numeric data type     |
| *        | Displays an asterisk in place of a leading space in a numeric data type |
|          | Specifies a decimal point position                                      |
| ,        | Specifies a comma position  |

Any other characters specified in the template overwrite the character at the same position in the source text input or output. If, however, you use the "R" function, non-template characters are inserted into the display. On GET input, the cursor will skip over these inserted characters.

**RANGE:** The RANGE option limits entry into date and numeric type variables by specifying the lower and upper allowable values. If the value input is not within the specified range, the GET will not be accepted. The field or memory variable cannot be altered until a value within the specified limits is entered.

**KEYWORDS:** TDBS adds four keywords to the GET and one keyword to the SAY command to allow special functions.

SHARED: This keyword may be added to the SAY command if a database field is being displayed. If this keyword is present, the SAY information is saved in a special form in the GETPOOL and if another user changes the displayed field during a READ command, that change will be automatically shown. This field cannot be edited during the READ.

NOENHANCE: This keyword allows a GET field to be displayed in normal mode with no delimiters even though SET ENHANCED and/or SET DELIMITED are ON. It overrides these commands for this field only.

**NOEDIT:** This keyword makes this GET field inaccessible during a READ command. It causes a GET to act the same as a SAY with the SHARED keyword.

**READONLY:** This keyword only has meaning if the GET field is a memo type. In this case, if the field is opened, the memo editor is entered in READONLY mode regardless of the current SET EDITOR setting. It allows overriding SET EDITOR for this memo field only.

**EDIT:** This keyword is the complement of READONLY. If the GET field is a memo type and the user opens it then the memo editor is allowed to update the field regardless of the current setting of the SET EDITOR command.

Notes:The length or type of a variable or a field may not be changed during<br/>a READ operating on a GET command. Any altered file fields are<br/>written to the file simultaneously at the end of the READ com-<br/>mand. If fields from a shared file are part of a multiuser<br/>GET/READ command sequence, then the TDBS Transparent<br/>Screen Update and Rollback on Collision feature is automatically<br/>invoked. This feature will immediately post any changes to your<br/>displayed fields (made by another user) to your screen as soon as<br/>they are committed, allowing you to view changes dynamically.

**Examples:** 

@ ROW()+2,6 SAY "Relative Screen Display"

This shows use of the ROW() function to position output relative to the current screen position. Note: These functions always return the screen position at the beginning of the command line.

This example shows establishing the length of the field by presetting it to a fixed length string.

```
Choice = 0

@ 15,30 SAY "Select a number";

GET Choice PICTURE "@Z 9"

READ
```

This example shows numeric input (the variable type and default value are established initially). The PICTURE command displays the zero as a space and limits input to a single numeric key.

```
@ 15,7 GET Charge->Amount
@ 16,7 GET Customer->Name
READ
```

This example shows the use of alias qualifiers to input data to two different database records in a single screen read. This operation can be tricky, and should be very carefully thought out.

See Also:

SET FORMAT CLEAR GETS READ SET CONFIRM SET BELL SET UPDATE BELL SET UPDATE BELL SET DELIMITERS SET INTENSITY SET COLOR TO SET EDITOR SET DEVICE COL(), ROW(), PCOL(), PROW(), SETPRC()

|            | @ TO   |  |  |
|------------|--|--|--|
| Syntax:    | @ <row>,<col/> TO <row2>,<col2> [DOUBLE]</col2></row2></row>   |  |  |
| Purpose:   | This command draws a box on the screen using either double or single lines.  |  |  |
| Arguments: | <row> and $<$ col> define the upper left corner of the box while<br><row2> and $<$ col2> define the lower right corner of the box. If<br><row> and $<$ row2> are the same, this command will draw a<br>horizontal line, while if $<$ col> and $<$ col2> are the same a vertical<br>line will be drawn. |  |  |
| Option:    | DOUBLE: If this option is specified, the box or line is drawn using<br>a double line. By default a single line is used.  |  |  |
| Usage:     | As with all of the @ command variations, this command requires that ANSI bet configured on in the user's TBBS profile.   |  |  |
| Example:   | € 0,0 TO 24,79 DOUBLE  |  |  |
|            | <b>Results:</b> A double line border is drawn around the entire screen   |  |  |
|            | € 10,20 CLEAR TO 20,40<br>€ 10,20 TO 20,40   |  |  |
|            | <b>Results:</b> First the box area is cleared, then a single line border is drawn around it.   |  |  |
| See Also:  | @ CLEAR  |  |  |

### ACCEPT

Syntax: ACCEPT [<prompt>] TO <memvar>

**Purpose:** Allow a text string to be entered from the keyboard into a specified memory variable.

**Argument:** < memvar > is the name of a memory variable where the text string entered from the keyboard is to be placed.

**Option: Prompt:** If the optional < prompt > string (or character expression) is present, it will be displayed before data input.

Usage: ACCEPT optionally prompts for user input and waits for a response. The response is stored as a character string in the specified memory variable, and may be from 1 to 254 characters in length. If the user presses only the return key, the specified memory variable is set to a null string (length=0). This command does NOT require ANSI to be set on in the TBBS user profile. Note: the specified memory variable must already exist, the ACCEPT command cannot be the first definition of the variable.

Example: ACCEPT "Enter your name" TO Name

See Also: @ ... SAY ... GET / READ INPUT WAIT INKEY()

## **APPEND BLANK**

Syntax:

APPEND BLANK

**Purpose:** This command adds a new blank record to the end of the database file in the currently selected work area.

Usage: The APPEND BLANK command adds a single blank record to the end of the database file. Any active index files are immediately updated and this record becomes the current record.

| Example: | USE Mail              | & & | Open database file |
|----------|-----------------------|-----|--------------------|
|          | SET FORMAT TO MScreen | & & | Define screen fmt  |
|          | ? RECNO()             | & & | 1 (1st rec)        |
|          | ? RECCOUNT()          | & & | 90 (last rec)      |
|          | APPEND BLANK          | & & | Add a blank rec    |
|          | ? RECNO()             | & & | Result: 91         |
|          | READ                  | 88  | full scrn fill-in  |
|          |                       |     |                    |

Multiuser:

When an APPEND BLANK is issued on a shared file, TDBS will transparently lock the file, add the record, and unlock the file. This process cannot fail in the TBBS environment, and thus all indexes and the database file are always kept consistent. If you wish to keep the record locked after the APPEND BLANK, you must issue a FLOCK() successfully before APPENDing. After the APPEND BLANK you may then issue an RLOCK() which cannot fail to release all of the file except the newly added blank record.

See Also: APPEND FROM SET FORMAT TO READ RECNO(), RECCOUNT() FLOCK(), RLOCK(), WAIT4FLOCK()

## **APPEND FROM**

| Syntax:   | APPEND FROM <file> [<scope>]<br/>[FOR <condition>] [WHILE <condition>]<br/>[TYPE] [SDF]/[DELIMITED [WITH BLANK/<char>]]</char></condition></condition></scope></file>  | ( |
|-----------|--|---|
| Purpose:  | This command appends database records from an external file to<br>the database file in the currently selected work area. The external<br>file may be either another database or an ASCII file.   |   |
| Argument: | < file > is the name of the external file to append. If the DELIMITED option is not specified, the extension is assumed to be .DBF, if the DELIMITED option is specified then the extension is assumed to be .TXT unless one is specified.   |   |
| Options:  | Scope: The <scope> limits the range of the records which are appended to the file.</scope>   |   |
|           | <b>Condition:</b> The FOR and WHILE < conditions > restrict which records will be appended to the file.  | ( |
|           | <b>Type:</b> There are three types of files which TDBS can import: SDF, DELIMITED, and .DBF files.   |   |
|           | <b>SDF:</b> This type is a System Data Format ASCII file. Each record is a fixed length, ends with a carriage return and line feed, and the end-of-file mark is a Ctrl-Z (0x1A).   |   |
|           | <b>DELIMITED:</b> This options identifies the external file as an ASCII file where fields are separated by commas and fields are bounded by double quote marks (the default delimiter). If the optional WITH $<$ char $>$ is specified, then $<$ char $>$ replaces double quotes as the delimiting character. Fields and records are variable length and each record ends with a carriage return and line feed. A $^Z$ (1Ah) is interpreted as an end-of-file mark if encountered before the physical end-of-file. |   |
| Usage:    | If the external file is a database file, only data with identical field<br>names and data type are appended to the current database. If the<br>width for a field in the active database is smaller than the width in<br>the external database, TDBS will truncate the field. If the field in   |   |

the external database is smaller than the active database TDBS will pad the data to fit.

If SET DELETED is OFF, records that are marked for deletion are appended to the current database (but are not marked deleted). If SET DELETED is ON, then records marked for deletion are not appended to the current active database.

Multiuser:TDBS will transparently interlock an APPEND FROM command<br/>on a record by record basis. This means that even if two or more<br/>users are appending data to the file at the same time, the database<br/>and index integrity is preserved. You only need to issue a FLOCK(<br/>) function, or open the file for EXCLUSIVE USE, if you wish to<br/>prevent any other records from being appended to the same file<br/>while this append is taking place.

**Example:** This example appends all records from the database file ORDTMP.DBF into the currently open ORDERS.DBF file. Records in ORDTMP which are marked for deletion, or which have no PartNo specified are not appended.

> USE Orders SET DELETED ON APPEND FROM OrdTmp FOR PartNo <> " "

This example appends a comma delimited ASCII file to the current work area's database file.

USE Orders APPEND FROM OrdAscii DELIMITED

See Also: COPY TO FLOCK(), WAIT4FLOCK()

### AVERAGE

| Syntax:    | AVERAGE <expn list=""> [<scope>] TO <memvar list=""><br/>[FOR <condition>] [WHILE <condition>]</condition></condition></memvar></scope></expn>                     |
|------------|--|
| Purpose:   | Calculates the average (arithmetic mean) of one or more specified numeric fields.  |
| Arguments: | < expN list > is a list of field names or expressions of numeric fields<br>which are to be averaged for each record processed.                                     |
|            | <memvar list=""> is a corresponding list of memory variables which will receive the averages.</memvar>   |
| Options:   | Scope: The < scope > option restricts the range of records which will be averaged.   |
|            | <b>Condition:</b> The FOR and WHILE < conditions > qualify which records within the database file will be averaged. Only those records which qualify will be used. |
| Example:   | The command  |
|            | AVERAGE Ext_Price TO AvgPrice  |
|            | calculates the average of the numeric field Ext_Price for all records<br>in the database, and stores the result in the numeric memory<br>variable named AvgPrice.  |
| See Also:  | SUM<br>COUNT   |

# CLEAR

| Syntax:   | CLEAR  |
|-----------|--|
| Purpose:  | To clear the screen and home the cursor.   |
| Usage:    | After CLEAR the screen is erased, and the cursor is at 0,0. Note:<br>This command also releases any pending GETs from the GET<br>buffer. |
| See Also: | @ CLEAR<br>CLEAR GETS  |

### **CLEAR ALL**

Syntax: CLEAR ALL

 Purpose:
 Closes any open files, releases all memory variables, releases any pending GETs from the GET buffer, and sets the SELECTed work area to 1.

See Also: CLEAR MEMORY CLEAR GETS CLOSE RELEASE

### **CLEAR GETS**

Syntax: CLEAR GETS

**Purpose:** Releases all pending GETs from the GET buffer.

Usage: As each @ ... GET command is executed, the screen position, variable and any associated PICTURE are stored in the GET buffer. The next READ command uses these stored GET commands to do full screen editing. If the SAVE option is used on a READ, or if a READ has not yet been performed, these commands remain in the buffer and new @ ... GET commands are added to the GET buffer. The CLEAR GETS command releases all stored GETs from the GET buffer.

See Also: @ ... CLEAR CLEAR CLEAR ALL

## **CLEAR MEMORY**

**CLEAR MEMORY** Syntax: Releases all memory variables, both PUBLIC and PRIVATE. **Purpose: Usage:** This command is used when you want to release all memory variables both public and private. This is in contrast to the RELEASE ALL command which only releases PRIVATE variables. All variable memory is returned to the free pool and may be used again. Example: **PUBLIC Var** var = SPACE(10)? TYPE ("Var") && Result: C CLEAR MEMORY ? TYPE ("Var") && Result: U See Also: **CLEAR ALL RELEASE ALL** TYPE()

## **CLEAR TYPEAHEAD**

Syntax:

**CLEAR TYPEAHEAD** 

**Purpose:** Empties the keyboard TYPEAHEAD buffer.

Usage: CLEAR TYPEAHEAD allows the program to delete any characters which the user has input, but which have not yet been read by the program. This is particularly useful if the program wants to be sure it waits for a fresh input. Note: Normally, TDBS has an output buffer as well, and the program execution may "run ahead" of the characters being sent to the user. CLEAR TYPEAHEAD will wait until all output has been sent to the user's terminal before purging the input buffer.

See Also: SET TYPEAHEAD LASTKEY(), NEXTKEY(), INKEY()

# CLOSE

| Syntax:   | CLOSE [ALL/ALTERNATE/DATABASES/FORMAT/INDEX]   |  |
|-----------|--|--|
| Purpose:  | To close the specified files.  |  |
| Options:  | ALL: Closes all alternate, database, and index files in all work areas.<br>In addition it releases all active filters, relations, and full screen<br>format (.FMT) files.        |  |
|           | <b>ALTERNATE:</b> Closes any currently open alternate file. Performs the same action as SET ALTERNATE TO with no arguments.  |  |
|           | <b>DATABASES:</b> Closes all open database and associated index files<br>in all work areas. Also releases all active filters. Does not affect<br>any SET FORMAT which is active. |  |
|           | FORMAT: Releases any active SET FORMAT file and returns the READ command to normal operation. Performs the same action as SET FORMAT TO with no argument.                        |  |
|           | <b>INDEX:</b> Closes any index files open in the current work area.<br>Leaves .DBF file positioned to the current record.  |  |
| Usage:    | Several other TDBS commands or actions do an implied CLOSE ALL command as follows:   |  |
|           | QUIT<br>HALT<br>An error which aborts the program<br>Loss of carrier or operator abort of the user   |  |
| See Also: | QUIT<br>HALT<br>SET ALTERNATE TO<br>SET INDEX TO<br>USE<br>CLEAR ALL   |  |

### CONTINUE

Syntax: CONTINUE

Purpose:

Resumes the pending LOCATE sequential search.

Usage:

CONTINUE searches from the current record position for the next record meeting the criteria of the most recent LOCATE command issued. If no match is found, EOF() is set to true (.T.) and FOUND() is set to false (.F.). If another record meets the criteria, then FOUND() is set to true (.T.) and the current record pointer is set to the matching record.

CONTINUE works only with LOCATE, it cannot be used with FIND or SEEK to find the next matching record. The search is sequential, and thus much slower than the indexed search of FIND or SEEK since all records in the file must be read in order to find matches. However, the field being searched does not need to be indexed to use LOCATE and CONTINUE.

Example:

| USE MyFile             |                    |
|------------------------|--------------------|
| ? RECCOUNT()           | && Result: 10      |
| LOCATE FOR State = "CO | )"<br>)            |
| ? FOUND(), RECNO(), EC | OF() && .T. 4 .F.  |
| CONTINUE               |                    |
| ? FOUND(), RECNO(), EC | OF() && .T. 6 .F.  |
| CONTINUE               |                    |
| ? FOUND(), RECNO(), EC | OF() && .F. 11 .T. |

See Also:

LOCATE FOUND()

## COPY TO

| Syntax:    | COPY TO <file> [<scope>] [FIELDS <field list="">]<br/>[FOR <condition>] [WHILE <condition>]<br/>[TYPE] [SDF]/[DELIMITED [WITH <char>]]</char></condition></condition></field></scope></file>   | ( |
|------------|--|---|
| Purpose:   | Copy all or part of the current database file to a new file.   |   |
| Arguments: | < file > is the name of the new file. If DELIMITED is not present,<br>the extension is .DBF and the new file will be a database type. If<br>the DELIMITED option is present, the new file will be an ASCII<br>file and the default extension is .TXT unless specified.   |   |
| Options:   | <b>FIELDS:</b> This option specifies a list of which fields to copy. If it is omitted, then all fields in the current database are copied.   |   |
|            | <b>Scope:</b> The <scope> limits the range of the records which are copied to the new file.</scope>  |   |
|            | <b>Condition:</b> The FOR and WHILE < conditions > restrict which records will be copied to the new file.  | ( |
|            | <b>SDF:</b> The SDF type outputs the file as a System Data Format ASCII file. Records are fixed length separated by a carriage return/line feed. Dates and logicals are treated as in <b>DELIMITED</b> below.  |   |
|            | <b>DELIMITED:</b> If this option is specified, then the new file is an ASCII file. Normally each field will be separated by a comma and enclosed in double quotes (the default delimiter). The WITH option allows the double quotes to be changed to any character you wish. On each field all leading and trailing blanks are removed. Dates are written as YYYYMMDD, and logical fields are written as T or F (without periods). |   |
| Example:   | COPY TO NameCO FOR State = "CO" DELIMITED  |   |
|            | Makes an ASCII file of all records from the state of Colorado.   | ( |
| See Also:  | APPEND FROM<br>SET DELETED   |   |

## **COPY FILE**

| ) | Syntax:    | COPY FILE <file>.<ext> TO <file2>.<ext2></ext2></file2></ext></file>   |
|---|------------|--|
|   | Purpose:   | To duplicate any type of file.   |
|   | Arguments: | <file>.<ext> is the name of the source file.<br/><file2>.<ext2> is the name of the destination file.</ext2></file2></ext></file>   |
|   | Usage:     | COPY FILE copies files to and from the HOMEPATH drive<br>unless a path is specified. If a file already exists by the name of the<br>destination file it will be overwritten. |
|   | Example:   | COPY FILE Data.bak TO Data.dbf   |
|   | See Also:  | CLOSE<br>COPY<br>USE   |

# **COPY STRUCTURE**

Syntax: COPY STRUCTURE [FIELDS < field list>] TO < file>

**Purpose:** Creates an empty database file with field definitions from the current database file.

**Arguments:** < file > is the name of the destination database file. The default extension is .dbf.

**Options:** FIELDS: The < field list > specifies which fields are to be used to define the new database file. If this option is not specified, then all fields are present in the new database file.

Example: USE Charges COPY STRUCTURE TO Summary; FIELDS CustNo, Amount USE Summary APPEND FROM Charges

Creates a new database with only the customer number and amount fields defined. Then the database is filled with values from the current database.

See Also: COPY STRUCTURE EXTENDED CREATE

## COPY STRUCTURE EXTENDED

Syntax:

#### COPY TO < file > STRUCTURE EXTENDED

**Purpose:** Create a database file whose contents are the field definitions of the current database file.

**Arguments:** < file > is the name of the structure extended database file.

Usage: COPY STRUCTURE EXTENDED creates a database file with four defined fields named: FIELD\_NAME, FIELD\_TYPE, FIELD\_LEN, and FIELD\_DEC. Each record of this file defines one field of the original database. This allows you to create and modify the structure of a database under program control. You can use CREATE FROM to create a new database file from the modified structure extended file.

Example:

USE File COPY TO Struc STRUCTURE EXTENDED USE Struc DO WHILE .NOT. EOF( ) ? Field\_Name, Field\_Type, Field\_Len,; Field\_Dec SKIP ENDDO

**Results:** 

FNAME C 15 0 LNAME C 15 0 ADDR1 C 20 0 ADDR2 C 20 0 STATE C 2 0 ZIP C 5 0 AMOUNT N 8 2

See Also:

CREATE CREATE FROM

# COUNT

I

| Syntax:   | COUNT [ <scope>] TO <memvar><br/>[FOR <condition>] [WHILE <condition>]</condition></condition></memvar></scope> | ( |
|-----------|---|---|
| Purpose:  | Tallies selected records from the current work area to a specified memory variable.                             |   |
| Argument: | < memvar > is a memory variable which will contain the count after<br>the command is executed.                  |   |
| Options:  | <b>Scope:</b> The < scope > option limits the range of records to count.  |   |
|           | <b>Condition:</b> The FOR and WHILE < conditions > select which records are counted.                            |   |
| Example:  | COUNT FOR Lname = "Smith" TO SmithCnt   |   |
|           | This will produce a count of the number of records in the database file where the last name is Smith.           | ( |
| See Also: | AVERAGE<br>SUM<br>TOTAL<br>RECCOUNT()   |   |

# CREATE

| ) | Syntax:   | CREATE <file></file>   |
|---|-----------|--|
|   | Purpose:  | Creates an empty STRUCTURE EXTENDED file.  |
|   | Argument: | < file > is the name of the empty structure extended file.   |
|   | Usage:    | This command allows you to create an empty structure extended<br>file. You can then write a subroutine which allows interactive or<br>"canned" input of records to this file to define a structure for a new<br>database. Then the file may be used in conjunction with the CRE-<br>ATE FROM command to create a new database with the defined<br>structure. |
|   |           | area after it is created.  |
| ) | See Also: | CREATE FROM<br>COPY STRUCTURE EXTENDED   |

۲

# **CREATE FROM**

| Syntax:    | CREATE <file1> FROM <file2></file2></file1>  |
|------------|--|
| Purpose:   | Creates a new database file from a structure extended file.  |
| Arguments: | <file1> is the name of the new database file to create.<br/><file2> is the name of the STRUCTURE EXTENDED file<br/>which specifies the structure of the new database file.</file2></file1>   |
| Usage:     | CREATE FROM produces a new database file whose field defini-<br>tions are taken from a structure extended file. See the COPY<br>STRUCTURE EXTENDED file for the definition of such a file.   |
| Example:   | <pre>CREATE "Struc"+ULINE()<br/>USE "Struc"+ULINE()<br/>DO WHILE .T.<br/>VN = SPACE(10)<br/>VT = SPACE(1)<br/>VL = 0<br/>VD = 0<br/>@ 5,0 SAY "Name: "GET VN<br/>@ 6,0 SAY "Type: "GET VT<br/>@ 7,0 SAY "Length: "GET VL PICT "999"<br/>@ 8,0 SAY "Decimals: "GET VD PICT "999"<br/>READ<br/>IF (.NOT. EMPTY(VN))<br/>APPEND BLANK<br/>REPLACE Field_Name WITH VN,;<br/>Field_Type WITH VT,;<br/>Field_Len WITH VL,;<br/>Field_Dec WITH VD<br/>ELSE<br/>EXIT</pre> |
|            | ENDIF<br>ENDDO<br>USE && Close Strucnn<br>CREATE NewFile FROM "Struc"+ULINE()<br>ERASE "Struc"+ULINE()+".DBF"  |

This example allows interactive creation of a database file with user specified structure. It also shows the use of the ULINE() function to guarantee unique temporary file names in a multi-line setting.

Caution: To comply with dBASE III Plus the CREATE FROM command will use all records in the structure extended file, even if they are marked deleted! Thus deleting a record will not remove it from a structure extended file. To remove a record you must not only delete it, but copy the file with SET DELETED ON so the record is physically removed.

#### COPY STRUCTURE EXTENDED CREATE

See Also:

# DECLARE

| Syntax:    | DECLARE <array1>[<expn1>]<br/>[,<array2>[<expn2>]]</expn2></array2></expn1></array1>  |
|------------|---|
| Purpose:   | Creates one or more memory variable arrays.   |
| Arguments: | <array> is the name of an array to create. Note that you can create more than one array with a single DECLARE statement.</array>  |
|            | $\langle expN \rangle$ is the number of elements in the array up to a maximum<br>of 4096. An array DECLARED with less than one element defaults<br>to one, more than 4096 causes an error message. (Note: Because<br>in TDBS 1.2 the MEMVAR area is limited to 6k, you will run out<br>of memory before you can allocate a 4096 element array. The real<br>limit is based on the memory available in this release).   |
|            | Note: The square brackets surrounding <expn> are<br/>a required part of the command syntax, and in this case<br/>do not signify an optional argument.</expn>  |
| Usage:     | DECLARE creates single dimensional PRIVATE arrays. Domain<br>operation for arrays is the same as for other memory variables, so<br>if any PUBLIC or higher level PRIVATE arrays by the same name<br>already exist, they are hidden until the program returns from this<br>level (See Memory Variable Domains in Chapter 2).   |
|            | Unlike memory variables, arrays cannot be saved in .MEM files using SAVE TO and RESTORE FROM.   |
|            | To assign a value to an array element, use the normal memory<br>variable assignment operator (=) or the STORE TO command.<br>To store a value to an entire array use the AFILL() function. To<br>retrieve a value from an array, refer to the element using a subscript<br>indicating its position in the array. The subscript is indicated in<br>square brackets following the array name, and must always be<br>present. The subscript may be any numeric expression. |

To determine the number of elements in an array, use LEN() by specifying only the array name (with no subscript) as the function argument.

**Data Type:** Elements within the same array may be mixed data type and obey all typing rules of ordinary memory variables. TYPE() returns an "A" for an array reference and the normal data type if an array element is referenced by subscript.

**Passing Parameters:** Array and Array elements can be passed as parameters to procedures the same as any other memory variables. Arrays are passed by reference while array elements are passed by value.

**PUBLIC arrays:** To create PUBLIC arrays, use the PUBLIC command. PUBLIC arrays follow the same rules as private arrays with the exception of their domain.

**Examples:** The following example creates an array, assigns values to the elements, and then displays them.

DECLARE array[5]
array[1] = "Hello"
array[2] = 1234
? array[1] && Result: Hello
? array[2] && Result: 1234

This example demonstrates macro array references.

x="V1[5]"
y="V1[3]
DECLARE &x &&create array V1 with 5 elements
&y = "abc" && assign "abc" to element 3

See Also: PRIVATE PUBLIC ADEL(), AFILL(), AINS(), ASORT(), ADSORT()

### DELETE

| Syntax:    | DELETE [ <scope>] [FOR <condition>]<br/>[WHILE <condition>]</condition></condition></scope>   |
|------------|---|
| Purpose:   | Marks one or more records for deletion in the current database.   |
| Options:   | <b>Scope:</b> This option limits the range of records in the file which will be deleted. If no scope is specified, then the default is the current record if no condition is specified. If a condition is specified, then the default scope is ALL.                     |
|            | <b>Condition:</b> The FOR and WHILE options specify the conditions used to test records in the file for deletion. Records which match the specified conditions are marked for deletion.   |
| Usage:     | DELETE marks records with the DELETED() attribute. These records may be filtered with SET DELETED ON. A record which is marked deleted, may have the mark removed by use of the RECALL command. If you need to remove all records from a database, use the ZAP command. |
| Example:   | Use File<br>? RECNO() && Result: 1<br>DELETE<br>? DELETED() && Result: .T.  |
| Multiuser: | When a file is shared, TDBS will transparently update a record<br>which is marked deleted. All other users of this record will imme-<br>diately see the record marked as deleted. No special record or file<br>locking is required.                                     |
| See Also:  | RECALL<br>SET DELETED<br>ZAP<br>DELETED()   |

| 100 |   |  |
|-----|---|--|
|     |   |  |
|     |   |  |
|     | - |  |

| ) | Syntax:<br>Purpose: | DIR [ <path>][<skeleton>]<br/>Displays a listing of files from the specified path.</skeleton></path>  |
|---|---------------------|---|
|   | Option:             | <b>Path:</b> This option may be used to specify a drive and/or path other than the HOMEPATH to list.  |
|   |                     | Skeleton: This option allows a wildcard specification to select only certain files to list in the directory.  |
|   | Usage:              | DIR displays one of two formats, depending on whether a skeleton<br>is given or not. If no skeleton is specified, then a listing of all .DBF<br>database files is given, including the name, number of records, and<br>last date modified for each file. If a skeleton is specified, then all<br>files which match are displayed with name, extension, number of<br>bytes, and date of last update. |
|   | See Also:           | HOMEPATH()  |

### DO

Syntax: DO < procedure > [WITH < parameter list >]

Purpose: Calls a procedure one level down and optionally passes parameters.

**Arguments:** cedure > is the name of the procedure to be executed.

- Option: WITH: Allows a < parameter list > to be specified to pass values to the called procedure. This is a list of up to 100 parameters separated by commas. Each parameter may be either a single memory variable, or an expression. Memory variables may be passed either by reference or by value.
- Using the WITH clause passes parameters to the specified procedure. If a parameter is a field or an expression, it is evaluated and the resulting value is passed to the subprocedure. A parameter consisting of a single memory variable is passed by reference if the receiving PARAMETERS command uses the same name as the name given in the WITH parameter list. Otherwise, memory variables are also passed by value. Memory variables passed by reference may be modified by the subprocedure to return values.

The PARAMETERS command in the subprocedure must have a parameter list with the same number of arguments to receive the passed parameters.

- **Compiling:** When the TDBS compiler encounters a DO statement with a procedure name which is not already known, it searches the default directory for a file by that name and compiles it if found. See the Installation section under AUTOCOMPILE for more details on this compiler feature.
- See Also: PARAMETERS PRIVATE, PUBLIC PROCEDURE RETURN SET PROCEDURE

# DO CASE

| Syntax:   | DO CASE<br>CASE < condition ><br>< commands ><br>[CASE < condition ><br>< commands >]<br>[OTHERWISE<br>< commands >]<br>ENDCASE  |
|-----------|--|
| Purpose:  | Creates a <b>programming structure</b> which allows the execution of<br>only one out of several blocks of commands depending on which<br>set of associated conditions is true (.T.).   |
| Options:  | <b>OTHERWISE:</b> If none of the CASE < condition > expressions<br>evaluate to true (.T.), the commands following the otherwise state-<br>ment are executed up to the next ENDCASE statement.  |
| Usage:    | The DO CASE structure is used when only one option out of several alternatives is to be selected. As soon as a single CASE < condition > is encountered which evaluates to true (.T.) then TDBS will perform all the commands between that CASE statement and either the next CASE statement, OTHERWISE statement, or ENDCASE statement. From that point on all other CASE statements are ignored, whether or not they are true, and execution resumes with the first command following the ENDCASE statement. If none of the CASE < condition > expressions evaluates as true, then any commands listed beneath the optional OTHERWISE statement will be executed. If none of the CASE < condition > expressions are true, and there is no OTHERWISE statement, then none of the commands between the DO CASE and the ENDCASE statement are executed. |
|           | DO CASE structures may be nested within other DO CASE struc-<br>tures up to a level of 255 levels.   |
| See Also: | IF ELSE ENDIF  |

17

|           | DO WHILE  |  |
|-----------|---|--|
| Syntax:   | DO WHILE < condition ><br>< commands ><br>[EXIT]<br>< commands ><br>[LOOP]<br>< commands ><br>ENDDO   |  |
| Purpose:  | Creates a <b>programming structure</b> which repeatedly executes a block of commands while a condition evaluates as true (.T.).   |  |
| Argument: | < condition > is evaluated each time the DO WHILE loop is<br>executed and controls the DO WHILE loop execution flow.  |  |
| Options:  | <b>EXIT:</b> The EXIT command unconditionally branches to the first command following the ENDDO command. This ends the loop no matter how the < condition > expression evaluates.   |  |
|           | <b>LOOP:</b> The LOOP command unconditionally branches to the most recently evaluated DO WHILE command. The controlling $<$ condition $>$ is again evaluated to determine the next cycle of the loop.   |  |
| Usage:    | The DO WHILE structure executes a block of commands<br>repeatedly as long as the controlling < condition > expression<br>evaluates to true (.T.). If the < condition > expression evaluates<br>to false (.F.) at the beginning of a loop, the program will branch to<br>the next instruction following the ENDDO command. |  |
|           | EXIT is used to terminate a DO WHILE structure based on some internal condition other than the controlling condition.   |  |
|           | LOOP is used to prevent execution of some commands within the DO WHILE based on some intermediate condition. It branches immediately back to the DO WHILE command line.   |  |

| Macros:   | Macros may be used in the DO WHILE control expression. They   |
|-----------|---|
|           | are re-evaluated on each loop as the expression is evaluated. |
| Examples: | An example of LOOPing within a DO WHILE structure:            |
|           | DO WHILE <condition></condition>                              |
|           | <initial commands=""></initial>                               |
|           | IF <intermediate condition=""></intermediate>                 |
|           | LOOP  |
|           | ENDIF   |
|           | <continued commands=""></continued>                           |
|           | ENDDO   |
|           | An example of Repeat Until looping in a DO WHILE structure:   |
|           | more = .T.  |
|           | DO WHILE more   |
|           | <commands></commands>   |
|           | <pre>more = <until condition=""></until></pre>                |
|           | ENDDO   |
|           | An example of EXITing a DO WHILE structure on a condition:    |
|           | DO WHILE .T.  |
|           | <commands></commands>   |
|           | IF <exit condition=""></exit>                                 |
|           | EXIT  |
|           | ENDIF   |
|           | ENDDO   |
| See Also: | DO CASE   |
|           | IF ELSE ENDIF   |
|           |   |

## DOTBBS

| Syntax:    | DOTBBS TYPE <expn> OPTDATA <expc></expc></expn>   |
|------------|---|
| Purpose:   | To allow a TDBS program to "shell" into TBBS and execute internal<br>TBBS functions, or chain to another option module.   |
| Arguments: | $\langle expN \rangle$ is a TYPE = number per the TBBS menu format. It must be in the range 1-255. Note: It must represent a valid TBBS internal function or installed option module function.  |
|            | <expc> is a 0 to 64 character string which supplies any optional data required by the specified TBBS command.</expc>  |
| Usage:     | This command allows a TDBS program to use the internal TBBS<br>menu functions as subroutines. This allows such capabilities as<br>downloading files from a TDBS program, etc. This command does<br>a "CLOSE ALL" before it calls the specified TBBS menu action<br>routine, so all work areas and any alternate file will be closed upon<br>return.                                     |
|            | Note: Commands which cause menu movement (TYPE = 5, 12, 35, and 45) or change user logon status (TYPE = 10 and 43) or which invoke an option module (TYPE = 200, 205, etc.) chain "one way" only. For these commands any ON DISCONNECT routine in the current TDBS program is called, all files are closed, and there is no return to this TDBS program.                                |
| Notes:     | If the underlying TBBS function suffers a non-recoverable error,<br>the calling program is not always able to detect that fact on the<br>return from a DOTBBS command. The DOTBBS command re-<br>quires TBBS 2.2 or newer, and will be rejected if it is attempted on<br>TBBS 2.1. The DOTBBS() function may be used to see if the<br>underlying TBBS system will support this command. |
| Examples:  | DOTBBS TYPE 4 OPTDATA "D:\FILES\FARLIST /NTL"   |
|            | DOTBBS TYPE 1 OPTDATA "C:\TEXT\FILE.DOC/D"  |
| See Also:  | DOTBBS()  |

# EJECT

| Syntax:   | EJECT  |
|-----------|--|
| Purpose:  | Advances printer to top-of-form.   |
| Usage:    | EJECT causes the currently selected printer to advance to the top<br>of page by sending a form feed character (ASCII 12). The internal<br>printer ROW and COL are reset to 0,0.  |
|           | You must use the SET PRINTER TO command or the WAIT4LPT<br>function to successfully request a printer before the EJECT com-<br>mand will have any effect. If you don't currently "own" a printer,<br>then the EJECT command will do nothing. |
| See Also: | SET PRINTER TO<br>WAIT4LPT()<br>PROW(), PCOL()   |
|           |  |

### ERASE

Syntax:

**Purpose:** Removes the specified file from the disk.

ERASE [<path>] <file>.<ext>

Argument: <file>.<ext> is the name of the file, including extension which is to be deleted. By default this file is assumed to exist in the HOMEPATH directory. You may optionally specify a full path to ERASE a file on any drive in any directory.

> A TDBS program has no restrictions on the files this command can delete. Do not delete any of the TBBS system control files. If you delete any of the TBBS system control files, system malfunction may result.

Example: ? FILE("Temp.txt") && Result: .T. ERASE Temp.txt && Result: .F. ? FILE("Temp.txt)

See Also:

USE CLOSE FILE() HOMEPATH()

# **FBREAD**

| Syntax:    | FBREAD <expn1> <memvar1> [memvar2/expN2&gt;<br/>[<expn3> [<expn4>]]]</expn4></expn3></memvar1></expn1>  |
|------------|---|
| Purpose:   | Read from a binary mode file.   |
| Arguments: | <expn1> is the handle of the binary file (returned by the FOPEN or FCREATE command).</expn1>  |
|            | <memvar1> receives the numeric count showing the number of<br/>bytes actually read by this command. If 0 bytes are read either the<br/>EOF was encountered or an error occurred. The FERROR()<br/>function will determine which it was. If <memvar1> is less than<br/>the read size, then EOF was encountered before the buffer was<br/>filled.</memvar1></memvar1>   |
|            | <memvar2 expn2=""> specify the buffer to use for the read. If<br/><memvar2> is used it must already exist, be character type and<br/>have the length already set by placing dummy data into the string.<br/>In this case, the string itself is used as the buffer for the read. If, on<br/>the other hand, &lt; expN2&gt; is used, it specifies the handle of an open<br/>flat file (usually this one). That file's buffer will then be used to<br/>determine the read size and it will be the read destination. You<br/>may open this file without a buffer and specify the buffer of another<br/>open file if you wish and thus share the memory used by buffering.</memvar2></memvar2> |
|            | <expn3> if specified indicates the first byte of the buffer to read<br/>into. 1 (or any number less than 1) specify the start of the buffer.<br/>Bytes "skipped over" in the buffer are not changed by the read.</expn3>  |
|            | $<\exp N4>$ if specified indicates the number of bytes to read from<br>the file in the range 0 to 32,767 bytes. If $<\exp N4>$ is larger than<br>the buffer, the extra bytes are skipped over in the file. If $<\exp N4>$<br>is smaller than the buffer size, then the extra bytes in the buffer are<br>left unchanged. If $<\exp N4>$ is not specified (or is -1) the size for<br>the read is the number of bytes between the first byte selected by<br>$<\exp N3>$ and the end of the buffer.   |
| Usage:     | FBREAD allows reading of binary data from a file which has been opened by FOPEN or FCREATE. You may use FBREAD to read  |

data into a TDBS string variable or into an internal buffer (if the record size is larger than 254 characters).

Example: FOPEN Handle TEMP.BIN 2 1296 IF Handle < 0 ? "Cannot Open File" ELSE FBREAD Handle Num\_read IF Num\_read = 0 IF FERROR(Handle) = 0 ? "EOF Hit" ELSE ? "Error Reading File." ENDIF Record = FBEXTRACT(Handle, 0, 128) ENDIF

This example reads a 128 byte record from the file "TEMP.BIN" in the homepath directory.

See Also:

FOPEN FCLOSE FSEEK FBWRITE FERROR(), FBEXTRACT(), FBINSERT(), FBMOVE(), FBFILL()

### **FBWRITE**

| Syntax:    | FBWRITE <expn1> <memvar> [<expc expn2=""><br/>[<expn3> [<expn4>]]]</expn4></expn3></expc></memvar></expn1>  |
|------------|---|
| Purpose:   | To write to a binary mode file.   |
| Arguments: | < expN1> is the handle of the binary file (returned by the FOPEN or FCREATE command).   |
|            | <memvar> is the variable which will receive the count of the<br/>number of bytes actually written to the file. If <memvar> returns<br/>zero, then a an error may have occurred. The FERROR() function<br/>should be used to determine the cause of the error.</memvar></memvar>   |
|            | $< \exp C/\exp N2 >$ specify the output buffer as follows: If $< \exp C >$ is used, the result of evaluating the character expression is the output buffer. If $< \exp N2 >$ is used, it is the handle of an open binary mode file which defined a buffer when it was opened. In that case the corresponding buffer is used. Note: If $< \exp N2 >$ evaluates to a -1, then this file's own buffer is used and must have been defined when the file was opened. This is also the default if this field is not present on the FBWRITE command. |
|            | < expN3> specifies the first byte of the buffer to write from. If $< expN3>$ is less than or equal to 1, then the first byte of the buffer is the first byte written. Any bytes skipped over by this parameter are ignored by the write.  |
|            | < expN4> specifies the number of bytes to write in the range 0 to 32,767. If < expN4> is larger than the buffer size, the extra bytes are written as binary zero or CHR(0) bytes. If < expN4> is less than the buffer size, the extra bytes in the buffer are ignored and not written to the file. Note: if < expN4> is zero, then no data is output to the file. However all DOS buffers are flushed to disk assuring that all data will be on disk if any interruption (such as a power failure) occurs.                                    |
| Usage:     | This command writes a record to a binary mode file. Data may be<br>written either from a file buffer (defined by FOPEN or FCREATE)<br>or directly from a TDBS character string expression.  |

#### **Chapter 4: TDBS Commands**

| Example: | FOPEN Handle TEMP.BIN 2 128   |
|----------|---|
|          | IF Handle $< 0$   |
|          | ? "Cannot Open File"  |
|          | ELSE  |
|          | FSEEK Handle CurSize 0 2 && Pos to EOF  |
|          | <pre>Dummy = FBINSERT(Handle, 1, "New End Rec")</pre>                                       |
|          | FBWRITE Handle Num Write 0 128  |
|          | IF Num Write = 0  |
|          | ? "Error Writing File."   |
|          | ENDIF   |
|          | FCLOSE Handle   |
|          | ENDIF   |
|          | This example appends a 128 byte record to the file "TEMP.BIN" in                            |
|          | the homepath directory. Note: Only the first 11 bytes of the record                         |
|          | are predictable, the rest of the record will have the previous con-<br>tents of the buffer. |

See Also:

FOPEN FCLOSE FSEEK FBREAD FERROR(), FBEXTRACT(), FBINSERT(), FBMOVE(), FBFILL()

#### **FCLOSE**

Syntax: FCLOSE [<expN>] **Purpose:** To close one or all open flat files. Argument:  $\langle expN \rangle$  is the handle of the single file to close (returned by FOPEN or FCREATE). If  $\langle expN \rangle$  is absent, all currently open flat files are closed. Usage: FCLOSE closes one or all currently open flat files. Any buffers associated with the file(s) are returned to the work pool. The status (success or failure) of an FCLOSE may be obtained by the FER-ROR() function. Attempts to close files which are not open are ignored without error. Example: FCREATE Handle TEST.FIL 3 FCLOSE Handle This example will create an empty file named "TEST.FIL" in the homepath directory. See Also: FOPEN FCREATE **FSEEK FLREAD FLWRITE FLFIND FBREAD FBWRITE** FERROR(), FBEXTRACT(), FBINSERT(), FBMOVE(), FBFILL()

#### FCREATE

Syntax:

FCREATE < memvar> <file> <expN1> [<expN2>
 [<expN3>]]

**Purpose:** To create a new file for flat file access.

Arguments:

< file > is the name of the file (with optional drive and path) that is to be created.

< expN1> is the mode of the create as follows: 3 = Binary Mode Read/Write Access 13 = Line Mode Read/Write Access 255 = Binary Mode "buffer only, no file"

Note: if  $\langle expN1 \rangle = 255$ , then no file is actually created. Rather a buffer is allocated and a handle assigned to it as an internal storage resource. Only FCLOSE operations may be done to such a handle, but the buffer may be used by any flat file operation and the FBINSERT(), FBEXTRACT() functions.

<expN2> specifies the DOS file attributes of the file. 0 (the default) indicates normal read/write attributes, 1 sets the read only attribute for this file which means you cannot write in the file again once it is closed.

