CA-Clipper®

For DOS

Version 5.3

Reference Guide

Volume 1

June 1995

COMPUTER ASSOCIATES®
Software superior by design.

© Copyright 1995 Computer Associates International, Inc.
One Computer Associates Plaza, Islandia, NY 11788-7000
All rights reserved.

Printed in the United States of America
Computer Associates International, Inc.
Publisher

No part of this documentation may be copied, photocopied, reproduced, translated, microfilmed, or otherwise duplicated on any medium without written consent of Computer Associates International, Inc.

Use of the software programs described herein and this documentation is subject to the Computer Associates License Agreement enclosed in the software package.
All product names referenced herein are trademarks of their respective companies.

Contents

Volume 1

Chapter 1: Introduction

Language Reference 1–1
Organization of Reference Items 1–2
Obsolete and Compatibility Items 1–3
Glossary 1–4
Symbols and Conventions 1–4
Metasymbols 1–5
Typography 1–6
Cross References 1–6

Chapter 2: Language Reference

#command | #translate directive 2–1
#define directive 2–12
#error directive 2–17
#ifdef directive 2–18
#ifndef directive 2–20
#include directive 2–22
#stdout directive 2–25
#undef directive 2–26
#xcommand | #xtranslate directive 2–28
$ operator 2–29
% operator 2–30
& operator 2–31
( ) operator 2–39

* operator 2–40
** operator 2–41
+ operator 2–42
++ operator 2–44
- operator 2–46
-- operator 2–48
-> operator 2–50
.AND. operator 2–53
.NOT. operator 2–54
.OR. operator 2–55
/ operator 2–56
: operator 2–57
:= operator 2–59
< operator 2–61
<= operator 2–63
<> != # operator 2–64
= (assign) operator 2–66
= (compound assign) operator 2–68
= (equality) operator 2–71
== operator 2–73
> operator 2–75
>= operator 2–77
? | ?? command 2–79
@ operator 2–81
@...BOX command 2–83
@...CLEAR command 2–85
@...GET command 2–86
@...GET CHECKBOX command 2–97
@...GET LISTBOX command 2–101
@...GET PUSHBUTTON command 2–106
@...GET RADIOGROUP command 2–110
@...GET TBROWSE command 2–113
@...PROMPT command 2–115
@...SAY command 2–117
@...TO command 2–122
[ ] operator 2–124

{ } operator 2–126
AADD() function 2–127
ABS() function 2–129
*ACCEPT command 2–130
ACHOICE() function 2–131
ACLONE() function 2–138
ACOPY() function 2–139
ADEL() function 2–141
ADIR()* function 2–142
AEVAL() function 2–144
AFIELDS()* function 2–146
AFILL() function 2–148
AINS() function 2–150
ALERT() function 2–151
ALIAS() function 2–153
ALLTRIM() function 2–154
ALTD() function 2–155
ANNOUNCE statement 2–156
APPEND BLANK command 2–157
APPEND FROM command 2–158
ARRAY() function 2–162
ASC() function 2–164
ASCAN() function 2–165
ASIZE() function 2–167
ASORT() function 2–168
AT() function 2–170
ATAIL() function 2–171
AVERAGE command 2–172
BEGIN SEQUENCE statement 2–173
BIN2I() function 2–176
BIN2L() function 2–177
BIN2W() function 2–178
BLOBDIRECTEXPORT() function 2–179
BLOBDIRECTGET() function 2–181
BLOBDIRECTIMPORT() function 2–183
BLOBDIRECTPUT() function 2–186

BLOBEXPORT() function 2–188
BLOBGET() function 2–190
BLOBIMPORT() function 2–192
BLOBROOTGET() function 2–194
BLOBROOTLOCK() function 2–196
BLOBROOTPUT() function 2–197
BLOBROOTUNLOCK() function 2–199
BOF() function 2–200
BREAK() function 2–202
BROWSE()* function 2–203
CALL* command 2–205
CANCEL* command 2–207
CDOW() function 2–208
CheckBox class 2–209
CHR() function 2–215
CLEAR ALL* command 2–217
CLEAR GETS command 2–218
CLEAR MEMORY command 2–219
CLEAR SCREEN command 2–220
CLEAR TYPEAHEAD command 2–221
CLOSE command 2–222
CMONTH() function 2–223
COL() function 2–224
COLORSELECT() function 2–225
COMMIT command 2–227
CONTINUE command 2–229
COPY FILE command 2–231
COPY STRUCTURE command 2–232
COPY STRUCTURE EXTENDED command 2–234
COPY TO command 2–236
COUNT command 2–240
CREATE command 2–241
CREATE FROM command 2–243
CTOD() function 2–246
CURDIR() function 2–248
DATE() function 2–249