<expN3> is the size of the associated buffer in bytes. If <expN3> is not specified (or -1 is used) the no buffer is explicitly associated with this file. In this case Line Mode files will assign temp buffers as needed (with loss in performance) and a binary mode file must "borrow" another file's buffer for read or write.

Example: FCREATE Handle TEST.FIL 3 FCLOSE Handle

This example will create an empty file named "TEST.FIL" in the homepath directory.

| See Also: | FOPEN  |
|-----------|--|
|           | FCLOSE   |
|           | FSEEK  |
|           | FLREAD   |
|           | FLWRITE  |
|           | FLFIND   |
|           | FBREAD   |
|           | FBWRITE  |
|           | FERROR(), FBEXTRACT(), FBINSERT(),               |
|           | <pre>FBMOVE(), FBFILL(), FLEN(), FMAXLEN()</pre> |
|           |  |

## FIND

| Syntax:   | FIND < character string >   |
|-----------|---|
| Purpose:  | Searches the master index for the first key which matches the specified $<$ character string $>$ and positions the record pointer to the corresponding record.  |
| Argument: | < character string > is all or part of the index key of the record you are searching for.   |
| Usage:    | FIND searches the master index starting with the first key, and<br>proceeds in index order until a match is found or a key is en-<br>countered which is greater than the search < character string > . If<br>there is a match, the record pointer is positioned to the record<br>number found in the index and FOUND() will report true (.T.). If<br>no match is found, then the record pointer is positioned past the<br>end of file if <b>SET SOFTSEEK</b> is OFF. If SET SOFTSEEK is ON,<br>then when a key is not found, the record pointer will be positioned<br>to the next highest key in the file.<br>Note: If the key is a character string, FIND is affected by SET<br>EXACT. If the search argument has leading blanks, it must be<br>delimited by quote marks and have the same number of leading |
| Example:  | blanks as the key text.<br>FIND 1001  |
|           | This will find the first record with a key $= 1001$ .   |
|           | Farg = "Smith"<br>FIND &Farg  |
|           | This will find the first record with a key = "Smith".   |
| See Also: | SEEK<br>SET EXACT<br>SET INDEX<br>SET ORDER<br>FOUND(), EOF()   |

#### **FLFIND**

FLFIND < expN1> < memvar> < expC> [< expN2>] Syntax: Find the next line in a Line Mode flat file which contains the Purpose: specified target string. Arguments: <expN1> is the handle of the Line Mode (ASCII text) file (returned by the FOPEN or FCREATE command). <memvar > is the numeric variable which will return the position of the first character of the target string within its line. 1 to 254 is the column where the target begins, 0 = no match, EOF hit in file, and -1 indicates an I/O error. The FERROR() function may be used to determine the type of I/O error.  $\langle \exp C \rangle$  is a character expression which forms the search target string. Note: Neither CHR(10) or CHR(13) may occur in this expression.  $<\exp N2>$  indicates the search mode. 0 is the default and indicates that the search is case sensitive (search is faster, but upper and lower case don't match). 1 indicates the search should ignore the difference between upper and lower case letters. Usage: FLFIND will search an open Line Mode flat file from its current position to the EOF for a match with the target text string. If <memvar > returns a non-zero value (indicating a match) the file is positioned to the START OF THE LINE that contains the string. Thus the next FLREAD issued will read the line which contains the matching text string, and < memvar > indicates the character within the line where the match begins. Since the file is positioned to the beginning of the line with the match, an FLREAD must be done prior to a repeated FLFIND in order to avoid finding the same string over and over. Once an FLREAD is issued, a repeat of the FLFIND will search for any other match following this line of the file.

Note: If the line is longer than 254 characters, then the file is positioned so that the matching string will be at the end of a 254 byte line fragment.

Note: If SET ESCAPE ON is in effect, the user may abort an FLFIND by pressing the  $\langle Esc \rangle$  key. In this case the FLFIND will immediately terminate with a "not found" condition and then the escape is processed normally.

Example: FOPEN Handle CHAPTER1.TXT 10 2048 FLFIND Handle Location "Abraham Lincoln" IF Location > 0 FLREAD Handle Size Record ? Record ELSE ? "No match found" ENDIF FCLOSE Handle

> This example locates the first line in the text file "CHAP-TER1.TXT" which contains the text "Abraham Lincoln". The search is case sensitive (i.e. upper and lower case must match). The line is then read into the string "Record" and the variable "Location" contains the offset within the string of the first character of the matched text.

See Also:

FOPEN FCREATE FCLOSE FSEEK FLREAD FLWRITE FERROR(), CRTRIM()

# **FLREAD**

| Syntax:    | FLREAD <expn> <memvar1> <memvar2></memvar2></memvar1></expn>  |
|------------|---|
| Purpose:   | To read a line from a Line Mode (ASCII text) flat file.   |
| Arguments: | <expn1> is the handle of the Line Mode (ASCII text) file<br/>(returned by the FOPEN or FCREATE command).</expn1>  |
|            | <memvar1> receives the numeric count of the number of bytes<br/>actually read. 0 indicates either EOF hit or an I/O error. The<br/>FERROR() function may be used to determine the error type.</memvar1>   |
|            | <memvar2> becomes a character string which is the next line of<br/>the file. The string placed in this variable contains the end of line<br/>sequence (the CRTRIM() function many be used to remove the<br/>end of line sequence). If the line is greater than 254 characters,<br/>only the first 254 characters are placed in this variable and the<br/>remainder of the line will be read with the next FLREAD.</memvar2> |
|            | Note: If the memvars do not exist, they are created as private to the current program level the same as $x = "abc"$ would do.   |
| Usage:     | The FLREAD command allows an ASCII text file (open in Line Mode) to be read one line at a time.   |
| Example:   | <pre>FOPEN Handle CHAPTER1.TXT 10 2048 DO WHILE .T.    FLREAD Handle Size Record    IF Size &gt; 0         ? CRTRIM(Record)    ELSE         EXIT    ENDIF ENDDO FCLOSE Handle</pre>   |
|            | This example displays the file "CHAPTER1.TXT" to the user's screen one line at a time.  |

#### **Chapter 4: TDBS Commands**

See Also:

FOPEN FCREATE FCLOSE FSEEK FLFIND FLWRITE FERROR(), CRTRIM()

## FLWRITE

| Syntax:    | FLWRITE <expn> <memvar> <expc></expc></memvar></expn>   |
|------------|---|
| Purpose:   | To write a line to a Line Mode (ASCII text) flat file.  |
| Arguments: | <expn1> is the handle of the Line Mode (ASCII text) file<br/>(returned by the FOPEN or FCREATE command).</expn1>  |
|            | <memvar> is the variable which will receive the count of the<br/>number of bytes actually written to the file. 0 bytes written indicates<br/>an error and the FERROR() function may be used to determine<br/>the type of error which occurred.</memvar>   |
|            | $<\exp C>$ is the character expression which is to be written to the file beginning at the current file position. The entire contents of the expression are written to the file. Note: The END-OF-LINE sequence MUST be part of this string! If $<\exp C>$ is zero length, then no data is output to the file. However all DOS buffers are flushed to disk assuring that all data will be on disk if any interruption (such as a power failure) occurs. |
| Usage:     | FLWRITE outputs a specified character string to a Line Mode (ASCII text) flat file.   |
| Example:   | <pre>FCREATE Handle TEST.TXT 13 2048 RecNum = 1 DO WHILE RecNum &lt;= 20 Record = STR(RecNum)+CHR(13)+CHR(10) FLWRITE Handle Size Record IF Size &lt; 1 EXIT &amp;&amp; Get out if write error ENDIF RecNum = RecNum + 1 ENDDO</pre>  |
|            | This example writes a file with 20 records each of which contains the ASCII number of the record.   |
| See Also:  | FOPEN, FCREATE, FCLOSE, FSEEK, FLFIND, FLWRITE<br>FERROR(), CRTRIM()  |

4-65

#### **FOPEN**

Syntax:

FOPEN <memvar> <file> <expN1> [<expN2>]

**Purpose:** To open a DOS file for flat file I/O.

Arguments: <memvar > will receive the numeric value of the handle assigned
to the opened file. A -1 indicates a failure in the open and the
FERROR() function will return the reason for the failure.

< file > is the name of the file (with optional drive and path) that is to be opened.

< expN1 > is the mode and type of open as follows:

- 0 =Binary Mode, Read Only
- 1 = Binary Mode, Write Only, Append
- 2 = Binary Mode, Read/Write
- 3 = Binary Mode, Read/Write, Append
- 10 = Line Mode, Read Only
- 11 = Line Mode, Write Only, Append
- 12 = Line Mode, Read/Write
- 13 = Line Mode, Read/Write, Append

Note: If  $\langle expN1 \rangle = 13$  (Line Mode, Read/Write, Append) the append will locate a logical EOF mark ( $^Z$  character) and append prior to it if the mark exists within the last 512 bytes of the file. If no  $^Z$  exists, the append uses the DOS EOF value. However,  $\langle expN1 \rangle = 11$  can only use the DOS EOF value to append, since it isn't allowed read access to locate any possible logical EOF mark.

 $< \exp N2 >$  specifies the size (in bytes) of an internal buffer to be allocated from the work pool and associated with this file until it is closed. If  $< \exp N2 >$  is not specified (or -1 is used) no buffer is associated with this file. If no buffer is specified for Line Mode files TDBS will use temporary internal buffers (at some loss of performance). For Binary Mode files you must "borrow" another file's buffer to do reads or writes to the file if you don't specify a buffer on the open. Examples: FOPEN Handle CONFIG.CTL 0 512

Open the file "CONFIG.CTL" in binary mode, read only, and set a 512 byte buffer. The file handle is placed in the variable "Handle".

ACCEPT "Enter File Name: " TO Fname FOPEN Handle2 &Fname 12 2048

Open the file entered by the user in Line Mode, Read/Write access, and assign a 2048 byte internal buffer for better performance during file operations.

See Also:

FCREATE FCLOSE FSEEK FLREAD FLWRITE FLFIND FBREAD FBWRITE FERROR(), FBEXTRACT(), FBINSERT(), FBMOVE(), FBFILL(), CRTRIM(), FLEN(), FMAXLEN()

## **FSEEK**

| Syntax:    | FSEEK <expn1> <memvar> <expn2> [<expn3>]</expn3></expn2></memvar></expn1>   |
|------------|---|
| Purpose:   | Position a flat file (either binary or text mode).  |
| Arguments: | <expn1> is the handle of the flat file to position (returned by the FOPEN or FCREATE command).</expn1>  |
|            | <memvar> is the numeric variable which will receive the position<br/>of the file (as bytes after BOF) after the file has been positioned.</memvar>  |
|            | $< \exp N2 >$ is the signed numeric value which indicates the number<br>of bytes to move the file (neg = backwards, pos = forward, 0 =<br>set to BOF, Current Position, or EOF (based on $< \exp N3 >$ ).     |
|            | <expn3> is a numeric value specifying the type of positioning<br/>to do as follows:<br/>0 = Position relative to BOF</expn3>  |
|            | <ul> <li>1 = Position relative to BOF</li> <li>1 = Position relative to current location (default)</li> <li>2 = Position relative to BOF</li> <li>10 = Position relative to BOF and SET AS NEW EOF</li> </ul> |
|            | 11 = Position relative to current location and SET AS<br>NEW EOF  |
|            | 12 = Position relative to EOF and SET AS NEW EOF.   |
| Usage:     | FSEEK allows you to position a flat file to any byte position within<br>it. It also allows you to determine the current byte position of the<br>file, or to set a new DOS EOF value.                          |
| Example:   | FSEEK Handle CurPos 0   |
|            | CurPos = Current Position of the file.  |
|            | FSEEK Handle NewPos 2048 0  |
|            | Position the file to byte 2048 from the beginning of the file.  |
|            |   |

FSEEK Handle NewPos -128 1

Position the file 128 bytes in front of its current position.

FSEEK Handle CurPos 0 11

Set the current file position as the new DOS EOF.

See Also:

FCREATE FOPEN FCLOSE FLREAD FLWRITE FLFIND FBREAD FBWRITE FERROR(), FBEXTRACT(), FBINSERT(), FBMOVE(), FBFILL(), CRTRIM()

# **GO/GOTO**

Syntax: GO/GOTO < expN > /BOTTOM/TOP

**Purpose:** Moves the record pointer to a specific record in the database file open in the current work area.

Argument:< expN> is the specific record number to move the record pointer<br/>to. GOTO moves the record pointer to this record, even if<br/>DELETED is on, or it falls outside the scope of the current SET<br/>FILTER TO conditions. In other words, even if it would normally<br/>be hidden, GOTO will access it. The number is always the physical<br/>record number in the file and is not affected by an index file.

**BOTTOM:** GO/GOTO BOTTOM moves to the last logical record in the current work area.

**TOP:** GO/GOTO TOP moves to the first logical record in the current work area.

Note: if there is an index, then it controls the first or last logical record for TOP or BOTTOM. Also, TOP and BOTTOM will skip records which are deleted if SET DELETE ON is active.

See Also: SKIP SET DELETED RECCOUNT(), RECNO()

# HALT

| Syntax:   | HALT < exp list >  |
|-----------|--|
| Purpose:  | Aborts a program with an optional error message.   |
| Argument: | < exp list $>$ is a list of expressions which are displayed identically to the ? command.  |
| Usage:    | This command will close all open files and print the error message.<br>Then it will print "Press Any Key" and when any key is pressed by<br>the user will return to the TBBS calling menu. |
| Example:  | HALT MESSAGE()   |
| See Also: | QUIT<br>?/??   |

### IF

| Syntax:   | IF < condition ><br>< commands ><br>[ELSE<br>< commands >]<br>ENDIF   |  |
|-----------|---|--|
| Purpose:  | Execute or skip a group of commands based upon one or more conditions.  |  |
| Argument: | < condition > is the control expression.  |  |
| Usage:    | If the control expression evaluates to true (.T.) then the commands<br>following the IF and up to either the ELSE (if present) or ENDIF<br>(if ELSE not present) are executed. If the control expression<br>evaluates false (.F.) then the commands after the IF are skipped,<br>and if an ELSE is specified, then the commands between the ELSE<br>and the ENDIF are executed.<br>IF structures may be nested within other IF structures up to 255<br>levels deep. |  |
| Example:  | <pre>number = 0<br/>INPUT "Enter test number" TO number<br/>IF number &lt; 50<br/>? "Less than 50"<br/>ELSE<br/>IF number = 50<br/>? "Equal to 50"<br/>ELSE<br/>? "Greater than 50"<br/>ENDIF<br/>ENDIF</pre>   |  |
|           | This example shows the nesting of IF structures.  |  |
| See Also: | DO CASE<br>IIF( )   |  |

#### **INDEX ON**

Syntax:

**Purpose:** 

INDEX ON < key exp> TO < file> Creates a file which contains an index to the records in the current work area's database file.

**Arguments:** < key exp > is an expression which returns the key value to place in the index for each record in the current database file. The maximum length of the key expression is 200 characters.

< file > is the name of the index file to create. The file extension is normally .NDX, but can be made anything you wish.

**Usage:** When an index file is used, the database records appear in key expression order although the index does not alter the physical order of the records in the database file. This allows you to create and maintain many different logical orders of records automatically.

Records which are marked for deletion, or filtered out (by a SET FILTER TO command) are still included in the index.

Character Date indexes are supported through the use of the DTOS() functions. Descending indexes are supported through the use of the DESCEND() function. Note: If you use the DESCEND() function to create descending indexes, you must also use it on the SEEK command.

Unique Indexes: If SET UNIQUE ON is active when the INDEX ON command is executed, the index created will have uniqueness as an attribute. As indexing proceeds, if two or more records have the same key expression value, only the first record will be included in the index. Whenever the unique index is updated via REPLACE, APPEND, etc. commands only unique records are indexed. Note that uniqueness becomes an attribute of the file after the INDEX ON command, and is unaffected by subsequent settings of SET UNIQUE.

#### **Chapter 4: TDBS Commands**

## INPUT

| Syntax:    | INPUT [ <prompt>] TO <memvar></memvar></prompt>   |  |  |
|------------|---|--|--|
| Purpose:   | Allows a number to be input from the keyboard into a memory variable.   |  |  |
| Arguments: | < memvar > is the name of the memory variable where the numeric input will be placed.   |  |  |
| Options:   | Prompt: If the optional < prompt > character expression is given<br>it is displayed to the screen before the numeric input is accepted. |  |  |
| Usage:     | INPUT cannot create a memory variable. The variable must be defined before the INPUT command is executed.                               |  |  |
| Example:   | Val = 0   |  |  |
|            | INPUT "Enter Number " TO Val<br>? Val && Result: Number input is displayed  |  |  |
| See Also:  | ACCEPT<br>WAIT  |  |  |

| LOCATE |
|--------|
|--------|

| Syntax:   | LOCATE [ <scope>] FOR <condition><br/>[WHILE <condition>]</condition></condition></scope>  |  |
|-----------|--|--|
| Purpose:  | Searches for first record in the current work area's database file<br>which matches the specified condition.   |  |
| Argument: | The FOR < condition > specifies the record to locate within the given scope.   |  |
| Options:  | Scope: The $<$ scope $>$ option limits the portion of the file which the locate will search. The default is ALL.   |  |
|           | WHILE: The WHILE option limits the range of the LOCATE command to those consecutive records for which the WHILE < condition > evaluates as true (.T.).   |  |
| Usage:    | LOCATE begins its search with the first record of the $<$ scope $>$ (or the first record of the file if no $<$ scope $>$ is given). If a matching record is located, the record pointer is positioned to it and FOUND() reports true (.T.). If no match is found, then the record pointer points past the end of the $<$ scope $>$ specified and the FOUND () reports false (.F.). |  |
|           | Subsequent searches on the same criteria are done using the CON-<br>TINUE command.   |  |
| Example:  | USE Orders<br>LOCATE FOR Customer = "45198"<br>? FOUND(), RECNO() &Results: .T. 17   |  |
| See Also: | CONTINUE<br>FOUND ()<br>FIND<br>SEEK   |  |

# NOTE/\*/&&

| Syntax:   | NOTE [ <text><br/>* [<text>]<br/>*@ [<command/>]<br/>[<command/>] &amp;&amp; [<text>]</text></text></text>  |  |  |
|-----------|---|--|--|
| Purpose:  | Allows comments in program and after command lines.   |  |  |
| Argument: | <text> is any string of characters used for commentary.</text>  |  |  |
| Usage:    | <b>NOTE</b> or * must be the first non-blank portion of a line. The remainder of the line is commentary. The && can occur anywhere on any line and ends any active command portion. The remainder of the line is treated as commentary.   |  |  |
|           | The *@ is a special form of the * command which may be used to<br>allow TDBS extended code to be considered commentary by other<br>dBASE language dialects. Such a line is normally treated as com-<br>mentary by the TDBS compiler unless the /XC switch is used on the<br>compiler command line. In this case, the remainder of the line<br>following the *@ is treated as a normal command line. |  |  |
| Example:  | NOTE this is a comment line<br>* This is also a comment line<br>SET CONFIRM ON && This portion is comment   |  |  |
|           | *@ SET UPDATE BELL ROLLBACK && Cmd if /XC   |  |  |

# **ON DISCONNECT**

Syntax: ON DISCONNECT [<command>]

 Purpose:
 Allows program control when a user accidentally disconnects via loss of carrier signal or operator abort.

**Option:** < command > is a single TDBS command which is executed in
response to a disconnect from loss of carrier or operator abort.
Normally this is a DO command (error procedure), but any command except another ON command or a structure command is
allowed. If no < command > is present, any current ON DISCONNECT command is disarmed.

Usage: The ON DISCONNECT command allows a program to provide its own "cleanup" procedure to be executed if the program is prematurely aborted due to a user disconnect or operator shutdown. Following are the considerations for an ON DISCONNECT procedure:

> The ON DISCONNECT takes effect between instructions, unless the current instruction is a multi-record instruction (e.g. APPEND FROM etc.). In that case the instruction is aborted at the end of the current record. If the instruction under execution at the time of the disconnect was a screen output or keyboard input instruction it is aborted and the ON DISCONNECT routine is entered.

> The ON DISCONNECT routine ends when a QUIT, CANCEL, HALT, RETURN TO MASTER, or a RETURN at the level of the ON DISCONNECT procedure. It will also end if any error occurs during ON DISCONNECT processing or if the maximum allowed number of instructions specified by the SET DISCONNECT command is exceeded. At the end of the ON DISCONNECT routine the internal TDBS cleanup routine closes all files and work areas which may be left open and releases any resources still held by the program.

> Once the ON DISCONNECT has been entered, all other exception routines (ON ERROR, ON ESCAPE, ON KEY) are inhibited for the remainder of the program's execution. Any new ON commands executed inside the ON DISCONNECT routine are ignored.

Any subsequent attempt to send output to the user's terminal is discarded, but any output routed to an ALTERNATE file or printer will be processed. (This allows an ALTERNATE file to record any errors which may occur during an ON DISCONNECT procedure).

Any attempt to read input from the user's keyboard will result in an immediate end of the ON DISCONNECT routine with a special error code [1215] "No keyboard reads allowed during ON DIS-CONNECT".

Since the ON DISCONNECT sequence never returns to the main program, any pending ON ERROR, ON ESCAPE, or ON KEY procedures are disarmed and ignored. If the ON DISCONNECT occurred during processing an ON ERROR procedure, the ERROR() function will continue to return the ORIGINAL error code. Normally ERROR() in an ON DISCONNECT routine will return the special error code [1210] "ON DISCONNECT triggered". This may be used if desired to see if an ERROR handler was partially complete and needs further cleanup.

**Example:** The following is an example of an ON DISCONNECT procedure which assures that a database field accurately reflects the logon or logoff state of a user.

SELECT 3 USE Status ON DISCONNECT DO Hangup REPLACE Logged WITH .T. && Show logged ... REPLACE C->Logged WITH .F. && Show OFF QUIT PROCEDURE Hangup

REPLACE C->Logged WITH .F. && Show OFF RETURN

See Also:

SET DISCONNECT ERROR()

## **ON ERROR**

Syntax: ON ERROR [<command>]

**Purpose:** Allows program control over most error conditions.

**Option:** < command > is a single TDBS command which is executed in
response to an error condition. Normally this is a DO command
(error procedure), but any command except another ON command
or a structure command is allowed. If no < command > is present,
any current ON ERROR command is disarmed.

Usage: The ON ERROR command responds to most errors by performing the < command >. Normally this is a DO command which invokes an ON ERROR handling procedure. Once an ON ERROR handler is armed by executing an ON ERROR with a command present, any fieldable error will invoke the ON ERROR command. Entry to the ON ERROR command disarms the ON ERROR handler until either a RETURN (or RETRY) to the original program level or another ON ERROR is issued. This means that if another error occurs within the ON ERROR handler, this error will not re-enter the handler but will abort the program. Note: If only a single instruction is given in the ON ERROR command, an implied RETURN to the program level occurs after that instruction.

Within an ON ERROR handler, the two functions ERROR() and MESSAGE() allow access to the error code and message text to allow conditional error handling.

The **RETRY** command when issued from an ON ERROR handler will re-execute the instruction which caused the error to occur. This instruction is restarted from its beginning, not resumed in the middle at the point where the error occurred. Thus if there are any "side effects" from functions evaluated in the instruction, these functions will be evaluated again from the beginning of the instruction.

The **RETURN** command when issued from an ON ERROR handler will return to the instruction FOLLOWING the one which caused the error to occur.

| Multiuser: | ON ERROR can be used to field explicit file locking errors in two<br>cases. First, it is used to do retries when your program wishes to<br>open files for exclusive access and needs to wait for another pro-<br>gram to close the file. Secondly, it can be used if your program is<br>written to use the <i>Transparent File Sharing</i> feature of TDBS but<br>wants to be able to share files with another program which may be<br>written using explicit file locking. |  |  |
|------------|---|--|--|
| Examples:  | The following is an example of a program which will wait for acce<br>to a file and notify the user with a message in the upper right har<br>corner of the screen if a wait is necessary.  |  |  |
|            | Msgout = 0<br>ON ERROR DO Filewa<br>USE File EXCLUSIVE  |  |  |
|            | ON ERROR  | && Disarm the handler  |  |
|            | IF Msgout <> 0  |  |  |
|            | 0,57  | && clear "Waiting" msg   |  |
|            | ENDIF   |  |  |
|            | PROCEDURE Filewait<br>IF ERROR() = 108 && File in use by another?   |  |  |
|            | IF Msgout = $0$   | "Waiting for file access"  |  |
|            | A=INKEY(1)  | && delay 1 second  |  |
|            | RETRY   | && try to open file again  |  |
|            | ENDIF<br>HALT MESSAGE()   | && Abort other error   |  |
|            |   | nple of a routine which allows a user to<br>ields the error if name he chooses is in use<br>er a new name. |  |
|            | IF .NOT. "." \$ New<br>NewName = NewNa  |  |  |
|            | ENDIF<br>ON ERROR DO FileE:<br>RENAME ThisFile.dl<br>ON ERROR   |  |  |
|            |   | we saratan ditivi nanatari   |  |

Here is an example of an error routine which could be used in a program which normally uses *Transparent File Sharing*, but wants to allow other programs to do explicit record locking on shared files.

ON ERROR DO Conflict && Arm Error Handler PROCEDURE Conflict IF ERROR() = 109 && Record in use? A=INKEY(1) && Delay 1 second RETRY && yes, retry access ENDIF HALT MESSAGE() && Abort Other error

See Also: RETRY RETURN ERROR(), MESSAGE(), INKEY()

### **ON ESCAPE**

Syntax:

ON ESCAPE [<command>]

**Purpose:** Allows a program to take "hot key" action if the user presses the <esc> key.

**Options:** < command > is a single TDBS command which is executed in
response to an <esc> key. Normally this is a DO command
(escape procedure), but any command except another ON command or a structure command is allowed. If no <command> is
present, any current ON ESCAPE command is disarmed.

Usage: The ON ESCAPE command responds only to the < esc > key, and only if the SET ESCAPE ON command is also in effect. ON ESCAPE is executed "between" commands. That is, each command instruction completes fully before the SET ESCAPE routine is entered. The RETURN command from a SET ESCAPE routine will continue program execution with the next instruction which would have been executed if the < esc > key had not been pressed. ON ESCAPE is activated whenever an < esc > is placed INTO the typeahead buffer. If both ON KEY and ON ESCAPE are set, only the ON ESCAPE command will respond to the < esc > key.

Once an ON ESCAPE routine has been entered, another  $\langle esc \rangle$  key will not cause it to interrupt itself. This status is cleared when the program returns to the level which was interrupted by the  $\langle esc \rangle$  key at which time the ON ESCAPE routine will again field  $\langle esc \rangle$  key presses.

See Also: SET ESCAPE ON/OFF ON KEY RETURN

#### **ON KEY**

Syntax: ON KEY [<command>]

**Purpose:** Allows a program to respond to input between commands.

**Options:** < command> is a single TDBS command which is executed in
response to a key waiting condition. Normally this is a DO command (hot key procedure), but any command except another ON
command or a structure command is allowed. If no < command>
is present, any current ON KEY command is disarmed.

Usage: ON KEY checks for a keystroke waiting in the typeahead buffer between each command in a TDBS program. If a key has been pressed, then the ON KEY command is executed. The key which triggered the ON KEY command stays in the input buffer until it is explicitly read by an input command.

> If an ON ESCAPE command is active, then the  $\langle esc \rangle$  key will not trigger the ON KEY command. If no ON ESCAPE command is active, then the  $\langle esc \rangle$  will trigger the ON KEY command and can be read as a normal character.

> The INKEY() function may be used to read the key and prevent repeated entry to the ON KEY routine. When the ON KEY routine is entered, another key will not interrupt it. If no ON KEY command is issued in the ON KEY routine to disarm the ON KEY function, then it will be automatically rearmed when the program returns to the level which was interrupted by the key.

See Also: ON ESCAPE SET TYPEAHEAD CLEAR TYPEAHEAD INKEY()

## **ON NEWMAIL**

Syntax:

:: ON NEWMAIL [<command>]

**Purpose:** Allows a program to take immediate action if any open mailbox is updated by another user.

**Options:** < command > is a single TDBS command which is executed in
response to another user has updated a field in one or more open
mailbox files. Normally this is a DO command (newmail procedure), but any command except another ON command or a structure command is allowed. If no < command > is present, any
current ON NEWMAIL command is disarmed.

Usage: The ON NEWMAIL command responds to the setting of the program's NEWMAIL flag. This flag is set whenever data is stored into any open mailbox field by another user, and remains set until a NEWMAIL() or WAIT4MAIL() function is executed.

If an ON NEWMAIL procedure is enabled by the ON NEWMAIL command, then it is entered at the end of the next instruction after mail is received. An ON NEWMAIL procedure exits with a RETURN.

Once the ON NEWMAIL procedure has been entered, ON ES-CAPE and ON KEY routines are inhibited until ON NEWMAIL exits.

When the ON NEWMAIL routine is entered the status of each active mailbox should be polled with the NEWMAIL() function. If additional new mail arrives during the ON NEWMAIL procedure execution, it is noted for subsequent detection when the ON NEWMAIL procedure is exited, but ON NEWMAIL will not interrupt itself. If any received mail exists for a work area which the ON NEWMAIL procedure did not issue a NEWMAIL() function, the ON NEWMAIL procedure will be immediately reentered when it exits. This effect can be minimized by looping checking each active mailbox with the NEWMAIL() function before exiting.

#### **Chapter 4: TDBS Commands**

| Example:  | The following allows a mailbox to be used to notify a current user<br>if a message has been sent to him. |  |  |  |  |
|-----------|--|--|--|--|--|
|           | ON NEWMAIL DO INMAIL   |  |  |  |  |
|           | SELECT 10<br>USE MailFile MAILBOX  |  |  |  |  |
|           |  |  |  |  |  |
|           | DO WHILE A=0   |  |  |  |  |
|           | A=INKEY(1)<br>ENDDO  |  |  |  |  |
|           |  |  |  |  |  |
|           | PROCEDURE InMail   |  |  |  |  |
|           | <pre>IF 10-&gt;MailFor = UNAME() .and. NEWMAIL(10) @ 23,1 SAY "Mail For You Waiting" ENDIF</pre>         |  |  |  |  |
|           | RETURN   |  |  |  |  |
| See Also: | USE MAILBOX  |  |  |  |  |
|           | NEWMAIL(), WAIT4MAIL()   |  |  |  |  |

## PARAMETERS

Syntax:

**Purpose:** Identifies memory variables which receive the parameters passed by the DO WITH command.

PARAMETERS < memvar list >

Argument:

Usage: Normally parameters are passed by value. That is, a new private version of the variable specified on the PARAMETER command is created, and the value of the corresponding expression on the DO WITH command is copied into this private variable. In this case, no original variables can be changed by the called procedure, and any memory used by the receiving variable is released when this procedure returns to its caller.

However, if the DO WITH parameter is a simple memory variable name or unsubscripted array name then the variable is passed by reference. That is, the called program is given access to the original variable directly (via the receiving variable name) and may modify it and return a value in it. Note: FIELDs are always passed by value.

Example: STORE 1 TO var1,var2,var3 DO Proc WITH var1,var2+var3,"string" ? var1, var2, var3 && Result: 3 1 1

> PROCEDURE Proc PARAMETERS var1,var2,var3 ? var1, var2, var3 && Result: 1 2 string STORE 3 TO var1, var2, var3 ? var1, var2, var3 && Result: 3 3 3 RETURN

See Also: DO PROCEDURE PRIVATE

#### PRIVATE

| Syntax:    | PRIVATE [ALL[LIKE/EXCEPT < skeleton > ]]<br>/< memvar list >/< array list >  |  |
|------------|--|--|
| Purpose:   | Creates a new instance of one or more memory variables or memory variable arrays at the current program level.   |  |
| Arguments: | <memvar list=""> is a list of variables to declare private.</memvar>   |  |
|            | Optionally ALL, ALL LIKE < skeleton >, or ALL EXCEPT<br>< skeleton > may be used to create new instances of any existing<br>variables which match (or fail to match) the skeleton.   |  |
| Usage:     | When a memory variable is declared PRIVATE in a procedure,<br>and existing definition of the same name, whether PUBLIC or<br>PRIVATE, is hidden until the current procedure RETURNs. The<br>domain of the new variable is the current procedure and all lower<br>level procedures. When the current procedure RETURNs, all<br>private variables it created are released and any previous defini-<br>tions again become accessible. This process may be nested to any<br>depth. |  |
|            | Note that PRIVATE, unlike PUBLIC, assigns no initial value to<br>the created variable. Instead the variable is undefined until a value<br>is explicitly assigned to it.  |  |
|            | Arrays: PRIVATE may be used in place of DECLARE to define<br>or redefine private domain arrays. Note that array definitions may<br>not be mixed with ALL LIKE or ALL EXCEPT skeletons.   |  |
| Example:   | PRIVATE ALL LIKE C*<br>PRIVATE var1, var2, array1[10], array2[20]  |  |
| See Also:  | DECLARE<br>PUBLIC<br>Memory Variable Domains (Chapter 2)   |  |

# PROCEDURE

| Syntax:   | PROCEDURE < procedure ><br>< commands ><br>[RETURN]   |
|-----------|---|
| Purpose:  | Identifies the beginning of a procedure.  |
| Argument: | < procedure > is the name of the procedure. The name may be up<br>to 8 characters in length, and must begin with an alphabetic char-<br>acter. Each name must be unique from all other procedures,<br>programs, and format (.FMT) file names in application.                          |
| Usage:    | A PROCEDURE is any executable block of code. It begins with<br>the PROCEDURE command and includes all code up to another<br>PROCEDURE command or the end of the file. TDBS allows<br>PROCEDURES to occur anywhere in any file, but they may not be<br>nested within other procedures. |
|           | A procedure is called with the DO command, and exits back to the calling program by using the RETURN command.   |
| Example:  | * Main Program<br>DO Proc1<br>DO Proc2<br>QUIT<br>PROCEDURE Proc1<br>? "Proc One?<br>RETURN<br>PROCEDURE Proc2<br>? "Proc Two"<br>RETURN<br>RETURN<br>RESULTS: Proc One<br>Proc Two   |
| 0         |   |
| See Also: | DO, RETURN<br>SET PROCEDURE TO  |

#### PUBLIC

Syntax: PUBLIC < memvar list >/< array list >

**Purpose:** Declares memory variables and or memory variable arrays as global, making them available to all procedures within a program.

# Argument: <memvar list> is the list of memory variables to declare as PUBLIC variables.

Usage: A memory variable may not exist when it is declared PUBLIC, or an error will occur. Declaring a variable PUBLIC creates a new variable with a type of logical, and an initial value of false (.F.). Once assigned this value, the variable has a domain of global meaning that all procedures may access it. While you cannot declare an existing variable PUBLIC, you can declare a PUBLIC variable to be PRIVATE which will temporarily hide it from other procedures.

Arrays: Declaring a PUBLIC array creates the array with the specified number of elements. Note that PUBLIC array elements are not defined until they are assigned a value.

**PUBLIC TDBS:** To include TDBS extensions in a program, and still allow the program to run properly under other dBASE language dialects, the special memory variable "TDBS" is initialized to true (.T.) when declared PUBLIC. Thus you may use the variable TDBS as the argument of IF ... ENDIF structures to hide TDBS specific code from other dBASE language dialects.

**Example:** PUBLIC TDBS, var1, array1[10], array2[10]

See Also: DECLARE PRIVATE PARAMETERS Memory Variable Domains (Chapter 2)

|           | QUIT  |
|-----------|---|
| Syntax:   | QUIT  |
| Purpose:  | Terminates program processing, closes all open files, and returns to the calling TBBS menu.   |
| Usage:    | QUIT may be used from anywhere in the program to terminate and<br>return to the TBBS menu. Any open files will be closed, and all<br>updates are cleanly written to disk. |
| Example:  | IF Answer \$ "Nn"<br>QUIT<br>ENDIF  |
|           | This example will terminate a program from any level if the variable<br>ANSWER contains either an upper or lower case N.  |
| See Also: | HALT  |

## READ

Syntax: READ [SAVE][FKEY][SELECT < field >]

**Purpose:** Allows full screen editing using either the pending GETs or a screen format (.FMT) file procedure.

**Options:** SAVE: The SAVE option retains the current set of pending GETs, allowing you to edit the same GETs by issuing another READ. If SAVE is not specified, the current GETs are released at the end of the READ command.

FKEY: The FKEY option causes the READ to terminate if the function keys F2 through F10 are pressed and no text is defined for them via the SET FUNCTION TO command. Following the READ the LASTKEY() function may be used to determine which function key ended the read, and the READKEY() command may be used to tell if any fields were updated.

SELECT < field >: The SELECT option positions the cursor initially to the named field. If this field is a memo field, the memo editor is entered instantly. If a memo field is selected in this way, the READ will ignore all other GETS and when the memo editor is exited by the user, the READ will terminate. In this special case the READKEY() function may be used to determine if the memo field was updated. READKEY() will return 14 if no changes were made, and 270 if the memo field was updated.

Note: The GET for a memo field must occur in a format file (as for any memo editing operation) for the automatic memo editor entry to operate correctly.

Usage: READ executes full screen editing using all GETs which have been issued since the last CLEAR, CLEAR GETS, CLEAR ALL, or READ (without the SAVE option) command.

If there is an active SET FORMAT file, then READ will call that file as a procedure before entering a full screen edit. If there are any READ commands in the format file, they are executed normally, and this READ command will operate on any GETs which are pending when the format procedure returns. Note: This is a TDBS extension to the dBASE language, which only allows @ commands in a format file.

Within a READ, the user can edit the contents of, and navigate between any pending GET field. Whenever a key is pressed which terminates a GET, the editing session is completed. The format of data which may be entered in each field is determined by the GET command and its associated PICTURE template and functions (if any).

**Communications:** Because TDBS uses serial communications ports to connect to terminals, it allows some VT-100, VT-52, and ANSI function key sequences as input. These key sequences are only recognized if they are sent at full speed, so they cannot be sent manually. If there is more than 100ms delay between the characters in a sequence, they will be treated as separate characters and not as a function sequence. The following table lists the VT/ANSI multi-key sequences along with the standard single key code. A program which is looking for explicit keys will only see the single key code, since TDBS maps the VT/ANSI keys to their single key value before passing them to the program. This is also true for all INKEY, LASTKEY, and NEXTKEY functions.

TDBS 1.2 also maps IBM PC scan code emulation (sometimes called "doorway mode") function keys. Remote keyboards which operate in this mode will allow all function keys to behave as would be expected on a local IBM PC keyboard (and as shown in the KEY column in the following table).

The following table lists the keys which perform the full screen editing functions in TDBS when in a READ command.

|                    | Ctrl | Inkey | VT/ANSI       |  |
|--------------------|------|-------|---------------|--|
| KEY                | Key  | Value | Sequences     | Effect on Eiting   |
| Up Arrow           | ^E   | 5     | EscA or Esc[A | Previous Field   |
| Down Arrow         | ^X   | 24    | EscB or Esc[B | Next Field   |
| Return             | ^м   | 13    |               | Next Field   |
| Left Arrow         | ^S†  | 19    | EscD or Esc[D | Character Left ‡   |
| <b>Right Arrow</b> | ^D   | 4     | EscC or Esc[C | Character Right ‡  |
| ^Left Arrow        | ^A   | 1     | Esc[H         | Word Left ‡  |
| ^ Right Arrow      | ^F   | 6     | Esc[K         | Word Right ‡   |
| Backspace          | ^н   | 8     |               | Destructive Backspace ‡  |
| Del                | ^G   | 7     |               | Delete char at cursor  |
| ^ Backspace        | ^T   | 20    |               | Delete word right  |
|                    | ^Y   | 25    |               | Delete to end of Field   |
| Ins                | ^v   | 22    |               | Toggle Insert Mode.  |
| ^END               | ^w   | 23    |               | Exit Read (save changes)   |
| Page Up            | ^R   | 18    | A A           | Exit Read (save changes)   |
| Page Down          | ^C   | 3     |               | Exit Read (save changes)   |
| Esc                | ^[   | 27    |               | Exit Read (discard any<br>changes to fields, save<br>changes to memvars) |

 $\dagger$  Since  $\land$ S is used for flow control (XOFF/XON) it cannot be directly input. Therefore TDBS will convert  $\land$ O internally to  $\land$ S to allow a single key input for cursor left.

‡ If the cursor is already at the "edge" of the field, these keys can advance to the previous or next field (depending on their direction).