DAY() function 2–250
DBAPPEND() function 2–251
DBCLEARFILTER() function 2–252
DBCLEARINDEX() function 2–253
DBCLEARRELATION() function 2–254
DBCLOSEALL() function 2–255
DBCLOSEAREA() function 2–256
DBCOMMIT() function 2–257
DBCOMMITALL() function 2–259
DBCREATE() function 2–261
DBCREATEINDEX() function 2–264
DBDELETE() function 2–266
DBEDIT() function 2–268
DBEVAL() function 2–276
DBF()* function 2–279
DBFIELDINFO() function 2–280
DBFILEGET() 2–282
DBFILEPUT() 2–284
DBFILTER() function 2–286
DBGOBOTTOM() function 2–288
DBGOTO() function 2–290
DBGOTOP() function 2–292
DBINFO() function 2–293
DBORDERINFO() function 2–297
DBRECALL() function 2–302
DBRECORDINFO() function 2–304
DBREINDEX() function 2–306
DBRELATION() function 2–307
DBRLOCK() function 2–309
DBRLOCKLIST() function 2–311
DBRSELECT() function 2–312
DBRUNLOCK() function 2–314
DBSEEK() function 2–315
DBSELECTAREA() function 2–317
DBSETDRIVER() function 2–319
DBSETFILTER() function 2–320

DBSETINDEX() function 2–322
DBSETORDER() function 2–323
DBSETRELATION() function 2–324
DBSKIP() function 2–326
DBSTRUCT() function 2–328
DBUNLOCK() function 2–330
DBUNLOCKALL() function 2–331
DBUSEAREA() function 2–332
DECLARE* statement 2–334
DELETE command 2–335
DELETE FILE command 2–337
DELETE TAG command 2–338
DELETED() function 2–340
DESCEND() function 2–341
DEVOUT() function 2–343
DEVOUTPICT() function 2–344
DEVPOS() function 2–346
DIR* command 2–348
DIRCHANGE() function 2–350
DIRECTORY() function 2–351
DIRMAKE() function 2–353
DIRREMOVE() function 2–354
DISKCHANGE() function 2–355
DISKNAME() function 2–356
DISKSPACE() function 2–357
DISPBEGIN() function 2–358
DISPBOX() function 2–360
DISPCOUNT() function 2–363
DISPEND() function 2–364
DISPLAY command 2–365
DISPOUT() function 2–367
DO* statement 2–369
DO CASE statement 2–371
DO WHILE statement 2–373
DOSERROR() function 2–375
DOW() function 2–377

DTOC() function 2–378
DTOS() function 2–379
EJECT command 2–381
EMPTY() function 2–382
EOF() function 2–384
ERASE command 2–386
Error class 2–387
ERRORBLOCK() function 2–393
ERRORLEVEL() function 2–395
EVAL() function 2–397
EXIT PROCEDURE statement 2–399
EXP() function 2–402
EXTERNAL* statement 2–403
FCLOSE() function 2–405
FCOUNT() function 2–406
FCREATE() function 2–407
FERASE() function 2–409
FERROR() function 2–410
FIELD statement 2–412
FIELDBLOCK() function 2–414
FIELDGET() function 2–416
FIELDNAME()/FIELD() function 2–417
FIELDPOS() function 2–419
FIELDPUT() function 2–420
FIELDWBLOCK() function 2–421
FILE() function 2–423
FIND* command 2–424
FKLABEL()* function 2–425
FKMAX()* function 2–426
FLOCK() function 2–427
FOPEN() function 2–429
FOR statement 2–431
FOUND() function 2–433
FREAD() function 2–435
FREADSTR() function 2–437
FRENAME() function 2–439

FSEEK() function 2–441
FUNCTION statement 2–443
FWRITE() function 2–449
GBMPDISP() function 2–451
GBMPLOAD() function 2–454
GELLIPSE() function 2–456
Get class 2–459
GETACTIVE() function 2–470
GETAPPLYKEY() function 2–471
GETDOSETKEY() function 2–473
GETENV() function 2–474
GETPOSTVALIDATE() function 2–476
GETPREVALIDATE() function 2–477
GETREADER() function 2–478
GFNTERASE() function 2–480
GFNTLOAD() function 2–481
GFNTSET() function 2–483
GFRAME() function 2–486
GGETPIXEL() function 2–489
GLINE() function 2–490
GMODE() function 2–492
GO command 2–497
GPOLYGON() function 2–498
GPUTPIXEL() function 2–500
GRECT() function 2–502
GSETCLIP() function 2–504
GSETEXCL() function 2–507
GSETPAL() function 2–510
GWRITEAT() function 2–512
HARDCR() function 2–515
HEADER() function 2–517
I2BIN() function 2–519
IF statement 2–520
IF() function 2–522
IIF() function 2–524
INDEX command 2–526

INDEXEXT() function 2–532
INDEXKEY() function 2–533
INDEXORD() function 2–536
INIT PROCEDURE statement 2–537
INKEY() function 2–540
*INPUT command 2–542
INT() function 2–543
ISALPHA() function 2–544
ISCOLOR() function 2–545
ISDIGIT() function 2–546
ISDISK() function 2–547
ISLOWER() function 2–548
ISPRINTER() function 2–549
ISUPPER() function 2–550

Glossary

Index

Volume 2

Chapter 2: Language Reference (cont.)