Since TDBS does not implement a status line or scoreboard line on a full screen read, the Insert function is reset on exit from a field. Thus the **Ins** key must be pressed to put the edit mode back to insert when each new field is entered.

| Multiuser: | In TDBS, the READ function activates the <i>Transparent Screen</i><br><i>Update and Rollback on Collision</i> feature if any of the pending<br>GETs refer to fields in a shared file. With this feature, a record<br>lock is never necessary. In addition, if another user updates any<br>field which is part of your READ operation in progress, that update<br>will be immediately displayed on your screen. If you have edited<br>the field which was changed, your edit is rolled back (since the file<br>data was updated). You have the option of being alerted with a<br>bell if any displayed field is changed by another user, or only if such<br>a change rolls back a field you edited, but have not yet committed.<br>None of your changes are committed to the shared file until you exit<br>the READ with either a Page Up, Page Down, or Ctrl-End key. |
|------------|---|
| See Also:  | @ GET<br>CLEAR<br>CLEAR GETS<br>SET FORMAT TO   |

SET UPDATE BELL LASTKEY(), READKEY()

|            |   | - |
|------------|---|---|
| Syntax:    | RECALL [ <scope>] [FOR <condition>]<br/>[WHILE <condition>]</condition></condition></scope>   |   |
| Purpose:   | Reinstates records which were marked for deleted.   |   |
| Options:   | <b>Scope:</b> The < scope > limits the records in the file to RECALL. If no scope is specified, the default is the current record if no condition is specified. If a condition is specified, then the default scope is ALL  |   |
|            | <b>Condition:</b> The FOR option specifies the conditional set of records to RECALL within the given scope. The WHILE option limits the RECALL to the set of records from the current record to the first record which fails to meet the < condition >.   |   |
| Usage:     | If SET DELETED is ON, RECALL only reinstates the current<br>record or a specific record if you specify the RECORD scope. In<br>other words, RECALL cannot find deleted records, other than a<br>specific single record, if SET DELETED is ON. Therefore it is<br>usually best to SET DELETED OFF before doing a RECALL. |   |
| Example:   | USE File<br>GOTO 5<br>DELETE<br>? DELETED() && Results: .T.<br>RECALL<br>? DELETED() && Results: .F.  |   |
| Multiuser: | If the file is being shared, no record locking is required. The RECALL is immediately made known to all users of the file transparently by TDBS.  |   |
| See Also:  | DELETE<br>SET DELETED<br>DELETED()  |   |

## RELEASE

| ) | Syntax:    | RELEASE < memvar list ><br>/[ALL[LIKE/EXCEPT < skeleton > ]]   |
|---|------------|--|
|   | Purpose:   | Deletes memory variables.  |
|   | Arguments: | <memvar list=""> is a list of memory variables to delete.<br/><skeleton> is a wildcard mask specifying a group of memory<br/>variables to delete (or exclude from deletion).</skeleton></memvar>   |
|   | Usage:     | The effect of the RELEASE command differs depending upon<br>whether the ALL option (with or without a $<$ skeleton $>$ ) is used.<br>If it is, then only PRIVATE memory variables created at the current<br>procedure level are deleted. If the $<$ memvar list $>$ is used instead,<br>then the most recent instance of each specified memory variable is<br>deleted, even if the variable is PUBLIC or was created by a higher<br>level procedure. |
|   |            | Note: RELEASEing the local (or most recent instance) of a variable does NOT cause previous hidden instances to become accessible. That still occurs only on the RETURN from the procedure level which created the instance of the variable which was released. It does, however, return the memory used by the variable to the memory variable pool immediately.   |
|   | Example:   | <pre>PUBLIC var1<br/>var1 = "1"<br/>DO Proc<br/>? TYPE("var1") &amp;&amp; Result: C</pre>  |
|   |            | PROCEDURE Proc<br>PRIVATE var1<br>one = "2"  |
|   |            | ? TYPE ("varl") && Result: C<br>RELEASE varl   |
| ) |            | ? TYPE ("var1") && Result: U<br>RETURN   |
|   | 0          |  |

See Also:

CLEAR ALL, CLEAR MEMORY, PRIVATE, PUBLIC

## RENAME

| Syntax:<br>Purpose: | <b>RENAME [<path>]<file>.<ext> TO <file2>.<ext></ext></file2></ext></file></path></b><br>Renames a file to a new name.                       |
|---------------------|--|
| Arguments:          | <file>.<ext> is the name of a file to rename.</ext></file>   |
|                     | <file2>.<ext2> is the new name of the file.</ext2></file2>   |
| Usage:              | The file is assumed to be in the HOMEPATH directory unless an explicit $< path >$ is specified. Then it can be on any drive and in any path. |
|                     | Do not rename any of the TBBS system control files.<br>System malfunction may result!  |
| Example:            | RENAME Orders.dbf TO FebOrder.dbf  |
| See Also:           | COPY FILE<br>ERASE<br>FILE()   |

# REPLACE

| Syntax:    | REPLACE [ <scope>] [<alias>-&gt;]<field> WITH<br/><exp><br/>[,[<alias2>-&gt;]<field2> WITH <exp2> ]<br/>[FOR <condition>] [WHILE <condition>]</condition></condition></exp2></field2></alias2></exp></field></alias></scope>   |
|------------|--|
| Purpose:   | Changes the contents of selected fields to the results of the specified expressions.   |
| Arguments: | < field > is the name of the target field to change.   |
|            | $<\exp>$ is the expression which establishes the new value. It must be the same type as the target field.  |
| Options:   | Alias: Fields in other than the current work area may be replaced<br>by preceding the field name with the alias and an arrow (->)<br>delimiter.  |
|            | <b>Scope:</b> The <scope> selects multiple records to replace in the database file. The default <scope> is the current record only unless a <condition> is specified in which case the default <scope> is ALL.</scope></condition></scope></scope>   |
|            | <b>Condition:</b> The FOR option selects which records in the < scope > to replace. The WHILE option limits the scope from the current record until the condition fails.   |
| Usage:     | When the master index key field is REPLACEd, the index is<br>updated and the relative position of the record pointer is changed<br>to the new position of this record. This means that < scope > and<br>WHILE limits can give unexpected results if you are replacing the<br>master index key field. |
| Example:   | USE Cust<br>SELECT 2<br>USE Invoices<br>APPEND BLANK && Blank record to fill in<br>REPLACE Charges WITH (Cust->Markup * Cost),;<br>Custid WITH Cust->Custid,;<br>Cust->Tran WITH DATE()  |
|            |  |

#### Chapter 4: TDBS Commands

UPDATE

| Multiuser: | The TDBS Transparent File Sharing feature will assure that a<br>REPLACE on a shared file is propagated to all users immediately.<br>It also assures that the data and index file integrity is always correct.<br>Thus no record or file locks are required. If you wish to lock a<br>record out for more than the single REPLACE instruction (such<br>as may be that case if multiple REPLACE commands are required<br>to update the record) then you must use explicit record locking. |
|------------|---|
|            | See the RLOCK() and WAIT4RLOCK() commands for explicit<br>record locking details. Be sure that all records involved are locked<br>if the REPLACE involves fields from multiple work areas.  |
| See Also:  | APPEND<br>SET ORDER   |

4-100

## RESTORE

Syntax: **RESTORE FROM < file > [ADDITIVE] Purpose:** Retrieves memory variables stored in a memory (.mem) file. **Argument:** < file > is the memory (.mem) file to restore. **Option:** ADDITIVE: When the ADDITIVE option is specified, memory variables restored from the memory file are added to the existing pool of memory variables. Memory variables with the same name are overwritten unless they are hidden first. Without this option, all existing memory variables are released before the memory file is restored. Usage: When you **RESTORE** memory variables, they are initialized as PRIVATE at the current procedure level unless they are declared PUBLIC prior to the RESTORE and the ADDITIVE option is used. Without ADDITIVE, all restored variables are PRIVATE at the current level. **Multiuser:** In order to assure that multiple copies of the same program do not collide by using the same file to save variables when what is wanted is unique private files, you may use the ULINE() function to create a unique file name as follows: RESTORE "Temp"+ULINE() See Also: PRIVATE PUBLIC SAVE TO Memory Variable Domains (Chapter 2) ULINE()

## RETURN

| Syntax:   | RETURN   |
|-----------|--|
| Purpose:  | Terminates a procedure or program and returns control to the calling program.  |
| Usage:    | When RETURN is executed, any PRIVATE variables which were<br>created at this level are deleted. In addition, any previous instances<br>of those variables are unhidden and may be accessed again. Then<br>control returns to the calling program following the calling instruc-<br>tion. |
|           | Normally the calling instruction is a DO command. However, in<br>the case of an ON ERROR, ON ESCAPE, or ON KEY "interrupt"<br>the return will be to the instruction which would have normally<br>executed next had the interrupting condition not occurred.                              |
| Example:  | DO Proc<br>? "Returned"  |
|           | PROCEDURE Proc<br>? "In Proc"<br>RETURN  |
|           | Results: In Proc<br>Returned   |
| See Also: | PRIVATE<br>PUBLIC<br>DO<br>ON ERROR<br>ON KEY<br>ON ESCAPE<br>QUIT   |
|           |  |

## **RETURN TO MASTER**

Syntax: RETURN TO MASTER

**Purpose:** Returns from any level to the main program.

**Usage:** RETURN TO MASTER performs a return at each level to release any PRIVATE variables declared there. Then it returns to the instruction following the most recent DO command issued by the main program level.

See Also: RETURN QUIT

## SAVE

| Syntax:    | SAVE TO <file> [ALL [LIKE/EXCEPT <skeleton>]]</skeleton></file>  |
|------------|--|
| Purpose:   | Saves memory variables to a memory (.mem) file.  |
| Arguments: | <file> is the name of the file where specified memory variables<br/>are SAVEed. If no extension is specified, .mem is assumed.</file>  |
|            | <skeleton> is the wildcard mask to specify a group of memory variables to SAVE.</skeleton>   |
| Usage:     | SAVE copies the specified memory variables to the memory file<br>without any reference to domain (PUBLIC and PRIVATE vari-<br>ables alike are SAVEd). Hidden memory variables are not written<br>to the file. Only those instances of memory variables which are<br>accessible at the current program level are SAVEd. |
|            | If no arguments are given, ALL is assumed.   |
| Multiuser: | In order to assure that multiple copies of the same program do not collide by using the same file to SAVE variables when what is wanted is unique private files, you may use the ULINE() function to create a unique file name as follows:   |
|            | SAVE TO "Temp" + ULINE()   |
|            | SAVE and RESTORE open files in exclusive access mode and will return a "File in use by another" error code on open collisions.   |
| Example:   | <pre>var1 = "Initial string" SAVE TO Temp var1 = "New string" RESTORE FROM Temp ? var1 &amp;&amp; Result: Initial string</pre>   |
| See Also:  | PUBLIC<br>PRIVATE<br>RESTORE   |

## SEEK

÷

| ) | Syntax:    | SEEK <exp></exp>   |
|---|------------|--|
|   | Purpose:   | Searches for the first index key matching a given expression.  |
|   | Arguments: | $\langle \exp \rangle$ is an expression of any type which is to be matched with the master index key. It must be the same type as the key.   |
|   | Usage:     | SEEK searches the current work area's master index starting with<br>the first key, and continues until a match is found or there is a key<br>value greater than the search argument. If there is a match, the<br>record pointer is positioned to the record number found in the<br>matching index, and FOUND() will report true (.T.). If there is<br>no match, then the record pointer is positioned past the end of file<br>if <b>SET SOFTSEEK</b> is OFF. If SET SOFTSEEK is ON, then when<br>a key is not found, the record pointer will be positioned to the next<br>highest key in the file. |
| ) |            | If the key is a character type, the SET EXACT will affect the comparison.  |
|   | Example:   | USE Invoice INDEX Custno<br>SEEK "45136"<br>? FOUND(), RECNO() && Result: .T. 425<br>This indicates that the first invoice for the customer with number  |
|   |            | "45136" is record number 425 in the file.  |
|   | See Also:  | FIND<br>SET EXACT<br>LOCATE<br>SET DELETED<br>SET INDEX<br>USE<br>EOF(), FOUND(), RECNO()  |
|   |            |  |

## SELECT

Syntax: SELECT < work area > / < alias > Changes the current work area. Purpose: Arguments: <work area> is a number from 1 to 10 which designates the desired work area. <alias> is the name of a currently open work area. The letters A through J are "fixed aliases" for work areas 1 through 10. Each work area also has as its alias either the name of the database file which is open (by default) or an alternate alias which was specified on the USE command with the ALIAS option. Usage: You may use up to 10 work areas at the same time in TDBS, as long as the maximum number of allowable files open is not exceeded. Select does not open or close files. Select makes the specified work area the current work area. See Also: USE SET INDEX ALIAS(), SELECT()

| SET | ALTE | RNATE |
|-----|------|-------|
|-----|------|-------|

|  | Syntax:    | SET ALTERNATE TO [[ <path]<file>[.<ext>]][APPEND]</ext></path]<file>   |  |  |
|--|------------|--|--|--|
|  |            | SET ALTERNATE ON/OFF   |  |  |
|  | Purpose:   | Directs output from ? and ?? commands to a text file.  |  |  |
|  | Options:   | TO: The SET ALTERNATE TO variation of this command opens<br>a standard DOS ASCII text file which can receive the output of the<br>? and ?? commands under program control. The default extension<br>of this file is .txt if no extension is given. By default this file will be<br>created in the HOMEPATH directory unless a specific directory<br>is given. If no file name is present, then any currently open AL-<br>TERNATE file will be closed. If APPEND is used, then new data<br>is appended to any existing file text. |  |  |
|  |            | ON/OFF: The SET ALTERNATE ON or OFF commands allow<br>the program to dynamically route the output of the ? and ?? to the<br>alternate file. Note: SET ALTERNATE OFF does not close the<br>file, so a new SET ALTERNATE ON can be used to append new<br>? and ?? output to the file.  |  |  |
|  | Usage:     | Alternate files do not use a work area, and only one may be open<br>at a time. An alternate file is closed by CLOSE ALTERNATE,<br>CLOSE ALL, or SET ALTERNATE TO with no file name.  |  |  |
|  | Multiuser: | Alternate files are opened for exclusive use. In order to avoid conflict in file names when several users are running the same program, you may use the ULINE() function to append the line number to the file name.   |  |  |
|  | Example:   | SET ALTERNATE TO "Tmpfile"+ULINE()<br>SET ALTERNATE ON<br>? "Testing 1 2 3"<br>CLOSE ALTERNATE   |  |  |
|  | See Also:  | CLOSE<br>?/??<br>ULINE()   |  |  |

#### SET BELL

Syntax: SET BELL ON/OFF Sets the sounding of the warning bell on full screen edit on or off. **Purpose:** Usage: When SET BELL ON is issued, the bell will sound a warning in the following cases during full screen editing: 1. You completely fill a memory variable on input. 2. You attempt to enter invalid data. This could be data of either the wrong type, or wrong range as dictated by the PICTURE on the GET and the type of variable. Note: This BELL is completely independent of the operation of the SET UPDATE BELL ON/OFF command, even though both apply to full screen editing. This bell applies to your input, while the UPDATE BELL applies to input from another user which affected the screen you were editing. See Also: SET CONFIRM SET UPDATE BELL

## **SET CENTURY**

Syntax: SET CENTURY ON/OFF

Sets display of century digits for date values on or off.

sage: Using SET CENTURY ON will cause all default date display

Using SET CENTURY ON will cause all default date displays to use a four digit number for the year. Using SET CENTURY OFF will cause all default date displays to use a two digit year.

SET CENTURY affects date displays in ?, ??, @...SAY and @...GET as well as the CTOD and DTOC functions.

**Example:** 

**Purpose:** 

? DATE() && Result: 07/16/89 SET CENTURY ON ? DATE() && Result: 07/16/1989 See Also: DATE() CTOD() DTOC() YEAR()

SET CENTURY OFF

#### **SET COLOR**

Syntax: SET COLOR TO [< standard >] [, < enhanced >] Purpose: Defines colors for the next screen output. **Options:** Each option (<standard> and <enhanced>) define both a foreground and optional background color as described below. <standard> defines the colors used for all normal output. <enhanced> defines the colors used by the data fields in an @...GET command. If no options are present, the colors are reset to their defaults. Note: TDBS will accept up to three more arguments on the SET COLOR TO command for syntax compatibility with other dBASE language dialects. However those arguments will have no effect in TDBS operation. Usage: To display a particular color combination a one or two letter code is used. In addition, attributes may be added to the color by use of the + (for high intensity), \* (for blinking), or (X) for inverse video. Note: TDBS uses the ANSI "Set Graphics Rendition" code sequences to communicate color and attribute changes to the user's terminal (See pages 2-23 & 2-24 of your TBBS manual for these codes). This means that your program should not set any colors which the terminal in use cannot support. All VT-100 or ANSI terminals will support the default color settings, high or low intensity, blink, and inverse video. Many do not respond to explicit color settings. Keep this in mind when coding colors in your program. Foreground and background color settings are divided by a slash character (which is explicitly coded in this command). The following table lists the color codes which you can use on the SET COLOR TO command:

| Color or Attribute       | Code                |
|--------------------------|---------------------|
| BLACK                    | Ν                   |
| BLUE                     | В                   |
| GREEN                    | G                   |
| CYAN                     | BG                  |
| RED                      | R                   |
| MAGENTA                  | RB                  |
| BROWN (YELLOW if + used) | GR                  |
| WHITE                    | W                   |
| INVISIBLE                | Х                   |
| UNDERLINE                | U (Monochrome only) |
| INVERSE VIDEO            | I                   |
| BLINKING                 | *                   |
| HIGH INTENSITY           | +                   |

**Examples:** 

| SET COLOR TO W+/N, X    | && GET field invisible |
|-------------------------|------------------------|
| Password = SPACE(6)     | && Set size of field   |
| @ 23,0 SAY "Password: " | GET Password           |
| READ                    |                        |
| SET COLOR TO            | && Return to defaults  |

This example allows the password to be invisible to the user while it is being typed in. After the password is input, normal colors are restored.

| Menu = "W+/R"      | & & | Brt White on Red |
|--------------------|-----|------------------|
| Text = "W/N"       | & & | White on Black   |
| SET COLOR TO &Menu |     |                  |
| DO DispMenu        |     |                  |
| SET COLOR TO &Text |     |                  |

This example shows the use of macros to allow parametric control over various categories of color settings.

See Also:

SET INTENSITY @ ... SAY ... GET

## **SET CONFIRM**

Syntax: SET CONFIRM ON/OFF

**Purpose:** Either terminates the current GET field input automatically, or requires an explicit terminating key press.

Usage: If SET CONFIRM OFF is in effect, then typing a character in the last position of a GET field will automatically terminate the GET input and advance to the next field. If SET CONFIRM ON is in effect, then only one of the following field terminating keys will terminate entry:

| Ctrl-END   | (^W) |
|------------|------|
| Up Arrow   | (^E) |
| Down Arrow | (^X) |
| Page Up    | (^R) |
| Page Down  | (^C) |
| Return     | (^M) |
| Esc        | (^[) |

In addition a Left Arrow from the FIRST field will terminate, as will a return or right arrow from the LAST field.

See Also:

@ ... GET READ SET BELL

## SET CONSOLE

Syntax:

SET CONSOLE ON/OFF

**Purpose:** Allows program control over sending ? and ?? to screen.

Usage: SET CONSOLE affects whether the output of the ? and ?? commands displays on the screen. It is usually used in conjunction with SET PRINTER or SET ALTERNATE to prevent ? and ?? output from going to the console as well as the alternate destination.

 Example:
 SET PRINTER TO LPT1
 && Acquire LPT1

 SET PRINTER ON
 && Route ? to LPT1

 SET CONSOLE OFF
 && Don't display it

 ? "Send this line to the printer"

 SET CONSOLE ON

 SET CONSOLE ON

 SET PRINTER OFF

 SET PRINTER OFF

 SET PRINTER TO

 SET PRINTER TO

See Also: SET PRINTER SET ALTERNATE

## SET DATE

| Syntax:   | SET DATE < format >  |  |  |  |  |
|-----------|--|--|--|--|--|
| Purpose:  | To set the format of the date display, input, and function calls.  |  |  |  |  |
| Argument: | The < format > argument establishes the date format until the next<br>SET DATE command is encountered. The < format > argument<br>must be one of the following:  |  |  |  |  |
|           | AMERICAN(Format is: mm/dd/yy)ANSI(Format is: yy.mm.dd)BRITISH(Format is: dd/mm/yy)FRENCH(Format is: dd/mm/yy)GERMAN(Format is: dd.mm.yy)ITALIAN(Format is: dd-mm-yy)   |  |  |  |  |
| Usage:    | SET DATE affects all date input and display formats. This allows<br>you a way to control date formatting for different countries from a<br>central location in your program.   |  |  |  |  |
| Example:  | SET DATE ANSI<br>? DATE() && Result: 89.07.16<br>SET DATE BRITISH<br>? DATE() && Result: 16/07/89<br>SET DATE GERMAN<br>? DATE() && Result: 16.07.89<br>SET DATE FRENCH<br>? DATE() && Result: 16/07/89<br>SET DATE ITALIAN<br>? DATE() && Result: 16-07-89<br>SET DATE AMERICAN<br>? DATE() && Result: 07/16/89 |  |  |  |  |
| See Also: | SET CENTURY<br>CTOD()<br>DTOC()  |  |  |  |  |

## SET DECIMALS

Syntax: SET DECIMALS TO < expN >

**Purpose:** Sets the number of decimal places to display for numeric values.

**Argument:** < expN > is the number of decimal places to display from 1 to 15.

Usage: The exact operation of SET DECIMALS depends on the current setting of the SET FIXED command. If SET FIXED is off, then numeric values are displayed to the last non-zero decimal and the SET DECIMAL value has no effect. No trailing zeros are displayed after the decimal point. If SET FIXED is ON, then the number of decimal places specified by the SET DECIMALS TO are always displayed, even if they are zero.

Note: SET DECIMALS only affects the display. The internal calculations are always done to 15.9 significant digits.

**Examples:** 

SET DECIMALS TO 2 SET FIXED ON ? 2/4 && Result: 0.50 2 1/3 && Result: 0.33 SET DECIMALS TO 4 ? 2/4 && Result: 0.5000 2 1/3 && Result: 0.3333 SET FIXED OFF ? 2/4 && Result: 0.5 ? 1/3 && Result: 0.33333333333333333

See Also:

@ ... SAY ... GET PICTURE ?/?? SET FIXED TRANSFORM()

#### **SET DELETED**

Syntax: SET DELETED ON/OFF

**Purpose:** Turn on or off automatic filtering of records marked deleted.

Usage: When SET DELETED is ON, most commands ignore deleted records. However the GOTO command, and any command which uses the RECORD < scope > access deleted records even with SET DELETED ON.

When SET DELETED is OFF, all commands will access deleted records. A deleted record may be detected by using the DELETED() function.

See Also:

DELETE RECALL DELETED()

| SET | DELIMITI | ERS |
|-----|----------|-----|
|-----|----------|-----|

Syntax: SET DELIMITERS TO [<expC>/DEFAULT] SET DELIMITERS ON/OFF **Purpose:** Defines the characters to delimit GET fields. Arguments: ON/OFF: If SET DELIMITERS is ON, then each GET field has the specified delimiters displayed. If SET DELIMITERS is OFF, the no delimiters are displayed. TO: The SET DELIMITERS TO  $\langle \exp C \rangle$  defines the delimiter characters which are used to bound GET field displays. The expression must resolve to a two character string. The first character is the leading field delimiter, the second character is the trailing field delimiter. If no string is specified, or if the keyword DEFAULT is specified, then both delimiters are set to the default value of colon. **Example:** mvar = "field" SET DELIMITERS TO "[]" SET DELIMITERS ON @ 0,0 SAY "Enter " GET mvar READ **Result:** Enter [field] See Also: @ ... GET READ

## **SET DEVICE**

| Syntax:   | SET DEVICE TO SCREEN/PRINT   |  |  |  |
|-----------|--|--|--|--|
| Purpose:  | Directs the output of @ SAY to either screen or printer.   |  |  |  |
| Options:  | SCREEN: Specifying SCREEN a from the @ SAY command to  | as the DEVICE directs all output the screen. This is the default.                                  |  |  |
|           | the @ SAY command to the pr  | DEVICE directs all output from<br>rinter. The printer must first have<br>PRINTER TO command or the |  |  |
| Usage:    | When the DEVICE is set to PRINT, then @ SAY commands<br>are sent to the printer, and are not sent to the screen. When sending<br>@ SAY commands to the printer, the cursor position may only<br>be advanced. if it appears to go backwards, then a new page is<br>ejected from the printer and the output is placed on a new page. |  |  |  |
| Example:  | SET PRINTER TO LPT1<br>SET DEVICE TO PRINT<br>@ 4,20 SAY "Put on Prin<br>EJECT<br>SET PRINTER TO   | && Acquire printer<br>&& route to it<br>ter"<br>&& Form Feed<br>&& Release printer                 |  |  |
| See Also: | @ SAY<br>EJECT<br>SET PRINTER TO<br>PROW( ), PCOL( ), SETPRC( )  |  |  |  |

## SET DISCONNECT

| Syntax:   | SET DISCONNECT [MAXINST < expN>]<br>[MAXREPS < expN>]   |
|-----------|---|
| Purpose:  | Allows control of limits during an ON DISCONNECT procedure.   |
| Argument: | <b>MAXINST</b> - $\langle expN \rangle$ specifies the maximum number of TDBS instructions which may be executed by an ON DISCONNECT routine. If $\langle expN \rangle$ evaluates to 0, then an unlimited number of instructions may be executed after the user accidentally disconnects. In this case, an ON DISCONNECT procedure which has an error and loops will never release the line it is on. The default is 250 instructions. |
|           | MAXREPS - <expn> specifies the maximum number of record cycles (repetitions) of any single multi-record instruction. The default is 0 for no limit on a single multi-record instruction.</expn>   |
| Usage:    | This command allows controlling the maximum time an ON DIS-<br>CONNECT procedure may execute. It prevents a "runaway" ON<br>DISCONNECT procedure from locking out a line. SET DISCON-<br>NECT with no arguments restores the default settings.  |
|           | If an ON DISCONNECT procedure exceeds the specified limits,<br>then all files are closed and execution is terminated.   |
| Example:  | SET DISCONNECT MAXINST 200  |
|           | This limits an ON DISCONNECT procedure to 200 instructions.   |
| See Also: | ON DISCONNECT   |

# SET DISPLAY RULES

Syntax: SET DISPLAY RULES TO TDBS/STD1/STD2

**Purpose:** Allows display to conform to other dBASE dialect standards in unusual cases.

**Argument:** TDBS: This is the default, and invokes what we feel are the display rules you are most likely to expect in all conditions.

**STD1:** This setting forces TDBS to display unusual conditions as the program dBASE III + does in all cases except where dBASE III + exhibits outright bugs.

STD2: This setting forces TDBS to display unusual conditions as the program dBASE IV does in all cases except where dBASE IV exhibits outright bugs.

**Usage:** This command is present in case some program depends on the unusual "quirks" in the two dBASE program display routines. This is unlikely, and thus this command will probably not be necessary.

Most of the "quirks" which TDBS altered are probably actual programming bugs in the two dBASE products. However, this command is provided in case some programmer has learned about them and has come to depend on them.

### SET DIVIDE BY ZERO

Syntax:

SET DIVIDE BY ZERO TO ERROR/INFINITY

**Purpose:** Allows choice of how to handle a divide by zero condition.

Argument:

**INFINITY:** This is the default, and is how dBASE III + handles divide by zero conditions. No error is reported, and the result of the operation is set to the largest possible number.

**ERROR:** If you issue the SET DIVIDE BY ZERO TO ERROR then a divide by zero condition will generate an error. This prevents you from accidentally dividing by zero and believing the result is significant.

#### **SET EDITOR**

Syntax:

#### SET EDITOR [READONLY][NOMENU][NOISTAT] [NORFILE][NORPATH][NOWFILE][NOWPATH][IGRAVE]

**Purpose:** Allows the appearance and features of the built in memo editor to be customized to the program's needs.

## Argument: READONLY - The editor will only display memo records. The user cannot make any changes, and all edit controls are absent from the editor menu.

**NOMENU** - The menu box is NOT displayed, allowing all but the first line of the screen to be used by the memo editor for text.

NOISTAT - The status line is always displayed in "normal" mode, even when SET INTENSITY ON is in effect.

**NORFILE** - This disables the  $^KR$  command, and it does not appear in the edit menu. The user may not access other disk files. **NORPATH** - This limits the  $^KR$  command to only be able to read files which are in the HOMEPATH. The user cannot access files in other directories.

NOWFILE - The  $^{KW}$  command is disabled and is not in the editor menu. the user cannot output a memo to a normal disk file. NOWPATH - The  $^{KW}$  can only write text files to the HOMEPATH. The user cannot create files in other directories.

**IGRAVE** - Causes the memo editor to consider 0x8D as a displayable character and not a soft return (which is ignored).

**Usage:** This command allows control of the access which a user of the memo editor has to other files in the system by limiting editing options. If a SET EDITOR command is issued operation is restored to the dBASE compatible mode.

**Example:** SET EDITOR NORFILE NOWFILE

This prevents access to text files outside the program itself.

#### SET ESCAPE

Syntax:

SET ESCAPE ON/OFF

**Purpose:** 

Usage:

In all cases, pressing Esc during a READ command will end the READ without changing any file fields. During ACCEPT or INPUT the Esc key will be treated as data. An INKEY() will read the Esc key as data also, if it is left in the input data stream as explained below.

Allows control over the Esc key interrupt action.

In all other program conditions, the Esc key can either be specified to be treated as data, or to cause a program interrupt. If SET ESCAPE OFF is in effect, the Esc key is always treated as any other key and is left in the input data stream.

If SET ESCAPE ON is in effect, then the Esc key will cause an interrupt when it is placed in the typeahead buffer. This means that it can cause an escape interrupt even if there are unread characters ahead of it in the typeahead buffer. In this case, any typeahead data is removed from the typeahead buffer and discarded. The type of interrupt is determined by the ON ESCAPE command. If there is no ON ESCAPE, TDBS will ask if you want to abort. Answering No allows the program to continue, answering YES will abort after closing all files. If the ON ESCAPE is in effect, the the interrupt will activate the ON ESCAPE routine.

Note: SET TYPEAHEAD 0 will inhibit all escape interrupt processing no matter what the setting of the ON ESCAPE or the SET ESCAPE command.

See Also:

ON ESCAPE SET TYPEAHEAD

#### SET EXACT

|           |   | _ |
|-----------|---|---|
| Syntax:   | SET EXACT ON/OFF  |   |
| Purpose:  | Determines how two character strings are compared.  |   |
| Usage:    | When SET EXACT OFF is in effect (the default setting) two<br>character strings are compared as follows:   |   |
|           | There are two strings in any text comparison, the target string and the comparison string. In the equation $A = B$ , B is the target string and A is the comparison string. In a command such as SEEK the index key is the target string, and the SEEK argument is the comparison string. |   |
|           | If the target string is null, return true (.T.) for the comparison.   |   |
|           | If LEN(target) is greater than LEN(comparison) return false (.F.) for the comparison.   |   |
|           | Compare all characters in the target with the comparison string. If<br>all characters in the target match the leading characters of the<br>comparison string then return true (.T.) otherwise return false (.F.).   |   |
|           | Note: This means the target string may be shorter than the com-<br>parison string and still return a match.   |   |
|           | If SET EXACT IS ON, then the two strings must match exactly.  |   |
| Example:  | SET EXACT OFF<br>? "123" = "12345" && Result: .F.<br>? "12345" = "123" && Result: .T.<br>? "123" = "" && Result: .T.<br>? "" = "123" && Result: .F.<br>SET EXACT ON<br>? "12345" = "123" && Result: .F.<br>? "123" = "123" && Result: .T.   |   |
| See Also: | FIND<br>LOCATE<br>SEEK  |   |

### SET EXCLUSIVE

Syntax:

SET EXCLUSIVE ON/OFF

**Purpose:** Sets the default file mode with which USE will open files.

Usage: The default is SET EXCLUSIVE ON. In this case, USE will open all files for EXCLUSIVE use only, and no files may be shared.

SET EXCLUSIVE OFF changes this default to shared so that more than one user can USE the same files and share them. You may still override this on each individual USE command by specifying the EXCLUSIVE option for that file (or set of files).

Multiuser: TDBS provides Transparent File Sharing automatically when files are shared, so no explicit record or file locking is required. You may still use explicit locking where your application requires locking out a multiple access for longer than a single TDBS command line. See chapter 3 for a complete discussion of the TDBS file sharing features.

See Also:

USE RLOCK(), FLOCK() Chapter 3 on file sharing

### **SET FILTER**

| Syntax:   | SET FILTER TO [< condition >]   |
|-----------|---|
| Purpose:  | To make a database file appear as if it contains only the records<br>meeting a specified condition.   |
| Argument: | < condition > is a logical expression that identifies a specific set of records for the current work area.  |
|           | If not condition is specified, then any existing SET FILTER for the currently selected work area is deactivated.  |
| Usage:    | When a FILTER condition is SET, the database acts as if it only<br>contains records matching the specified condition. Each work area<br>can have its own separate SET FILTER condition.   |
|           | When a FILTER is SET, it is not activated until the next time the record pointer is moved. A filter has no effect on an INDEX or REINDEX command, and you may directly access filtered records by using the GOTO command or RECORD $< n >$ scope.   |
| Examples: | USE Customers<br>SET FILTER TO Zipcode = 80014<br>GO TOP<br>DO WHILE .NOT. EOF()<br>? CustName<br>ENDDO<br>SET FILTER TO  |
|           | This example lists that name of all customers in the 80014 ZIP code.<br>Note: If you set a filter that selects only a small number of records<br>from a very large database, the SET FILTER command has very<br>poor performance. You should read each record and filter them<br>in the program in such cases for good performance. |
| See Also: | SET DELETED   |

#### **SET FIXED**

Syntax: Purpose: **SET FIXED ON/OFF** 

Control the display of numeric output based on the current SET DECIMALS command setting.

**Options:** 

ON: When SET FIXED ON is in effect, all numeric output is displayed to the number of decimal places specified in by the SET DECIMALS TO command. The default is 2 places if no SET DECIMALS has been issued).

> OFF: When SET FIXED OFF is in effect, then numeric output is displayed to either the last significant digit, or up to but not including the first zero after the decimal point. No decimal point is displayed if the number is an integer. SET DECIMALS has no effect when SET FIXED OFF is in effect.

See Also:

SET DECIMALS

### **SET FORMAT**

Syntax: SET FORMAT TO [<procedure>] [NOCLEAR]

Purpose:Activates (or deactivates) a screen format procedure which will be<br/>performed by each READ command.

**Options:** < procedure > is a format (.FMT) file, a program file, or a procedure which is to become the format procedure for READs.

NOCLEAR: Normally the screen is cleared before execution of a format procedure READ command. This option suppresses this automatic screen clear to allow complete customization of the format by the procedure.

Note: SET FORMAT TO with no options will deactivate any current format procedure.

Usage: In TDBS format procedures may have any commands and are not just limited to screen format commands and the READ command. This extension allows format procedures to even call subprocedures if necessary. Multiple-page format files are supported by TDBS, except that the Page Up key will not "back up" to the previous screen in the format file. This action can be implemented under program control, however, since program structures are valid in TDBS format files.

A SET FORMAT TO command which occurs in a format file will be ignored.

A RETURN is not required at the end of a format file, but one may be specified anywhere you wish.

**Compiling:** Unless you use a .TDB compiler control file, a format file must be renamed to have a .prg extension. Otherwise the autocompile feature of the TDBS compiler will not find it.

See Also: @ ... SAY ... GET READ

### **SET FUNCTION**

| Syntax:   | SET FUNCTION <key> TO [<expc>]</expc></key>  |
|-----------|--|
| Purpose:  | To assign a character string to a function key.  |
| Argument: | <key> is the function key number (2-10).</key>   |
|           | $< \exp C >$ is the character string to assign to the function key.  |
| Usage:    | SET FUNCTION assigns a character string to a function key.<br>When that function key is pressed, this string is stuffed in the input<br>buffer. Any valid character may be in this string. |
|           | If the string ends in a semicolon, a carriage return is generated at the end of the string.  |
|           | If the string is empty or not specified, then the indicated function key is emptied of any previous string value.  |
|           | Strings of up to 254 characters may be loaded into function keys.  |
| Example:  | SET FUNCTION 3 TO "Name;"  |
|           | This will send the string Name followed by a carriage return to the program input when function key 2 is pressed.  |
| See Also: | FKLABEL()<br>FKMAX()   |

#### **SET INDEX**

| Syntax:   | SET INDEX TO < file list >  |
|-----------|---|
| Purpose:  | Opens specific index file(s) in the current work area.  |
| Argument: | <file list=""> is one or more index (.ndx) file names separated by commas. Each file name may be preceded by a &lt; path &gt; specification, and if it is not the file is assumed to be in the HOMEPATH directory. The first file in <file list=""> becomes the master index for the current work area.</file></file>   |
|           | SET INDEX TO without a < file list > closes any index files open<br>in the current work area.   |
| Usage:    | When more than one index is opened for a database, the first file<br>specified becomes the master (or controlling) index. The record<br>pointer is initially positioned to the first logical record in that index<br>file. During database processing ALL index files specified are<br>updated whenever their key value is appended or changed. To<br>change the master index file without issuing another SET INDEX<br>command, use the SET ORDER command. |
| Macros:   | Index files may be specified using macros, but each file name must<br>be in a separate macro variable. For example:<br>ndx1 = "Name"  |
|           | ndx2 = "CustNo"<br>SET INDEX TO &ndx1, &ndx2  |
| See Also: | USE<br>CLOSE<br>CLOSE INDEX<br>SET ORDER  |

#### SET INTENSITY

Syntax:

SET INTENSITY ON/OFF

**Purpose:** Sets the display of GET screen edit fields to take place in either enhanced of standard color settings.

Usage:

If SET INTENSITY is ON (the default condition) then all GET edit fields will be displayed using the enhanced color settings.

If SET INTENSITY is OFF, then all GET edit fields will be displayed using the standard color settings.

The color settings are controlled by the SET COLOR TO command.

This command allows all the areas on the screen which may be modified during a full screen edit to be highlighted, or optionally to be displayed with the same attributes as the rest of the screen characters.

See Also:

@ ... SAY ... GET READ SET COLOR

#### **SET MARGIN**

Syntax: SET MARGIN TO < expN> Purpose: Sets the left margin for all printer output. **Argument:**  $\langle \exp N \rangle$  specifies the column which is to become the left margin. **Usage:** SET MARGIN has no effect on the screen display. It only affects output sent to the printer. Note: PCOL() will reflect the absolute printer column position, it is therefore "biased" if any margin other than zero is set. The default margin is 0. Example: && Acquire LPT2 SET PRINTER TO LPT2 SET PRINTER ON && route ? to LPT2 && don't display ? SET CONSOLE OFF SET MARGIN TO 10 && set LPT margin ? "This is line 1" ? "This is line 2" EJECT && Send top-of-form && Release LPT2 SET PRINTER TO This example prints the two lines beginning in column 10 of the printer page. See Also: SET PRINTER TO SET PRINTER SET DEVICE SET CONSOLE ?/?? @ ... SAY

#### SET MEMOWIDTH

Syntax: SET MEMOWIDTH TO <expN>

? ??

6

**Purpose:** To set the column width of memo field screen output.

Usage: The default width for memo field displays is the user's TBBS width setting. The SET MEMOWIDTH command may be used to alter this width. SET MEMOWIDTH affects only memos display by the "?" or "??" commands.

**Example:** The following will display the memo field "text\_info" with a width of 40 columns:

SET MEMOWIDTH TO 40 ? Text info

See Also:

4-133

#### **SET ORDER**

| Syntax:<br>Purpose: | SET ORDER TO [ <expn>]<br/>Selects a new master (controlling) index file.</expn>  |
|---------------------|---|
| Argument:           | < expN > specifies the new master index by pointing to its position<br>in the list of open index files in the current work area.  |
|                     | SET ORDER TO 0 resets the database to its "natural" order (record<br>number order) by indicating that there is NO master or controlling<br>index. The index files are still remembered however, and are<br>properly updated if any key field is modified. They do not control<br>the logical record order however, and logical record order becomes<br>physical record order. |
|                     | The default is SET ORDER TO 1, which places the first named index file as the master (controlling) index.   |
| See Also:           | SET INDEX<br>USE  |

#### SET PRINT

Syntax:

**SET PRINT ON/OFF** 

**Purpose:** 

Controls output of ? and ?? command text to the printer.

Usage:

By default PRINT is OFF, and the output from the ? and ?? commands does not go to the printer. SET PRINT ON causes the output of these two commands to be directed to the printer. This does NOT stop the output of these commands from going to the screen as well. If you only want the output of the ? and ?? commands to go to the printer, you must also use the SET CONSOLE OFF command.

Note: You must have acquired a printer before issuing this command, or all printer output will be discarded.

Example:

See Also:

USE CustFile SET PRINTER TO LPT1 SET PRINT ON SET CONSOLE OFF DO WHILE .NOT. EOF() ? Customer SKIP ENDDO EJECT SET PRINT OFF SET CONSOLE ON SET PRINTER TO CLOSE DATABASES

&& request a printer && ?/?? to printer && ?/?? not to screen && Print Cust Names && next record && top of form LPT1 && return ?/?? normal && Return LPT1 && Close the file

EJECT SET CONSOLE SET PRINTER TO SET DEVICE

### **SET PRINTER TO**

Syntax: SET PRINTER TO [LPT1/LPT2/LPT3/LPT4]

**Purpose:** Requests and assigns this program a printer to use.

Arguments: LPTn: Specifies which one of a possible four printers which may be connected to a TDBS system is being requested.

SET PRINTER TO with no arguments will release any printer currently assigned to this program for use by others.

Usage: By default a TDBS program has no access to a printer. Any printed output goes to the NUL: device and is discarded. Before a TDBS program can output data to a printer it must first request that printer and successfully have it assigned.

- Multiuser:If the requested printer is in use by another program, then the SET<br/>PRINTER TO command will generate an error. This error may be<br/>fielded by an appropriate ON ERROR routine to retry the request.<br/>Optionally the WAIT4LPT(n) command may be used to wait for a<br/>specific printer to become available.
- See Also: @ ... SAY SET DEVICE SET PRINT SET CONSOLE WAIT4LPT(n)

#### SET PROCEDURE

Syntax:

Purpose:

SET PROCEDURE TO [<file>]

Compiles all procedures in the specified file so they may be accessed by the rest of the TDBS program.

## **Argument:** < file > is the name of a procedure file. The file has the extension of .PRG and contains TDBS source code statements.

**Usage:** A procedure file may contain any number of procedures. When a SET PROCEDURE TO command is encountered, TDBS compiles the procedure file into the current program.

In TDBS separate procedure files are not required, as multiple procedures may be placed in any program file including the main program. All procedure names must be unique.

For compatibility with interpretive dBASE language dialects, TDBS treats the CLOSE PROCEDURE and SET PROCEDURE with no arguments as no operation commands if encountered.

See Also:

DO PROCEDURE RETURN

#### **SET RELATION**

Syntax:

### SET RELATION TO [<key exp>/RECNO() INTO <alias>] [ADDITIVE]

Neither file has to be indexed for this relationship.

**Purpose:** Relates two work areas so they perform as a single database file.

Arguments: <key exp > is an expression which both files have in common, and which is indexed on the linked file (<alias>). RECNO() may be used instead of a common key expression, and establishes a one-toone relationship based on the logical record number of the files.

< alias > specifies the "child" work area where the file to be related to has been opened.

SET RELATION TO with no arguments dissolves any current relationship in effect.

**Option:** ADDITIVE: Adds this relation to any other current relations for the currently open work area. Otherwise all current relations are released before this new relation is set.

Usage: The SET RELATION command links the database in the current work area to one or more other open database(s). The current work area is one side of a one-to-many relationship. The common key expression must have the same type and size in both files. It doesn't need to be indexed in the current work area, but it must be the master index in the "child" work area.

The SET RELATION command operates as follows:

• When a record is positioned in the current work area, the value of any "related" key expression is used to do a SEEK on the "child" work area. This makes the "child" database file(s) always reflect the relationship which has been set automatically. If the value of the "related" key expression has no corresponding value in the "child" file then the record pointer in the "child" file is set to EOF() regardless of the current setting of SET SOFTSEEK. You may establish up to nine relations per work area. Note: You may not set up a circular relationship. That is, you cannot relate a database file either directly or indirectly to itself.

Multiuser: If a SET RELATION is in effect, then both files will be locked and unlocked simultaneously by the RLOCK() and FLOCK() functions. The TDBS Transparent File Sharing feature will also assure that all updates to linked files are properly interlocked automatically for the entire instruction duration to assure the integrity of both files in a relationship and all of their indexes. If an interlock is required across multiple REPLACE commands to assure integrity, you must use the RLOCK() function.

Example:

SELECT A USE Cust INDEX Custno SELECT B USE Invoices SET RELATION TO Custno INTO Cust DO WHILE .NOT. EOF() ? InvNum, Cust->CustName SKIP ENDDO CLOSE DATABASES

This example lists all invoice numbers and the customer name for each. The two databases are related by customer number.

See Also:

INDEX ON REPLACE SET INDEX SET ORDER USE RLOCK(), FLOCK(), WAIT4RLOCK(), WAIT4FLOCK()

### SET SOFTSEEK

Syntax: SET SOFTSEEK ON/OFF

**Purpose:** Sets relative or absolute SEEKing.

**Usage:** If SOFTSEEK is ON and a match for a SEEK (or FIND) is not found, the record pointer is set to the record with the next highest index key. If there is no record with a higher key, the file is positioned past the end and EOF() is set true (.T.).

If SOFTSEEK is OFF, and a match for a SEEK is not found, the record pointer is positioned to past the end of the file and EOF() is set true (.T.). This is the default setting and is dBASE compatible.

FOUND() operates the same regardless of the setting of SOFTSEEK and is set true (.T.) if a key match occurs, and set to false (.F.) if a match is not found.

Note: SET EXACT affects the string comparison if the index file has character keys.

See Also: SET EXACT SET RELATION SEEK FIND FOUND()

### **SET TYPEAHEAD**

| Syntax:   | SET TYPEAHEAD TO < expN >  |
|-----------|--|
| Purpose:  | Sets the size of the keyboard input buffer.  |
| Argument: | $<\exp N>$ determines the size of the typeahead buffer and is in the range 0 to 255.   |
| Usage:    | By default the typeahead buffer is set to 255.   |
|           | Note: SET TYPEAHEAD 0 will disable any ON ESCAPE or ON KEY interrupt routines. In addition, NEXTKEY and INKEY will always return 0 since there can be no characters waiting.   |
|           | SET TYPEAHEAD 0 will also disable the wait portion of the TDBS extended functions WAIT4RLOCK(), WAIT4FLOCK(), WAIT4MAIL(), and WAIT4LPT(n) since no typed ahead key can be detected to end the wait. Thus these functions will make a single attempt to acquire the resource they are asked to wait for, and if they fail will immediately return false (.F.) and will not wait. |
|           | If the size of the typeahead buffer is changed, any characters which were in the buffer at the time are discarded.   |
| See Also: | ACCEPT<br>INPUT<br>READ<br>ON ESCAPE<br>ON ERROR<br>SET ESCAPE<br>INKEY(), NEXTKEY(), LASTKEY(), READKEY()   |
|           |  |

### **SET UNIQUE**

| Syntax:   | SET UNIQUE ON/OFF   |
|-----------|---|
| Purpose:  | Determine whether index files are to include non-unique keys when creating a new index.   |
| Usage:    | The status flag set by this command is only used by the INDEX and<br>ZAP commands. These commands will set the specified index<br>file(s) to indicate if these indexes are to include duplicate keys or<br>not. This status becomes part of the index file itself, and as new<br>records are added, or record keys are updated, each index file is<br>handled according to its own status setting. This means that UNI-<br>QUE and non-UNIQUE index files may be mixed on the same<br>database file, and the current SETting of UNIQUE does not matter<br>when records are updated. |
|           | A UNIQUE index file will not contain index keys to more than one<br>record with each unique key. If duplicate keys exist in records, then<br>the remainder of the records with the same key value will not appear<br>to exist if this index is used to access the database.   |
| See Also: | FIND<br>SEEK<br>USE<br>SET INDEX<br>ZAP   |

### SET UPDATE BELL

Syntax:

Purpose:

Usage:

Determines alert condition for a shared file screen update collision.

SET UPDATE BELL ON/OFF/ROLLBACK

This command allows setting an option which will alert the user if a READ full screen edit is in progress on database field values and another user updates these same fields in the same record.

The TDBS *Transparent Screen Update and Rollback on Collision* feature allows multiple users to do full screen editing on a shared file. This feature will immediately post any changes made by another user to the record you are editing on your screen.

The SET UPDATE BELL command determines what alert (if any) will be given to you if another user changes one or more of the fields you are currently editing.

**OFF:** No alert will be given when another user changes a field which is part of the full screen edit you are doing. The changed values will be immediately displayed, and any of the changed fields you have edited will be rolled back to their new values as input by the other user.

ON: This setting will give an alert bell if any field which is currently part of a GET on your screen is updated by another user, even if you haven't changed it in this READ.

**ROLLBACK:** This setting (the default) will only give an alert bell if another user changes a field you have edited during this READ, but have not yet committed. Changes to other fields which are part of your GET list, but which you haven't changed during this edit will have the new value displayed on the screen, but no alert bell will sound.

See Also:

READ Transparent Screen Update in Chapter 2

#### SKIP

SKIP [<expN>] Syntax: Moves the record pointer relative to the current position for the Purpose: database file in the current work area. Argument: <expN> specifies the number of records to move the record pointer from the current position. A positive value moves the record pointer forward, a negative value moves it backwards. If  $\langle \exp N \rangle$  is omitted, the default is +1 (skip to next record). Usage: SKIPping backward beyond the beginning of the file moves the pointer to the first record and BOF() returns true (.T.). Another skip backwards when BOF is true will return an error. SKIPping forward past the end of file positions the record pointer to RECCOUNT() + 1 and EOF() returns true (.T.). Another skip forwards when EOF is true will return an error. If a master (controlling) index is in use, the skip takes place in the index itself, and the record pointer is set to the resulting database record. Example: **USE Filel** && Result: 1 ? RECNO() SKIP ? RECNO() && Result: 2 SKIP 28 && Result: 30 ? RECNO() SKIP -5 ? RECNO() && Result: 25 See Also: GOTO FIND SEEK

> LOCATE CONTINUE BOF(), EOF(), RECNO(), RECCOUNT()

#### STORE

| Syntax:    | STORE < exp > TO < memvar list ><br>or  |
|------------|---|
|            | <memvar> = <exp></exp></memvar>   |
| Purpose:   | Initializes and/or assigns a value to one or more memory variables.   |
| Arguments: | $\langle exp \rangle$ is an expression which results in a value of any data type.<br>This data type is assigned to the target memory variable(s).   |
|            | <memvar list=""> are the memory variables to initialize and assign<br/>the value and type to.</memvar>  |
| Usage:     | STORE (and its alternate form equate) both creates and assigns a value to memory variables. If the variable name already exists, then STORE assigns the value and type of the expression to the current instance of the variable. If the variable does not exist, STORE will create it first. The domain of all memory variables which STORE creates automatically is PRIVATE at the current program level. |
|            | STORE can only modify memory variables. Fields are modified by<br>using the REPLACE command. Either fields or memory variables<br>may be modified by the READ command.  |
|            | Note: Both Fields and Memory Variables may have the same name.<br>When there is a name conflict, the field will be accessed by expressions. If you want the memory variable instead, the conflict may be resolved using an $<$ alias> qualifier. Memory variables have an $<$ alias> of M-> which may be used as a field $<$ alias> normally is used to qualify accesses.                                   |
| Example:   | var1 = M->Name&& Use Memvar Valuevar2 = Name&& Use Field ValueSTORE 0 TO var1, var2, var3, var4   |
| See Also:  | CLEAR MEMORY<br>RELEASE<br>PRIVATE, PUBLIC<br>SAVE, RESTORE   |

### SUM

| Syntax:    | SUM [ <scope>] <expn list=""> TO <memvar list=""><br/>[FOR <condition>] [WHILE <condition>]</condition></condition></memvar></expn></scope>   |
|------------|---|
| Purpose:   | Sums a series of numeric expressions to memory variables for a range of records in the current work area.   |
| Arguments: | <expn list=""> is the list of numeric values to SUM for each record processed.</expn>   |
|            | <memvar list=""> is the list of variables which will receive the SUM values. This list must contain one variable for each expression.</memvar>  |
| Options:   | Scope: The $<$ scope $>$ is the portion of the database file to SUM.<br>The default $<$ scope $>$ is ALL.   |
|            | <b>Condition:</b> The FOR option filters records within the scope. Only records which match the FOR $<$ condition $>$ are summed. The WHILE condition limits the scope to the set of records beginning with the current record until the $<$ condition $>$ fails. |
| See Also:  | AVERAGE<br>TOTAL  |

### TEXT

| Syntax:    | TEXT<br><text be="" displayed="" to=""><br/>ENDTEXT</text>   |
|------------|--|
| Purpose:   | Displays a block of text.  |
| Arguments: | <text be="" displayed="" to=""> is a block of literal characters to be displayed.</text>   |
| Usage:     | The TEXT ENDTEXT command display follows the same routing rules as the ? command, and thus may be routed to the printer using the SET PRINT command.                           |
|            | Macros within the text block are expanded, however no word<br>wrapping will take place. The expanded lines will still have hard<br>returns at the end of each individual line. |
| See Also:  | ?/??<br>@ SAY  |

### TYPE

.

| Syntax:   | TYPE <file> [TO PRINT]</file>   |
|-----------|---|
| Purpose:  | Displays the contents of a text file to the screen or printer.  |
| Argument: | < file > is the text file to display. An optional < path > may be<br>added to locate the file on any drive or directory, the HOMEPATH<br>directory is assumed by default.   |
| Option:   | TO PRINT: This option routes the display to the currently selected<br>printer instead of the screen. Note: You must have used the SET<br>PRINTER TO command to assign a printer or all printed output<br>is sent to the NUL: device resulting in no output. |
| Usage:    | Output to the screen may be paused using $^S$ , with $^Q$ to restart.<br>Note that you may interrupt a TYPE listing with Esc between each line displayed.   |
| See Also: | COPY FILE<br>SET PRINTER TO   |

### UNLOCK

Syntax:UNLOCK [ALL]Purpose:Releases file or record locks set by the current user.Option:ALL: If this option is specified, then all locks in all work areas are released. If not specified, then only locks in the current work area (and any related work areas) are released.See Also:SET EXCLUSIVE USE ... EXCLUSIVE FLOCK(), WAIT4FLOCK() RLOCK(), WAIT4FLOCK() TDBS multiuser features - Chapter 3

#### USE

| Syntax:   | USE [ <file>] [INDEX <file list="">] [ALIAS <alias>]<br/>[EXCLUSIVE][READONLY]</alias></file></file>   |
|-----------|--|
| Purpose:  | Opens an existing database (.dbf) file, and any associated index files in the currently selected work area.  |
| Argument: | <file> is the name of the database file to open. An optional<br/><path> may be specified. By default the file is assumed to reside<br/>in the HOMEPATH directory.</path></file>  |
| Options:  | <b>INDEX:</b> This option may be used to specify the names of up to 7 index files to associate with the database file. The first file named will become the master (or controlling) index file and determines the logical record order.  |
|           | Alias: This option is used to give the work area an $< alias > name$ other than the file name. By default the $< alias > is$ the database file name.   |
|           | <b>EXCLUSIVE:</b> This option forces the file to be opened for the exclusive use of this program. By default the file will be opened either EXCLUSIVE or SHARED based on the current SET EXCLUSIVE TO setting. If the file is in use by another, an error is generated. An ON ERROR routine may be used to field and retry a USE with the EXCLUSIVE option.  |
|           | <b>READONLY:</b> Normally TDBS requires read/write access to all files<br>in a work area (.DBF, .DBT, .NDX) to correctly implement<br>transparent file sharing. The READONLY option allows access<br>to files which are restricted from write access (e.g. by LAN file<br>security or because they are on CD-ROM). Any attempt to alter a<br>field in a file opened READONLY will generate an error. |
|           | Note: TDBS expects you to be consistent in the use of the READONLY option! If you open a file that isn't restricted with READONLY and another program opens the same file without READONLY, TDBS will not correctly be able to protect file integrity. In some cases <b>this can cause file damage</b> . So never mix  |

the use of READONLY and non-READONLY USE on the same file!

**Usage:** When a database file is opened, the file is positioned to the first logical record in the file (record 1 if no index is specified).

USE with no arguments closes any open database and/or index files in the current work area.

See Also: CLOSE SELECT SET INDEX SET ORDER

### **USE MAILBOX**

| Syntax:    | USE [<.dbf file>] [ALIAS <alias>] MAILBOX [JOURNAL]</alias>   |
|------------|---|
| Purpose:   | Opens intraprogram communications channel which simulates a database file in the current work area.   |
| Arguments: | <.dbf file > is the name of a database file with a single record in it which will define the structure of the mailbox channel. A < path > may be used, but by default the file is assumed to reside in the HOMEPATH directory.  |
|            | MAILBOX: Defines this as a mailbox channel instead of a normal database file.   |
| Options:   | <b>ALIAS:</b> Allows the use of an $<$ alias $>$ name other than the database file name. By default the $<$ alias $>$ is the name of the file.  |
|            | <b>JOURNAL:</b> This option forces the mailbox channel image to be checkpointed to the database file every time any data is updated. This option has no use in TDBS 1.0, but is fully implemented.  |
| Usage:     | A mailbox channel appears in general to be the same as a one<br>record database file. However it is a very efficient intraprogram<br>communications channel. See TDBS Mailboxes in Chapter 3 for a<br>full discussion of how to use mailboxes for program to program<br>communications. |
| See Also:  | CLOSE<br>NEWMAIL( ), WAIT4MAIL( )   |

### WAIT

| ) | Syntax:   | WAIT [ <prompt>] [TO <memvarc>]</memvarc></prompt>   |                         |
|---|-----------|--|-------------------------|
|   | Purpose:  | Pauses program execution until a key is press  | ed.                     |
|   | Options:  | Prompt: The optional < prompt > is displayed<br>is paused. If no < prompt > is used, the defau<br>of "Press any key to continue" is displayed. |                         |
|   |           | <memvarc> If the TO <memvarc> optio<br/>which is pressed is returned to this variable.</memvarc></memvarc>                                     | n is specified, the key |
|   | Usage:    | WAIT returns the character entered to the variable.  | e specified memory      |
|   | Example:  | WAIT "Press a key to resume" TO  | Кеу                     |
| ) | See Also: | ACCEPT<br>INPUT<br>INKEY()   |                         |

|            | ZAP  |   |
|------------|--|---|
| Syntax:    | ZAP  | ( |
| Purpose:   | Removes all records from the database file in the current work area.   |   |
| Usage:     | ZAP does not mark all records deleted, it actually removes them<br>from the database file. Any currently associated index files also<br>have all records removed from them.              |   |
|            | Note: Any index files which are specified are also reset to zero records. The UNIQUE status of all index files is set to the current SET UNIQUE status.                                  |   |
| Multiuser: | No other user may be using the file when the ZAP is executed, or<br>an error will result. This error may be avoided if the file is opened<br>for EXCLUSIVE use before the ZAP is issued. |   |
| See Also:  | CLEAR<br>DELETE<br>SET UNIQUE<br>USE   | ( |

# FUNCTIONS

FUNCTIONS



## **Function Overview**

TDBS functions are internal operations which return a single character, numeric, or logical value. Functions are used in conjunction with TDBS commands to perform operations on either individual items of data, or to perform operations and status evaluation which return true or false answers as a logical data item. Functions may have arguments which are the input data item or items they operate upon. Arguments are enclosed in parentheses after the function name. Functions which have no arguments, still require the use of parentheses so that the function name may be recognized as not being a variable or field name.

A function may be used in place of any memory variable in an expression. Functions may be used as arguments to other functions as well, and this process may be nested up to 20 levels deep.

**Example:** 

chars = LTRIM(RTRIM(string))

This is an example of using a function as an argument for another function. RTRIM(string) produces a copy of string with the trailing (right hand) blanks removed. LTRIM takes this string as an argument and removes the leading (left hand) blanks from it. Thus the result of this expression is a character string which is a copy of "string" with all leading and trailing blanks removed. The memory variable chars is then assigned this string as its value.

TDBS provides all of the standard dBASE language functions. In addition many extended functions are provided. These functions allow access to many elements of the TDBS environment, and also ease many common programming tasks. Note: If you use any of these extended functions, your program will not be compatible with other dBASE language dialects.

## **Function Syntax**

The TBDS functions in this chapter are described with the same syntax as is used in Chapter 4 to describe commands. To identify the element types used in the syntax descriptions in this chapter, see pages 4-1 and 4-2 where these element types are explained.

## **Summary of TDBS Functions**

ABS(<expN>)

Returns the absolute value of  $\langle \exp N \rangle$ .

### ACOPY(<array1>,<array2>[,<expN1>[,<expN2> [,<expN3>]]])

Copies elements from < array1 > to < array2 >.

#### ADEL(<array>,<expN>)

Deletes element < expN> from < array>.

#### AFIELDS(<array1>[,<array2>[,<array3>[,<array4]]])

Fills a series of arrays with field definition information for the current work area and returns the number of fields.

#### $AFILL(\langle array \rangle, \langle exp \rangle [, \langle expN1 \rangle [, \langle expN2 \rangle ]])$

Fills all or part of  $\langle array \rangle$  with  $\langle exp \rangle$ .

#### AINS(<array>,<expN>)

Inserts new element into < array> at position < expN>.

#### ALIAS([<expN>])

Returns a string with the < alias> name of work area < expN>.

#### ASC(<expC>)

Returns ASCII numeric value of the first character of <expC>.

#### ASCAN(< array >, < exp > [, < expN1 > [, < expN2 > ]]

Searches  $\langle array \rangle$  for  $\langle exp \rangle$  starting with element  $\langle expN1 \rangle$ .

#### ASORT/ADSORT(<array>[,<expN1>[,<expN2>]]

Sorts < array> in ascending (or descending) order starting at element < expN1>.

#### AT(<expC1>,<expC2>)

Searches for string  $< \exp C1 >$  inside string  $< \exp C2 >$ .

#### BOF()

Returns .T. if beginning-of-file has been reached.

#### CAPFIRST(<expC>)

Returns a copy of  $\langle expC \rangle$  with the first letter forced upper case.

#### CDOW(<expD>)

Returns the day of the week or < expD> as a character string.

#### CEILING(<expN>)

Returns integer number equal to or next higher than <expN>.

#### CHR(<expN>)

Convert < expN> to ASCII character value.

#### CMONTH(<expD>)

Returns the name of the month of  $\langle \exp D \rangle$  as a character string.

#### COL()

Returns the number of the current screen column position.

#### CRTRIM(<expC>)

Removes any end-of-line sequence from a text string.

#### CTOD(<expC>)

Converts  $\langle expC \rangle$  text date to a date type.

#### DATE()

Returns today's date as a value in date type.

### DAY(<expD>)

Returns numeric value of the day of the month from  $\langle \exp D \rangle$ .

#### DBF()

Returns the name of the currently selected database file.

#### DEC2HEX(<expN>)

Returns string of < expN> converted to hexadecimal.

#### DELETED()

Returns .T. if the current record is marked for deletion.

## DESCEND()

Allows creation and SEEKing of descending order indexes.

## DISKSPACE()

Returns the number of bytes of free space available on the current work area's database file drive.

## DOTBBS()

Detects if the DOTBBS command is supported.

### DOW(<expD>)

Returns the number of the day of the week of  $\langle \exp D \rangle$ .

### DTOC(<expD>)

Returns a text string in date form for the date of <expD>.

### DTOS(<expD>)

Returns a string of the date < expD> in the form "yyyymmdd".

## EMPTY(<exp>)

Returns .T. if  $\langle \exp \rangle$  is blank.

## EOF()

Returns .T. if the current database file is past the end-of-file.

## ERROR()

Returns the number of the error which triggered an ON ERROR.

## EXP(<expN>)

Returns the value of the constant e raised to the power  $\langle expN \rangle$ .

## FBEXTRACT(<expN1>,<expN2>[,<expN3>])

Extract characters from a flat file I/O buffer into a character string.

## $\mathsf{FBFILL}(<\mathsf{expN1}>[,<\mathsf{expN2}>[,<\mathsf{expC}>[,<\mathsf{expN3}>]]])$

Fill a flat file I/O buffer with a pattern of characters.

## FBINSERT(<expN1>,<expN2>,<expC>[,<expN3>]

Insert data from a character string into a flat file I/O buffer.

```
FBMOVE(<expN1>,<expN2>,<expN3>[,<expN4>
[,<expN5>]]
```

Move data from one flat file I/O buffer to another.

#### FCOUNT()

Returns the number of fields in the current work area.

#### FDATE(<expC>)

Returns date of file name given in  $\langle \exp C \rangle$ .

#### FERROR([<expN>])

Returns error code from flat file I/O operation.

#### FIELD(<expN>)

Returns the text name of the field in the current database file which corresponds to the numeric position of  $\langle expN \rangle$ .

#### FILE(<expC>)

Returns .T. if the file name given in  $\langle \exp C \rangle$  exists.

## FINDFIRST(< memvar1>, < expC1>[, < expC2> [, < memvar2>]])

Returns name of first file matching skeleton in  $\langle \exp C \rangle$ .

#### FINDNEXT(<memvar1>[,<memvar2>])

Returns name of next file matching previous FINDFIRST.

#### FKLABEL(<expN>)

Returns the text name assigned to function key < expN>.

#### FKMAX()

Returns the maximum number of programmable function keys.

#### FLEN(<expN>)

Returns buffer length for an open flat file.

#### FLOCK()

Attempts to lock a file and returns a logical value based on the success or failure of the attempt.

FLOOR(<expN>)

Returns integer number equal to or next lower than  $\langle \exp N \rangle$ .

#### FMAXLEN()

Returns maximum flat file I/O buffer which can be allocated.

#### FOUND()

Returns a logical value based on the success or failure of the last SEEK, FIND, LOCATE, or CONTINUE command.

#### FSIZE(<expC>)

Returns size of file name given by  $\langle \exp C \rangle$ .

#### FTIME(<expC>)

Returns time stamp of file name given by  $\langle \exp C \rangle$ .

#### GETENV(<expC>)

Returns the contents of the DOS environment variable < expC>.

#### GETLPT(<expn>)

Requests access to LPT < expN >. Returns true (.T.) if granted.

#### HARDCR(<expC>)

Returns string with any 0x8D characters changed to 0x0D.

#### HEX2DEC(<expC>)

Returns number which is  $\langle expC \rangle$  converted from hexadecimal.

#### HOMEPATH()

Returns a string with the home path (from Opt Data).

#### IIF(<expL>,<exp true>,<exp false>)

Evaluates < expL>. If it evaluates as true (.T.) returns the value of < exp true >. If it is false (.F.) returns the value of < exp false >.

#### INDEXEXT()

Returns string with .NDX to indicate index file extension.

#### INDEXKEY(<expN>)

Returns the key expression of the specified index file in the current database work area.

#### INDEXORD()

Returns a numeric value of the controlling index within the current list of index files.

#### INKEY([<expN>])

Reads and returns numeric value of the next keycode in the typeahead buffer.

#### INT(<expN>)

Returns  $\langle expN \rangle$  with any fractional portion truncated.

#### ISALPHA(<expC>)

Returns .T. if the first character of  $\langle \exp C \rangle$  is alphabetic.

#### ISINT(<expN>)

Returns .T. if  $\langle expN \rangle$  is an integer value.

#### ISLASTDAY(<expD>)

Returns .T. if  $\langle expD \rangle$  is the lst day of the month.

#### ISLEAP(<expD>)

Returns .T. if  $\langle \exp D \rangle$  is a leap year.

#### ISLOWER(<expC>)

Returns .T. if the first character of  $\langle \exp C \rangle$  is lower case.

#### ISSHARE(<expC>)

Determine if a database or mailbox is currently being shared.

#### ISSTATE(<expC>)

Returns .T. if  $\langle expC \rangle$  is a valid post office state abbreviation.

#### ISUPPER(<expc>)

Returns .T. if the first character of  $\langle \exp C \rangle$  is upper case.

### LASTDAY(<expD>)

Returns date which is the last day of the month of  $\langle \exp D \rangle$ .

#### LASTKEY()

Returns the numeric ASCII value of the last key which has been read and processed by TDBS.

### LEFT(<expC>,<expN>)

Returns  $< \exp N >$  characters from the left of  $< \exp C >$ .

#### LEN(<expC>)

Returns the number of characters in  $\langle expC \rangle$ .

#### LJUST(<expC>)

Returns a copy of  $\langle \exp C \rangle$  which is left justified.

#### LOG(<expN>)

Returns natural logarithm of  $\langle \exp N \rangle$ .

#### LOWER(<expC>)

Returns  $\langle expC \rangle$  converted to lower case.

#### LTRIM(<expC>)

Returns  $\langle expC \rangle$  with any leading blanks removed.

#### LUPDATE()

Returns the last date the current database was modified.

#### MAX(<expN1>,<expN2>)

Returns the greater of  $\langle \exp N1 \rangle$  or  $\langle \exp N2 \rangle$ .

#### MESSAGE([<expN>])

Returns a character string with the text of the error which caused an ON ERROR condition. If  $\langle expN \rangle$  is given, returns the text for the specified error code.

#### MIN(<expN1>,<expN2>)

Returns the lesser of  $\langle \exp N1 \rangle$  or  $\langle \exp N2 \rangle$ .

#### MOD(<expN1>,<expN2>)

Returns  $\langle expN1 \rangle$  modulo  $\langle expN2 \rangle$ . For positive numbers this is the remainder of  $\langle expn1 \rangle$  divided by  $\langle expN2 \rangle$ .

#### MONTH(<expD>)

Returns the month of  $\langle \exp D \rangle$  as a numeric value.

#### NDX(<expN>)

Returns character string with the name of index file < expN>.

#### NEWMAIL()

Returns .T. if new mail received since last NEWMAIL() function.

#### **NEXTKEY()**

Returns the numeric ASCII code of the next key pending in the keyboard typeahead buffer.

#### **NMYUSERS()**

Returns the number of users of this TDBS program.

#### NUSERS()

Returns the number of users of any TDBS program.

#### **OPTDATA()**

Returns the TBBS menu Opt Data field as a string.

#### OS()

Returns the name of the operating system.

#### PCOL()

Returns current numeric column of printer.

#### **PROCLINE()**

Returns the current TDBS source code line number.

#### **PROCNAME()**

Returns the name of the current procedure as a character string.

#### PROW()

Returns current numeric row (line) of printer.

### RAT(<expC1>,<expC2>)

Returns starting position of the LAST occurrence of  $\langle expC1 \rangle$  within  $\langle expC2 \rangle$  as a numeric value.

#### **READKEY()**

Returns number indicating the key which ended the last full screen READ, and if any data was changed during the read.

#### RECCOUNT()/LASTREC()

Returns number of records in the current database file.

#### **RECNO()**

Returns current record number of the current database file.

#### **RECSIZE()**

Returns the record length of the current database file.

#### REPLICATE(<expC>,<expN>)

Returns a string made up of < expC> repeated < expN> times.

#### RIGHT(<expC>,<expN>)

Returns < expN> characters from the right of < expC>.

#### RJUST(<expC>)

Returns a copy of  $\langle \exp C \rangle$  which is right justified.

#### RLOCK()/LOCK()

Attempts to lock the current record and returns a logical value indicating the success or failure of the lock attempt.

#### ROUND(<expN1>,<expN2>)

Returns the value of <expN1> rounded to <expN2> decimals.

#### ROW()

Returns the number of the current screen row position.

#### RTRIM(<expC>) / TRIM(<expC>)

Returns a copy of  $\langle \exp C \rangle$  with any trailing blanks removed.

#### SECONDS()

Returns time of day as hundredths of seconds since midnight.

#### SELECT()

Returns the currently selected work area number.

#### SETPRC(<expN1>,<expN1>)

Set the internal PROW() and PCOL() values.

#### SOUNDEX(<expC>)

Returns SOUNDEX code for <expC>.

#### SPACE(<expN>)

Returns a character string of  $\langle \exp N \rangle$  spaces.

#### SQRT(<expN>)

Returns the square root of a positive  $\langle \exp N \rangle$ .

#### STATENAME(<expC>)

Returns a string which is the full state name for the post office state abbreviation in  $\langle \exp C \rangle$ .

#### STR(<expN1>[,<expN2>[,<expN3>]])

Returns a character string with the value of  $\langle expN1 \rangle$  converted to numbers. Optionally fixed at  $\langle expN2 \rangle$  characters in length and  $\langle expN3 \rangle$  decimal places.

#### STUFF(<expC1>,<expN1>,<expN2>,<expC2>)

Returns string which is a copy of  $\langle expC1 \rangle$  with  $\langle expN2 \rangle$  characters beginning at character  $\langle expN1 \rangle$  replaced with  $\langle expC2 \rangle$ .

#### SUBSTR(<expC>,<expN1>[,<expN2>])

Returns a string of  $\langle expN2 \rangle$  characters extracted from  $\langle expC \rangle$  starting at position  $\langle expN1 \rangle$ . The length may be limited to a maximum of  $\langle expN2 \rangle$  characters.

#### TIME()

Returns system time of day as a string of the form "hh:mm:ss".

#### TRANSFORM(<exp>,<expC>)

Returns a string which is  $\langle exp \rangle$  converted according to the format PICTURE specified by  $\langle expC \rangle$ .

#### TYPE(<expC>)

Returns the type of an expression, memory variable, or field.

#### UANSI()

Returns .T. if the user profile has ANSI = YES.

#### UAUTH(<expN>)

Returns string with the user's  $A(\langle expN \rangle)$  authorization flags.

#### UIBM()

Returns .T. if the user profile has IBM GRAPHICS = YES.

#### **ULOCATION()**

Returns a string with the user's location.

#### UMORE()

Returns the number of lines per page from the user profile.

#### UNAME()

Returns a string with the user's logon id.

#### **UNOTES()**

Returns a string with the user's userlog notes field.

#### ULINE()

Returns a single character string with the line identifier (0 - W).

#### ULREPLACE(<field>[,<expN>],<exp>)

Updates named fields in the user's TBBS userlog record.

#### ULPEEK(<offset>,<type>[,<length>])

Allows reading any field in the user's TBBS userlog record.

#### ULPOKE(<offset<,<type>[,<length>])

Allows updating any field in the user's TBBS userlog record.

#### UPDATED()

Determines if the last READ updated any fields.

#### UPPER(<expC>)

Returns a copy of  $\langle expC \rangle$  converted to upper case.

#### UPRIV()

Returns a number with the user's PRIVilege value.

#### USING([<expN>])

Determine which lines are currently using a shared work area.

#### UWIDTH()

Returns the number of characters per line from the user profile.

#### VAL(<expC>)

Returns a number which is  $\langle expC \rangle$  converted from text.

#### VERSION()

Returns current version number of TDBS as a string.

#### WAIT4FLOCK([<expN>])

Waits for File Lock, key press, or  $\langle expN \rangle$  seconds. Returns .T. if file lock was successful.

#### WAIT4LPT(<expN1>[,<expN2>])

Waits for access to LPT<expN1>, key press, or <expN2> seconds. Returns .T. if LPT<expN1> was acquired.

#### WAIT4MAIL([<expN>])

Waits for new mail, key press, or < expN> seconds. Returns .T. if new mail was received.

#### WAIT4RLOCK([<expN>])

Waits for record lock, key press, or < expN> seconds. Returns .T. if record lock was successful.

#### YEAR(<expD>)

Returns the year from  $\langle \exp D \rangle$  as a numeric value.

# ABS()

| Syntax:<br>Purpose: | ABS( <expn>)<br/>Finds the absolute value of <expn>.</expn></expn>   |  |
|---------------------|--|--|
| Argument:           | <expn> is any numeric expression.</expn>   |  |
| Returns:            | A numeric value.   |  |
|                     | Usage:   |  |
|                     | ABS() returns the absolute value of $\langle expN \rangle$ . This means that if $\langle expN \rangle$ evaluates to a positive number or zero, that number is returned. If $\langle expN \rangle$ evaluates to a negative number, then the sign is removed and the magnitude is returned as a positive number. |  |
| Examples:           | a = 12<br>b = 20<br>? ABS(a-b) && Result: 8<br>? ABS(b-a) && Result: 8<br>? ABS(0) && Result: 0<br>? ABS(-25) && Result: 25<br>? ABS(25) && Result: 25   |  |
| Compatibility:      | dBASE Standard, no extensions  |  |

# ACOPY()

| Syntax:        | ACOPY( <array1>,<array2>[,<expn1>[,<expn2><br/>[,<expn3>]]])</expn3></expn2></expn1></array2></array1>               |
|----------------|--|
| Purpose:       | Copy elements from one array to another.   |
| Arguments:     | <array1> is the source array.</array1>   |
|                | <array2> is the destination array.</array2>  |
|                | <expn1> is the starting element position in the source array.</expn1>  |
|                | $< \exp N2 >$ is the number of elements to copy from the source array beginning with $< \exp N1 >$ .                 |
|                | <expn3> is the starting element position in the target array.</expn3>  |
| Return:        | A logical .F.  |
| Example:       | <pre>PRIVATE one[5], two[5]<br/>I = 1<br/>DO WHILE I &lt;= 5<br/>one[I] = I<br/>ENDDO<br/>dummy=ACOPY(one,two)</pre> |
| See Also:      | ADEL(), AFIELDS(), AFILL(), AINS(), ASCAN(), ASORT()   |
| Compatibility: | TDBS extended, no dBASE equivalent, Clipper compatible.  |

# ADEL()

| Syntax:        | ADEL( <array>,<expn>)</expn></array>  |  |
|----------------|---|--|
| Purpose:       | Deletes an array element.   |  |
| Arguments:     | < array> is the name of the target array.   |  |
|                | $<\exp N>$ is the index of the element to delete.   |  |
| Returns:       | A logical .F.   |  |
| Usage:         | The contents of the specified array element are discarded and all<br>elements from that position to the end of the array are shifted up<br>one element. The last element in the array becomes undefined until<br>a new value is assigned to it. |  |
| Example:       | <pre>PRIVATE array[5] array[1] = 1 array[2] = 2 array[3] = 3 ? array[2] &amp;&amp; Result: 2 dummy=ADEL(array,2) ? array[2] &amp;&amp; Result: 3</pre>  |  |
| See Also:      | ACOPY(), AFIELDS(), AFILL(), AINS(), ASCAN(), ASORT()   |  |
| Compatibility: | TDBS extended, no dBASE equivalent, Clipper compatible.   |  |

# AFIELDS()

| Syntax:        | AFIELDS( <array1>[,<array2>[,<array3><br/>[,<array4>]]])</array4></array3></array2></array1>   |
|----------------|--|
| Purpose:       | Fills a series of arrays with field names, field types, field lengths, and field decimals.   |
| Arguments:     | <array1> is the array to fill with field names. Each element is character type.</array1>   |
|                | <array2> is the array to fill with the type of fields in <array1>.<br/>Each element is character type.</array1></array2>   |
|                | <array3> is the array to fill with widths of fields in <array1>.<br/>Each element is numeric type.</array1></array3>   |
|                | <array4> is the array to fill with the number of decimals defined<br/>for the fields in <array1>. Each element is numeric type. If the<br/>corresponding field is not numeric, the element is set to zero.</array1></array4>             |
| Returns:       | An integer numeric value.  |
|                | AFIELDS() returns the number of fields or the length of the shortest array, whichever is less.   |
| Usage:         | AFIELDS() fills a series of arrays with the information normally<br>placed in a STRUCTURE EXTENDED file. This information is<br>taken from the current work area. If there is no database currently<br>in use, AFIELDS() returns a zero. |
| Example:       | To select some attributes while skipping others pass a dummy variable. For example to obtain field names and lengths only:   |
|                | <pre>PRIVATE fname[FCOUNT()], flen[FCOUNT()] dummy = "" nfld = AFIELDS(fname, dummy, flen)</pre>   |
| Compatibility: | TDBS extended, no dBASE equivalent, Clipper compatible.  |

# AFILL()

| Syntax:        | AFILL( <array>,<exp>[,<expn1>[,<expn2>]])</expn2></expn1></exp></array>  |
|----------------|--|
| Purpose:       | Fills an array with a specified value.   |
| Arguments:     | <array> is the array to fill.</array>  |
|                | $\langle exp \rangle$ is the value to place in each array element. It may be an expression of any data type.   |
|                | $<\exp N1>$ is the position of the first element to fill. If this argument is omitted, filling begins with element 1.  |
|                | <expn2> is the number of elements to fill. This argument is optional, and if it is omitted all elements from the starting point to the end of the array are filled.</expn2>                                  |
| Returns:       | A logical .F.  |
| Usage:         | <exp> is evaluated only once, at the beginning of the instruction.<br/>Thus it is not possible to use AFILL to place different (e.g. in-<br/>crementing or decrementing) values in different elements.</exp> |
| Example:       | PRIVATE alogic[15]<br>dummy = AFILL(alogic, .T.)<br>dummy = AFILL(alogic, .F., 5, 10)  |
|                | Elements 1 through 4 are filled with .T. and elements 5 through 15 are filled with .F. in this example.  |
| See Also:      | ACOPY(), ADEL(), AFIELDS(), AINS(), ASCAN(), ASORT()   |
| Compatibility: | TDBS extended, no dBASE equivalent, Clipper compatible.  |

# AINS()

| Syntax:<br>Purpose: | AINS( <array>,<expn>)<br/>Inserts an undefined element into an array.</expn></array>  |  |
|---------------------|---|--|
| Arguments:          | < array > is the array into which a new element is inserted.  |  |
|                     | <expn> is the position to insert the new element.</expn>  |  |
| Returns:            | A logical .F.   |  |
| Usage:              | The newly inserted position remains undefined until a new value is<br>assigned to it. After the insertion, the last element is discarded and<br>all elements after the insertion point are shifted up one position in<br>the array. |  |
| Example:            | <pre>DECLARE array[3]<br/>array[1] = 1<br/>array[2] = 2<br/>array[3] = 3<br/>? array[2]</pre>   |  |
| See Also:           | ACOPY(), ADEL(), AFIELDS(), AFILL(),<br>ASCAN(), ASORT()  |  |
| Compatibility:      | TDBS extended, no dBASE equivalent, Clipper compatible.   |  |

+

# ALIAS()

| Syntax:        | ALIAS([ <expn>])</expn>   |   |
|----------------|---|---|
| Purpose:       | Obtains the < alias > nam   | e for a work area.  |
| Argument:      | < expN $>$ evaluates to the number (1 - 10) of the desired work area. |   |
| Returns:       | A character string.   |   |
| Usage:         | the specified work area has   | of the alias of the specified work area. If<br>s no open database file, then a null string<br>rea number is given, then the currently<br>med. |
| Example:       | (/  | Orders<br>&& Result: ORDERS<br>&& Result: CUST  |
| See Also:      | SELECT, USE, SELECT   | ()  |
| Compatibility: | TDBS Extended, no dBASE equivalent                                    |   |

# ASCAN()

| Syntax:<br>Purpose: | ASCAN( <array>,<exp>[,<expn1>[,<expn2>]])<br/>Scans an array for a specific value.</expn2></expn1></exp></array>  |  |
|---------------------|---|--|
| Arguments:          | < array > is the array to scan.   |  |
|                     | $< \exp >$ is the expression to scan for and may be any data type.  |  |
|                     | $< \exp N1 >$ is the first element to scan. If it is omitted, scanning begins with element 1.   |  |
|                     | $<\exp N2>$ is the number of elements to scan. If it is omitted, all elements from the starting element to the end of the array are scanned.  |  |
| Returns:            | An integer numeric value.   |  |
| Usage:              | ASCAN() searches the specified array and returns the numeric element position of the first matching array element. If no element matches <exp> then ASCAN() returns a zero.</exp>                                 |  |
|                     | Note that ASCAN() is sensitive to the setting of EXACT. If SET EXACT is on, then the element must match the result of $\langle \exp \rangle$ character for character if $\langle \exp \rangle$ is character type. |  |
| See Also:           | ACOPY(), ADEL(), AFIELDS(), AFILL(), AINS(), ASORT()  |  |
| Compatibility:      | TDBS extended, no dBASE equivalent, Clipper compatible.   |  |
|                     | Purpose:<br>Arguments:<br>Returns:<br>Usage:<br>See Also:   |  |

-

# ASORT()/ADSORT()

| Syntax:        | ASORT/ADSORT( <array>[,<expn1>[,<expn2>]])</expn2></expn1></array>  |  |
|----------------|---|--|
| Purpose:       | Sorts the contents of an array in either ascending or descending order.   |  |
| Arguments:     | < array> is the array to sort.  |  |
|                | $< \exp N1 >$ is the first element to sort. If it is omitted, sorting begins with element 1.  |  |
|                | $< \exp N2 >$ is the number of elements to sort. If it is omitted, then<br>all elements from the starting element to the end of the array are<br>sorted.  |  |
| Returns:       | A logical .F.   |  |
| Usage:         | All elements in the range of the array being sorted must be the same<br>data type. If they are different data types, they are first sorted by<br>data type (C, D, N, L) and then sorted within each category. Note<br>that character string sorts are NOT affected by the setting of SET<br>EXACT but are always character for character with shorter but<br>equal strings sorting first. |  |
| Example:       | DECLARE array[3]<br>array[1] = "AA"<br>array[2] = "CC"<br>array[3] = "BB"<br>dummy = ASORT(array)<br>? array[1] && Result: "AA"<br>? array[2] && Result: "BB"<br>? array[3] && Result: "CC"   |  |
| See Also:      | ACOPY(), ADEL(), AFIELDS(), AFILL(), AINS(), ASCAN()  |  |
| Compatibility: | TDBS extended, no dBASE equivalent, Clipper compatible.   |  |

# ASC()

| Syntax:        | ASC( <expc>)</expc>   |  |
|----------------|---|--|
| Purpose:       | Convert character type to nume  | ric ASCII equivalent.  |
| Argument:      | <expc> is the character expre</expc>  | ession to convert.   |
| Returns:       | A numeric value in the range 0  | to 255.  |
| Usage:         | The first character of the string is returned as its numeric ASCII value. If $\langle expC \rangle$ is longer than one character, ASC ignores all but the first character. You may use ASC where you need to do calculations on the ASCII value of a character. |  |
| Example:       | <pre>? ASC("A") ? ASC("Anchor") ? ASC("*") ? ASC("Z")-ASC("A") ? ASC("")</pre>  | && Result: 65<br>&& Result: 65<br>&& Result: 42<br>&& Result: 25<br>&& Result: 0 |
| See Also:      | CHR(), INKEY(), NEXTKEY(), LASTKEY()  |  |
| Compatibility: | dBASE standard, no extensions.  |  |

.

# AT()

| Syntax:        | AT( <expc1>,<expc2>)</expc2></expc1>    |   |  |
|----------------|---|---|--|
| Purpose:       | Locates the first instance of one       | e character string in another.  |  |
| Arguments:     | <expc1> is the string to search</expc1> | ch for.   |  |
|                | <expc2> is the string to be se</expc2>  | earched.  |  |
| Returns:       | A numeric value.                        |   |  |
|                | <b>U</b>                                | ed in string <expc2>, the AT()<br/>character position of the substring.<br/>zero is returned.</expc2> |  |
|                |   | f one string is contained in another,<br>ise AT() when you need to know                               |  |
| Example:       | FNames = "JoeLarryStev                  | veBobFrank"   |  |
|                | ? AT("Bob", FNames)                     | && Result: 14   |  |
|                | ? AT("a","abcde")                       | && Result: 1  |  |
|                | ? AT("A", "abcde")                      | && Result: 0  |  |
| See Also:      | SUBSTR()                                |   |  |
| Compatibility: | dBASE standard, no extension            | S   |  |

|                | BOF()   |  |
|----------------|---|--|
| Syntax:        | BOF()   |  |
| Purpose:       | Determine if a skip past the beginning of file was attempted.   |  |
| Returns:       | A logical value.  |  |
| Usage:         | <ul> <li>BOF() returns a .T. only when you attempt to move the record pointer before the first logical record in the current database file. When this happens, the record pointer is positioned at the first logical record in the file. If the current database contains no records, then both BOF() and EOF() are true.</li> <li>An attempt to skip backwards again after BOF() reports true will cause a run time error. SKIP is the only command which can set BOF() true.</li> </ul> |  |
| Example:       | USE Customer<br>GO TOP<br>? RECNO() && Result: 1<br>? BOF() && Result: .F.<br>SKIP -1<br>? RECNO() && Result: 1<br>? BOF() && Result: .T.   |  |
| See Also:      | SKIP, EOF()   |  |
| Compatibility: | dBASE standard, no extensions.  |  |

## **Chapter 5: TDBS Functions**

e .