JOIN command 2–551
KEYBOARD command 2–553
L2BIN() function 2–555
LABEL FORM command 2–556
LASTKEY() function 2–558
LASTREC() function 2–560
LEFT() function 2–562
LEN() function 2–563
LIST command 2–565
ListBox class 2–567
LOCAL statement 2–580
LOCATE command 2–583
LOG() function 2–585
LOWER() function 2–587
LTRIM() function 2–588
LUPDATE() function 2–590
MAX() function 2–591
MAXCOL() function 2–592
MAXROW() function 2–593
MCOL() function 2–594
MDBLCLK() function 2–595
MEMOEDIT() function 2–596
MEMOLINE() function 2–605
MEMOREAD() function 2–607
MEMORY() function 2–609
MEMOSETSUPER() function 2–610
MEMOTRAN() function 2–612
MEMOWRIT() function 2–614
MEMVAR statement 2–615
MEMVARBLOCK() function 2–617
MenuItem class 2–619

MENUMODAL() function 2–623
MENU TO command 2–625
MHIDE() function 2–627
MIN() function 2–628
MLCOUNT() function....................2–629
MLCTOPOS() function 2–631
MLEFTDOWN() function....................2–633
MLPOS() function 2–634
*MOD() function....................2–635
MONTH() function 2–637
MPOSTOLC() function 2–638
MPRESENT() function 2–640
MRESTSTATE() function 2–641
MRIGHTDOWN() function 2–642
MROW() function....................2–643
MSAVESTATE() function....................2–644
MSETBOUNDS() function 2–645
MSETCLIP() function 2–646
MSETCURSOR() function 2–648
MSETPOS() function 2–649
MSHOW() function 2–650
MSTATE() function 2–652
NETERR() function 2–655
NETNAME() function....................2–657
NEXTKEY() function....................2–658
NOSNOW() function....................2–660
NOTE* command....................2–661
ORDBAGEXT() function....................2–663
ORDBAGNAME() function 2–664
ORDCOND() function....................2–666
ORDCONDSET() function 2–669
ORDCREATE() function....................2–673
ORDDESCEND() function 2–675
ORDDESTROY() function 2–677
ORDFOR() function 2–678
ORDISUNIQUE() function....................2–680

ORDKEY() function 2–682
ORDKEYADD() function 2–684
ORDKEYCOUNT() function 2–686
ORDKEYDEL() function 2–688
ORDKEYGOTO() function 2–691
ORDKEYNO() function 2–693
ORDKEYVAL() function 2–695
ORDLISTADD() function 2–697
ORDLISTCLEAR() function 2–699
ORDLISTREBUILD() function 2–700
ORDNAME() function 2–701
ORDNUMBER() function 2–703
ORDSCOPE() function 2–704
ORDSETFOCUS() function 2–706
ORDSETRELATION() function 2–708
ORDSKIPUNIQUE() function 2–710
OS() function 2–712
OUTERR() function 2–713
OUTSTD() function 2–714
PACK command 2–715
PAD() function 2–716
PARAMETERS statement 2–718
PCOL() function 2–720
PCOUNT() function 2–722
PopUpMenu class 2–724
PRIVATE statement 2–734
PROCEDURE statement 2–736
PROCLINE() function 2–741
PROCNAME() function 2–743
PROW() function 2–745
PUBLIC statement 2–747
PushButton class 2–749
QOUT() function 2–757
QUIT command 2–759
RadioButto class 2–761
RadioGroup class 2–768