# CAPFIRST()

| Syntax:<br>Purpose: | CAPFIRST( <expc>)<br/>Capitalize the first character of a string.</expc>   |  |
|---------------------|--|--|
| Argument:           | <expc> is the string to capitalize.</expc>   |  |
| Returns:            | A character string.  |  |
| Usage:              | CAPFIRST will return a copy of $\langle expC \rangle$ with the first character forced to upper case. The remainder of the string is returned as it was in $\langle expC \rangle$ . |  |
| Example:            | <pre>var = "JOHN" ? CAPFIRST(LOWER(var)) &amp;&amp; Result: John ? CAPFIRST("abcde") &amp;&amp; Result: Abcde ? CAPFIRST("Billy ray") &amp;&amp; Result: Billy ray</pre>           |  |
| See Also:           | UPPER(), LOWER()   |  |
| Compatibility:      | TDBS extended, no dBASE equivalent   |  |

# CDOW()

| Syntax:<br>Purpose: | <b>CDOW(<expd>)</expd></b><br>Return the day of the week in text for a given date.  |  |  |
|---------------------|---|--|--|
| Argument:           | <expd> is the date for which to</expd>  | find the day of the week.                |  |
| Returns:            | A character string.   |  |  |
| Usage:              | CDOW() returns the name of the day of the week with the first<br>letter in upper case, and the rest of the string in lower case. The<br>maximum returned string length is 9 characters for "Wednesday". |  |  |
| Example:            | <pre>? Date() ? CDOW(DATE())</pre>  | && Result: 07/17/89<br>&& Result: Monday |  |
| See Also:           | CMONTH(), CTOD(), DATE(), DAY(), DOW(), DTOC,<br>DTOS(), MONTH(), YEAR()  |  |  |
| Compatibility:      | dBASE standard, no extensions.  |  |  |

# CEILING()

2

| Syntax:        | CEILING( <expn>)</expn>  |   |  |  |
|----------------|--|---|--|--|
| Purpose:       | Finds the integer which is equal to  | or higher in value for $< \exp N >$ .   |  |  |
| Argument:      | <expn> is the numeric expression to find the CEILING of.</expn>  |   |  |  |
| Returns:       | An integer numeric value.  |   |  |  |
| Usage:         | Note that the CEILING of a num<br>as magnitude. The CEILING of a<br>integer, is the next highest intege<br>negative number which is not an<br>number which is the next smaller | a positive number which is not an<br>er number. The CEILING of a<br>integer is the next larger valued |  |  |
| Examples:      | <pre>? CEILING(1.5)<br/>? CEILING(-1.5)<br/>? CEILING(1.0)<br/>? CEILING(-1.0)<br/>? CEILING(0.333)<br/>? CEILING(-0.333)</pre>  | && Result: 2<br>&& Result: -1<br>&& Result: 1<br>&& Result: -1<br>&& Result: 1<br>&& Result: 0        |  |  |
| See Also:      | FLOOR(), INT()   |   |  |  |
| Compatibility: | TDBS extended, no dBASE equiv  | valent.   |  |  |

# CHR()

| Syntax:        | CHR( <expn>)</expn>  |  |  |  |
|----------------|--|--|--|--|
| Purpose:       | Converts an ASCII numeric number to a character.   |  |  |  |
| Argument:      | <expn> is the numeric value to convert to a character.</expn>  |  |  |  |
| Returns:       | A character value.   |  |  |  |
| Usage:         | CHR() returns the character corresponding to the ASCII value of $\langle expN \rangle$ . If $\langle expN \rangle$ is not in the range 1 to 255, a null string is returned.                    |  |  |  |
|                | CHR() allows embedding non-printable characters in strings.<br>These may be used to explicitly output ANSI screen control sequen-<br>ces, printer control sequences, ring the user's bell etc. |  |  |  |
| Example:       | <pre>? CHR(72) &amp;&amp; Result: H<br/>? CHR(61) &amp;&amp; Result: =<br/>? REPLICATE(CHR(61),10) &amp;&amp; Result: ====================================</pre>                               |  |  |  |
| See Also:      | ASC(), INKEY()   |  |  |  |
| Compatibility: | dBASE Standard, no extensions.   |  |  |  |

.

# CMONTH()

| >)  |  |  |  |
|---|--|--|--|
| ,   |  |  |  |
| text month from the da  | te < expD>.  |  |  |
| $<\exp D>$ is the date from which to extract the month.   |  |  |  |
| A character string.   |  |  |  |
| CMONTH() returns the name of the month from a date value. The first letter is upper case, the rest are lower case. The maximum length of the returned string is 9 characters for "September". |  |  |  |
| )) &&   | Result: July   |  |  |
| )+45) &&  | Result: August   |  |  |
| H(DATE(),1,3) &&  | Result: Jul  |  |  |
| ), DATE( ), DAY( ), I   | DOW( ), DTOC( ),   |  |  |
| (), YEAR()  |  |  |  |
| o extensions.   |  |  |  |
|   | ns the name of the montl<br>case, the rest are lowe<br>ed string is 9 characters |  |  |

|                | COL()   |
|----------------|---|
| Syntax:        | COL()   |
| Purpose:       | Returns current screen column position.   |
| Returns:       | An integer numeric value.   |
| Usage:         | COL() is used when you want to position the cursor relative to the current position. With COL(), and its companion function ROW() you may write position independent screen displays. |
| Example:       | var = "This is"<br>@ 10,15 SAY var<br>@ 10,COL()+1 SAY "a string"   |
|                | <b>Result:</b> This is a string   |
| See Also:      | @ SAY GET, @ TO, PCOL( ), PROW( ), ROW( )   |
| Compatibility: | dBASE standard, no extensions.  |

8

.

# CRTRIM()

| Syntax:        | CRTRIM( <expc>)</expc>  |
|----------------|---|
| Purpose:       | Removes end-of-line sequence from a character string.   |
| Argument:      | <expc> is a character string which has usually been read by the FLREAD flat file I/O function. CRTRIM removes the end-of-line sequence from the string.</expc>  |
| Returns:       | A character string.   |
| Usage:         | The CRTRIM() function provides a quick way to remove the<br>end-of-line sequence from a line read by the FLREAD command.<br>There are four different possible end-of-line sequences and<br>CRTRIM() will detect and remove any of them. Note: If the<br>end-of-line sequence is encountered prior to the end of the string,<br>any data following the end-of-line sequence is also removed. |
| Example:       | FLREAD Handle Size Record<br>DispRec = CRTRIM(Record) && Remove EOL<br>? DispRec  |
|                | This example reads a line from a file and displays it after removing the end-of-line sequence.  |
| Compatibility: | TDBS extended, no dBASE equivalent.   |

| CTOD( | ) |
|-------|---|
|-------|---|

| Syntax:        | CTOD( <expc>)</expc>  |   |  |
|----------------|---|---|--|
| Purpose:       | Converts a date character string  | to a date value.  |  |
| Argument:      | <expc> is a character string consisting of numbers representing<br/>the day, month, and year separated by a delimiter character. The<br/>order of the input string is determined by the SET DATE format.<br/>The default is AMERICAN format: "mm/dd/yy".</expc> |   |  |
|                | <b>Century:</b> The twentieth century specified for the year.   | is the default if only two digits are                               |  |
|                | <b>Empty date:</b> To specify a null date or a string of " / / ".   | te use either a string of all spaces                                |  |
| Returns:       | A date value.   |   |  |
| Usage:         | CTOD() is the only way to initial date. It may be used anywhere a   | ize a date variable with a constant<br>date type value is required. |  |
| Example:       | dvar = CTOD("07/17/89")   | )   |  |
|                | ? TYPE("dvar")  | && Result: D  |  |
|                | ? dvar<br>SET DATE ANSI   | && Result: 07/17/89   |  |
|                | ? dvar  | && Result: 89.07.17   |  |
| See Also:      | SET DATE, CDOW(), CMON<br>DOW(), DTOC(), DTOS(), M  |   |  |
| Compatibility: | dBASE standard, no extensions.  |   |  |
|                |   |   |  |

|                | DATE()  |  |
|----------------|---|--|
| Syntax:        | DATE()  |  |
| Purpose:       | Returns the current date as a date value.   |  |
| Returns:       | A date value.   |  |
| Usage:         | DATE() returns the current system date as a date value. Date<br>values are an internal format, and always the same. When a date is<br>converted from a date type value to text for display, the format of<br>the converted or displayed text is specified by the SET DATE<br>command.<br>You may perform date arithmetic only with a date variable. |  |
| Example:       | <pre>? DATE() &amp;&amp; Result: 07/17/89<br/>? DATE()+30 &amp;&amp; Result: 08/16/89<br/>vdate = DATE()<br/>? CMONTH(vdate) &amp;&amp; Result: July<br/>SET CENTURY ON<br/>SET DATE ANSI<br/>? vdate &amp;&amp; Result: 1989.07.17</pre>   |  |
| See Also:      | SET CENTURY, SET DATE, CDOW(), CMONTH(),<br>CTOD(), DAY(), DOW(), DTOC(), DTOS(), MONTH()<br>YEAR()   |  |
| Compatibility: | dBASE standard, no extensions.  |  |

## DAY()

| Syntax:        | DAY( <expd>)</expd>   |  |                            |                  |
|----------------|---|--|----------------------------|------------------|
| Purpose:       | Extracts the day of the month from  | mac  | date value.                |                  |
| Argument:      | <expd> is the date from which</expd>  | $<\exp D>$ is the date from which to extract the day of the month. |                            |                  |
| Returns:       | An integer numeric value.   |  |                            |                  |
| Usage:         | DAY() returns a number in the r<br>month of <expn>. If the mo<br/>accounted for and the 29th is pro<br/>null or empty date, then a zero is</expn> | nth<br>perly   | is February<br>y returned. | , leap years are |
| Example:       | <pre>? DATE() ? DAY(DATE()) ? DAY(DATE())+1 ? DAY(CTOD("12/25/88"))</pre>   | & &<br>& &   | Result:<br>Result:         | 18               |
| See Also:      | CDOW(), CMONTH(), CTOD<br>DTOC(), DTOS(), MONTH(),  |  |                            | OW( ),           |
| Compatibility: | dBASE standard, no extensions.  |  |                            |                  |

-

|                |   | DBF()  |  |
|----------------|---|--|--|
| Syntax:        | DBF()   |  |  |
| Purpose:       | Returns the drive and nar   | ne of the current database file.   |  |
| Returns:       | A character string.   |  |  |
| Usage:         | name of the current datal<br>directory information, just              | ter string which contains the drive and<br>base file. This string will not contain the<br>at the drive and file name. If there is no<br>urrent work area, a null string is returned. |  |
| Example:       | USE Customer  |  |  |
|                | ? DBF()<br>Close databases  | && Result: C:CUSTOMER  |  |
|                | ? DBF()   | && Result:   |  |
| See Also:      | ALIAS(), FIELD(), SELECT(), NDX(), LUPDATE()<br>RECCOUNT(), RECSIZE() |  |  |
| Compatibility: | dBASE standard, no exte   | nsions.  |  |

#### DEC2HEX()

| Syntax:        | DEC2HEX( <expn>)</expn>   |            |  |         |
|----------------|---|------------|--|---------|
| Purpose:       | Converts number to hexadecimal representation in text string.   |            |  |         |
| Argument:      | <expn> is the number to be converted to hexadecimal.</expn>   |            |  |         |
| Returns:       | A character string.   |            |  |         |
| Usage:         | DEC2HEX will convert a number to a hexadecimal representation<br>in a character string. If $\langle expN \rangle$ results in a number with a<br>fractional portion, it will be truncated before conversion. That is<br>an equivalent of INT( $\langle expN \rangle$ ) will occur. |            |  |         |
| Example:       | <pre>var = 43981 ? DEC2HEX(var) ? DEC2HEX(16) hvar = DEC2HEX(var) ? TYPE("hvar") ? hvar</pre>   | & &<br>& & | Result:<br>Result:<br>Result:<br>Result: | 10<br>C |
| See Also:      | HEX2DEC()<br>TDBS extended, no dBASE equivalent.  |            |  |         |
| Compatibility: |   |            |  |         |

## DELETED()

| Syntax:        | DELETED()   |  |  |
|----------------|---|--|--|
| Purpose:       | Determines if the current record is marked for deletion.  |  |  |
| Returns:       | A logical value.  |  |  |
| Usage:         | DELETED() returns a logical true (.T.) if the current record in<br>the current work area is marked for deletion; otherwise false (.F.)<br>is returned. If no database file is open in the current work area, a<br>false (.F.) is also returned. |  |  |
| Example:       | USE Customer<br>? RECNO() && Result: 1<br>? DELETED() && Result: .F.<br>DELETE<br>? DELETED() && Result: .T.<br>RECALL<br>? DELETED() && Result: .F.  |  |  |
| See Also:      | DELETE, RECALL, SET DELETED   |  |  |
| Compatibility: | dBASE standard, no extensions.  |  |  |

.

# DESCEND()

| Syntax:   | DESCEND( <exp>)</exp>  |  |
|-----------|--|--|
| Purpose:  | To create and SEEK descending order indexes.   |  |
| Argument: | $< \exp >$ is an expression of any data type.  |  |
| Usage:    | DESCEND() is designed to be used in combination with INDEX<br>and SEEK to allow for the creation of descending order indexes.<br>It returns the arithmetic complement of the input expression. |  |
| Examples: | To use DESCEND() in an INDEX expression, use the following syntax:   |  |
|           | INDEX ON DESCEND(Sales_date) TO date_dwn   |  |
|           | To SEEK on the descending index, use the following syntax:   |  |
|           | SEEK DESCEND(find_date)  |  |
| See Also: | INDEX  |  |
|           | SEEK   |  |

## DISKSPACE()

| Syntax:<br>Purpose: | DISKSPACE([ <expn>/<expc>])<br/>Determine number of bytes of available space on a disk drive.</expc></expn>   |  |
|---------------------|---|--|
| Options:            | $< \exp N >$ is a number indicating the drive, where $1 = A$ , $2 = B$ etc.   |  |
|                     | <expc> is a string where the first letter indicates the drive.</expc>   |  |
| Returns:            | An integer numeric value.   |  |
| Usage:              | DISKSPACE() returns the number of bytes of empty space on the specified disk drive. If no drive is specified, then the space available on the HOMEPATH drive is returned. |  |
| Example:            | FileSize = 25000<br>IF DISKSPACE() < FileSize<br>? "Not enough disk space available"<br>ENDIF   |  |
| See Also:           | HOMEPATH()  |  |
| Compatibility:      | dBASE standard plus extensions.   |  |

## DOTBBS()

| ) | Syntax:        | DOTBBS()   |
|---|----------------|--|
|   | Purpose:       | Detects whether the underlying TBBS will support the DOTBBS command.   |
|   | Returns:       | A logical value.   |
|   | Usage:         | The DOTBBS command is not supported by all versions of TBBS which will run TDBS 1.2. This function will return .T. if the DOTBBS command can be used, and .F. if it cannot be. |
|   | Compatibility: | TDBS extended, no dBASE equivalent.  |

.

# DOW()

| Syntax:        | DOW( <expd>)</expd>  |   |  |
|----------------|--|---|--|
| Purpose:       | To extract the day of the week from a date value.  |   |  |
| Argument:      | $<\exp D>$ is the date for which to find the day of the week.  |   |  |
| Returns:       | An integer numeric value in the range 0 to 7.  |   |  |
| Usage:         | DOW() returns a number representing the day of the week on which $\langle expD \rangle$ occurs. The first day of the week is considered to be Sunday and is given the number 1. Monday is 2, Tuesday is 3, etc. up to Saturday which is 7. If $\langle expD \rangle$ is an empty date, then DOW() will return a 0.   |   |  |
| Example:       | <pre>? DATE() &amp;&amp; Result: 07/17/89<br/>? DOW(DATE()) &amp;&amp; Result: 2<br/>? CDOW(DATE()) &amp;&amp; Result: Monday<br/>? DOW(DATE()-2) &amp;&amp; Result: 7<br/>? CDOW(DATE()-2) &amp;&amp; Result: Saturday<br/>dvar = DATE()<br/>weekof = dvar-DOW(dvar)+2<br/>IF DOW(dvar)=1<br/>weekof = weekof-7 &amp;&amp; Sunday = prev week<br/>ENDIF</pre> | ( |  |
|                | The above example generates the date of the Monday preceding<br>the date in "dvar". Note: This routine considers both Saturday and<br>Sunday part of the previous week. To count Sunday as part of the<br>week beginning with the Monday following it, remove the IF clause.   |   |  |
| See Also:      | CDOW(), CMONTH(), CTOD(), DATE(), DAY(),<br>DTOC(), DTOS(), MONTH(), YEAR()  |   |  |
| Compatibility: | dBASE standard, no extensions.   |   |  |

# DTOC()

| Syntax:        | DTOC( <expd>)</expd>   |  |  |
|----------------|--|--|--|
| Purpose:       | Decode a date value into a character string formatted date.  |  |  |
| Argument:      | $< \exp D >$ is the date value to decode and format.   |  |  |
| Returns:       | A character string.  |  |  |
| Usage:         | DTOC() returns a character string formatted version of the date<br>in <expd>. The format is based on the current setting of the SET<br/>DATE and SET CENTURY commands. The default format is<br/>SET CENTURY OFF, SET DATE AMERICAN and is<br/>"mm/dd/yy". An empty date returns " / / ".</expd> |  |  |
| Example:       | <pre>? DATE() &amp;&amp; Result: 07/17/89 ? DTOC(DATE()) &amp;&amp; Result: 07/17/89 ? "Today is "+DTOC(DATE())</pre>  |  |  |
|                | Result: Today is 07/17/89  |  |  |
| See Also:      | SET CENTURY, SET DATE, CDOW(), CMONTH(),<br>CTOD(), DATE(), DAY(), DOW(), DTOS(), MONTH(),<br>TRANSFORM(), YEAR()  |  |  |
| Compatibility: | dBASE standard, no extensions.   |  |  |
|                |  |  |  |

.

#### **Chapter 5: TDBS Functions**

۶,

## DTOS()

| Syntax:        | DTOS( <expd>)</expd>  |  |  |
|----------------|---|--|--|
| Purpose:       | Converts date value to a standard string format which collates properly by date in a sort.  |  |  |
| Argument:      | $<\exp D>$ is the date value to convert.  |  |  |
| Returns:       | A character string.   |  |  |
| Usage:         | DTOS() returns a string which is always 8 characters long. The format of this string is "yyyymmdd". If $\langle \exp D \rangle$ is a blank date, a string of 8 spaces is returned. This format for a date assures that for any sorting or collating, that dates from different months and years will always collate in a proper ascending sequence. |  |  |
| Example:       | <pre>? DATE() &amp;&amp; Result: 07/17/89 ? DTOS(DATE()) &amp;&amp; Result: 19890717</pre>  |  |  |
| See Also:      | CDOW(), CMONTH(), CTOD(), DATE(), DAY(), DOW(),<br>DTOC(), MONTH(), YEAR()  |  |  |
| Compatibility: | TDBS extended, no dBASE equivalent.   |  |  |

#### EMPTY()

| Syntax:   | EMPTY( <exp>)</exp>  |                                       |  |
|-----------|--|---------------------------------------|--|
| Purpose:  | Determines whether the result of an expression is empty.   |                                       |  |
| Argument: | <exp> is an exp</exp>  | pression of any da                    | ata type to test for empty.  |
| Returns:  | A logical value.   |                                       |  |
| Usage:    | <b>sage:</b> EMPTY() returns a logical true (.T.) if expression. An expression is considered e data type if it meets the following criteria:       |                                       | sidered empty depending on its   |
|           | Character:<br>Numeric:<br>Date:<br>Logical:  | Null or all space<br>0<br>Null<br>.F. | s & tabs.  |
| Example:  | <pre>? EMPTY(SPA<br/>? EMPTY("")<br/>? EMPTY(0)<br/>? EMPTY(CTC<br/>? EMPTY(.F.<br/>? EMPTY(1)<br/>? EMPTY("<br/>? EMPTY(DAT<br/>? EMPTY(.T.</pre> | DD(""))<br>)<br>a")<br>E())           | && Result: .T.<br>&& Result: .T.<br>&& Result: .T.<br>&& Result: .T.<br>&& Result: .T.<br>&& Result: .F.<br>&& Result: .F.<br>&& Result: .F.<br>&& Result: .F. |

Compatibility:

TDBS extended, no dBASE equivalent.

| EOF() |  |
|-------|--|
|       |  |

Syntax: EOF() Determines if the database file in the current work area has been Purpose: moved past its end-of-file. **Returns:** A logical value. Usage: EOF() returns true (.T.) when the current database file has been positioned past the last logical record. When EOF() becomes true, the record pointer is positioned to RECCOUNT() + 1. Any attempt to move the file forward after EOF() is set, will result in an error. If the current database file contains no records, then both EOF() and BOF() are true. **Example:** USE Customer GOTO BOTTOM && Result: .F. ? EOF() SKIP && Result: .T. ? EOF() See Also: CONTINUE, FIND, LOCATE, SEEK, SKIP BOF(), FOUND(), RECCOUNT(), RECNO() **Compatibility:** dBASE standard, no extensions.

# ERROR()

| Syntax:        | ERROR()  |   |  |
|----------------|--|---|--|
| Purpose:       | Returns the error code which triggered an ON ERROR condition.  |   |  |
| Returns:       | An integer numeric value.  |   |  |
| Usage:         | ERROR() returns the error code which triggered entry into an ON<br>ERROR handler. If the program is not in an ON ERROR handler<br>then ERROR() will always return a zero. This function allows the<br>error handler to handle different errors properly. |   |  |
| Example:       | ON ERROR DO ERRHAND  | && Connect Handler  |  |
|                | PROCEDURE ERRHAND<br>DO CASE<br>CASE ERROR()=108<br>A=INKEY(1)<br>RETRY  | && File Lock Error?<br>&& Delay 1 second<br>&& Retry the open |  |
|                | OTHERWISE<br>? MESSAGE()<br>HALT "Aborting P<br>ENDCASE  | && Print Error Msg<br>rogram"                                 |  |
| See Also:      | ON ERROR, USE EXCLUSIVE, MESSAGE()   |   |  |
| Compatibility: | dBASE standard, no extensions.   |   |  |

.

## EXP()

| Syntax:<br>Purpose: | <b>EXP(<expn>)</expn></b><br>Calculates e <sup><expn></expn></sup> where e is the base of a natural logarithm.  |   |  |
|---------------------|---|---|--|
| Argument:           | <expn> is the number which is the power of e to calculate.</expn>   |   |  |
| Returns:            | A numeric value.  |   |  |
| Usage:              | EXP() returns the requested power of e (2.7182818285). This value is calculated to 15.9 significant digits. This value is useful in many mathematical calculations. |   |  |
| Example:            | ? EXP(1)<br>? LOG(EXP(1))   | && Result: 2.7182818285<br>&& Result: 1 |  |
| See Also:           | LOG()   |   |  |
| Compatibility:      | dBASE standard, no extensions.  |   |  |

## FBEXTRACT()

| ) | Syntax:        | FBEXTRACT( <expn1>[,<expn2>[,<expn3>]])</expn3></expn2></expn1>   |
|---|----------------|---|
|   | Purpose:       | Extract data from a flat file I/O buffer into a string.   |
|   | Arguments:     | <expn1> is the handle of the flat file with which the buffer is associated. This file must be open in binary mode.</expn1>  |
|   |                | $<\exp N2>$ is the first byte of the buffer to extract from. Any<br>number less than or equal to 1 is the start of the buffer. If not<br>specified, the first byte of the buffer is assumed.  |
|   |                | $<\exp N3>$ is the number of bytes to extract. If not specified, all bytes to the end of the buffer are assumed.  |
|   | Returns:       | A character string.   |
|   | Usage:         | FBEXTRACT() allows placing a portion of a binary flat file record<br>into a TDBS character string for manipulation. Note: If the result<br>of the extraction exceeds 254 characters, then the string "****" is<br>returned instead. |
|   | Example:       | FOPEN Handle BINARY.FIL 0 2048<br>FBREAD Handle Size<br>Field3 = FBEXTRACT(Handle, 253, 25)   |
|   | •              | Bytes 253 - 277 of the 2k binary record are extracted and placed into the character string memvar "Field3".   |
|   | See Also:      | FBINSERT(), FBFILL(), FBMOVE()  |
|   | Compatibility: | TDBS extended, no dBASE equivalent.   |
|   |                |   |

## FBFILL()

| Syntax:<br>Purpose: | <pre>FBFILL(<expn1>[,<expn2>[,<expc>[,<expn3>]]]) Fill a binary flat file I/O buffer with a pattern of characters.</expn3></expc></expn2></expn1></pre>   |  |  |  |  |  |
|---------------------|---|--|--|--|--|--|
| Arguments:          | <expn1> is the handle of the flat file with which the buffer is associated. This file must be open in binary mode.</expn1>  |  |  |  |  |  |
|                     | <expn2> is the first byte of the buffer to fill. Any number less<br/>than or equal to 1 is the start of the buffer. If not specified, the first<br/>byte of the buffer is assumed.</expn2>  |  |  |  |  |  |
|                     | $\langle expC \rangle$ is the pattern to fill the buffer with. If not specified, the buffer is filled with binary zeroes (CHR(0) bytes).  |  |  |  |  |  |
|                     | <expn3> is the number of repetitions of string <expc>to fill.<br/>If not specified, all bytes to the end of the buffer are filled.</expc></expn3>   |  |  |  |  |  |
| Returns:            | A numeric value.  |  |  |  |  |  |
| Usage:              | FBFILL() allows you to fill a binary flat file record with a pre-deter-<br>mined pattern. The returned number is the next byte position in<br>the buffer after the fill ended. If the returned number is<br>FLEN(Handle) + 1 then the entire buffer was filled. |  |  |  |  |  |
| Example:            | FOPEN Handle BINARY.FIL 0 2048<br>FBFILL(Handle, 1, "BLANK")  |  |  |  |  |  |
|                     | The character string "BLANK" will be repeated into the buffer until all bytes of the buffer are filled.   |  |  |  |  |  |
| See Also:           | FBINSERT(), FBEXTRACT(), FBMOVE()   |  |  |  |  |  |
|                     |   |  |  |  |  |  |

#### FBINSERT()

| Syntax:        | FBINSERT( <expn1>[,<expn2>[,<expc><br/>[,<expn3>]]])</expn3></expc></expn2></expn1>  |  |  |  |  |  |
|----------------|--|--|--|--|--|--|
| Purpose:       | Insert data from a string into a binary flat file I/O buffer.  |  |  |  |  |  |
| Arguments:     | $< \exp N1 >$ is the handle of the flat file with which the buffer is associated. This file must be open in binary mode.   |  |  |  |  |  |
|                | $< \exp N2 >$ is the first byte of the buffer to insert into. Any number less than or equal to 1 is the start of the buffer. If not specified, the first byte of the buffer is assumed.  |  |  |  |  |  |
|                | $<\exp C>$ is the character string which contains the data to insert into the binary file I/O buffer.  |  |  |  |  |  |
|                | <expn3> is the number of bytes to insert. If not specified, all bytes to the end of the buffer are assumed.</expn3>  |  |  |  |  |  |
| Returns:       | A numeric value.   |  |  |  |  |  |
| Usage:         | FBINSERT() allows you to insert a character string into a binary<br>flat file record at the desired location. The returned number is the<br>next byte position in the buffer after the fill ended. If the returned<br>number is FLEN(Handle) + 1 then the entire buffer was filled. If<br>the buffer would have overflowed insertion stops and -1 is returned. |  |  |  |  |  |
| Example:       | FOPEN Handle BINARY.FIL 0 2048<br>FSEEK Handle Position 0 1 && Save Position<br>FBREAD Handle Size && Read Record<br>Field = ASC(FBEXTRACT(Handle, 253, 1))+3<br>Dummy = FBINSERT(Handle, 253, CHR(Field), 1)<br>FSEEK Handle Dummy Position 0 && Re-position<br>FBWRITE Handle Size && Re-write changed data  |  |  |  |  |  |
|                | This example adds 3 to byte 253 of the 1st record of the file.   |  |  |  |  |  |
| See Also:      | FBEXTRACT(), FBFILL(), FBMOVE()  |  |  |  |  |  |
| Compatibility: | TDBS extended, no dBASE equivalent.  |  |  |  |  |  |

## FBMOVE()

| Syntax:    | FBMOVE( <expn1>,<expn2>,<expn3>[,<expn4><br/>[,<expn5>]])</expn5></expn4></expn3></expn2></expn1>   |  |  |  |
|------------|---|--|--|--|
| Purpose:   | Move data from one binary flat file I/O buffer to another.  |  |  |  |
| Arguments: | $< \exp N1 >$ is the handle of the flat file with which the destination buffer is associated. This file must be open in binary mode.  |  |  |  |
|            | $<\exp N2>$ is the first byte of the buffer to insert moved data. Any number less than or equal to 1 is the start of the buffer. If not specified, the first byte of the buffer is assumed.   |  |  |  |
|            | <expn3> is the handle of the flat file with which the source buffer<br/>is associated. This file must be open in binary mode.</expn3>   |  |  |  |
|            | <expn4> is the first byte of the source buffer to move. Any<br/>number less than or equal to 1 is the start of the buffer. If not<br/>specified, the first byte of the buffer is assumed.</expn4>   |  |  |  |
|            | $< \exp N5 >$ is the number of bytes to move. If not specified, all bytes to the end of the source buffer are moved.  |  |  |  |
| Returns:   | A numeric value.  |  |  |  |
| Usage:     | FBMOVE() allows you to move data from one binary flat buffer to<br>another. The returned number is the next byte position in the<br>destination buffer after the move ended. If the returned number is<br>FLEN(Handle) + 1 then the entire buffer was filled. If the buffer<br>would have overflowed the move stops and -1 is returned. |  |  |  |

| Example:       | FOPEN Handle BINARY.FIL 0 2048<br>FOPEN Handle2 OTHER.FIL 2 4096<br>Check = FBMOVE(Handle2, 1, Handle, 1, 512) |  |  |  |  |
|----------------|--|--|--|--|--|
|                | The first 512 bytes of the buffer for BINARY.FIL is moved to the first 512 bytes of the buffer for OTHER.FIL.  |  |  |  |  |
| See Also:      | FBINSERT(), FBEXTRACT(), FBFILL()  |  |  |  |  |
| Compatibility: | TDBS extended, no dBASE equivalent.  |  |  |  |  |

4

-

#### FCOUNT()

| Syntax: FCOUNT() |  |  |  |  |  |  |
|------------------|--|--|--|--|--|--|
| Purpose:         | Obtains the number of fields in the current database.  |  |  |  |  |  |
| Returns:         | An integer value.  |  |  |  |  |  |
| Usage:           | FCOUNT() returns the number of fields in the database file open<br>in the current work area. If no database file is open, a zero is<br>returned. |  |  |  |  |  |
| Example:         | USE Sales<br>? FCOUNT() && Result: 5<br>COPY STRUCTURE EXTENDED TO Temp<br>USE Temp<br>? RECCOUNT() && Result: 5                                 |  |  |  |  |  |
| See Also:        | FIELD(), TYPE(), AFIELDS()   |  |  |  |  |  |
| Compatibility:   | TDBS extended, no dBASE equivalent, Clipper compatible.  |  |  |  |  |  |

# FDATE()

| Syntax:        | FDATE( <expc>)</expc>  |
|----------------|--|
| Purpose:       | Obtains the date, from the operating system, that a specified file was last modified.  |
| Argument:      | $<\exp C>$ is the file character string. Drive and path may be included, if they are absent the HOMEPATH() is assumed.                               |
| Returns:       | A date value.  |
| Usage:         | FDATE() returns the date of a file. If the specified file does not exist, a blank date is returned.  |
| Example:       | ? FDATE("TEST.FIL") && Print the file date<br>Fname = "D:\TBBS\DATA.DOC"<br>File_Date = FDATE(Fname)<br>? File_Date && Show date of D:\TBBS\DATA.DOC |
| See Also:      | FTIME(), FSIZE(), FINDFIRST(), FINDNEXT()  |
| Compatibility: | TDBS extended, no dBASE equivalent.  |

.

## FERROR()

| Syntax:        | FERROR([ <expn>])</expn>  |  |  |  |  |
|----------------|---|--|--|--|--|
| Purpose:       | Obtain error status of last flat file I/O function.   |  |  |  |  |
| Argument:      | <expn> is the handle of the flat file for which error status is desired. If absent, the status of the most recent flat file I/O operation on any handle is returned.</expn> |  |  |  |  |
| Returns:       | A numeric value.  |  |  |  |  |
| Usage:         | FERROR() returns an error code, or 0 if there was no error. The error code may be converted to a text message by using the function MESSAGE(ecode).                         |  |  |  |  |
| Example:       | Ecode = FERROR() && Last Flat File FCN<br>Ecode = FERROR(Handle) && Last FCN for Handle<br>? "Error:",MESSAGE(Ecode)  |  |  |  |  |
| See Also:      | FOPEN, FCREATE, FCLOSE, FSEEK,<br>FBREAD, FBWRITE,<br>FLREAD, FLWRITE, FLFIND<br>MESSAGE()  |  |  |  |  |
| Compatibility: | TDBS extended, no dBASE equivalent.   |  |  |  |  |

## FIELD()

| Syntax:<br>Purpose: | FIELD( <expn>)<br/>Returns the name of the specified field in the current database.</expn>  |  |  |  |  |  |
|---------------------|---|--|--|--|--|--|
| Argument:           | $< \exp N >$ is the position of the desired field in the database.  |  |  |  |  |  |
| Returns:            | A character string.   |  |  |  |  |  |
| Usage:              | FIELD() returns the name of the specified field in a character<br>string. The first field in a file is one, the second is two, etc. If the<br>field specified is larger than the number of fields in the file, then a<br>null string is returned. Field names are returned in upper case. |  |  |  |  |  |
| Example:            | USE Customer<br>? FIELD(1) && Result: FNAME   |  |  |  |  |  |
| See Also:           | ALIAS(), NDX(), SELECT(), DBF()   |  |  |  |  |  |
| Compatibility:      | dBASE standard, no extensions.  |  |  |  |  |  |

## FILE()

| Syntax:        | FILE( <expc>)</expc>   |     |                               |     |  |  |
|----------------|--|-----|-------------------------------|-----|--|--|
| Purpose:       | Determines whether a file exists.  |     |                               |     |  |  |
| Argument:      | $<\exp C>$ is the name of the file to locate.  |     |                               |     |  |  |
| Returns.       | A logical value.   |     |                               |     |  |  |
| Usage:         | FILE() returns true (.T.) if the specified file exists. You may<br>optionally put a drive and path on the file name to locate it on any<br>drive in any directory. If no path is specified, TDBS looks for the<br>file in the HOMEPATH directory. If the specified drive or path do<br>not exist, then false (.F.) is returned, and no error is generated. |     |                               |     |  |  |
| Example:       | <pre>? FILE("TRANSACT.PRG") ? FILE("C:\TBBS\INFO1.TXT") ? FILE("F:\FILE1.DBF")</pre>   | & & | Result:<br>Result:<br>Result: | .т. |  |  |
| See Also:      | ERASE, RENAME, HOMEPATH()  |     |                               |     |  |  |
| Compatibility: | dBASE standard, no extensions.   |     |                               |     |  |  |

#### **FINDFIRST()**

Syntax:

FINDFIRST(< memvar1>, < expC1>[, < expC2>
[,memvar2>]])

**Purpose:** Locate file name and attributes of the first file in a directory which meets the specified search criteria.

Arguments:

<expC1> is a character expression containing the search skeleton. This skeleton may contain a drive and a path and the HOMEPATH() is used if no path is given. The file name and extension may have wildcard characters (either \* or ?) to search for files using the normal DOS wildcard rules.

 $<\exp C2>$  is an optional five character string which specifies the attributes of files which qualify for the search. The mask has the following form:

"XXXXX" where each position contains an "X" to allow files with that attribute, and a "." to disallow files with that attribute. The positions equate to attributes as follows:

"X...." = DIRECTORY Names ".X..." = VOLUME LABEL "..X.." = SYSTEM file "...X." = HIDDEN file "....X" = READONLY file

Note: More than one "X" may be set to allow searches to find multiple file types. READONLY and NORMAL attribute files are always found in any search, so you must use the returned attributes to determine if it is the type of file you are searching for.

#### **Chapter 5: TDBS Functions**

|                | <memvar2> is an optional returned six character string which<br/>indicates the file attributes of the file name located by the search.<br/>The six characters show an "X" if the file has the corresponding<br/>attribute, and a "." if it does not. The returned attributes have the<br/>following meaning:</memvar2> |  |
|----------------|--|--|
|                | <pre>"X" = ARCHIVE (file has been modified since backup) ".X" = DIRECTORY name "X" = VOLUME LABEL "X." = SYSTEM file "X." = HIDDEN file "X" = READONLY file</pre>  |  |
| Returns:       | A character string.  |  |
| Usage:         | FINDFIRST() returns the name of the first file that matches the search skeleton in the form "filename.ext". If no file matches, an empty string of zero length is returned.  |  |
| Example:       | <pre>Fname = FINDFIRST(Dta, "*.*", ".XXXX") DO WHILE LEN(Fname) &gt; 0    ? Fname    Fname = FINDNEXT(Dta) ENDDO</pre>   |  |
|                | This example will list the names of all files in the HOMEPATH() directory.   |  |
| See Also:      | FDATE(), FTIME(), FSIZE(), FINDNEXT()  |  |
| Compatibility: | TDBS extended, no dBASE equivalent.  |  |

| FI | Ν | D | Ν | E) | KT | () |
|----|---|---|---|----|----|----|
|    |   |   |   |    |    | V  |

| Syntax:        | FINDNEXT( <memvar1>[,<memvar2>])</memvar2></memvar1>   |
|----------------|--|
| Purpose:       | Locate file name and attributes of the next file in a directory which meets the specified search criteria (set by FINDFIRST).  |
| Arguments:     | <memvar1> is the variable which received the DOS internal<br/>information from a previous FINDFIRST or FINDNEXT. This<br/>data contains the search mask and search attribute restrictions<br/>along with DOS internal data so the search may be continued.</memvar1> |
|                | <memvar2> is an optional returned six character string which<br/>indicates the file attributes of the file name located by the search.<br/>The returned attributes have the following meaning:</memvar2>   |
|                | "X" = ARCHIVE (file has been modified since backup)<br>"X" = DIRECTORY name<br>"X" = VOLUME LABEL  |
|                | " $\dots$ X $\dots$ " = SYSTEM file  |
|                | " $\dots$ x." = HIDDEN file  |
|                | " $\dots x$ " = READONLY file  |
| Returns:       | A character string.  |
| Usage:         | FINDNEXT() returns the name of the first file that matches the search skeleton in the form "filename.ext". If no remaining file matches, an empty string of zero length is returned.   |
| Example:       | <pre>Fname = FINDFIRST(Dta, "*.*", ".XXXX")</pre>  |
|                | DO WHILE LEN(Fname) > 0<br>? Fname   |
|                | Fname = FINDNEXT(Dta)<br>ENDDO   |
|                | This example will list the names of all files in the HOMEPATH() directory.   |
| See Also:      | FDATE(), FTIME(), FSIZE(), FINDFIRST()   |
| Compatibility: | TDBS extended, no dBASE equivalent.  |
|                |  |

## FKLABEL()/FKMAX()

| Syntax:        | FKLABEL( <expn>)</expn>  |  |
|----------------|--|--|
|                | FKMAX()  |  |
| Purpose:       | Obtains the name assigned to the specified function key:               |  |
| Argument:      | <expn> specifies the function key desired.</expn>                      |  |
| Returns:       | FKLABEL() returns a character string.                                  |  |
|                | FKMAX() returns a numeric value.                                       |  |
| Usage:         | FKLABEL() will return the name of any programmable function keys.      |  |
|                | FKMAX() Returns the maximum number of programmable func-<br>tion keys. |  |
| Compatibility: | dBASE Standard, no extensions.   |  |

# FLEN()

| Syntax:        | FLEN( <expn>)</expn>  |
|----------------|---|
| Purpose:       | Obtain the size of the buffer associated with an open flat file.  |
| Argument:      | <expn> is the handle of the open flat file for which you wish to determine the buffer size (returned by FOPEN or FCREATE).</expn>   |
| Returns:       | A numeric value.  |
| Usage:         | FLEN() returns the size in bytes of the buffer associated with the specified flat file handle. 0 is returned if no buffer is associated, and -1 is returned if the handle is invalid. |
| Example:       | FOPEN Handle BINARY.FIL 0 2048<br>BSize = FLEN(Handle) &Result: Bsize=2048  |
| See Also:      | FOPEN, FCREATE, FMAXLEN()   |
| Compatibility: | TDBS extended, no dBASE equivalent.   |
|                |   |

s"

## FLOCK()

| Syntax:        | FLOCK()   |
|----------------|---|
| Purpose:       | Locks the current database file and reports success or failure.   |
| Returns:       | A logical value.  |
| Usage:         | FLOCK() attempts to explicitly lock all of the records in the current database file. If this locking attempt is successful it returns a logical true (.T.). If the lock attempt fails a false (.F.) is returned. The file lock remains in place until you issue either another record or file lock attempt, issue the UNLOCK command, or close the database. Note: any form of program termination will close the database and release any locks. |
|                | Explicit program locks are not required in TDBS unless the application strategy requires exclusive use of a record of file for longer than one program command. See Chapter 3 for a full discussion of TDBS multiuser programming techniques and capabilities.  |
| Example:       | DO WHILE .NOT. FLOCK() && Wait for file lock<br>ENDDO<br>APPEND FROM Updates<br>UNLOCK  |
| See Also:      | SET EXCLUSIVE, UNLOCK, USE EXCLUSIVE,<br>RLOCK( ), WAIT4RLOCK( ), WAIT4FLOCK( )   |
| Compatibility: | dBASE standard, no extensions.  |

# FLOOR()

| Syntax:        | FLOOR( <expn>)</expn>   |   |
|----------------|---|---|
| Purpose:       | Finds the integer which is equal  | to or lower in value for $< \exp N >$ .   |
| Argument:      | <expn> is the numeric expres</expn>   | sion to find the FLOOR of.  |
| Returns:       | An integer numeric value.   |   |
| Usage:         | magnitude. The FLOOR of a integer, is the integer portion of  | ber is dependent on sign as well as<br>positive number which is not an<br>of the number. The FLOOR of a<br>integer is the next smaller valued<br>magnitude negative number. |
| Examples:      | <pre>? FLOOR(1.5)<br/>? FLOOR(-1.5)<br/>? FLOOR(1.0)<br/>? FLOOR(-1.0)<br/>? FLOOR(0.333)<br/>? FLOOR(-0.333)</pre> | && Result: 1<br>&& Result: -2<br>&& Result: 1<br>&& Result: -1<br>&& Result: 0<br>&& Result: -1   |
| See Also:      | CEILING(), INT()  |   |
| Compatibility: | TDBS extended, no dBASE equ   | iivalent.   |

.