RAT() function 2–777
RDDLIST() function 2–778
RDDNAME() function 2–780
RDDSETDEFAULT() function 2–781
READ command 2–782
READEXIT() function 2–786
READFORMAT() function 2–787
READINSERT() function 2–788
READKEY()* function 2–789
READKILL() function 2–791
READMODAL() function 2–792
READUPDATED() function 2–794
READVAR() function 2–795
RECALL command 2–797
RECCOUNT()* function 2–798
RECNO() function 2–799
RECSIZE() function 2–800
REINDEX command 2–801
RELEASE command 2–803
RENAME command 2–804
REPLACE command 2–806
REPLICATE() function 2–808
REPORT FORM command 2–809
REQUEST statement 2–812
RESTORE command 2–814
RESTORE SCREEN* command 2–816
RESTSCREEN() function 2–818
RETURN statement 2–820
RIGHT() function 2–822
RLOCK() function 2–823
ROUND() function 2–825
ROW() function 2–827
RTRIM() function 2–829
RUN command 2–831
SAVE command 2–833
SAVE SCREEN* command 2–835

SAVESCREEN() function 2–837
SCROLL() function 2–839
Scrollbar class 2–841
SECONDS() function 2–847
SEEK command 2–848
SELECT command 2–850
SELECT() function 2–852
SET ALTERNATE command 2–853
SET BELL command 2–855
SET CENTURY command 2–856
SET COLOR* command 2–857
SET CONFIRM command 2–861
SET CONSOLE command 2–862
SET CURSOR command 2–864
SET DATE command 2–865
SET DECIMALS command 2–867
SET DEFAULT command 2–868
SET DELETED command 2–870
SET DELIMITERS command 2–871
SET DESCENDING command 2–873
SET DEVICE command 2–874
SET EPOCH command 2–875
SET ESCAPE command 2–876
SET EVENTMASK command 2–877
SET EXACT* command 2–878
SET EXCLUSIVE* command 2–880
SET FILTER command 2–882
SET FIXED command 2–883
SET FORMAT* command 2–884
SET FUNCTION command 2–886
SET INDEX command 2–887
SET INTENSITY command 2–889
SET KEY command 2–890
SET MARGIN command 2–892
SET MEMOBLOCK command 2–893
SET MESSAGE command 2–894

SET OPTIMIZE command 2–895
SET ORDER command 2–896
SET PATH command 2–898
SET PRINTER command 2–900
SET PROCEDURE* command 2–903
SET RELATION command 2–904
SET SCOPE command 2–906
SET SCOPEBOTTOM command 2–907
SET SCOPETOP command 2–908
SET SCOREBOARD command 2–909
SET SOFTSEEK command 2–910
SET TYPEAHEAD command 2–912
SET UNIQUE* command 2–913
SET VIDEOMODE command 2–914
SET WRAP* command 2–915
SET() function 2–916
SETBLINK() function 2–919
SETCANCEL() function 2–920
SETCOLOR() function 2–922
SETCURSOR() function 2–925
SETKEY() function 2–927
SETMODE() function 2–929
SETPOS() function 2–930
SETPRC() function 2–931
SKIP command 2–933
SORT command 2–935
SOUNDEX() function 2–937
SPACE() function 2–938
SQRT() function 2–939
STATIC statement 2–940
STORE* command 2–943
STR() function 2–945
STRTRAN() function 2–947
STUFF() function 2–948
SUBSTR() function 2–950
SUM command 2–952

TBColumn class 2–953
TBrowse class 2–958
TEXT* command 2–974
TIME() function 2–976
TONE() function 2–977
TopBarMenu class 2–979
TOTAL command 2–986
TRANSFORM() function 2–988
TRIM() function 2–991
TYPE command 2–993
TYPE() function 2–994
UNLOCK command 2–997
UPDATE command 2–999
UPDATED() function 2–1001
UPPER() function 2–1003
USE command 2–1005
USED() function 2–1010
VAL() function 2–1011
VALTYPE() function 2–1012
VERSION() function 2–1014
WAIT* command 2–1015
WORD()* function 2–1017
YEAR() function 2–1019
ZAP command 2–1020

Glossary

Index

Categorized Contents

Volume 1

Statements

ANNOUNCE statement 2–156
BEGIN SEQUENCE statement 2–173
DECLARE* statement 2–334
DO* statement 2–369
DO CASE statement 2–371
DO WHILE statement 2–373
EXIT PROCEDURE statement 2–399
EXTERNAL* statement 2–403
FIELD statement 2–412
FOR statement 2–431
FUNCTION statement 2–443
IF statement 2–520
INIT PROCEDURE statement 2–537

Preprocessor Directives

#command | #translate directive 2–1
#define directive 2–12
#error directive 2–17
#ifdef directive 2–18
#ifndef directive 2–20
#include directive 2–22
#stdout directive 2–25
#undef directive 2–26
#xcommand | #xtranslate directive 2–28

Commands

? | ?? command 2–79
@...BOX command 2–83
@...CLEAR command 2–85
@...GET command 2–86
@...GET CHECKBOX command 2–97
@...GET LISTBOX command 2–101
@...GET PUSHBUTTON command 2–106
@...GET RADIOGROUP command 2–110
@...GET TBROWSE command 2–113
@...PROMPT command 2–115
@...SAY command 2–117
@...TO command 2–122
*ACCEPT command 2–130
APPEND BLANK command 2–157
APPEND FROM command 2–158
AVERAGE command 2–172
CALL* command 2–205
CANCEL* command 2–207
CLEAR ALL* command 2–217
CLEAR GETS command 2–218
CLEAR MEMORY command 2–219
CLEAR SCREEN command 2–220
CLEAR TYPEAHEAD command 2–221
CLOSE command 2–222
COMMIT command 2–227
CONTINUE command 2–229
COPY FILE command 2–231
COPY STRUCTURE command 2–232
COPY STRUCTURE EXTENDED command 2–234
COPY TO command 2–236
COUNT command 2–240
CREATE command 2–241
CREATE FROM command 2–243
DELETE command 2–335
DELETE FILE command 2–337

DELETE TAG command 2–338
DIR* command 2–348
DISPLAY command 2–365
EJECT command 2–381
ERASE command 2–386
FIND* command 2–424
GO command 2–497
INDEX command 2–526
*INPUT command 2–542

Functions

AADD() function 2–127
ABS() function 2–129
ACHOICE() function 2–130
ACLONE() function 2–138
ACOPY() function 2–139
ADEL() function 2–141
ADIR()* function 2–142
AEVAL() function 2–144
AFIELDS()* function 2–146
AFILL() function 2–148
AINS() function 2–150
ALERT() function 2–151
ALIAS() function 2–153
ALLTRIM() function 2–154
ALTD() function 2–155
ARRAY() function 2–162
ASC() function 2–164
ASCAN() function 2–165
ASIZE() function 2–167
ASORT() function 2–168
AT() function 2–170
ATAIL() function 2–171
BIN2I() function 2–176
BIN2L() function 2–177

BIN2W() function 2–178
BLOBDIRECTEXPORT() function 2–179
BLOBDIRECTGET() function 2–181
BLOBDIRECTIMPORT() function 2–183
BLOBDIRECTPUT() function 2–186
BLOBEXPORT() function 2–188
BLOBGET() function 2–190
BLOBIMPORT() function 2–192
BLOBROOTGET() function 2–194
BLOBROOTLOCK() function 2–196
BLOBROOTPUT() function 2–197
BLOBROOTUNLOCK() function 2–199
BOF() function 2–200
BREAK() function 2–202
BROWSE()* function 2–203
CDOW() function 2–208
CHR() function 2–215
CMONTH() function 2–223
COL() function 2–224
COLORSELECT() function 2–225
CTOD() function 2–246
CURDIR() function 2–248
DATE() function 2–249
DAY() function 2–250
DBAPPEND() function 2–251
DBCLEARFILTER() function 2–252
DBCLEARINDEX() function 2–253
DBCLEARRELATION() function 2–254
DBCLOSEALL() function 2–255
DBCLOSEAREA() function 2–256
DBCOMMIT() function 2–257
DBCOMMITALL() function 2–259
DBCREATE() function 2–261
DBCREATEINDEX() function 2–264
DBDELETE() function 2–266
DBEDIT()* function 2–268

DBEVAL() function 2–276
DBF()* function 2–279
DBFIELDINFO() function 2–280
DBFILEGET() 2–282
DBFILEPUT() 2–284
DBFILTER() function 2–286
DBGOBOTTOM() function 2–288
DBGOTO() function 2–290
DBGOTOP() function 2–292
DBINFO() function 2–293
DBORDERINFO() function 2–297
DBRECALL() function 2–302
DBRECORDINFO() function 2–304
DBREINDEX() function 2–306
DBRELATION() function 2–307
DBRLOCK() function 2–309
DBRLOCKLIST() function 2–311
DBRSELECT() function 2–312
DBRUNLOCK() function 2–314
DBSEEK() function 2–315
DBSELECTAREA() function 2–317
DBSETDRIVER() function 2–319
DBSETFILTER() function 2–320
DBSETINDEX() function 2–322
DBSETORDER() function 2–323
DBSETRELATION() function 2–324
DBSKIP() function 2–326
DBSTRUCT() function 2–328
DBUNLOCK() function 2–330
DBUNLOCKALL() function 2–331
DBUSEAREA() function 2–332
DELETED() function 2–340
DESCEND() function 2–341
DEVOUT() function 2–343
DEVOUTPICT() function 2–344
DEVPOS() function 2–346