#### FMAXLEN()

|                | V  |
|----------------|--|
| Syntax:        | FMAXLEN()  |
| Purpose:       | Obtain the maximum number of bytes available in the work pool which could be assigned to a flat file I/O buffer.   |
| Returns:       | A numeric value.   |
| Usage:         | FMAXLEN() returns the maximum number of bytes that could<br>currently be allocated from the work pool for flat file I/O buffers.<br>Values greater than 256 are rounded down to the nearest 256 byte<br>boundary. Since flat file I/O buffers are allocated from the TDBS<br>work pool (see Understanding Work Pool Allocation in Chapter 2)<br>which is used by all file I/O, this function tells you the available size<br>at this moment in your program's execution. The available size may<br>be increased by closing any database work areas or other flat file<br>which have buffers associated with them that you don't need at this<br>point in your program. |
| Example:       | <pre>BufSize = 2048 IF FMAXLEN() &lt; BufSuze Bufsize = FMAXLEN() ENDIF FOPEN Handle TEST.TXT 10 BufSize This example opens a file with a buffer of 2048 bytes if possible, or with the largest buffer that will fit if 2048 bytes are not available.</pre>  |
| See Also:      | FOPEN, FCREATE, FLEN()   |
| Compatibility: | TDBS extended, no dBASE equivalent.  |

## FOUND()

| Syntax:        | FOUND()   |
|----------------|---|
| Purpose:       | Determines whether the previous file search via FIND, SEEK, LOCATE or CONTINUE command found a match.   |
| Returns:       | A logical value.  |
| Usage:         | FOUND() returns a logical true (.T.) if the last search command was successful.   |
|                | Any record movement other than one of the four search commands<br>(or an implied SEEK due to a SET RELATION) will reset the<br>FOUND() flag to false (.F.).   |
|                | Each work area has a separate FOUND() flag, so that if a SET<br>RELATION is active a single search may result in a FOUND() of<br>true (.T.) in the current work area, and a FOUND() of false (.F.)<br>in a related work area. |
| Example:       | USE Customer INDEX LName<br>SEEK "Smith"  |
|                | ? FOUND(), EOF() && Result: .TF.<br>SEEK "Shelly"   |
|                | ? FOUND(), EOF() && Result: .FT.<br>SEEK "Smith"  |
|                | ? FOUND(), EOF() && Result: .TF.<br>SKIP  |
|                | ? FOUND(), EOF() && Result: .FF.  |
| See Also:      | CONTINUE, FIND, LOCATE, SEEK, SET RELATION, EOF( )  |
| Compatibility: | dBASE standard, no extensions.  |
|                |   |

-

#### FSIZE()

| Syntax:<br>Purpose: | <b>FSIZE(<expc>)</expc></b><br>Obtains the file size, from the operating system, for a specified file.  |
|---------------------|---|
| Argument:           | <expc> is the file character string. Drive and path may be</expc>   |
| Argument.           | included, if they are absent the HOMEPATH() is assumed.   |
| Returns:            | A numeric value.  |
| Usage:              | FSIZE() returns the DOS size of a file. If the specified file does<br>not exist, a zero is returned. Note: A file may exist with a length of<br>zero, so an FSIZE() return of zero does not necessarily mean that<br>a file does not exist. |
| Example:            | ? FSIZE("TEST.FIL") && Print the file size<br>Fname = "D:\TBBS\DATA.DOC"  |
|                     | File_Size = FSIZE(Fname)<br>? File_Size && Show size of D:\TBBS\DATA.DOC  |
| See Also:           | FTIME(), FDATE(), FINDFIRST(), FINDNEXT()   |
| Compatibility:      | TDBS extended, no dBASE equivalent.   |

# FTIME()

| S | yntax:        | FTIME( <expc>)</expc>   |
|---|---------------|---|
| Ρ | urpose:       | Obtains the time, from the operating system, that a specified file was last modified.   |
| A | rgument:      | $<\exp C$ is the file character string. Drive and path may be included, if they are absent the HOMEPATH() is assumed.   |
| R | eturns:       | A character string.   |
| U | sage:         | FTIME() returns the time of a file in a string of the form "HH:MM:SS". If the specified file does not exist, a zero length empty string is returned.                              |
| E | xample:       | <pre>? FTIME("TEST.FIL) &amp;&amp; Print the file time<br/>Fname = "D:\TBBS\DATA.DOC"<br/>File_Time = FTIME(Fname)<br/>? File_Time &amp;&amp; Show time of D:\TBBS\DATA.DOC</pre> |
| S | ee Also:      | FDATE(), FSIZE(), FINDFIRST(), FINDNEXT()   |
| С | ompatibility: | TDBS extended, no dBASE equivalent.   |

4

.

## GETENV()

| Syntax:        | GETENV( <expc>)</expc>   |
|----------------|--|
| Purpose:       | Returns the contents of a DOS environment variable.  |
| Argument:      | <expc> is the name of a DOS environmental variable.</expc>   |
| Returns:       | A character string.  |
| Usage:         | GETENV() returns the text which follows the "=" from the requested DOS environmental variable. If the requested variable does not exist a null string is returned.   |
|                | DOS environmental variables are established by using the DOS SET command. Note: Due to a "quirk" of the DOS SET command, a variable with a space between the variable name and the " $=$ " is not considered the same as one without such a space. It is generally good practice to never try to take advantage of this quirk, and to not place spaces in the DOS SET command on either side of the equals operator. |
|                | You may use this capability to pass configuration from the DOS SET command to your TDBS program.   |
| Example:       | ? GETENV("COMSPEC")  |
|                | Result: C:\COMMAND.COM   |
| Compatibility: | dBASE standard, no extensions.   |

## GETLPT()

| Syntax:<br>Purpose: | GETLPT( <expn>)<br/>Requests access to LPT &lt; expN&gt;. Returns true (.T.) if granted.</expn>   |
|---------------------|---|
| Argument:           | $< \exp N >$ is the number of the printer to request (1 to 4).  |
| Returns:            | A logical value.  |
| Usage:              | GETLPT() requests the specified printer. If this request is successful the printer is assigned to you and a logical true (.T.) is returned. If the request fails a false (.F.) is returned. The printer remains yours to use until you release it with a SET PRINTER TO command, or by requesting a different printer. Note: any form of program termination will also release the printer. |
| Example:            | See Chapter 3 for a full discussion of TDBS printer support.<br>IF .NOT. GETLPT(1)<br>? "LPT1 is not available"<br>ELSE<br>EJECT && Send top of form<br>ENDIF   |
| See Also:           | SET PRINTER TO, EJECT, WAIT4LPT()   |
| Compatibility:      | TDBS extended, no dBASE equivalent.   |

## HARDCR()

| Syntax:        | HARDCR( <expc>)</expc>   |
|----------------|--|
| Purpose:       | Converts any 0x8D soft returns in a string to 0x0D hard returns.                                       |
| Argument:      | $< \exp C >$ is the text string to convert.  |
| Result:        | A character string.  |
| Usage:         | HARDCR turns any soft returns in a string to hard returns.   |
| Example:       | A = HARDCR(B)  |
|                | Result: String A becomes a copy of string B with any 0x8D soft returns converted to 0x0D hard returns. |
| Compatibility: | TDBS extended, no dBASE equivalent.  |

# HEX2DEC()

| Syntax:        | HEX2DEC( <expc>)</expc>   |  |  |
|----------------|---|--|--|
| Purpose:       | Converts a text string hexadecim  | al number to a numeric value.  |  |
| Argument:      | $<\exp C>$ is the text hexadecima   | l number to convert.   |  |
| Result:        | A numeric value.  |  |  |
| Usage:         | ber and will convert it to a 32 binnumber. The range is +2,147,48<br>The conversion will skip any lead non-blank character. It will end w | ing blanks and begin with the first<br>with either the end of the < expC ><br>mal character. Either upper or |  |
| Example:       | <pre>? HEX2DEC("10") ? HEX2DEC("FFFFFFFE") ? HEX2DEC(" A test") ? HEX2DEC("1b")</pre>   |  |  |
| See Also:      | DEC2HEX()   |  |  |
| Compatibility: | TDBS extended, no dBASE equ   | ivalent.   |  |

#### HOMEPATH()

| Syntax:        | HOMEPATH()   |  |  |
|----------------|--|--|--|
| Purpose:       | Returns the default path from the Opt Data menu field.   |  |  |
| Returns:       | A character string.  |  |  |
| Usage:         | <ul> <li>HOMEPATH() returns the path where the .TPG program which is executing resides. This is the default drive and directory for this program.</li> <li>Since TDBS operates in a multiuser TBBS environment, the DOS default drive and directory have little meaning. So the HOMEPATH is used instead as the default. This allows you to place a .TPG program and all associated files in a separate directory. When you specify the location of the program in the Opt Data field of the TBBS menu entry, you also specify the default directory for all of the associated files automatically.</li> </ul> |  |  |
|                |  |  |  |
| Example:       | ? HOMEPATH() && Result: C:\TDBS  |  |  |
| Compatibility: | TDBS extended, no dBASE equivalent.  |  |  |

|  |   |     | Λ |
|--|---|-----|---|
|  | I | - ( |   |
|  |   |     | v |

| Syntax:        | <pre>IIF(<expl>,<exp true="">,<exp false="">)</exp></exp></expl></pre>   |
|----------------|--|
| Purpose:       | Returns one of two specified expressions depending on the logical value of the given logical expression.   |
| Arguments:     | <expl> is the logical expression to evaluate.</expl>   |
|                | < exp true $>$ is the value to return if $<$ expL $>$ is true (.T.).   |
|                | $< \exp \text{ false} > \text{ is the value to return if } < \exp L > \text{ is false (.F.).}$   |
| Returns:       | A value of any data type.  |
| Usage:         | IIF() returns the value of the argument determined by the control-<br>ling $\langle expL \rangle$ logical expression. Since $\langle exp$ true $\rangle$ and $\langle exp$ false $\rangle$ may be of different types, the value returned is the type of whichever expression is evaluated. |
|                | The expression which is not returned, is NOT evaluated. Thus any<br>side effects its evaluation may have had will not occur. Only the<br>logical expression and the expression whose value is returned are<br>evaluated.   |
| Example:       | <pre>IIF(Tax,(Qty*UPrice*1.06),(Qty*UPrice))</pre>   |
|                | This example will add sales tax to the total price calculation only if the logical variable "Tax" is true (.T.).   |
| See Also:      | DO CASE, IF  |
| Compatibility: | dBASE standard, no extensions.   |
|                |  |

,

#### **INDEXEXT()**

Syntax:INDEXEXT()Purpose:Determines the type of index file structure in use.Returns:A character string.Usage:INDEXEXT() always returns ".NDX" in TDBS 1.2, because<br/>dBASE III Plus compatible index files are the only format sup-<br/>ported. This function is included to allow programs to be written<br/>compatibility:Compatibility:TDBS extended, no dBASE equivalent, Clipper compatible.

# INDEXKEY()

| Syntax:        | INDEXKEY( <expn>)</expn>  |  |  |
|----------------|---|--|--|
| Purpose:       | Determines the key expression of the specified index file.  |  |  |
| Argument:      | < expN > is the position of the desired index file in the list of index files in the currently selected work area. If $< expN >$ is zero, then the controlling index is desired regardless of its position in the list. |  |  |
| Returns:       | A character string.   |  |  |
| Usage:         | INDEXKEY() returns the key expression of the specified index. If<br>there is no index at the specified position, the a null string is<br>returned.  |  |  |
| Example:       | USE CustFile INDEX Name, CustNo<br>SET ORDER TO 2<br>? INDEXKEY(1) && Result: "Lname+Fname"<br>? INDEXKEY(2) && Result: "CustNo"<br>? INDEXKEY(0) && Result: "CustNo"   |  |  |
| Compatibility: | TDBS extended, no dBASE equivalent, Clipper compatible.   |  |  |

a

,a

#### INDEXORD()

| Syntax:        | INDEXORD()   |  |  |
|----------------|--|--|--|
| Purpose:       | Determines the position of the controlling index in the list of index files for the currently selected database.   |  |  |
| Returns:       | An integer numeric value.  |  |  |
| Usage:         | INDEXORD() returns the position of the controlling index in the<br>list of open index files for the current work area. If there is no<br>controlling index in the current work area, a zero is returned. |  |  |
|                | INDEXORD() is useful to record the controlling index prior to changing it, so that it may be restored later.   |  |  |
| Example:       | USE CustFile INDEX Name, CustNo<br>save_ord = INDEXORD()<br>SET ORDER TO 2<br>? INDEXORD() && Result: 2<br>SET ORDER TO save_ord<br>? INDEXORD() && Result: 1  |  |  |
| Compatibility: | TDBS extended, no dBASE equivalent, Clipper compatible.  |  |  |

#### INKEY()

Syntax: INKEY([<expN>]) Purpose: Reads a character from the keyboard typeahead buffer. Argument: <expN> specifies the number of seconds INKEY() waits for a key to be pressed. Specifying zero waits indefinitely for a key from the keyboard. If < expN > is omitted, INKEY() will not wait for a key press, but will return a key value if one was typed ahead. Returns: An integer numeric value. Usage: INKEY() returns the ASCII key code for a key which has been pressed. If the typeahead buffer is empty INKEY() will return a zero. If the time limit specified expires without a key press, then INKEY(<expN>) will return a zero. Function Keys: F1 returns 28, F2 through F10 return -1 thru -9 Example: DO WHILE LASTKEY() <> 27 ? "Press any key: " key = INKEY(0)?? "Character:",CHR(key),"ASCII code:",key ENDDO This routine displays the key code and repeats until Esc is pressed. key = 0DO WHILE key=0 DO calcs key = INKEY()ENDDO This routine repeatedly calls "calcs" until a key is pressed. See Also: SET TYPEAHEAD, LASTKEY(), NEXTKEY(), CHR() Compatibility: dBASE standard, plus TDBS extensions.

# INT()

| Syntax:        | INT( <expn>)</expn>   |  |   |                         |  |
|----------------|---|--|---|-------------------------|--|
| Purpose:       | Converts a numeric value to an integer by truncating all digits after<br>the decimal point.   |  |   |                         |  |
| Argument:      | <expn> is the numeric expression to convert.</expn>   |  |   |                         |  |
| Returns:       | An integer numeric value.   |  |   |                         |  |
| Usage:         | INT does not round an expression to the nearest integer, instead it<br>zeroes all digits after the decimal point. This discards any fractional<br>portion of the number leaving only the integer portion. |  |   |                         |  |
| Example:       | <pre>? INT(1.5)<br/>? INT(-1.5)<br/>? INT(1.0)<br/>? INT(-1.0)<br/>? INT(0.333)<br/>? INT(-0.333)<br/>? INT(0.999999999)</pre>  | 5 5<br>5 5<br>5 5<br>5 5<br>5 5<br>5 5<br>5 5<br>5 5<br>5 5<br>5 5 | Result:<br>Result:<br>Result:<br>Result:<br>Result:<br>Result:<br>Result: | -1<br>1<br>-1<br>0<br>0 |  |
| See Also:      | FLOOR(), CEILING(), RO  | UND()  |   |                         |  |
| Compatibility: | dBASE standard, no extensio   | ons.   |   |                         |  |

# ISALPHA()

| Syntax:        | ISALPHA( <expc>)</expc>   |  |  |
|----------------|---|--|--|
| Purpose:       | Determines if a character string begins with a letter.  |  |  |
| Argument:      | $< \exp C >$ is the character string to test.   |  |  |
| Returns:       | A logical value.  |  |  |
| Usage:         | ISALPHA returns true (.T.) if the first character of $\langle \exp C \rangle$ is alphabetic. Any upper or lower case letter A to Z is considered alphabetic. If $\langle \exp C \rangle$ is a null string, or begins with any non-alphabetic character, then ISALPHA returns false (.F.). |  |  |
| Example:       | <pre>? ISALPHA("ABC") ? ISALPHA("abc") ? ISALPHA("123") ? ISALPHA(".NOT.")</pre>  | && Result: .T.<br>&& Result: .T.<br>&& Result: .F.<br>&& Result: .F. |  |
| See Also:      | ISLOWER(), ISUPPER(), L<br>CAPFIRST()   | OWER(), UPPER(),   |  |
| Compatibility: | dBASE standard, no extension  | s.   |  |

# ISINT()

| Syntax:        | ISINT( <expn>)</expn>  |  |  |
|----------------|--|--|--|
| Purpose:       | Determines if a numeric value is an integer.   |  |  |
| Argument:      | <expn> is the numeric value to test.</expn>  |  |  |
| Returns:       | A logical value.   |  |  |
| Usage:         | ISINT returns true (.T.) if all digits of < expN> which follow the decimal point are zero. If there is any significant fractional portion to the number, then ISINT returns false (.F.). |  |  |
| Example:       | <pre>? ISINT(1.0) ? ISINT(25572) ? ISINT(1.1) ? ISINT(0.3333333)</pre>   | && Result: .T.<br>&& Result: .T.<br>&& Result: .F.<br>&& Result: .F. |  |
| See Also:      | CEILING(), FLOOR(), INT  | r(), round()   |  |
| Compatibility: | TDBS extended, no dBASE e  | equivalent.  |  |

# ISLASTDAY()

| Syntax:<br>Purpose: | ISLASTDAY( <expd>)<br/>Determines if a date value falls on the last day of the month.</expd>  |  |  |  |
|---------------------|---|--|--|--|
| Argument:           | $<\exp D>$ is the date value to test.   |  |  |  |
| Returns:            | A logical value.  |  |  |  |
| Usage:              | ISLASTDAY returns true (.T.) if the date given falls on the last day<br>of the month. Leap year is properly accounted for in this test. If<br>the specified date is empty or does not fall on the last day of the<br>month, then ISLASTDAY returns false (.F.). |  |  |  |
| Example:            | <pre>? ISLASTDAY(CTOD("01/10/88")) &amp;&amp; Result: .F.<br/>? ISLASTDAY(CTOD("02/28/87")) &amp;&amp; Result: .T.<br/>? ISLASTDAY(CTOD("02/28/88")) &amp;&amp; Result: .F.<br/>? ISLASTDAY(CTOD("09/30/88")) &amp;&amp; Result: .T.</pre>                      |  |  |  |
| See Also:           | LASTDAY(), ISLEAP()   |  |  |  |
| Compatibility:      | TDBS extended, no dBASE equivalent.   |  |  |  |

#### ISLEAP()

| Syntax:        | ISLEAP( <expd>)</expd>  |  |  |  |
|----------------|---|--|--|--|
| Purpose:       | Determines if a date value occurs in a leap year.   |  |  |  |
| Argument:      | $< \exp D >$ is the date value to test.   |  |  |  |
| Returns:       | A logical value.  |  |  |  |
| Usage:         | ISLEAP returns true (.T.) if the date given falls in a leap year. If<br>the specified date is empty or is from a year which is not a leap year,<br>then ISLEAP returns false (.F.). |  |  |  |
| Example:       | <pre>? ISLEAP(CTOD("01/10/88")) &amp;&amp; Result: .T.<br/>? ISLEAP(CTOD("02/28/87")) &amp;&amp; Result: .F.<br/>? ISLEAP(CTOD("09/30/88")) &amp;&amp; Result: .T.</pre>            |  |  |  |
| See Also:      | LASTDAY(), ISLASTDAY()  |  |  |  |
| Compatibility: | TDBS extended, no dBASE equivalent.   |  |  |  |

# ISLOWER()

| Syntax:        | ISLOWER( <expc>)</expc>  |  |     |
|----------------|--|--|-----|
| Purpose:       | Determines if the specified string begins with a lower case letter.  |  |     |
| Argument:      | $< \exp C >$ is the character string to test.  |  |     |
| Returns:       | A logical value.   |  |     |
| Usage:         | ISLOWER returns true (.T.) if the first character of the specified string is lower case (a to z). Otherwise ISLOWER will return false. |  |     |
| Example:       | <pre>? ISLOWER("ABC") ? ISLOWER("abc") ? ISLOWER("aBC")</pre>  | && Result:<br>&& Result:<br>&& Result: | .т. |
| See Also:      | ISALPHA(), ISUPPER(), LOWER(), UPPER()   |  |     |
| Compatibility: | dBASE standard, no extensions.   |  |     |

# ISSHARE()

| Syntax:<br>Purpose: | ISSHARE( <expc>)<br/>Determine if a work area is currently being shared.</expc>  |
|---------------------|--|
| Argument:           | $\langle expC \rangle$ is a character string from the USING() function to be tested.   |
| Returns:            | A logical value.   |
| Usage:              | ISSHARE() returns a logical .T. if the specified < expC> has an "X" character anywhere other than the user's own slot. Normal usage is in combination with the USING() function to determine if a database or mailbox is currently being shared. |
| Example:            | USE CustFile   |
|                     | ? ISSHARE(USING())   |
|                     | Results: .T. if file is shared, .F. if it is not.  |
| Compatibility:      | TDBS extended, no dBASE equivalent.  |

#### ISSTATE()

| Syntax:<br>Purpose: | <b>ISSTATE(<expc>)</expc></b><br>Determines if the specified string is a valid US state abbreviation.   |            |
|---------------------|---|------------|
| Argument:           | <expc> is the character string to test.</expc>  |            |
| Returns:            | A logical value.  |            |
| Usage:              | ISSTATE returns true (.T.) if the specified character string is<br>exactly two characters long and is a valid US post office state<br>abbreviation. Otherwise ISSTATE will return false (.F.). Upper<br>or lower case is not significant for this test. |            |
| Example:            | <pre>? ISSTATE("ABC") &amp;&amp; Result: .F.<br/>? ISSTATE("CO") &amp;&amp; Result: .T.<br/>? ISSTATE("CO") &amp;&amp; Result: .T.<br/>? ISSTATE("co") &amp;&amp; Result: .T.<br/>? ISSTATE("Minn") &amp;&amp; Result: .F.</pre>                        |            |
| See Also:           | STATENAME()   |            |
| Compatibility:      | TDBS extended, no dBASE e   | quivalent. |

# ISUPPER()

| Syntax:<br>Purpose: | ISUPPER( <expc>)<br/>Determines if the specified string begins with an upper case letter.</expc>                                       |                            |       |     |
|---------------------|--|----------------------------|-------|-----|
| Argument:           | <expc> is the character string to test.</expc>   |                            |       |     |
| Returns:            | A logical value.   |                            |       |     |
| Usage:              | ISUPPER returns true (.T.) if the first character of the specified string is upper case (A to Z). Otherwise ISUPPER will return false. |                            |       |     |
| Example:            | <pre>? ISUPPER("ABC") ? ISUPPER("abc") ? ISUPPER("aBC")</pre>  | && Re:<br>&& Re:<br>&& Re: | sult: | .F. |
| See Also:           | ISALPHA(), ISLOWER(), LOWER(), UPPER()   |                            |       |     |
| Compatibility:      | dBASE standard, no extensions.   |                            |       |     |

#### LASTDAY()

| Syntax:        | LASTDAY( <expd>)</expd>   |   |  |
|----------------|---|---|--|
| Purpose:       | Returns the last day of the month in which $\langle \exp D \rangle$ falls.                                      |   |  |
| Argument:      | <expd> is the date for which to find the last day of the month.</expd>  |   |  |
| Returns:       | A date value.   |   |  |
| Usage:         | LASTDAY will return the date whi<br>in which <expd> falls. If <expl<br>empty date is returned.</expl<br></expd> | •   |  |
| Example:       | <pre>dvar = CTOD("07/17/89") ? LASTDAY(dvar) &amp; dvar = CTOD("02/10/88")</pre>                                | && Result: 01/31/88<br>&& Result: 07/31/89<br>&& Result: 02/29/88 |  |
| See Also:      | ISLASTDAY()   |   |  |
| Compatibility: | TDBS extended, no dBASE equiva  | lent.   |  |

.

.

# LASTKEY()

| Syntax:        | LASTKEY()   |  |  |
|----------------|---|--|--|
| Purpose:       | Returns the last key which was read by TDBS.  |  |  |
| Returns:       | An integer value.   |  |  |
| Usage:         | LASTKEY returns the integer ASCII key code for the last key<br>which TDBS read as part of any keyboard input. This can be used<br>to determine the last key read by an INKEY function, or an<br>ACCEPT, INPUT, or READ command. |  |  |
|                | If you want to know the next unread key waiting in the typeahead buffer, use the NEXTKEY() function.  |  |  |
|                | LASTKEY() is not affected by SET TYPEAHEAD 0 since it is reporting the last key read, not a key waiting in the typeahead buffer.  |  |  |
|                | Function Keys: F1 returns 28, F2 through F10 return -1 thru -9  |  |  |
| Example:       | <pre>svar = var &amp;&amp; Save original value @ 10,20 SAY "New Value: " GET var READ IF LASTKEY() = 27 &amp;&amp; was exit w/Esc?</pre>  |  |  |
|                | <pre>var = svar &amp;&amp; yes, ignore any changes ENDIF</pre>  |  |  |
|                | This routine will roll back a memory variable change if the READ<br>ended with an Esc key. Normally Esc only rolls back field changes<br>made during a READ.  |  |  |
| See Also:      | CHR(), INKEY(), NEXTKEY(), READKEY()  |  |  |
| Compatibility: | TDBS extended, no dBASE equivalent.   |  |  |

| LEFT() |
|--------|
|--------|

| Syntax:        | LEFT( <expc>,<expn>)</expn></expc>  |  |  |
|----------------|---|--|--|
| Purpose:       | Extracts the specified number of characters from the left end of a character string.  |  |  |
| Arguments:     | $< \exp C >$ is the character string from which to extract characters.  |  |  |
|                | < expN > is the number of characters to extract.  |  |  |
| Returns:       | A character string.   |  |  |
| Usage:         | LEFT returns the leftmost < expN> characters of < expC>. If<br>< expN> is negative or zero, then a null string is returned. If<br>< expN> is larger than the length of < expC> then the entire<br>original < expC> is returned. |  |  |
| Example:       | ? LEFT("September",3) && Result: Sep  |  |  |
|                | ? LEFT("Monday",3) && Result: Mon<br>? LEFT("ABC",10) && Result: ABC  |  |  |
|                |   |  |  |
| See Also:      | AT(), LTRIM(), RIGHT(), RTRIM(), STUFF(), SUBSTR()  |  |  |
| Compatibility: | dBASE standard, no extensions.  |  |  |

.

#### LEN()

| Syntax:        | LEN( <expc>/<array>)</array></expc>  |                                  |   |
|----------------|--|----------------------------------|---|
| Purpose:       | Returns the number of character number of elements in an array.              | ers in a character string or the |   |
| Argument:      | <expc> is the character string f</expc>                                      | or which to find the length.     |   |
|                | < array> is the array to count ele   | ements in.                       |   |
| Returns:       | An integer numeric value.  |                                  |   |
| Usage:         | LEN returns the number of cha<br><expc>. If <expc> is a null s</expc></expc> |                                  |   |
| Example:       | ? LEN("")  | && Result: 0                     |   |
|                |  | && Result: 3                     |   |
|                | ? LEN("Testing 1 2 3.")  | && Result: 14                    |   |
|                | DECLARE my_array[10]   |                                  | ( |
|                | ? LEN(my_array)<br>my array[5] = "ABC"                                       | && Result: 10                    |   |
|                |  | && Result: 3                     |   |
| See Also:      | LTRIM( ), RTRIM( ), RIGHT( )<br>STUFF( )                                     | ), LEFT( ), SUBSTR( ),           |   |
| Compatibility: | dBASE standard plus Clipper arr  | ray extensions.                  |   |

#### LJUST()

| Syntax:<br>Purpose: | LJUST( <expc>)<br/>Left justifies a character string.</expc>  |  |  |
|---------------------|---|--|--|
| Argument:           | $<\exp C>$ is the character string to left justify.   |  |  |
| Returns:            | A character string.   |  |  |
| Usage:              | LJUST returns a string of the same length as $\langle \exp C \rangle$ with any leading blanks moved to the end of the string. This left justifies any text in the input character string. |  |  |
| Example:            | ? LJUST(" ABC") && Result: ABC<br>? LJUST("1 23") && Result: 1 23<br>? LJUST(" 45")+"X" && Result: 45 X   |  |  |
| See Also:           | RJUST(), LTRIM(), RTRIM(), SUBSTR(), STUFF()  |  |  |
| Compatibility:      | TDBS extended, no dBASE equivalent.   |  |  |

4

-

# LOG()

| Syntax:        | LOG( <expn>)</expn>  |  |
|----------------|--|--|
| Purpose:       | Calculates the natural logarithm of a number.  |  |
| Argument:      | <expn> is the positive number for which to calculate the natural logarithm.</expn>                                     |  |
| Returns:       | A numeric value.   |  |
| Usage:         | LOG calculates $LOG_e < expN >$ . If this value is referred to as "X" then the following equation is true:             |  |
|                | $e^{X} = \langle expN \rangle$   |  |
|                | where e is a mathematical constant which is approximately equal to 2.7182818285.                                       |  |
|                | Note: If < expN> is zero or a negative number a numeric overflow is returned (which prints as a row of all asterisks). |  |
| Example:       | ? LOG(2.7182818285) && Result: 1.0   |  |
| See Also:      | EXP()  |  |
| Compatibility: | dBASE standard, no extensions.   |  |

#### LOWER()

| Syntax:        | LOWER( <expc>)</expc>   |   |
|----------------|---|---|
| Purpose:       | Converts any letters in a string to lower case.   |   |
| Argument:      | <expc> is the character string to convert to lower case.</expc>   |   |
| Returns:       | A character string.   |   |
| Usage:         | LOWER returns a copy of $\langle \exp C \rangle$ with any letters forced to lower case. All other characters are left as they were. |   |
| Example:       | <pre>? LOWER("STRING") ? LOWER("5 Chars")</pre>   | && Result: string<br>&& Result: 5 chars |
| See Also:      | ISLOWER(), ISUPPER(), UPPER()   |   |
| Compatibility: | dBASE standard, no extensions.  |   |

.

-

#### LTRIM()

| Syntax:        | LTRIM( <expc>)</expc>   |                 |  |
|----------------|---|-----------------|--|
| Purpose:       | Removes any leading blanks from the specified character string.                 |                 |  |
|                |   |                 |  |
| Argument:      | $< \exp C >$ is the string from which to remove leading blanks.                 |                 |  |
| Returns:       | A character string.   |                 |  |
| Usage:         | LTRIM returns a copy of $\langle expC \rangle$ with any leading blanks removed. |                 |  |
| Example:       | string = " 5678"  |                 |  |
|                | ? LEN(string)   | && Result: 8    |  |
|                |   | && Result: 5678 |  |
|                | ? LEN(LTRIM(string))  | && Result: 4    |  |
| See Also:      | RTRIM(), SUBSTR(), STUFF(   | )               |  |
| Compatibility: | dBASE standard, no extensions.  |                 |  |

# LUPDATE()

| Syntax:        | LUPDATE()   |  |  |  |  |
|----------------|---|--|--|--|--|
| Purpose:       | Returns a date value with the date the database in the current work area was last modified.   |  |  |  |  |
| Returns:       | A date value.   |  |  |  |  |
| Usage:         | LUPDATE returns the date of the last change to the current work<br>area database file. If no file is currently open, a blank date is<br>returned. |  |  |  |  |
| Example:       | ? LUPDATE() && Result: 07/17/89   |  |  |  |  |
| Compatibility: | dBASE standard, no extensions.  |  |  |  |  |

#### MAX()

| Syntax:<br>Purpose: | MAX( <expn1>,<expn2>)<br/>Returns the larger of two numeric expressions.</expn2></expn1>              |  |  |  |
|---------------------|---|--|--|--|
| Arguments:          | <expn1> is the first numeric value to compare.</expn1>  |  |  |  |
|                     | <expn2> is the second numeric value to compare.</expn2>   |  |  |  |
| Returns:            | A numeric value.  |  |  |  |
| Usage:              | MAX returns the larger of the two arguments. This allows for limit checking a value in an expression. |  |  |  |
| Example:            | ? MAX(5,10)       && Result: 10         ? MAX(5*2,7)       && Result: 10                              |  |  |  |
| See Also:           | MIN()   |  |  |  |
| Compatibility:      | dBASE standard, no extensions.  |  |  |  |

# MESSAGE()

| Syntax:        | MESSAGE([ <expn>])</expn>  |                |   |  |
|----------------|--|----------------|---|--|
| Purpose:       | Returns the error message for a given error code.  |                |   |  |
| Argument:      | $< \exp N >$ is the error code for which the error message is desired.<br>If this argument is omitted, then the error message for the error<br>which triggered an ON ERROR condition is given. |                |   |  |
| Returns:       | A character string.  |                |   |  |
| Usage:         | MESSAGE returns a character s<br>corresponding to the error code a<br>may be used with no error code a<br>handler to return the text of the<br>ON ERROR condition to occur.                    | giver<br>irgur | h by <expn>. MESSAGE<br/>nent inside an ON ERROR</expn> |  |
|                | MESSAGE() is equivalent to MI  | ESSA           | AGE(ERROR( )).  |  |
| Example:       | ON ERROR ERRHAND   | & &            | Connect handler   |  |
|                | PROCEDURE ERRHAND<br>IF ERROR() = 108<br>INKEY(1)<br>RETRY<br>ENDIF<br>HALT MESSAGE()  | 88             | Delay if file lock<br>Try again<br>abort w/error msg    |  |
| See Also:      | ON ERROR, ERROR()  |                |   |  |
| Compatibility: | dBASE standard plus TDBS exte  | nsio           | ns.   |  |

# MIN()

| Syntax:        | MIN( <expn1>,<expn2>)</expn2></expn1>                                |                              |
|----------------|--|------------------------------|
| Purpose:       | Returns the lesser of two numeric                                    | expressions.                 |
| Arguments:     | <expn1> is the first numeric val</expn1>                             | ue to compare.               |
|                | <expn2> is the second numeric</expn2>                                | value to compare.            |
| Returns:       | A numeric value.   |                              |
| Usage:         | MIN returns the lesser of the two checking a value in an expression. |                              |
| Example:       | ? MIN(5,10)<br>? MIN(5*2,7)  | && Result: 5<br>&& Result: 7 |
| See Also:      | MAX()  |                              |
| Compatibility: | dBASE standard, no extensions.                                       |                              |

ę

# MOD()

| Syntax:<br>Purpose: | <pre>MOD(<expn1>,<expn2>) Returns the value of <expn1> modulo <expn2>.</expn2></expn1></expn2></expn1></pre>  |  |  |  |  |
|---------------------|---|--|--|--|--|
| Arguments:          | $< \exp N1 >$ is the number to take the modulus of.   |  |  |  |  |
|                     | $< \exp N2 >$ is the base to take the modulus to.   |  |  |  |  |
| Returns:            | A numeric value.  |  |  |  |  |
| Usage:              | MOD returns the mathematical result of the calculation of <expn1> MOD <expn2>. If <expn1> is x, and <expn2> is y, then the complete formula for the modulus operation is:</expn2></expn1></expn2></expn1> |  |  |  |  |
|                     | $x \mod y = x - y \lfloor x/y \rfloor$ , if $y \neq 0$ ; $x \mod 0 = x$ .   |  |  |  |  |
|                     | The result of $x - (x \mod y)$ is an integral multiple of y, so you may think of x mod y as <i>the remainder when x is divided by y</i> .   |  |  |  |  |
| Example:            | ? MOD(3,-2)       && Result: -1         ? MOD(-3,2)       && Result: 1         ? MOD(5,3)       && Result: 2  |  |  |  |  |
| Compatibility:      | dBASE standard, no extensions.  |  |  |  |  |

8

#### MONTH()

| Syntax:        | MONTH( <expd>)</expd>                                  |   | ( |
|----------------|--|---|---|
| Purpose:       | Extracts the number of the m                           | onth from a date value.   |   |
| Argument:      | <expd> is the date value fr</expd>                     | rom which to extract the month.   |   |
| Returns:       | An integer numeric value.                              |   |   |
| Usage:         |  | or of the month in the specified date ary etc. If $\langle expD \rangle$ is a blank date, |   |
| Example:       | <pre>? DATE() ? MONTH(DATE()) ? MONTH(DATE()+30)</pre> | && Result: 07/17/89<br>&& Result: 7<br>&& Result: 8                                       |   |
| See Also:      | CDOW(), CMONTH(), CI<br>DTOC(), DTOS(), YEAR(          | COD(), DATE(), DAY(), DOW() )   | ( |
| Compatibility: | dBASE standard, no extension                           | ons.  |   |

# NDX()

| Syntax:        | NDX( <expn>)</expn>  |  |  |
|----------------|--|--|--|
| Purpose:       | Returns the name of the index file corresponding to $\langle expN \rangle$ .   |  |  |
| Argument:      | $< \exp N >$ is an integer numeric value representing the position of the desired index file in the index file list for the current work area.   |  |  |
| Returns:       | A character string.  |  |  |
| Usage:         | NDX will return the name of the desired index file from the currently selected work area. If no such index file exists, a null string is returned. $\langle expN \rangle$ must be between 1 and 7, since only 7 index files may be open at a time. |  |  |
| Example:       | USE Mail INDEX Names, Zips, Dates  |  |  |
|                | ? NDX(3) && Result: C:Dates.ndx  |  |  |
|                | ? NDX(2) && Result: C:Zips.ndx   |  |  |
|                | ? NDX(1) && Result: C:Names.ndx  |  |  |
|                | ? NDX(4) && Result: A Null String  |  |  |
| See Also:      | SET ORDER, SET INDEX, DBF(), SELECT()  |  |  |
| Compatibility: | dBASE standard, no extensions.   |  |  |

#### NEWMAIL()

| Syntax:        | NEWMAIL([ <expn>])</expn>   |
|----------------|---|
| Purpose:       | To see if a mailbox channel has received any new mail.  |
| Argument:      | $<\exp N>$ is a numeric value specifying the work area the mailbox to be checked is open in. If this argument is omitted, the current work area is assumed.   |
| Returns:       | A logical value.  |
| Usage:         | NEWMAIL will return a true (.T.) if the specified mailbox channel<br>has received any new mail since the last NEWMAIL() or<br>WAIT4MAIL() function was executed. Note: When a true is<br>returned, the newmail flag is automatically cleared and the NEW-<br>MAIL function will again return false (.F.) until more mail is<br>received in that mailbox. Each mailbox open has its own newmail<br>flag, and may be checked separately. If the specified work area is<br>not open to a mailbox, false (.F.) is returned. |
| Example:       | IF NEWMAIL()<br>? "Mail waiting"<br>ENDIF   |
| See Also:      | USE MAILBOX, WAIT4MAIL()  |
| Compatibility: | TDBS extended, no dBASE equivalent.   |

#### **NEXTKEY()**

Syntax:

NEXTKEY()

**Purpose:** Reads the next pending key in the typeahead buffer without removing if from the buffer.

Returns: An integer numeric value.

Usage: NEXTKEY returns the ASCII key code for the next character waiting to be read from the typeahead buffer. If there is no character waiting, then a zero is returned. Since NEXTKEY does not remove the key from the typeahead buffer, this same key will be read by the next INPUT, ACCEPT, READ, WAIT, or INKEY operation. If the next key in the typeahead buffer is a function key (or multiple key VT52/VT100 Esc sequence supported by TDBS) NEXTKEY will report the single control key equivalent of this sequence.

Note: SET TYPEAHEAD 0 will cause NEXTKEY to always report a zero, since there is no typeahead buffer to examine.

Function Keys: F1 returns 28, F2 through F10 return -1 thru -9

| Example: | ? | NEXTKEY() | & & | Result: | 65 | (ASCII | "A") |
|----------|---|-----------|-----|---------|----|--------|------|
|          | ? | INKEY()   | & & | Result: | 65 |        |      |
|          | ? | NEXTKEY() | 33  | Result: | 66 | (ASCII | "B") |
|          | ? | LASTKEY() | & & | Result: | 65 |        |      |
|          |   |           |     |         |    |        |      |

This example assumes "AB" originally in the typeahead buffer.