DIRCHANGE() function 2–350
DIRECTORY() function 2–351
DIRMAKE() function 2–353
DIRREMOVE() function 2–354
DISKCHANGE() function 2–355
DISKNAME() function 2–356
DISKSPACE() function 2–357
DISPBEGIN() function 2–358
DISPBOX() function 2–360
DISPCOUNT() function 2–363
DISPEND() function 2–364
DISPOUT() function 2–367
DOSERROR() function 2–375
DOW() function 2–377
DTOC() function 2–378
DTOS() function 2–379
EMPTY() function 2–382
EOF() function 2–384
ERRORBLOCK() function 2–393
ERRORLEVEL() function 2–395
EVAL() function 2–397
EXP() function 2–402
FCLOSE() function 2–405
FCOUNT() function 2–406
FCREATE() function 2–407
FERASE() function 2–409
FERROR() function 2–410
FIELDBLOCK() function 2–414
FIELDGET() function 2–416
FIELDNAME()/FIELD() function 2–417
FIELDPOS() function 2–419
FIELDPUT() function 2–420
FIELDWBLOCK() function 2–421
FILE() function 2–423
FKLABEL()* function 2–425
FKMAX()* function 2–426

FLOCK() function . . . 2–427
FOPEN() function . . . 2–429
FOUND() function . . . 2–433
FREAD() function . . . 2–435
FREADSTR() function . . . 2–437
FRENAME() function . . . 2–439
FSEEK() function . . . 2–441
FWRITE() function . . . 2–449
GBMPDISP() function . . . 2–451
GBMPLOAD() function . . . 2–454
GELLIPSE() function . . . 2–456
GETACTIVE() function . . . 2–470
GETAPPLYKEY() function . . . 2–471
GETDOSETKEY() function . . . 2–473
GETENV() function . . . 2–474
GETPOSTVALIDATE() function . . . 2–476
GETPREVALIDATE() function . . . 2–477
GETREADER() function . . . 2–478
GFNTERASE() function . . . 2–480
GFNTLOAD() function . . . 2–481
GFNTSET() function . . . 2–483
GFRAME() function . . . 2–486
GGETPIXEL() function . . . 2–489
GLINE() function . . . 2–490
GMODE() function . . . 2–492
GPOLYGON() function . . . 2–498
GPUTPIXEL() function . . . 2–500
GRECT() function . . . 2–502
GSETCLIP() function . . . 2–504
GSETEXCL() function . . . 2–507
GSETPAL() function . . . 2–510
GWRITEAT() function . . . 2–512
HARDCR() function . . . 2–515
HEADER() function . . . 2–517
I2BIN() function . . . 2–519
IF() function . . . 2–522

IIF() function 2–524
INDEXEXT() function 2–532
INDEXKEY() function 2–533
INDEXORD() function 2–536
INKEY() function 2–540
INT() function 2–543
ISALPHA() function 2–544
ISCOLOR() function 2–545
ISDIGIT() function 2–546
ISDISK() function 2–547
ISLOWER() function 2–548
ISPRINTER() function 2–549
ISUPPER() function 2–550

Classes

CheckBox class 2–209
Error class 2–387
Get class 2–459

Operators

$ operator 2–29
% operator 2–30
& operator 2–31
( ) operator 2–39
* operator 2–40
** operator 2–41
+ operator 2–42
++ operator 2–44
- operator 2–46
-- operator 2–48
-> operator 2–50
.AND. operator 2–53
.NOT. operator 2–54

.OR. operator 2–55
/ operator 2–56
: operator 2–57
:= operator 2–59
< operator 2–61
<= operator 2–63
<> != # operator 2–64
= (assign) operator 2–66
= (compound assign) operator 2–68
= (equality) operator 2–71
== operator 2–73
> operator 2–75
>= operator 2–77
@ operator 2–81
[ ] operator 2–124
{ } operator 2–126

Chapter 1
Introduction

This is the *Reference Guide* for CA-Clipper Release 5.3. It is divided into two volumes and contains specification-level documentation for the CA-Clipper language. Use the appropriate volume whenever you want to know how a particular aspect of the CA-Clipper language (e.g., command, function) works and what syntax is required to use it.

Note: The *Reference Guide, Volume 2* includes a glossary, which is a comprehensive dictionary of terms used throughout the CA-Clipper documentation. Each glossary entry consists of the item name, the identity of one or more categories to which the item belongs, and a short definition.

For a brief summary of the *Reference Guide* information, use the *Quick Reference Guide*. For an online version of the *Reference Guide*, accessible while operating your program editor or any other development utility, use the *Guide To CA-Clipper* (see the *Programming and Utilities Guide* for documentation).

Language Reference

This chapter is the complete reference to the CA-Clipper language. The language reference is organized alphabetically without regard to categorization, making it easier for you to locate the information you need. With this organization, you need to know only the name of an item to find the information you want.

Each element in the CA-Clipper language falls into one of the following categories: statement, directive, command, function, class, or operator. There is a separate table of contents for each category, allowing you even quicker access to the reference material.

Organization of Reference Items

Each statement, directive, command, function, class, and operator is an item. Items are arranged in alphabetical order by name. If there are synonyms (e.g., LASTREC() and RECCOUNT()), each item is replicated in full. Each item begins on a page by itself so you can easily find the item you need.

Each entry is divided into the subsections listed below. If a subsection is inappropriate for a particular item, it is omitted:

- ***Item name*** is the name of the item followed by its category—statement, directive, command, function, class, or operator.

- ***Short descriptor*** describes the purpose or action of the item.

- ***Syntax*** defines a prototypical usage of the specified item with metasymbols for user input. If the item is a function, it is specified followed by a return value metasymbol. See the Symbols and Conventions section below for more information on metasymbols and other special symbols used in the syntax representations.

  Note: The semicolon is the correct symbol for line continuation in a program. It has been purposely omitted from the language definitions in this guide in order that it not be confused as a required part of the syntax.

- ***Arguments*** defines the meaning of each item argument. Arguments are variables and/or keywords used at the invocation of an item. Arguments are sometimes referred to as actual parameters, especially when invoking a function. If the item is a command, a clause is treated as an argument.

- ***Operands*** defines the meaning of each operand. This section is similar to the Arguments section but is used for operators only. Thus, you can think of operands as the arguments of an operator.

- ***Returns*** defines the return value of a function, including its data type.

- ***Type*** lists the data types of an operator. The data type of an operator is analogous to the return value of a function.

- ***Description*** describes the basic operation and usage of the item.

- *Notes* describes operations that demand special note, including eccentric or exceptional behavior. This section also describes synergistic or conflicting behaviors that relate to other items.
- *Examples* are code fragments with resulting output, if appropriate. Examples generally serve three purposes: usage demonstration, validation of operational rules and/or return values, and illumination of a programming concept.
- *Files* is the list of files on which the current item depends. These files fall into the following categories:
  - Libraries—usually located in \CLIP53\LIB
  - Source files—usually located in \CLIP53\SOURCE\...
  - Header files—usually located in \CLIP53\INCLUDE
- *See also* is an alphabetical list of items related to the usage of the current item.

Note: The structure of items is designed to facilitate use based on the level of expertise you may have with an item. If you are experienced, a quick glance at the item name, short descriptor, and syntax sections should give enough information. If you are less experienced, read the more detailed information thoroughly.

Obsolete and Compatibility Items

The asterisk following an item name indicates an item that is obsolete or that exists for compatibility with previous releases of CA-Clipper. Obsolete items, in general, are inconsistent with the current CA-Clipper programming philosophy. We strongly discourage their use since they may not be supported in future releases of CA-Clipper.

As you begin to incorporate the new features of CA-Clipper into your existing applications, it is advisable to review the code to determine any obsolete items you might be using. All items marked as obsolete are superseded by one or more improved features that are referenced in the description of the item. This information should be helpful to you in updating your programs to CA-Clipper.

Glossary

The Glossary is a comprehensive dictionary of terms used throughout the CA-Clipper documentation. Each glossary entry consists of the item name, the identity of one or more categories to which the item belongs, and a short definition.

Symbols and Conventions

This document uses symbols and metasymbols to specify items in a general way. Symbols identify special items and behaviors. For example, keywords and metasymbols enclosed in square brackets are optional. The following is a complete list of symbols used in syntax:

Symbols

<table>
<tr><th>Symbols</th><th>Description</th></tr>
<tr><td>< ></td><td>Indicates user input item.</td></tr>
<tr><td>( )</td><td>Within syntax specification, indicates function argument list.
At the end of syntax specification, describes multiple behaviors.</td></tr>
<tr><td>[ ]</td><td>Indicates optional item or list.</td></tr>
<tr><td>{ }</td><td>Indicates a code block or a literal array.</td></tr>
<tr><td>| |</td><td>Indicates a code block argument list.</td></tr>
<tr><td>→</td><td>Indicates a function return value.</td></tr>
<tr><td>...</td><td>Repeated elements if followed by a symbol.
Intervening code if followed by a keyword.</td></tr>
<tr><td>,</td><td>List item separator.</td></tr>
<tr><td>|</td><td>Indicates two or more mutually exclusive options.</td></tr>
<tr><td>@</td><td>Indicates that an argument must be passed by reference.</td></tr>
<tr><td>*</td><td>Indicates a compatibility command or function.</td></tr>
</table>

Metasymbols

Metasymbols describe the general nature of basic syntax elements. A metasymbol is a data type designator followed by a logical descriptor. The data type designation is a prefix chosen from the table below.