See Also: SET TYPEAHEAD, INKEY(), LASTKEY()

**Compatibility:** TDBS extended, no dBASE equivalent.

# NMYUSERS()

| Syntax:        | NMYUSERS()   |  |  |
|----------------|--|--|--|
| Purpose:       | Returns the number of users of this TDBS program.  |  |  |
| Returns:       | An integer numeric value.  |  |  |
| Usage:         | NMYUSERS reports the number of users currently running this program. This will be a number between 1 and 33. |  |  |
| Example:       | ? NMYUSERS() && Result: 3  |  |  |
| See Also:      | NUSERS()   |  |  |
| Compatibility: | TDBS extended, no dBASE equivalent.  |  |  |

# NUSERS()

| Syntax:        | NUSERS()   |
|----------------|--|
| Purpose:       | Returns the number of users currently running any TDBS program.  |
| Returns:       | An integer numeric value.  |
| Usage:         | NUSERS returns the number of users currently running a TDBS program. This number does not include users which may be logged on to TBBS but who are not running a TDBS program. |
| Example:       | ? NUSERS() && Result: 5  |
|                | ? NMYUSERS() && Result: 2  |
| See Also:      | NMYUSERS()   |
| Compatibility: | TDBS extended, no dBASE equivalent.  |

# OPTDATA()

| Syntax:        | OPTDATA()   |
|----------------|---|
| Purpose:       | Returns the OPT DATA string from the calling menu entry.  |
| Returns:       | A character string.   |
| Usage:         | This function allows a TDBS program to accept options from the<br>Opt Data line. The entire Opt Data line is returned, including the<br>TDBS program invocation. The Opt Data string which is returned<br>may be scanned by the TDBS program for program specific control<br>information. |
| Example:       | ? OPTDATA() && Result: String Prints  |
| Compatibility: | TDBS extended, no dBASE equivalent.   |

|                | OS()  |  |  |
|----------------|---|--|--|
| Syntax:        | OS()  |  |  |
| Purpose:       | Returns the name of the operating system.                                   |  |  |
| Returns:       | A character string.   |  |  |
| Usage:         | OS() returns the name and version of the operating system currently in use. |  |  |
| Example:       | ? OS() && Result: DOS 3.30  |  |  |
| See Also:      | GETENV(), VERSION()   |  |  |
| Compatibility: | dBASE standard, no extensions.  |  |  |

| PCOL | .() |
|------|-----|
|------|-----|

| Syntax:        | PCOL()  |
|----------------|---|
| Purpose:       | Returns the current printer column position.  |
| Returns:       | An integer numeric value.   |
| Usage:         | PCOL returns the column on the page where the next character<br>sent to the printer will be printed. It is used to keep track of the<br>printer position. Note: PCOL may become "out of sync" with the<br>printer if paper jams, or is not properly adjusted. PCOL is set to 0<br>with every top-of-form. |
| Example:       | SET PRINTER TO LPT1 && Assign LPT1 to us<br>SET DEVICE TO PRINT && Route @ SAY<br>@ 5,0 SAY "Line of"<br>@ 5, PCOL()+1 SAY "print"<br>SET DEVICE TO SCREEN<br>'SET PRINTER TO && Return printer   |
|                | Result: Line of print   |
| See Also:      | PROW(), COL(), ROW()  |
| Compatibility: | dBASE standard, no TDBS extensions.   |

# PROCLINE()

| Syntax:        | PROCLINE()  |
|----------------|---|
| Purpose:       | Returns the source code line of the current command line.   |
| Returns:       | An integer numeric value.   |
| Usage:         | PROCLINE will return the line number of the command line within<br>its own source file. This command will operate correctly whether<br>or not the /DB option was used on the TDBS compiler command<br>line. It can be used to locate a debug or error message within the<br>program code to ease debugging. |
| Example:       | <pre>? "Error at line", PROCLINE(), "In ", PROCNAME()</pre>   |
| See Also:      | PROCNAME()  |
| Compatibility: | TDBS extended, no dBASE equivalent.   |

#### PROCNAME()

| Syntax:<br>Purpose: | <b>PROCNAME()</b><br>Returns the name of the current procedure level.   |
|---------------------|---|
| Returns:            | A character string value.   |
| Usage:              | PROCNAME returns the name of the current TDBS procedure<br>level. It can be used to locate a debug or error message within the<br>program code to ease debugging. This procedure will return the<br>correct name regardless of whether or not the /DB option was used<br>on the TDBS compiler when this program was compiled. |
| Example:            | <pre>? "Error at line",PROCLINE(),"In ",PROCNAME()</pre>  |
| See Also:           | PROCLINE()  |
| Compatibility:      | TDBS extended, no dBASE equivalent.   |

## PROW()

| Syntax:        | PROW()   |                            |  |
|----------------|--|----------------------------|--|
| Purpose:       | Returns the current row (line) pos   | sition                     | n of the printer.  |
| Returns:       | An integer numeric value.  |                            |  |
| Usage:         | PROW returns the line on the p<br>printer will be printed on. It is us<br>ment and page sizes.   | ~                          |  |
|                | EJECT reset PROW() to zero.  |                            |  |
| Example:       | USE Invoices<br>SET PRINTER TO LPT1<br>SET PRINT ON<br>SET CONSOLE OFF<br>DO WHILE .NOT. EOF()<br>IF PROW() > 55<br>EJECT<br>ENDIF<br>? CustNo, PartNo, Qty<br>SKIP<br>ENDDO | 22<br>22<br>23<br>23<br>23 | Assign LPT1<br>?/?? to printer<br>and not to screen<br>Page the output |
|                | SET PRINT OFF  |                            | printer off  |
|                | SET CONSOLE ON<br>SET PRINTER TO   |                            | screen back on<br>Return printer                                       |
| See Also:      | PCOL(), ROW(), COL()   |                            |  |
| Compatibility: | dBASE standard, no extensions.   |                            |  |
|                |  |                            |  |

# RAT()

|                     | RAT()  |  |  |  |
|---------------------|--|--|--|--|
| Syntax:<br>Purpose: | RAT( <expc1>,<expc2>)<br/>Find the LAST occurrence of one string within another string.</expc2></expc1>  |  |  |  |
| Arguments:          | $< \exp C1 >$ is the character string to find.   |  |  |  |
|                     | $<\exp C2>$ is the character string to be searched for $<\exp C1>$ .   |  |  |  |
| Returns:            | An integer numeric value.  |  |  |  |
| Usage:              | If $\langle expC1 \rangle$ is contained within $\langle expC2 \rangle$ , RAT() returns the starting character position of the rightmost (last) occurrence of the string. If $\langle expC1 \rangle$ is not contained within $\langle expC2 \rangle$ , then RAT() returns a zero. |  |  |  |
|                     | RAT() is similar to the AT() function except that it finds the last instance of $\langle \exp C1 \rangle$ instead of the first.  |  |  |  |
| Compatibility:      | TDBS extended, no dBASE equivalent, Clipper compatible.  |  |  |  |

#### **READKEY()**

Syntax:

READKEY()

Purpose:

Returns an integer corresponding to the key pressed to exit a full screen READ command, and indicates whether changes were made to the data during that command.

Returns: An integer numeric value.

Usage: Depending on the value returned by READKEY(), you can determine what to do next after the user exits from a full screen READ. READKEY() returns one of two possible values from each keypress which can terminate a READ, depending on whether any data on the screen were altered as follows:

| Key Pressed                        | If no update | If update |
|------------------------------------|--------------|-----------|
| Left Arrow or Backspace (^H or ^S) | 0            | 256       |
| Right Arrow (^D)                   | 1            | 257       |
| ^Left Arrow (^A)                   | 2            | 258       |
| ^Right Arrow (^F)                  | 3            | 259       |
| Up Arrow (^E)                      | 4            | 260       |
| Down Arrow (^X)                    | 5            | 261       |
| Page Up (^R)                       | 6            | 262       |
| Page Down (^C)                     | 7            | 263       |
| Esc                                | 12           | 268       |
| ^END (^W)                          | 14           | 270       |
| Return (^M)                        | 15           | 271       |

Note that if data were altered during the read, 256 is added to the READKEY() code which is returned. The following example shows how READKEY() might be used. In this example, if the user types past the end of the last field the RETURN code (15) is returned. If no changes are made, then this example skips to the next record. If the user made any changes to the record, and then typed past the end of the screen, the routine prompts for any further changes. It also allows Page Up and Page Down to traverse the file.

#### **Chapter 5: TDBS Functions**

| xample:        | SET FORMAT TO SCrFmt          |                        |
|----------------|-------------------------------|------------------------|
|                | DO WHILE .NOT. EOF()          |                        |
|                | READ                          |                        |
|                | STORE READKEY() TO            | KitKey, Modified       |
|                | IF Modified < 256             |                        |
|                | Modified = $0$                |                        |
|                | ELSE                          |                        |
|                | XitKey = XitKey               | - 256                  |
|                | ENDIF                         |                        |
|                | DO CASE                       |                        |
|                | CASE XitKey = 6               |                        |
|                | SKIP -1                       | && Page Up             |
|                | LOOP                          |                        |
|                | CASE XitKey = 7               |                        |
|                | SKIP                          | && Next Page if        |
|                | LOOP                          | && Page Down           |
|                | OTHERWISE                     |                        |
|                | IF Modified >                 |                        |
|                | Decide = "                    |                        |
|                |                               | && Position cursor     |
|                |                               | re Changes?" TO Decide |
|                | IF Decide                     |                        |
|                | LOOP<br>ELSE                  | && Same page again     |
|                | SKIP                          | && Next Page           |
|                | LOOP                          |                        |
|                | ENDIF                         |                        |
|                | ENDIF                         |                        |
|                | SKIP                          | && Other exit, no chg  |
|                | LOOP                          | && next page           |
|                | ENDCASE                       |                        |
|                | ENDDO                         |                        |
|                | SET FORMAT TO                 | && Cancel screen fmt   |
| ee Also:       | ON KEY, READ, LASTKEY(        | ), INKEY()             |
| Compatibility: | dBASE standard, no extensions |                        |

|                | RECCOUNT()/LASTREC()  |
|----------------|---|
| Syntax:        | RECCOUNT() / LASTREC()  |
| Purpose:       | Returns the total number of records in the currently selected database file. This is also the last record number.   |
| Returns:       | An integer numeric value.   |
| Usage:         | RECCOUNT returns the number of records in a database file<br>without moving the record pointer. This count includes all records<br>regardless of the number of deleted records or any filter condition.<br>This may also be considered the record number of the last record<br>in the file. If the database is empty, a zero is returned. |
| Example:       | USE Customer<br>? RECCOUNT() && Result: 25000<br>DELETE<br>? RECCOUNT() && Result: 25000  |
| See Also:      | RECNO(), RECSIZE()  |
| Compatibility: | dBASE standard, no extensions.  |

#### RECNO()

| Syntax:        | RECNO()   |   |
|----------------|---|---|
| Purpose:       | Returns the number of the current database record.  |   |
| Returns:       | An integer numeric value.   |   |
| Usage:         | RECNO returns the current record pointer for the database file in<br>the currently selected work area. If the file contains no records, the<br>RECNO() returns 1 and both BOF() and EOF() return true (.T.).<br>If the record pointer is positioned past the last record in the file,<br>RECNO() returns the number of records in the file plus one, and<br>EOF() returns true (.T.). |   |
| Example:       | USE Invoices<br>GOTO 3<br>? RECNO() && Result: 3<br>SKIP<br>? RECNO() && Result: 4<br>GO TOP && Result: 1   | ( |
| See Also:      | RECCOUNT(), RECSIZE()   |   |
| Compatibility: | dBASE standard, no extensions.  |   |

# RECSIZE()

| Syntax:        | RECSIZE()   |
|----------------|---|
| Purpose:       | Returns the number of bytes in a single record in the currently selected database file.   |
| Returns:       | An integer numeric value.   |
| Usage:         | RECSIZE returns the size of a single record in the database file in<br>the currently selected work area. If no database file is open in the<br>selected work area, RECSIZE will return a zero.  |
| Example:       | USE DataFile<br>NumFields = X && X is number of fields<br>HeadSize = 32 * NumField + 35<br>NumRecs = (DISKSPACE()-HeadSize)/RECSIZE()<br>IF NumRecs < RECCOUNT()<br>WAIT "Not enough room?"<br>ELSE<br>COPY Datafile TO Backup<br>ENDIF |
| See Also:      | RECCOUNT(), DISKSPACE()   |
| Compatibility: | dBASE standard, no extensions.  |

## REPLICATE()

| Syntax:        | REPLICATE( <expc>,<expn>)</expn></expc>   |
|----------------|---|
| Purpose:       | Repeats a character string the specified number of times.   |
| Arguments:     | <expc> is the character string to be repeated.</expc>   |
|                | <expn> is the number of times to repeat <expc>.</expc></expn>   |
| Returns:       | A character string.   |
| Usage:         | REPLICATE returns a string which consists of $\langle \exp C \rangle$ repeated $\langle \exp N \rangle$ times. Note: if the resultant string is more than 254 characters in length, an error will result. If $\langle \exp N \rangle$ is negative or zero, a null string is returned. |
| Example:       | <pre>? REPLICATE("=",80) &amp;&amp; Result: dbl line<br/>? REPLICATE(CHR(219),80) &amp;&amp; Result: bar</pre>  |
| See Also:      | SPACE(), SUBSTR(), STUFF()  |
| Compatibility: | dBASE standard, no extensions.  |

# RIGHT()

| Syntax:        | RIGHT( <expc>,<expn>)</expn></expc>  |
|----------------|--|
| Purpose:       | Returns the specified number of characters from the right end of the specified character string.   |
| Arguments:     | $<\exp C>$ is the string from which to extract the characters.   |
|                | $<\exp N>$ is the number of characters to extract.   |
| Returns:       | A character string.  |
| Usage:         | RIGHT returns the rightmost $\langle expN \rangle$ characters from the string $\langle expC \rangle$ . If $\langle expN \rangle$ is negative or zero, then a null string is returned. If $\langle expN \rangle$ is larger than the number of characters in $\langle expC \rangle$ then the original $\langle expC \rangle$ string is returned. |
| Example:       | <pre>? RIGHT("Abcdefg",4) &amp;&amp; Result: defg ? RIGHT("A",4) &amp;&amp; Result: A</pre>  |
|                | ? RIGHT("A",4) && Result: A  |
| See Also:      | LEFT(), LTRIM(), RTRIM(), SUBSTR(), STUFF()  |
| Compatibility: | dBASE standard, no extensions.   |

# RJUST()

| Syntax:<br>Purpose: | RJUST( <expc>)<br/>Right justifies a character string.</expc>   |
|---------------------|---|
| Argument:           | <expc> is the character string to right justify.</expc>   |
| Returns:            | A character string.   |
| Usage:              | RJUST returns a string of the same length as $\langle \exp C \rangle$ with any trailing blanks moved to the beginning of the string. This right justifies any text in the input character string. |
| Example:            | <pre>? RJUST("ABC ") &amp;&amp; Result: ABC<br/>? RJUST("1 45") &amp;&amp; Result: 1 45<br/>? RJUST("12 ")+"X" &amp;&amp; Result: 12X</pre>   |
| See Also:           | LJUST(), LTRIM(), RTRIM(), SUBSTR(), STUFF()  |
| Compatibility:      | TDBS extended, no dBASE equivalent.   |

#### RLOCK()/LOCK()

Syntax: RLOCK() / LOCK() **Purpose:** Locks the current record in the current work area. **Returns:** A logical value. Usage: RLOCK() attempts to lock the current record in the current work area. It returns true (.T.) if the lock attempt succeeds. If the lock attempt fails, or if no file is open in the currently selected work area, then false (.F.) is returned. TDBS will attempt to lock any related work area records if a SET RELATION is pending. In this case either all locks succeed, or no locks result. Only one record may be locked at a time in a work area. Attempting to lock any record releases all locks which may be currently in place in that work area (and any related work area) regardless of the success or failure of the attempted lock. Closing a file or terminating a program removes any record locks this program has established. See Chapter 3 for a discussion of the TDBS multiuser Transparent File Sharing feature which usually removes the requirement for record and file locking, as well as a discussion of explicit record and file locking operation. Example: IF RLOCK() DELETE UNLOCK ENDIF See Also: SET EXCLUSIVE, USE, UNLOCK, WAIT4RLOCK(), FLOCK(), WAIT4FLOCK() **Compatibility:** dBASE standard, no extensions.

## ROUND()

| Syntax:        | ROUND( <expn1>,<expn2>)</expn2></expn1>  |                 |
|----------------|--|-----------------|
| Purpose:       | Returns a numeric value rounded to the specified numb decimal places.  | per of          |
| Arguments:     | <expn1> is the number to be rounded.</expn1>   |                 |
|                | $< \exp N2 >$ is the number of decimal places to round to.   |                 |
| Returns:       | A numeric value.   |                 |
| Usage:         | ROUND returns the value of $\langle expN1 \rangle$ rounded to the nu<br>of decimal places specified by $\langle expN2 \rangle$ . If the $\langle expN1 \rangle$<br>negative, then ROUND begins rounding in the integer port<br>the number as shown in the examples below. If $\langle expN2 \rangle$ is<br>the the number is rounded to the nearest integer value. | 2> is<br>ion of |
| Example:       | <pre>? ROUND(12.3456,3) &amp;&amp;&amp; Result: 12.346<br/>? ROUND(12.3456,2) &amp;&amp;&amp; Result: 12.35<br/>? ROUND(12.3456,1) &amp;&amp;&amp; Result: 12.3<br/>? ROUND(12.3456,0) &amp;&amp;&amp; Result: 12<br/>? ROUND(12.3456,-1) &amp;&amp;&amp; Result: 10<br/>? ROUND(12.3456,-2) &amp;&amp;&amp; Result: 0</pre>                                       |                 |
| See Also:      | SET DECIMALS, SET FIXED, INT(), FLOOR(), CEILI   | NG()            |
| Compatibility: | dBASE standard, no extensions.   |                 |

**Chapter 5: TDBS Functions** 

# ROW()

| Syntax:        | ROW()   |
|----------------|---|
| Purpose:       | Returns the current row position of the screen cursor.  |
| Returns:       | An integer numeric value.   |
| Usage:         | ROW() returns the row (line) position of the current screen cursor.<br>It is commonly used for screen relative cursor addressing. Note:<br>The cursor position which ROW() and COL() return is the cursor<br>position at the beginning of the command. Thus if the cursor moves<br>during the command, these values will always return the position of<br>the cursor at the beginning of the command. |
| Example:       | <pre>@ 10,5 SAY "This is line" @ ROW(),COL()+1 SAY ROW() PICTURE "99" Result: This is line 10</pre>   |
| See Also:      | @ SAY GET, COL( ), PCOL( ), PROW( )   |
| Compatibility: | dBASE standard, no extensions.  |

# RTRIM()/TRIM()

| Syntax:        | RTRIM( <expc>) / TRIM(<ex< th=""><th>pC&gt;)</th></ex<></expc>                 | pC>)  |
|----------------|--|---|
| Purpose:       | Removes any trailing blanks from   | the specified character string.                 |
| Argument:      | <expc> is the string from which</expc>   | to remove trailing blanks.                      |
| Returns:       | A character string.  |   |
| Usage:         | RTRIM returns a copy of < expC blanks removed.                                 | > with any trailing (right most)                |
| Example:       | <pre>string = "1234 " ? LEN(string) ? RTRIM(string) ? LEN(RTRIM(string))</pre> | && Result: 8<br>&& Result: 1234<br>&& Result: 4 |
| See Also:      | LTRIM(), SUBSTR(), STUFF(  | )   |
| Compatibility: | dBASE standard, no extensions.   |   |

## SECONDS()

| Syntax:        | SECONDS()  |
|----------------|--|
| Purpose:       | Returns the number of seconds since midnight.  |
| Returns:       | A numeric value.   |
| Usage:         | SECONDS() returns the system time as a numeric value. This value is the number of seconds since midnight to the nearest hundredth of a second. The range is from 0.00 to 86399.99 seconds. |
| Example:       | ? TIME() && Result: 10:00:00   |
|                | ? SECONDS() && Result: 36000.00  |
| See Also:      | TIME()   |
| Compatibility: | TDBS extended, no dBASE equivalent.  |



## SELECT()

ž

| Syntax:        | SELECT()   |
|----------------|--|
| Purpose:       | Returns the currently selected work area number.                 |
| Returns:       | An integer numeric value.  |
| Usage:         | SELECT() returns the number of the currently selected work area. |
| Example:       | SELECT 4<br>? SELECT() && Result: 4                              |
| See Also:      | SELECT, USE, ALIAS()   |
| Compatibility: | TDBS extended, no dBASE equivalent.                              |

# SETPRC()

| Syntax:        | SETPRC( <expn1>,<expn2>)</expn2></expn1>  |
|----------------|---|
| Purpose:       | Sets the internal PROW() and PCOL() to the specified values.  |
| Arguments:     | <expn1> is the new internal value for PROW().</expn1>   |
|                | <expn2> is the new internal value for PCOL().</expn2>   |
| Returns:       | A logical .F.   |
| Usage:         | SETPRC() is useful when you send a setup string to a printer<br>without changing the head position. In addition this function can<br>be used to suppress page ejects or compensate for other special<br>printer conditions. |
| Compatibility: | TDBS extended, no dBASE equivalent, Clipper compatible.   |

#### SOUNDEX()

| Syntax:   | SOUNDEX( <expc>)</expc>   |   |
|-----------|---|---|
| Purpose:  | Obtains a phonetic match (or "sounds like") code for a keyword.   |   |
| Argument: | <expc> is the character string to obtain a SOUNDEX code for.</expc>   |   |
| Returns:  | A character string.   |   |
| Usage:    | SOUNDEX() returns a phonetic match or "sound-alike" character<br>code of the form "letter digit digit digit" for an input string. Strings<br>which produce the same soundex code will tend to sound like each<br>other. This four character code may be used to do "fuzzy" searches<br>or indexing for words or names that sound similar. The soundex<br>code generation rules are: |   |
|           | 1. Leading blanks in the input string are ignored.  |   |
|           | 2. The upper case of the first letter in the string becomes the first character of the four character output string. If the first non-blank in the string is not a letter the code "0000" is returned.  | ( |
|           | 3. After the first letter, the letters A, E, H, I, O, U, W, and Y are ignored in producing the code.  |   |
|           | 4. The remaining letters are assigned a code as follows:<br>B, F, P, and V = 1<br>C, G, J, K, Q, S, X, and Z = 2<br>D and T = 3<br>L = 4<br>M and N = 5<br>R = 6  |   |
|           | 5. The code for the next letter is added to the output string un-   |   |

5. The code for the next letter is added to the output string unless it is a repeat of the code of the previous source string character in which case it is ignored.

6. The scan stops at the first non-alpha character (including blank) and the code is padded with ASCII "0" if it comes up short.

| Example:       | USE Customer<br>INDEX ON SOUNDEX(Fname) TO Fnsdx<br>USE Customer INDEX Fnsdx<br>SEEK SOUNDEX("Bill")<br>? FOUND(), Fname &&Result: .T. Bill<br>SEEK SOUNDEX("Billy")<br>? FOUND(), Fname &&Result: .T. Bill |
|----------------|---|
| Compatibility: | TDBS extended, no dBASE III Plus equivalent,<br>Clipper similar, dBASE IV compatible.   |

## SPACE()

| Syntax:        | SPACE( <expn>)</expn>   |
|----------------|---|
| Purpose:       | Returns a string of spaces of a specified length.   |
| Argument:      | $< \exp N >$ is the length of the string to return.   |
| Result:        | A character string.   |
| Usage:         | SPACE creates a string of $\langle expN \rangle$ spaces and returns it. If $\langle expN \rangle$ is negative or zero then a null string is returned. If $\langle expN \rangle$ is greater than 254, then an error will result. |
| Example:       | Name = SPACE(20)<br>@ 10,20 SAY "Enter Name: " GET Name<br>READ   |
|                | This example uses SPACE() to create a character variable of the desired length for GET and READ to operate on.  |
| See Also:      | REPLICATE()   |
| Compatibility: | dBASE standard, no extensions.  |

# SQRT()

| Syntax:<br>Purpose: | SQRT( <expn>)<br/>Returns the square root of the specified number.</expn>   |                          |                 |
|---------------------|---|--------------------------|-----------------|
| Argument:           | <expn> is the number to extract the square root of.</expn>  |                          |                 |
| Returns:            | A numeric value.  |                          |                 |
| Usage:              | SQRT calculates the square root of $\langle expN \rangle$ and returns it as a numeric value. If $\langle expN \rangle$ is negative, an error results. |                          |                 |
| Example:            | ? SQRT(25)<br>? SQRT(2)   | && Result:<br>&& Result: | 5<br>1.41421356 |
| See Also:           | SET DECIMAL, SET FIXED, ABS()   |                          |                 |
| Compatibility:      | dBASE standard, no extensions.  |                          |                 |

.

.

#### STATENAME()

| Syntax:        | STATENAME( <expc>)</expc>   |  |  |
|----------------|---|--|--|
| Purpose:       | Returns the full name for a US post office state abbreviation.  |  |  |
| Argument:      | $< \exp C >$ is the abbreviation to return the full state name for.   |  |  |
| Returns:       | A character string.   |  |  |
| Usage:         | STATENAME returns the full state name for the abbreviation<br>given by $\langle \exp C \rangle$ . If $\langle \exp C \rangle$ is not exactly two characters in<br>length, or is not a valid US post office state abbreviation, then a null<br>string is returned. The first letter of a state name is upper case, and<br>the remainder of the letters in the name are lower case. The case<br>of $\langle \exp C \rangle$ is not significant. |  |  |
| Example:       | <pre>? STATENAME("Co") &amp;&amp; Result: Colorado<br/>? STATENAME("NY") &amp;&amp; Result: New York<br/>? STATENAME("ut") &amp;&amp; Result: Utah<br/>? STATENAME("Minn") &amp;&amp; Result: Null Str</pre>  |  |  |
| See Also:      | ISSTATE()   |  |  |
| Compatibility: | TDBS extended, no dBASE equivalent.   |  |  |

| ST | R() |
|----|-----|
|----|-----|

| Syntax:        | STR( <expn1>[,<expn2>[,<expn3>]])</expn3></expn2></expn1>  |        |               |               |  |
|----------------|--|--------|---------------|---------------|--|
| Purpose:       | Converts a numeric value into a t  | ext st | tring display | equivalent.   |  |
| Arguments:     | <expn1> the numeric value to</expn1>   | conv   | ert to a char | acter string. |  |
|                | <expn2> the length of the string to return, including all digits, decimal point, and minus sign.</expn2>   |        |               |               |  |
|                | <expn3> the number of decimation of decimati</expn3> | al pla | ces to displa | ay.           |  |
| Usage:         | STR returns a string with the display representation of $\langle expN1 \rangle$ as a decimal number. The length argument $\langle expN2 \rangle$ sets the total length of the string to be returned, and the decimal argument $\langle expN3 \rangle$ sets the number of decimal places to be included. If the length is omitted, the default of 10 is used, and if decimal places is omitted, the default is 0. If you specify the length as smaller than that which is required to contain the number, then asterisks are returned in the digit positions to indicate field overflow occurred. The converted display number is rounded to the last displayed digit.  |        |               |               |  |
|                |  |        |               |               |  |
| Example:       | Number = $1234.56$   |        | 0.000         |               |  |
|                | ? STR(Number)  |        | Result:       |               |  |
|                | ? STR(Number, 7, 2)  |        | Result:       |               |  |
|                | ? STR(Number, 8, 3)  |        |               | 1234.560      |  |
|                | ? STR(Number, 6, 1)  | 88     | Result:       | 1234.6        |  |
| See Also:      | TRANSFORM(), VAL()   |        |               |               |  |
| Compatibility: | dBASE standard, no extensions.   |        |               |               |  |
|                |  |        |               |               |  |

# STUFF()

| Syntax:<br>Purpose: | STUFF( <expc1>,<expn1>,<expn2>,<expc2>)<br/>Combines two character strings to produce a third character string.</expc2></expn2></expn1></expc1>   |
|---------------------|---|
| Arguments:          | <expc1> is the target character string.</expc1>   |
|                     | $< \exp N1 >$ is the starting position in the target for replacement.   |
|                     | <expn2> is the number of characters to replace.</expn2>   |
|                     | $< \exp C2 >$ is the replacement string.  |
| Returns:            | A character string.   |
| Usage:              | STUFF replaces $\langle expN2 \rangle$ characters in $\langle expC1 \rangle$ beginning at<br>position $\langle expN2 \rangle$ with the string $\langle expC2 \rangle$ . The first number<br>determines where to stuff the second string into the first, and the<br>second number determines how many characters from the first<br>string are overwritten. If the second number is zero or negative, no<br>characters are overwritten. |
| Example:            | <pre>Target = "This is a long sentence"<br/>Bullet = "<inserted>"<br/>? STUFF(Target,5,0,Bullet)<br/>* Result: This<inserted> is a long sentence<br/>? STUFF(Target,5,0," "+Bullet)<br/>* Result: This <inserted> is a long sentence<br/>? STUFF(Target,6,4,Bullet)<br/>* Result: This <inserted> long sentence</inserted></inserted></inserted></inserted></pre>   |
| See Also:           | AT(), LEFT(), RIGHT(), SUBSTR()   |
| Compatibility:      | dBASE standard, no extensions.  |

# SUBSTR()

| ) | Syntax:        | SUBSTR( <expc>,<expn1>[,<expn2>])</expn2></expn1></expc>   |
|---|----------------|--|
|   | Purpose:       | Extracts a portion of a string and returns it as a new string.   |
|   | Arguments:     | <expc> is the source character string.</expc>  |
|   |                | <expn1> is the starting character of the substring to extract.</expn1>   |
|   |                | $< \exp N2 >$ is the length of the substring.  |
|   | Returns:       | A character string.  |
|   | Usage:         | SUBSTR extracts $< \exp N2 >$ characters from the string $< \exp C >$ beginning at character $< \exp N1 >$ . If $< \exp N2 >$ is omitted, then the string begins at $< \exp N1 >$ and continues to the end of the source string. |
| ) | Example:       | String = "Your dog has fleas"<br>? SUBSTR(String,6,3) && Result: dog   |
|   |                | <pre>Week = "Monday Tuesday WednesdayThursday " Week = Week+"Friday " Day = 1 DO WHILE Day &lt;= 5     ? SUBSTR(Week,(Day-1)*9+1,9)     Day = Day+1 ENDDO</pre>  |
|   |                | Result: Monday<br>Tuesday<br>Wednesday<br>Thursday<br>Friday   |
| ) | See Also:      | AT(), RIGHT(), LEFT(), STUFF()   |
|   | Compatibility: | dBASE standard, no extensions.   |
|   |                |  |

#### TIME()

| Syntax:        | тіме()   |
|----------------|--|
| Purpose:       | Returns the current system time as a character string.   |
| Returns:       | A character string.  |
| Usage:         | TIME returns a character string in the format "hh:mm:ss". Because<br>TIME() returns a character string, you cannot do time arithmetic<br>directly. You may either convert the string to a number (as shown<br>in the example below) or use the SECONDS() function to get the<br>time directly in numeric form. |
| Example:       | <pre>? TIME() &amp;&amp; Result: 17:29:30 CTime = TIME() Seconds = VAL(LEFT(CTime,2)*3600+;</pre>  |
|                | This example converts the current time to seconds since midnight.  |
| See Also:      | DATE(), SECONDS()  |
| Compatibility: | dBASE standard, no extensions.   |

## TRANSFORM()

| Syntax:        | TRANSFORM( <exp>,<expc>)</expc></exp>  |
|----------------|--|
| Purpose:       | Returns a character string which is the value of $\langle \exp \rangle$ formatted according to the PICTURE contained in $\langle \exp C \rangle$ .   |
| Arguments:     | $< \exp >$ is an expression of any type to convert to a character string and format.   |
|                | <expc> is the PICTURE format which controls the conversion.</expc>   |
| Returns:       | A character string.  |
| Usage:         | TRANSFORM() takes the result of an expression of any data type<br>and returns a formatted character string according to the PIC-<br>TURE functions and/or template specified in <expc>.</expc>         |
|                | TRANSFORM() follows the same rules for PICTURE conversions as the @ SAY command. See this command for details on the PICTURE functions and template for conversion.                                    |
|                | Since TRANSFORM returns a character string, it allows you to do formatted output with the ? and ?? commands, and thus use formatted output when the user does not have ANSI = YES in the TBBS profile. |
| Example:       | Amount = 12345.67<br>Picture = "999,999,999.99"<br>? TRANSFORM(Amount,Picture)<br>* Result: 12,345.67  |
| See Also:      | @ SAY PICTURE, STR( ), LOWER( ), UPPER( ),<br>LJUST( ), RJUST( ), UANSI( )   |
| Compatibility: | dBASE standard, no extensions.   |

#### TYPE()

| Syntax:  | TYPE( <expc>)</expc>   | ( |  |  |
|--|--|---|--|--|
| Purpose:   | Determine the data type of the specified character expression.   |   |  |  |
| Argument:  | <exp> is a character expression which is the name of a database field, memory variable, or expression of any type.</exp> |   |  |  |
| Returns:   | A character string.  |   |  |  |
| <b>Usage:</b> TYPE returns the type of the specified expression as follows |  |   |  |  |
|  | C = character $N = numeric$ $D = date$ $L = logical$ $M = memo$  |   |  |  |
|  | A = array<br>U = undefined<br>UE = undefined array element   | ( |  |  |
|  | Arrays: References to array names will return an "A". References to array elements will return the type of the element.  |   |  |  |
| Example:   | STORE 0 TO memvar<br>? TYPE("memvar") && Result: N<br>STORE "test" TO memvar   |   |  |  |
|  | ? TYPE("memvar") && Result: C<br>STORE DATE() TO memvar  |   |  |  |
|  | ? TYPE("memvar") && Result: D<br>STORE .T. TO memvar   |   |  |  |
|  | ? TYPE("memvar") && Result: L<br>RELEASE memvar  |   |  |  |
|  | ? TYPE("memvar") && Result: U<br>DECLARE memvar[5]   |   |  |  |
|  | ? TYPE["memvar") && Result: A  | ( |  |  |
| Compatibility:   | dBASE standard, Clipper compatible array extension.  |   |  |  |

#### UANSI()

| Syntax:        | UANSI()   |
|----------------|---|
| Purpose:       | Determine the setting of the ANSI parameter in the user profile.  |
| Returns:       | A logical value.  |
| Usage:         | UANSI returns a logical true (.T.) if the user profile has ANSI set<br>to YES. Otherwise false (.F.) is returned.         |
| Example:       | <pre>IF UANSI()     @ 5,0 SAY "Line 5 of screen" ELSE     CLEAR     ? REPLICATE(CHR(13),4),"Line 5 of screen" ENDIF</pre> |
|                | This is an example of using ANSI and non-ANSI output based on the user's terminal profile.                                |
| See Also:      | UAUTH(), UIBM(), ULOCATION(), UMORE(),<br>UNAME(), UNOTES(), UPRIV(), UWIDTH()  |
| Compatibility: | TDBS extended, no dBASE equivalent.   |

# UAUTH()

| Syntax:        | UAUTH( <expn>)</expn>  | ( |
|----------------|--|---|
| Purpose:       | Returns the user's TBBS authorization flag settings.   |   |
| Argument:      | <expn> is a number from 1 to 4 to select A1, A2, A3, or A4.</expn>   |   |
| Returns:       | A character string.  |   |
| Usage:         | UAUTH returns an 8 character long string with the specified authorization flags. If $\langle \exp N \rangle$ is not 1, 2, 3, or 4 a null string is returned. |   |
|                | This function allows you to pass the TBBS user authorization to your TDBS program for whatever reasons you choose.   |   |
| Example:       | ? UAUTH(2) && Result: XX   |   |
| See Also:      | UANSI(), UIBM(), ULOCATION(), UMORE(), UNAME(),<br>UNOTES(), UPRIV(), UWIDTH()   | ( |
| Compatibility: | TDBS extended, no dBASE equivalent.  |   |
|                |  |   |

| U | IB | Μ | () |
|---|----|---|----|
| - |    |   | V  |

| Syntax:        | UIBM()  |
|----------------|---|
| Purpose:       | Determine the setting of the IBM GRAPHICS parameter in the user profile.  |
| Returns:       | A logical value.  |
| Usage:         | UIBM returns a logical true (.T.) if the user profile has IBM GRAPHICS set to YES. Otherwise false (.F.) is returned.   |
|                | Note: This is not required in general, since TBBS will automatically<br>convert graphics characters to ASCII equivalents if the user does<br>not have graphics set. However, if a program uses block fill<br>graphics etc. it may want to tailor the output based on this profile<br>setting. |
| Example:       | IF UIBM()<br>? "Graphics enabled"<br>ELSE<br>? "Graphics disabled"<br>ENDIF   |
| See Also:      | UANSI(), UAUTH(), ULOCATION(), UMORE(),<br>UNAME(), UNOTES(), UPRIV(), UWIDTH()   |
| Compatibility: | TDBS extended, no dBASE equivalent.   |

#### ULINE()

Syntax: ULINE()

Purpose: Returns the line number identifier.

**Returns:** A character string.

Usage: ULINE() Returns a two character string with the ASCII decimal value of the line indentifier. This will be in the range "00" to "64". ULINE() = "00" indicates the program is running on the local console.

Note: If this TDBS program is invoked with the /OU switch on the Opt Data line, ULINE() will return a single character string with the line identifier. This is in the range of 0 to W to indicate lines 0 through 32. Letters are always upper case for lines 10 through 32. Line 10 is A, 11 is B, etc. up to W. This operation is for compatibility with older compiled programs which ran on TBBS 2.1, and CANNOT BE USED on systems with more than 32 lines!

# ULOCATION()

| Syntax:<br>Purpose: | ULOCATION()<br>Returns the LOCATION field of the user's userlog record.                               |  |
|---------------------|---|--|
| Returns:            | A character string.   |  |
| Usage:              | ULOCATION returns the LOCATION field of the current user's TBBS userlog record as a character string. |  |
| Example:            | ? ULOCATION() && Result: AURORA, CO   |  |
| See Also:           | UANSI(), UAUTH(), UIBM(), UMORE(), UNAME(),<br>UNOTES(), UPRIV(), UWIDTH()                            |  |
| Compatibility:      | TDBS extended, no dBASE equivalent.   |  |

# ULPEEK()

| Syntax:<br>Purpose: | ULPEEK( <expn1>,<expn2>[,<expn3>])<br/>Allows reading any field in the user's TBBS userlog record.</expn3></expn2></expn1>   |  |
|---------------------|--|--|
| Arguments:          | $< \exp N1 >$ is the byte offset to the desired userlog field.   |  |
|                     | <expn2> is the format type of the field as follows:</expn2>  |  |
|                     | <ul> <li>1 - Byte Numeric</li> <li>2 - Word Numeric</li> <li>3 - Message Number Numeric</li> <li>4 - Double Word Numeric</li> <li>5 - Flag String Byte ("XXXXXXX" character format)</li> <li>6 - Character string literal</li> <li>7 - Date Format (3 bytes, MM DD YY in binary)</li> <li>&lt; expN3 &gt; if the format type is 6, this field specifies the number of</li> </ul> |  |
|                     | characters in the field. It has no meaning for other types.  |  |
| Returns:            | A numeric or character string value.   |  |
| Usage:              | ULPEEK() allows the program to read any field within the user's TBBS userlog record. The most common fields have their own functions, but ULPEEK allows access to any field desired.   |  |
| Example:            | The following sample will read the user's expiration date:   |  |
|                     | exp_date = ULPEEK(440,7)<br>? type("exp_date") && Result: "D"  |  |
| See Also:           | ULPOKE(), ULREPLACE()  |  |
| Compatibility:      | TDBS extended, no dBASE equivalent.  |  |

# ULPOKE()

| ULPOKE( <expn1>,<expn2>,<exp>[,<expn3>])<br/>Allows changing any field within the userlog record.</expn3></exp></expn2></expn1>  |
|--|
|  |
|  |
| $<\exp N1>$ is the byte offset to the desired userlog field.   |
| <expn2> is the format type of the field as follows:</expn2>  |
| 1 - Byte Numeric   |
| 2 - Word Numeric   |
| 3 - Message Number Numeric   |
| 4 - Double Word Numeric  |
| 5 - Flag String Byte ("XXXXXXX" character format)  |
| <ul><li>6 - Character string literal</li><li>7 - Date Format (3 bytes, MM DD YY in binary)</li></ul>   |
| 7 - Date Format (5 bytes, MM DD I I in binary)   |
| $< \exp >$ is an expression of the proper type (numeric or character)  |
| to match the specified format. This value is placed in the selected  |
| userlog record field in the specified format.  |
| <expn3> if the format type is 6, this field specifies the number of characters in the field. It has no meaning for other types.</expn3>  |
| A logical value.   |
| ULPOKE() allows any value in the user's TBBS userlog record to<br>be altered. It converts $\langle exp \rangle$ from its TDBS internal format to<br>the specified userlog record format and stores it in the userlog<br>record. ULPOKE() returns true (.T.) if the update was performed,<br>and false (.F.) if there was an error. |
| Note: Altering the Name field will cause the user to not be able to log on until the userlog is re-indexed!  |
| Caution!! you can destroy a userlog record and cause malfunction by incorrect use of this function!  |
|  |

#### **Chapter 5: TDBS Functions**

|                | Most alterations to the userlog record made with ULPOKE() will<br>not affect the control of the user until the next logon. |  |
|----------------|--|--|
| Example:       | dummy = ULPOKE(440,7,CTOD("02/02/91"))   |  |
|                | This example sets the user's expiration date to ground hog day 1991.   |  |
| Compatibility: | TDBS extension, no dBASE equivalent.   |  |

# ULREPLACE()

| Syntax:        | ULREPLACE( <field>[,<select>],<exp>)</exp></select></field>  |
|----------------|--|
| Purpose:       | Updates common TBBS userlog record fields for the current user.  |
| Arguments:     | <field> is the name of the userlog field as follows:</field>   |
|                | UANSI - Update ANSI profile setting (logical)<br>UIBMG - update Graphics profile setting (logical)<br>UWIDTH - update screen width setting (numeric)<br>UMORE - update lines per page setting (numeric)<br>ULOCATION - update location field (character)<br>UNOTES - update the notes field (character)<br>UPRIV - update user privilege setting (numeric)<br>UAUTH - update A1, A2, A3, or A4 flags (character "XXXX")<br>< select > only has meaning if < field > is UAUTH. It is 1, 2, 3, |
|                | or 4 to select the Authorization flag set to be updated.<br><exp> is the expression of the proper type which is to be placed<br/>in the specified field of the current user's TBBS userlog record.</exp>   |
| Returns:       | A logical .T.  |
| Usage:         | ULREPLACE() updates all userlog fields for which there is a specific read function. Updates of UPRIV and UAUTH take place immediately with this function.  |
| Example:       | <pre>dummy = ULREPLACE(ULOCATION, "Aurora, CO") dummy = ULREPLACE(UAUTH, 2, "XXX.")</pre>  |
| Compatibility: | TDBS extension, no dBASE equivalent.   |

# UMORE()

| Syntax:        | UMORE()  |  |
|----------------|--|--|
| Purpose:       | Returns the number of lines per display page from the user profile.  |  |
| Returns:       | A numeric value.   |  |
| Usage:         | UMORE returns the number of lines per page from the user's TBBS profile. If the user has the -more- feature of TBBS turned off, then a zero is returned.                               |  |
|                | TDBS does not do any automatic -more- handling. You may<br>program such a capability and through the use of this function<br>determine the number of lines to consider a display page. |  |
| Example:       | ? UMORE() && Result: 24  |  |
| See Also:      | UANSI(), UAUTH(), UIBM(), ULOCATION(), UNAME(),<br>UNOTES(), UPRIV(), UWIDTH()   |  |
| Compatibility: | TDBS extended, no dBASE equivalent.  |  |

# UNAME()

| Syntax:        | UNAME()  |  |
|----------------|--|--|
| Purpose:       | Returns the user's TBBS logon ID as a character string.  |  |
| Returns:       | A character string.  |  |
| Usage:         | UNAME returns the user's TBBS logon ID as a character string.<br>The form of this ID depends on the logon method in use. If the<br>logon method makes the ID the caller's name, then the full name is<br>returned. |  |
| Example:       | ? UNAME() && Result: PHIL BECKER   |  |
| See Also:      | UANSI(), UAUTH(), UIBM(), ULOCATION(), UMORE(),<br>UNOTES(), UPRIV(), UWIDTH()   |  |
| Compatibility: | TDBS extended, no dBASE equivalent.  |  |

# UNOTES()

| Syntax:<br>Purpose: | <b>UNOTES()</b><br>Returns the NOTES field from the user's TBBS userlog record.  |
|---------------------|--|
| Returns:            | A character string.  |
| Usage:              | UNOTES returns the user's TBBS userlog record NOTES field as<br>a character string. If the notes field is empty, a null string is<br>returned. |
| Example:            | ? UNOTES() && Result: Userlog Notes Field  |
| See Also:           | UANSI(), UAUTH(), UIBM(), ULOCATION(), UMORE(),<br>UNAME(), UPRIV(), UWIDTH()  |
| Compatibility:      | TDBS extended, no dBASE equivalent.  |

### **UPDATED()**

| Syntax:        | UPDATED()  |
|----------------|--|
| Purpose:       | Determines if a change was made to any of the pending GETS during the last READ.   |
| Returns:       | A logical value.   |
| Usage:         | Each time the READ command executes, it resets UPDATED() to false (.F.). Any change to a GET within the READ command will set UPDATED() to true (.T.). Thus after the READ is exited, UPDATED() will allow the program to tell if any variable or field was altered by the READ command. |
| Example:       | USE Accounts<br>m_id = id<br>@ 1,0 SAY "Enter new ID" GET m_id<br>READ<br>IF UPDATED()<br>REPLACE id WITH m_id<br>ENDIF  |
| Compatibility. | TDBS extended, no dBASE equivalent, Clipper compatible.  |

# UPPER()

| Syntax:        | UPPER( <expc>)</expc>   |                          |
|----------------|---|--------------------------|
| Purpose:       | Converts any letters in a string to   | upper case.              |
| Argument:      | <expc> is the character string t</expc>   | o convert to upper case. |
| Returns:       | A character string.   |                          |
| Usage:         | UPPER returns a copy of $\langle \exp C \rangle$ with any letters forced to upper case. All other characters are left as they were. |                          |
| Example:       | ? UPPER("string")   | && Result: STRING        |
|                | ? UPPER("5 Chars")  | && Result: 5 CHARS       |
| See Also:      | ISLOWER(), ISUPPER(), LOWER()   |                          |
| Compatibility: | dBASE standard, no extensions.  |                          |

# UPRIV()

| Syntax:        | UPRIV()  |  |
|----------------|--|--|
| Purpose:       | Returns the user's TBBS PRIV value as a number from 0 to 255.  |  |
| Returns:       | An integer numeric value.  |  |
| Usage:         | UPRIV returns the user's PRIV value from the TBBS userlog record. This is number in the range 0 to 255. This may be used to pass the TBBS security level to your TDBS program. |  |
| Example:       | ? UPRIV() && Result: 150   |  |
| See Also:      | UANSI(), UAUTH(), UIBM(), ULOCATION(), UMORE(),<br>UNAME(), UNOTES(), UWIDTH()   |  |
| Compatibility: | TDBS extended, no dBASE equivalent.  |  |

### USING()

Syntax: USING([ < expN > ])**Purpose:** To determine which lines are currently sharing a database or mailbox work area. Argument: <expN> is an optional work area number. If this argument is omitted the current work area is used. **Returns:** A 65 character long string with an "X" for every user which has this file open, and a "." for every user who does not. Usage: The USING() function is used to obtain a complete list of all users currently sharing the DBF file open in the specified work area. This function will work on either mailbox or normal database file work areas. This function will always return a 65 character string (to allow future growth of TBBS up to 64 lines plus the console) and each character represents a line. The first character represents line 0 (the local console). If a line is sharing this file, it will show an X. Lines which do not exist on this installation always show a period indicating they are not sharing the file. **Compatibility:** TDBS extended, no dBASE equivalent.

# UWIDTH()

| Syntax:        | UWIDTH()  |  |
|----------------|---|--|
| Purpose:       | Returns the number of characters per line from the user profile.  |  |
| Returns:       | An integer numeric value.   |  |
| Usage:         | UWIDTH returns the user's configured characters per line. TDBS does not do any automatic word wrapping. You may use this function to write your TDBS program to adjust to different user screen widths. |  |
| Example:       | ? UWIDTH() && Result: 80  |  |
| See Also:      | UANSI(), UAUTH(), UIBM(), ULOCATION(), UMORE(),<br>UNAME(), UNOTES(), UPRIV()   |  |
| Compatibility: | TDBS extended, no dBASE equivalent.   |  |

# VAL()

| Syntax:        | VAL( <expc>)</expc>   |
|----------------|---|
| Purpose:       | Converts a character string number to a numeric value.  |
| Argument:      | <expc> is the character string number to convert.</expc>  |
| Returns:       | A numeric value.  |
| Usage:         | VAL converts a character string number to a numeric value. If the argument contains leading numeric characters followed by non-numeric characters, VAL converts the leading numeric characters to a number. If the argument consists of leading non-numeric characters, VAL will return a zero. |
| Example:       | Address = "15200 E. Girard Avenue"<br>? VAL(Address) && Result: 15200<br>? VAL(" 12.5") && Result: 12.5<br>? VAL(" 12.5") && Result: 12   |
| See Also:      | SET DECIMALS, STR()   |
| Compatibility: | dBASE standard, no extensions.  |

# VERSION()

| Syntax:        | VERSION()  |
|----------------|--|
| Purpose:       | Returns the version number of TDBS.  |
| Returns:       | A character string.  |
| Usage:         | VERSION returns a character string with the version of TDBS which is running your program. This will be the version of TDBSOM, not the compiler which compiled your program. |
| Example:       | ? VERSION() && Result: TDBS Version 1.2  |
| Compatibility: | dBASE standard, modified.  |

### WAIT4FLOCK()

| Syntax:        | WAIT4FLOCK([ <expn>])</expn>  |
|----------------|---|
| Purpose:       | Waits for file lock, key press, or timeout.   |
| Argument:      | $<\exp N>$ is the number of seconds to wait without being able to lock the file. If omitted, then the wait is unlimited by time.  |
| Returns:       | A logical value.  |
| Usage:         | WAIT4FLOCK attempts to lock the file in the current work area.<br>If it is successful, a true (.T.) is returned. If it fails, it will automat-<br>ically retry the operation every 500 milliseconds until it either<br>succeeds or one of the following occur:  |
|                | • A key is pressed by the user. Note: if SET TYPEAHEAD 0 is<br>in effect, this cannot occur. The key is not read and is still in<br>the typeahead buffer when the WAIT4FLOCK function<br>returns.   |
|                | • If < expN> is not zero, and < expN> seconds elapses.  |
|                | This function allows an extended attempt for file locking to occur<br>with almost zero system impact. The normal tight loop waiting for<br>a file lock, leaves the program in a 100% CPU bound loop as well<br>as being more complex to code. While TBBS schedules this sort of<br>wait well, the system impact is reduced by using WAIT4FLOCK. |
| Example:       | IF WAIT4FLOCK(60)<br>ZAP<br>UNLOCK<br>ELSE<br>? "Cannot Lock File Now"<br>ENDIF   |
| See Also:      | UNLOCK, FLOCK(), RLOCK(), WAIT4RLOCK()  |
| Compatibility: | TDBS extended, no dBASE equivalent.   |

# WAIT4LPT()

| Syntax:        | WAIT4LPT( <expn1>[,<expn2>])</expn2></expn1>  |
|----------------|---|
| Purpose:       | Waits for printer assignment, key press, or timeout.  |
| Argument:      | $<\exp N1>$ is the number of the printer (1, 2, 3, or 4) which you want assigned to your program.   |
|                | $<\exp N2>$ is the number of seconds to wait without being able to acquire the printer. If omitted, then the wait is unlimited by time.   |
| Returns:       | A logical value.  |
| Usage:         | WAIT4LPT requests the specified printer. If it is successful, a true (.T.) is returned. If it fails, it will automatically retry the operation every 500 milliseconds until it either succeeds or one of the following occur:   |
|                | • A key is pressed by the user. Note: if SET TYPEAHEAD 0 is<br>in effect, this cannot occur. The key is not read and is still in<br>the typeahead buffer when the WAIT4LPT function returns.  |
|                | • If <expn2> is not zero, and <expn2> seconds elapses.</expn2></expn2>  |
|                | This function allows an extended attempt to request a printer with<br>almost zero system impact. The normal tight loop waiting for a<br>printer leaves the program in a 100% CPU bound loop as well as<br>being more complex to code. While TBBS schedules this sort of<br>wait well, the system impact is reduced by using WAIT4LPT. |
| Example:       | IF WAIT4LPT(1,60)<br>COPY File TO PRN<br>ELSE<br>? "LPT1 in use by another program."<br>ENDIF   |
| See Also:      | SET PRINTER TO, GETLPT()  |
| Compatibility: | TDBS extended, no dBASE equivalent.   |
|                |   |

# WAIT4MAIL()

| WAIT4MAIL([ <expn>])</expn>  |   |
|--|---|
| Waits for new mail, key press, or timeout.   |   |
| <expn> is the number of seconds to wait without receiving new mail. If omitted, then the wait is unlimited by time.</expn>   |   |
| A logical value.   |   |
| WAIT4MAIL waits until new mail is received in the current work<br>area. If it is successful, a true (.T.) is returned. If it fails, it will<br>automatically retry the operation every 500 milliseconds until it<br>either succeeds or one of the following occur:   |   |
| • A key is pressed by the user. Note: if SET TYPEAHEAD 0 is<br>in effect, this cannot occur. The key is not read and is still in<br>the typeahead buffer when the WAIT4MAIL function returns.  |   |
| • If <expn> is not zero, and <expn> seconds elapses.</expn></expn>   |   |
| This function allows an extended wait for received mail to occur<br>with almost zero system impact. A tight loop waiting for mail,<br>leaves the program in a 100% CPU bound loop as well as being<br>more complex to code. While TBBS schedules this sort of wait well,<br>the system impact is reduced by using WAIT4MAIL. |   |
| IF WAIT4MAIL(60)<br>? "New mail received"<br>ELSE<br>? "No new mail now"<br>ENDIF  |   |
| USE MAILBOX, NEWMAIL( )  |   |
| TDBS extended, no dBASE equivalent.  |   |
|  | <ul> <li>Waits for new mail, key press, or timeout.</li> <li><expn> is the number of seconds to wait without receiving new mail. If omitted, then the wait is unlimited by time.</expn></li> <li>A logical value.</li> <li>WAIT4MAIL waits until new mail is received in the current work area. If it is successful, a true (.T.) is returned. If it fails, it will automatically retry the operation every 500 milliseconds until it either succeeds or one of the following occur:</li> <li>A key is pressed by the user. Note: if SET TYPEAHEAD 0 is in effect, this cannot occur. The key is not read and is still in the typeahead buffer when the WAIT4MAIL function returns.</li> <li>If &lt; expN&gt; is not zero, and &lt; expN&gt; seconds elapses.</li> <li>This function allows an extended wait for received mail to occur with almost zero system impact. A tight loop waiting for mail, leaves the program in a 100% CPU bound loop as well as being more complex to code. While TBBS schedules this sort of wait well, the system impact is reduced by using WAIT4MAIL.</li> <li>IF WAIT4MAIL(60)     <ul> <li>* "New mail received"</li> <li>ELSE         <ul> <li>* "No new mail now"</li> <li>ENDIF</li> </ul> </li> <li>USE MAILBOX, NEWMAIL()</li> </ul></li></ul> |

# WAIT4RLOCK()

| Syntax:        | WAIT4RLOCK([ <expn>])</expn>   |
|----------------|--|
| Purpose:       | Waits for record lock, key press, or timeout.  |
| Argument:      | $<\exp N>$ is the number of seconds to wait without being able to lock the record. If omitted, then the wait is unlimited by time.   |
| Returns:       | A logical value.   |
| Usage:         | WAIT4RLOCK attempts to lock the current record in the current<br>work area. If it is successful, a true (.T.) is returned. If it fails, it will<br>automatically retry the operation every 500 milliseconds until it<br>either succeeds or one of the following occur:   |
|                | • A key is pressed by the user. Note: if SET TYPEAHEAD 0 is<br>in effect, this cannot occur. The key is not read and is still in<br>the typeahead buffer when the WAIT4RLOCK function<br>returns.  |
|                | • If <expn> is not zero, and <expn> seconds elapses.</expn></expn>   |
|                | This function allows an extended attempt for record locking to<br>occur with almost zero system impact. A tight loop waiting for a<br>record lock, leaves the program in a 100% CPU bound loop as well<br>as being more complex to code. While TBBS schedules this sort of<br>wait well, the system impact is reduced by using WAIT4RLOCK. |
| Example:       | IF WAIT4RLOCK(60)<br>DELETE<br>UNLOCK<br>ELSE<br>? "Cannot Lock Record Now"<br>ENDIF   |
| See Also:      | UNLOCK, FLOCK(), RLOCK(), WAIT4FLOCK()   |
| Compatibility: | TDBS extended, no dBASE equivalent.  |

# YEAR()

| Syntax:<br>Purpose: | YEAR( <expd>)<br/>Extracts the year from the</expd> | specified date value.                                |
|---------------------|---|--|
| Argument:           | <expd> is the date value</expd>                     | e from which to extract the year.                    |
| Returns:            | A numeric value.                                    |  |
| Usage:              | YEAR returns the year century digits.               | from the specified date, including the               |
| Example:            | ()  | && Result: 07/17/89<br>&& Result: 1989               |
| See Also:           | SET CENTURY, CDOW<br>DAY(), DOW(), DTOC             | (), CMONTH(), CTOD(), DATE(),<br>(), DTOS(), MONTH() |
| Compatibility:      | dBASE standard, no exter                            | nsions.  |

# TECHNICAL

TECHNICAL

TECHNICAL

#### How TDBS handles loss of carrier

In some applications it can be important to know exactly what happens when a TDBS program is running and an unexpected disconnect occurs. An unexpected disconnect is either a loss of carrier, a user interrupt via an < Escape > key and an abort, or an operator shutdown of the user.

An unexpected disconnect can occur between any two TDBS instructions. It cannot occur during an instruction cycle and thus any instruction will complete fully except in the case of a repetitive instruction which operates on several file records. These instructions (e.g. APPEND FROM, REPLACE ALL etc.) will complete the full operation on the current database record cycle before a disconnect is honored. Thus for this type of instruction an unexpected disconnect is treated as though a WHILE CONNECT = .T. condition were part of the instruction, and failed on the next record processed after the disconnect. (Note: No such condition can actually be coded, but repetitive instructions act as if such a condition is always coded).

When an unexpected disconnect occurs in the absence of an ON DISCONNECT command, all open files are closed and any unwritten file buffers are written to disk. This makes the full effect of an unexpected disconnect the same as if a QUIT instruction had been placed after the last instruction executed. Thus file integrity is always assured.

Because disconnects can occur between any two instruction cycles, if a transaction is done in more than one REPLACE instruction the possibility of a partial transaction exists. If a transaction is done either with a single REPLACE command or via the @ ... GET and READ commands, then no partial transaction is possible. Either the entire transaction will occur properly, or it will all be discarded if an unexpected disconnect occurs.

If explicit cleanup is required, the program may use the command ON DISCONNECT to force execution of a cleanup routine in the case of a disconnection.

#### dBASE File Compatibility

The following file types used by TDBS uses are 100% compatible with dBASE III +:

.DBF .NDX .MEM .PRG .FMT

Compatible in this case means that TDBS and dBASE III + may use the files interchangeably with no problems. It does not mean that the files created by each are always identical, but that if a file is written by one program, it may be read or modified by the other with correct results.

This means that the same database may be modified by both TDBS and dBASE III + programs without any file conversions required.

Caution! It is not permissible to attempt to access the same database files with both TDBS and dBASE III + at the same time via a multitasker or network. This will result in file damage!

The following formats describe the files as they are created by TDBS. No representation is made that these formats are *identical* with those created by dBASE III +, however they are always correctly interpreted by dBASE III + as well as TDBS.

#### **TDBS**.DBF File Format

The TDBS .DBF database file contains a header followed by individual data records. Data records are always in ASCII and have a length and format which is defined by the field descriptors in the header. The header record begins with a 32 byte field which contains the fixed header information as follows:

#### **Fixed Header Format**

| Byte 0      | Bit 7:      | 1 if Memo File (.DBT) associated                  |
|-------------|-------------|---|
|             | Bits 4-6:   |   |
|             | Bit 3:      | 1 if dBASE IV style memo fields. TDBS             |
|             |             | requires a 0 here.                                |
|             | Bits 0-2:   | Program ID byte. TDBS puts a 03 here if it        |
|             |             | created the file.                                 |
| Byte 1      |             | Year of last update                               |
| Byte 2      |             | Month of last update                              |
| Byte 3      |             | Day of last update                                |
| Bytes 4     | I-7         | Long Integer number of records in file            |
| Bytes 8     | 8-9         | Integer number of bytes in header                 |
|             |             |   |
| Bytes 1     | <b>0-11</b> | Integer number of bytes per data record           |
| Bytes 12-31 |             | Reserved. TDBS always makes 0, but allows         |
|             |             | anything to be here. While TDBS has a file        |
|             |             | open, it stores file sharing information in this  |
|             |             | area in bytes 12, 13, 29, 30, and 31. These bytes |
|             |             | are set back to zero when TDBS closes the file.   |

Following this fixed portion of the header, is an array of field descriptor entries. There is one such entry for each field defined in the DBF file. The order of these entries corresponds to the the order of the fields in the DBF definition.

#### **Header Field Descriptor Format**

This array of field descriptors defines how each ASCII data record in the file is to be interpreted. The structure of a 32 byte data field descriptor entry is as follows:

| Bytes 0-10  | Field Name String. Left Justified, Null Filled.   |
|-------------|---|
| Byte 11     | Field Type as follows:  |
|             | N = Numeric:<br>ASCII digits, right justified, left padded with<br>spaces. Decimal point embedded per the<br>descriptor's decimals specification.       |
|             | C = Character:<br>ASCII data, left justified, right padded with<br>spaces. Not null terminated.   |
|             | L = Logical:<br>A single character from the set TtFfYyNn.   |
|             | D = Date:<br>Stored as ASCII YYYYMMDD.  |
|             | M = Memo:<br>A 10-digit ASCII number giving the block<br>number of the associated text within the .DBT<br>memo text file. (Not implemented in TDBS 1.0) |
| Bytes 12-15 | Reserved. TDBS makes 0, allows anything.  |
| Bytes 16-17 | Integer field width.  |
| Byte 18-19  | Integer number of decimal places.   |
| Byte 20-31  | Reserved. TDBS makes 0, allows anything.  |

The array of field descriptors is followed by a single ASCII carriage return (0Dh) to indicate the end of the DBF file header.

Data records are preceded by a single character which is either an ASCII blank or asterisk (\*). An asterisk indicates that this record has been marked for deletion, while a space indicates it has not been marked for deletion.

Following this indicator byte are the actual data bytes for each record as described by the header field descriptor array.

#### **TDBS**.NDX File Format

Each Data Base File (DBF) may be associated with up to seven index files. These files allow records to be located quickly based on the key which the index files catalogs. Each NDX file contains a unique entry for each DBF data record which it indexes. This entry contains the associated DBF record's key data value, and its DBF record number and is known as the "level 0 leaf" of the index file structure.

In addition, there may be higher level index structures in an NDX file to allow an individual key expression value to be located quickly. These records are known as level 1 through level n leaf records. In the dBASE III + index structure, there may be a maximum of 16 levels of such leaves, but normally there are only a very few levels to the structure. The number of levels indicates the worst case search time to find a key. For example if there are a maximum of 4 levels of index structures, then any key may be located with a maximum of 4 disk accesses. This structure is known as a modified BTREE index structure.

An NDX file record is 512 bytes in length. Record 0 of an NDX file is the file header. It points to the top level index "leaf" which begins a path that eventually leads to to the appropriate level 0 leaf entry for each key.

The structure of an index file header record is as follows:

#### **Chapter 6: Technical**

.

| Index File Header Record Format |   |  |
|---------------------------------|---|--|
| Byte 0-3                        | Long Integer record number of the root leaf.  |  |
| Byte 4-7                        | Long Integer total number of index records in the index file.   |  |
| Bytes 8-11                      | Reserved. TDBS puts zeros, allows anything.   |  |
| Bytes 12-13                     | Integer Key Size in bytes.  |  |
| Bytes 14-15                     | Integer maximum number of keys per leaf.  |  |
| Byte 16                         | Key type indicator. $0 = $ Char, $1 = $ Numeric   |  |
| Byte 17                         | Reserved. TDBS puts zero, allows anything.  |  |
| Bytes 18-19                     | Integer index key entry size.   |  |
| Bytes 20-22                     | Reserved. TDBS puts zero, allows anything.  |  |
| Byte 23                         | 1 = UNIQUE index, 0 = not UNIQUE  |  |
| Byte 24-244                     | Index Key Expression. ASCII text, zero<br>terminated. (TDBS version 1.0 only allows<br>a simple field name here). |  |
| Byte 245-507                    | Reserved. TDBS puts zero, allows anything.  |  |
| Byte 508-511                    | TDBS file sharing information while file is<br>open. TDBS puts zero here when the file is<br>closed.              |  |

#### ... . ... . . ---.

#### Index Leaf Record Format

| Bytes 0-3         | Long Integer number of active key entries in this leaf   |
|-------------------|--|
| Bytes 4-n         | Index key entries (as many as indicated in bytes 0-3 of this record). See below for format.  |
| Bytes n + 1-n + 4 | Long integer. Zero if this is a level 0 leaf. If this<br>is a higher level leaf, this is the record number<br>of the next lower level leaf record which begins<br>with the next greater key than the last key in<br>this leaf. |
| Bytes n + 5-511   | Undefined. May be anything.  |

#### **Index Key Entry Format**

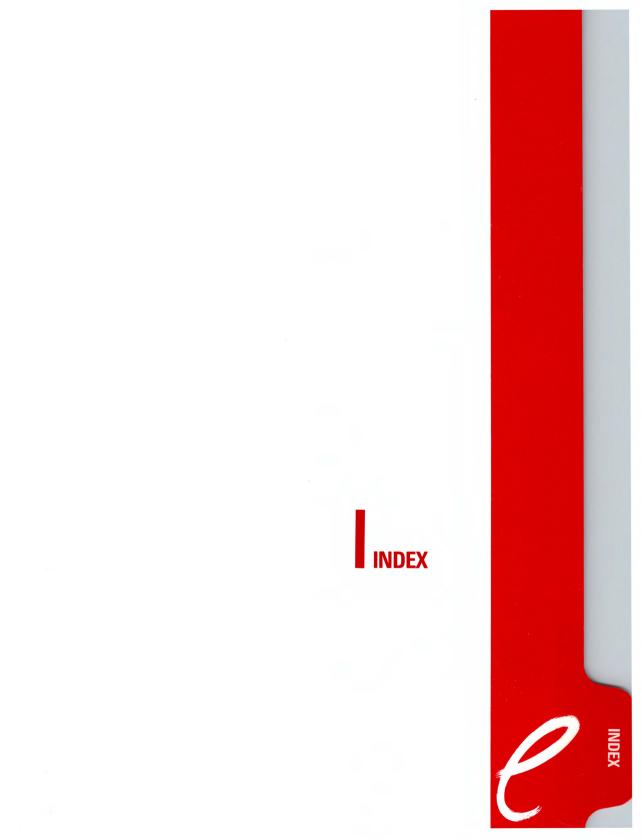
Each index record key entry has the following format.

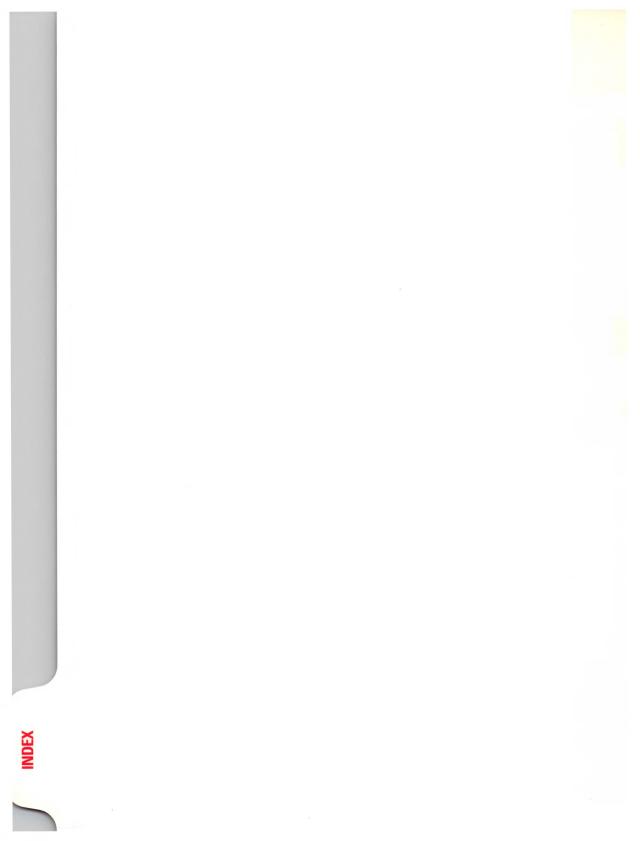
| Bytes 0-3 | Long Integer. Zero if this is a level 0 leaf entry.<br>Record number of associated index record in the<br>next lower level if not level 0 leaf entry.   |
|-----------|---|
| Bytes 4-7 | Long Integer. Zero if this is not a level 0 leaf<br>entry. If this is a level 0 leaf entry, then this is<br>the DBF record number of the database record<br>which is associated with this key.  |
| Bytes 8-n | Key Field Data. If index field is numeric or date<br>this is the floating point equivalent of the index<br>key data value. If the index field is character<br>this is the ASCII key value. Note: Numeric and<br>Date keys are always 8 bytes long. Character<br>keys are the length of the field, however the<br>entry here is rounded up to the next four byte<br>multiple. The actual key text is left justified in<br>this field. Any resulting padding is undefined<br>and may be anything. |

### **TDBS**.MEM File Format

The SAVE TO and RESTORE FROM commands create and read a memory variable file. This file saves the currently visible memory variables in the executing program. TDBS can restore a .MEM file which is saved by dBASE III+, and dBASE III+ will properly restore a .MEM file which is saved by TDBS. This file consists of a series of entries, one for each memory variable which is saved. The format of a TDBS .MEM file entry is as follows:

| Bytes 0-10  | Variable Name Text. Left Justified, zero filled.<br>Note: If byte 0 is 1Ah, then this is the end of the<br>.MEM file.  |
|-------------|--|
| Byte 11     | Bit 7 is always 1.<br>Bits 6-0 is the variable type as an ASCII letter.<br>(N = Numeric, C = Character etc.)   |
| Bytes 12-15 | Reserved. TDBS puts zero, allows anything.   |
| Bytes 16-17 | If Type = C, this is length (including null byte)<br>If Type = N, this is always 010Ch<br>If Type = D, this is always 0<br>If Type = L, this is always 1                                     |
| Byte 18     | Variable Domain. $0 =$ Public, $1 =$ Private in main $2 =$ Private one level down, etc.  |
| Byte 19-31  | Reserved. TDBS puts 0, allows anything.  |
| Byte 32-n   | Variable value. If Type = N or D, this is an 8<br>byte floating point number. If Type = L this is<br>a single byte. If Type = C this is the character<br>string plus a null terminator byte. |





| & (Macro)                |  |
|--------------------------|--|
| && (Comment)             |  |
| * (Comment)              |  |
| *@ (Conditional Comment) |  |
| = (Assign Value)         |  |
| ?/?? (Display Result)    |  |
| @ CLEAR                  |  |
| @ GET                    |  |
| @ SAY                    |  |
| @TO                      |  |
|                          |  |

### A

| ABS()   | 5-14   |
|---|--|
| Absolute Maximum Limits   | 2-20   |
| ACCEPT  | 4-22   |
| ACOPY()   | 5-15   |
| ADEL()  | 5-16   |
| ADSORT()  |  |
| AFIELDS()   |  |
| AFILL()   | 5-18   |
| AINS()  |  |
| ALIAS()   |  |
| ANSI output   | 2-14   |
| APPEND  |  |
|   |  |
| BLANK   | 4-23   |
| BLANK   |  |
|   | 4-24   |
| FROM<br>Arrays  | 4-24<br>-22, 5-92  |
| FROM  | 4-24<br>-22, 5-92<br>5-23  |
| FROM<br>Arrays  | 4-24<br>-22, 5-92<br>5-23<br>5-21                                |
| FROM<br>Arrays2-8, 4-42, 4-88, 4-90, 5-15 - 5-19, 5-21 - 5<br>ASC()<br>ASCAN()                    | 4-24<br>-22, 5-92<br>5-23<br>5-21<br>5-22                        |
| FROM<br>Arrays  | 4-24<br>-22, 5-92<br>5-23<br>5-21<br>5-22<br>5-24                |
| FROM<br>Arrays2-8, 4-42, 4-88, 4-90, 5-15 - 5-19, 5-21 - 5<br>ASC()<br>ASCAN()<br>ASORT()<br>AT() | 4-24<br>-22, 5-92<br>5-23<br>5-21<br>5-22<br>5-24<br>1-13        |
| FROM  | 4-24<br>-22, 5-92<br>5-23<br>5-21<br>5-22<br>5-24<br>1-13<br>3-2 |

### В

| Block Structured Language | 2-12 |
|---------------------------|------|
| BOF()                     |      |
| Box Drawing               |      |

### С

| CAPFIRST()                 |
|----------------------------|
| Carrier Loss               |
| CASE                       |
| CDOW()                     |
| CEILING()                  |
| Character Strings2-7       |
| SET EXACT4-124             |
| CHR()5-29                  |
| CLEAR                      |
| ALL4-28                    |
| GETS                       |
| MEMORY4-30                 |
| SCREEN                     |
| TYPEAHEAD4-31              |
| CLOSE                      |
| ALL4-32                    |
| ALTERNATE                  |
| DATABASES4-32              |
| FORMAT4-32                 |
| INDEX                      |
| CMONTH()                   |
| COL()5-31                  |
| Command Summary4-3         |
| Command Syntax             |
| Commands not supported2-21 |
| Compatibility              |
| Commands2-21               |
| Compiler Options1-9        |
| Compiling                  |
| .TDB file                  |
| Autocompiling1-13          |
| Command Line Syntax1-9     |
| Conditional Compiling2-5   |
| Options1-9                 |

| CONFIG.SYS         |             |
|--------------------|-------------|
| BUFFERS = setting  | 1-8         |
| FILES = Setting    |             |
| CONTINUE           |             |
| Control Structures | 2-12        |
| COPY               |             |
| FILE               |             |
| STRUCTURE          |             |
| STRUCTURE EXTENDED |             |
| то                 |             |
| COUNT              |             |
| CREATE             |             |
| CREATE FROM        |             |
| CRTRIM()           |             |
| CTOD()             | <b>F</b> 00 |
|                    |             |

### D

| Data Types               | 2-7   |
|--------------------------|-------|
| Database File Format     | 6-3   |
| Database Files           | 2-5   |
| DATE()                   | 5-34  |
| Dates                    |       |
| Range                    | 2-7   |
| SET CENTURY              | 4-109 |
| SET DATE                 | 4-114 |
| DAY()                    | 5-35  |
| dBASE Compatibility      | 2-3   |
| Extended Commands        | 2-23  |
| Extended Functions       | 2-25  |
| File Sharing             | 2-28  |
| Macros                   | 2-30  |
| Unimplemented Commands   | 2-21  |
| dBASE File Compatibility | 6-2   |
| DBF File Format          | 6-3   |
| DBF()                    | 5-36  |
| DEC2HEX()                | 5-37  |
| DELCARE                  |       |
| DELETE                   |       |
| DELETED()                | 5-38  |
| DIR                      |       |
| DISKSPACE()              | 5-40  |
|                          |       |

| DO         |      |
|------------|------|
| CASE       |      |
| PARAMETERS |      |
| WHILE      |      |
| DOTBBS     |      |
| DOTBBS()   | 5-41 |
| DOW()      |      |
| DTOC()     |      |
| DTOS()     |      |

### Ε

| EJECT                   | 4-51 |
|-------------------------|------|
| ELSE                    | 4-72 |
| ЕМРТҮ()                 | 5-45 |
| EMS Memory usage        |      |
| ENDCASE                 |      |
| ENDDO                   |      |
| ENDIF                   |      |
| EOF()                   |      |
| ERASE                   |      |
| ERROR()                 |      |
| Exclusive File Use      |      |
| EXIT                    |      |
| EXP()                   |      |
| Explicit File Locking   |      |
| Explicit Record Locking |      |
| Extended Keys           |      |
|                         |      |

#### F

| FBEXTRACT()           | 5-49 |
|-----------------------|------|
| FBFILL()              |      |
| FBINSERT()            | 5-51 |
| FBMOVE()              | 5-52 |
| FBREAD                |      |
| FBWRITE               |      |
| FCLOSE                | 4-57 |
| FCOUNT()              | 5-54 |
| FCREATE               |      |
| FDATE()               | 5-55 |
| FERROR()              |      |
| Field & Record Memory | 2-17 |
|                       |      |

| FIELD()                   | 5-57           |
|---------------------------|----------------|
| File Limits               | 2-29           |
| File Sharing              |                |
| Automatic Features        | 2-28           |
| Automatic Locking         |                |
| dBASE Compatible          |                |
| Exclusive or Shared Use   | 3-6            |
| Exclusive Use             | 3-3            |
| Explicit Locking          | 3-2, 3-5       |
| Mailboxes                 | 3-2, 3-5, 3-15 |
| ON ERROR                  | 3-9            |
| Record Locking            | 3-3            |
| Transparent File Sharing  | 3-2, 3-4, 3-12 |
| Transparent Locking       |                |
| Transparent Screen Update |                |
| USE                       |                |
| FILE()                    |                |
| FIND                      |                |
| FINDFIRST()               |                |
| FINDNEXT()                |                |
| FKLABEL()                 |                |
| FKMAX()                   |                |
| Flat File I/O             |                |
| CRTRIM()                  | 5-32           |
| FBEXTRACT()               |                |
| FBFILL()                  |                |
| FBINSERT()                |                |
| FBMOVE()                  |                |
| FBREAD                    |                |
| FBWRITE                   |                |
| FCLOSE                    |                |
| FCLOSE                    |                |
| FERROR()                  |                |
| 0                         |                |
| FLFIND                    |                |
| FLREAD                    |                |
| FLWRITE                   |                |
| FOPEN                     |                |
| FSEEK                     |                |
| FLEN()                    |                |
| FLFIND                    |                |
| FLOCK()                   |                |
| FLOOR()                   |                |
| FLREAD                    |                |
| FLWRITE                   |                |

| FMAXLEN()        |      |
|------------------|------|
| FOPEN            |      |
| FOUND()          |      |
| FSEEK            |      |
| FSIZE()          | 5-68 |
| FTIME()          | 5-69 |
| Function Keys    | 2-16 |
| Function Summary | 5-2  |

### G

| Get Pool Memory        | 2-18 |
|------------------------|------|
| GETENV()               |      |
| GETLPT()               |      |
| Global Glossary Memory |      |
| GO/GOTO                |      |
|                        |      |

### Η

| HARDCR()           |
|--------------------|
| HEX2DEČ()          |
| Homepath Directory |
| HOMEPATH()         |

### I

| IF ELSE ENDIF        | 4-72 |
|----------------------|------|
| IIF() [Immediate IF] | 5-75 |
| Index File Format    |      |
| Index Files          |      |
| dBASE compatibility  | 2-6  |
| INDEX ON             |      |
| INDEXEXT()           |      |
| INDEXKEY()           |      |
| INDEXORD()           |      |
| INKEY()              |      |
| INPUT                |      |
| Installing TDBS      | 1-3  |
| INT()                |      |
| ISALPHA()            |      |
| ISINT()              |      |
| v                    |      |

| ISLASTDAY() |  |
|-------------|--|
| ISLEAP()    |  |
| ISLOWER()   |  |
| ISSHARE()   |  |
| ISSTATE()   |  |
| ISUPPER()   |  |
| 0           |  |

### Κ

| Keyboard Mapping | 2-16 |  |
|------------------|------|--|
|------------------|------|--|

#### L

| LASTDAY()       | 5-89  |
|-----------------|-------|
| LASTKEY()       | F 00  |
| LASTREC()       | 5-117 |
| LEFT()          |       |
| LEN()           |       |
| LJUST()         |       |
| LOCATE          | 4-76  |
| LOCK()          |       |
| LOG()           |       |
| Logical Data    |       |
| LOOP            |       |
| Loss of Carrier |       |
| LOWER()         |       |
|                 |       |
| LUPDATE()       |       |
|                 |       |

### Μ

| Macro(&)             |  |
|----------------------|--|
| Mailboxes            |  |
| NEWMAIL()            |  |
| USE                  |  |
| WAIT4MAIL()          |  |
| MAX()                |  |
| Maximum Files Open   |  |
| Maximum Program Size |  |
| MEM File Format      |  |
| Memo fields          |  |
| Memory Allocation    |  |
|                      |  |

| Memory Variable Domains |      |
|-------------------------|------|
| Hidden Variables        | 2-32 |
| Parameter Passing       | 2-33 |
| Private Variables       |      |
| Program Levels          | 2-31 |
| Public Variables        |      |
| Memory Variable Size    |      |
| Memvar Memory           |      |
| Menu Entries            |      |
| MESSAGE()               |      |
| MIN()                   |      |
| MOD()                   |      |
| MONTH()                 |      |
| Multiuser File Access   |      |
| Multiuser Programming   | 3-1  |

### Ν

| NDX File Format | 6-5   |
|-----------------|-------|
| NDX()           | 5-103 |
| NEWMAIL()       |       |
| NEXTKEY         |       |
| NMYUSERS()      |       |
| NOTE            |       |
| Numeric Data    | 2-7   |
| NUSERS()        |       |

### 0

| OM CODE Size   |      |
|----------------|------|
| OM UDATA Size  |      |
| ON DISCONNECT  |      |
| ON ERROR       |      |
| ON ESCAPE      |      |
| ON KEY         |      |
| ON NEWMAIL     |      |
| Operators      |      |
| .AND           | 2-11 |
| .NOT.          | 2-11 |
| .OR            | 2-11 |
| Addition       |      |
| Character (\$) |      |
| Division       |      |
|                |      |

| Equal                 | 2-11  |
|-----------------------|-------|
| Exponentiation        |       |
| Greater Than          |       |
| Greater Than or Equal |       |
| Less Than             |       |
| Less Than or Equal    |       |
| Logical               |       |
| Mathematical          |       |
| Multiplication        |       |
| Not Equal             |       |
| Order of evaluation   |       |
| Relational            |       |
| Subtraction           |       |
| OPTDATA()             | 5-108 |
| OS()                  |       |
| OTHERWISE             |       |

### Ρ

| Parameter Passing        |       |
|--------------------------|-------|
| PARAMETERS               |       |
| PCOL()                   |       |
| PICTURE                  |       |
| Function Symbols         | 4-17  |
| Template Symbols         | 4-17  |
| TRANSFORM()              | 5-139 |
| Printer Control          | 3-21  |
| Printer Sharing          |       |
| Printer Support          |       |
| EJECT                    |       |
| GETLPT()                 |       |
| PCOL()                   | 5-110 |
| PROW()                   |       |
| SET CONSOLE              | 4-113 |
| SET DEVICE               | 4-118 |
| SET MARGIN               |       |
| SET PRINT                |       |
| SET PRINTER TO           |       |
| WAIT4LPT()               | 5-161 |
| PRIVATE                  |       |
| Private Memory Variables |       |
| PROCEDURE                |       |
| PROCLINE()               |       |
|                          |       |

| 5-112 |
|-------|
| 2-12  |
| 2-18  |
| 5-113 |
|       |
| 2-32  |
|       |

### Q

| QUIT | Γ4- | 91 |
|------|-----|----|
|------|-----|----|

### R

| RAT()            | 5-114 |
|------------------|-------|
| READ             |       |
| READKEY()        | 5-115 |
| RECALL           |       |
| RECCOUNT()       | 5-117 |
| RECNO()          | 5-118 |
| Record Locking   | 3-3   |
| RECSIZE()        | 5-119 |
| RELEASE          |       |
| RENAME           |       |
| REPLACE          |       |
| REPLICATE()      | 5-120 |
| RESTORE          | 4-101 |
| RETRY            |       |
| RETURN           |       |
| RETURN TO MASTER | 4-103 |
| RIGHT()          | 5-121 |
| RJUST()          |       |
| RLOCK()          | 5-123 |
| ROUND()          | 5-124 |
| ROW()            |       |
| RTRIM()          | 5-126 |

### S

| SAVE               | 4-104 |
|--------------------|-------|
| Save File Format   |       |
| Screen I/O         | 2-14  |
| Searching for text |       |
| 3                  |       |

|          | DS()                                    |       |
|----------|---|-------|
|          |   |       |
|          | ••••••••••••••••••••••••••••••••••••••• |       |
|          | 0                                       |       |
| SET ALT  | ERNATE                                  | 4-107 |
|          | L                                       |       |
|          | NTURY                                   |       |
|          | LOR                                     |       |
|          | NFIRM                                   |       |
|          | NSOLE                                   |       |
|          | ٢Ε                                      |       |
| SET DEC  | CIMALS                                  | 4-115 |
|          | ETED                                    |       |
| SET DEL  | LIMITERS                                | 4-117 |
|          | /ICE                                    |       |
| SET DISC | CONNECT                                 | 4-119 |
| SET DIS  | PLAY RULES                              | 4-120 |
| SET DIVI | IDE BY ZERO                             | 4-121 |
| SET EDIT | TOR                                     | 4-122 |
| SET ESC  | CAPE                                    | 4-123 |
| SET EXA  | NCT                                     | 4-124 |
| SET EXC  | CLUSIVE                                 | 4-125 |
| SET FILT | rer                                     | 4-126 |
| SET FIXE | ED                                      | 4-127 |
|          | RMAT TO                                 |       |
| SET FUN  |   | 4-129 |
|          | EX                                      |       |
|          | ENSITY                                  |       |
| SET MAP  | RGIN                                    | 4-132 |
| SET ORD  | DER                                     | 4-134 |
|          | NT                                      |       |
|          | NTER TO                                 |       |
| SET REL  | ATION TO                                | 4-138 |
| SET SOF  | FTSEEK                                  | 4-140 |
| SET TYP  | PEAHEAD                                 | 4-141 |
| SET UNI  | QUE                                     | 4-142 |
| SET UPD  | DATE BELL                               | 4-143 |
| SETPRC   | ;()                                     | 5-129 |
| SKIP     | -                                       | 4-144 |
| SOUNDE   | EX()                                    | 5-130 |
|          |   |       |
|          |   |       |
|          | AME()                                   |       |
|          |   |       |
|          |   |       |

| STR()                |     |
|----------------------|-----|
| STUFF()              |     |
| Subroutines          |     |
| SUBSTR()             |     |
| SUM                  |     |
| Syntax               | 2-4 |
| Syntax Element Types |     |

### Т

| TBBS Command Switch        | 1-4   |
|----------------------------|-------|
| TBBS Menu Entries          |       |
| /HP Opt Data Switch        |       |
| /OU Opt Data Switch        |       |
| /Q Opt Data Switch         |       |
| /U Opt Data Switch         |       |
| TYPE = 200 Command         |       |
| TDBS Compiler              |       |
| TDBS Functions             | 5-1   |
| TDBS Language              |       |
| Commands and Functions     |       |
| TDBS Program Structure     | 2-1   |
| TDBS variable              |       |
| TDBSOM Memory Requirements |       |
| TEXT ENDTEXT               | 4-147 |
| Text File Display          | 4-1   |
| Text Searching             | 4-61  |
| TIME()                     | 5-138 |
| TRANSFORM()                | 5-139 |
| Transparent File Sharing   |       |
| Transparent Screen Sharing | 4-143 |
| Transparent Screen Update  |       |
| ТҮРЕ                       |       |
| TYPE()                     | 5-140 |

### U

| UANSI()     | 5-141 |
|-------------|-------|
| UAUTH()     |       |
| UIBM()      |       |
| ULINE()     |       |
| ULOCATION() | 5-145 |
| ULPEEK()    |       |
|             |       |

| ULPOKE()              | 5-147 |
|-----------------------|-------|
| ULREPLACE()           | 5-149 |
| UMORE()               | 5-150 |
| UNAME()               | 5-151 |
| Unexpected Disconnect | 6-1   |
| UNLOCK                |       |
| UNOTES()              | 5-152 |
| UPDATED()             |       |
| UPPER()               |       |
| UPRIV()               |       |
| USE                   |       |
| USE MAILBOX           |       |
| USING()               |       |
| UWIDTH()              |       |
|                       |       |

#### V

| VAL()     | 5-158 |
|-----------|-------|
| VERSION() | 5-159 |

#### W

| WAIT         |  |
|--------------|--|
| WAIT4FLOCK() |  |
| WAIT4LPT()   |  |
| WAIT4MAIL()  |  |
| WAIT4RLOCK() |  |

#### Y

| YEAR() | 5-164 |
|--------|-------|
| Z      |       |

| 54 |
|----|
| -  |