Prefixes are always lowercase and logical descriptors are always capitalized. Logical descriptors describe the meaning of the argument the metasymbol represents. For example, *cString* represents any character string and *xcFilespec* represents a file whose specification is either a literal string or a character expression enclosed in parentheses. If an argument supports more than one data type, a prefix symbol is specified for each data type. For example, RANGE <*dnLower*>, <*dnUpper*>.

Metasymbol Prefixes

Type Prefix	Description
a	Array
b	Code block
c	Character expression
d	Date expression
exp	Expression of any type
id	Literal identifier
l	Logical expression
m	Memo field
n	Numeric expression
obj, o	Object
u	Undetermined
x	Extended expression

Typography

The *Reference Guide* uses capitalization and type styles to distinguish between language elements and discussion of them. The following list defines the convention for each element:

Typographic Conventions

Element Example	Description
`ABS(<nNumber>)`	Syntax specification
`<nNumber>`	Metasymbol in syntax specification
<*nNumber*>	Metasymbol in text
nNumber	Function or method return value in syntax specification.
`? ABS()`	Program example.
COPY TO	CA-Clipper keywords, commands, and functions are all uppercase in syntax and text.
ON | off	Default SET command options are uppercase.
#include	Preprocessor directives are all lowercase.
`Assert()`	User-defined functions and procedures are capitalized in syntax and text.
ClassName	Class names are always capitalized.
pageUp()	Method and instance variable names begin with a lowercase character.
Class:selector()	Messages are always formatted with the class name followed by a colon (:) character and the method name.
CLIPPER.EXE	DOS files are uppercase, including extension.
Main.prg	CA-Clipper files, including source and database files, are capitalized.
Ctrl+Up arrow	Keynames.

Cross References

The following conventions are used:

- Guide name in italic:

  See the *Workbench User Guide*.

- Chapter name in double quotes:

  See "Using the Form Editor" in the *Workbench User Guide*.

- Section name as it appears in the document:

  See the Setting System-Wide Options section.

Chapter 2
Language Reference

#command | #translate directive

Specify a user-defined command or translation directive

Syntax

```
#command    <matchPattern> => <resultPattern>
#translate   <matchPattern> => <resultPattern>
```

Arguments

<matchPattern> is the pattern the input text should match.

<resultPattern> is the text produced if a portion of input text matches the *<matchPattern>*.

The => symbol between *<matchPattern>* and *<resultPattern>* is, along with #command or #translate, a literal part of the syntax that must be specified in a #command or #translate directive. The symbol consists of an equal sign followed by a greater than symbol with no intervening spaces. Do not confuse the symbol with the >= or the <= comparison operators in the CA-Clipper language.

Description

#command and #translate are translation directives that define commands and pseudofunctions. Each directive specifies a translation rule. The rule consists of two portions: a match pattern and a result pattern. The match pattern matches a command specified in the program (.prg) file and saves portions of the command text (usually command arguments) for the result pattern to use. The result pattern then defines what will be written to the result text and how it will be written using the saved portions of the matching input text.

#command and #translate are similar, but differ in the circumstance under which their match patterns match input text. A #command directive matches only if the input text is a complete statement, while #translate matches input text that is not a complete statement. #command defines a complete command and #translate defines clauses and pseudofunctions that may not form a complete statement. In general, use #command for most definitions and #translate for special cases.

#command and #translate are similar to but more powerful than the #define directive. #define, generally, defines identifiers that control conditional compilation and manifest constants for commonly used constant values such as INKEY() codes. Refer to any of the header files in the \CLIP53\INCLUDE directory for examples of manifest constants defined using #define.

#command and #translate directives have the same scope as the #define directive. The definition is valid only for the current program (.prg) file unless defined in Std.ch or the header specified with the /U option on the compiler command line. If defined elsewhere, the definition is valid from the line where it is specified to the end of the program file. Unlike #define, a #translate or #command definition cannot be explicitly undefined. The #undef directive has no effect on a #command or #translate definition.

As the preprocessor encounters each source line preprocessor, it scans for definitions in the following order of precedence: #define, #translate, and #command. When there is a match, the substitution is made to the result text and the entire line is reprocessed until there are no matches for any of the three types of definitions. #command and #translate rules are processed in stack-order (i.e., last in-first out, with the most recently specified rule processed first).

In general, a command definition provides a way to specify an English language statement that is, in fact, a complicated expression or function call, thereby improving the readability of source code. You can use a command in place of an expression or function call to impose order of keywords, required arguments, combinations of arguments that must be specified together, and mutually exclusive arguments at compile time rather than at runtime. This can be important since procedures and user-defined functions can now be called with any number of arguments, forcing any argument checking to occur at runtime. With command definitions, the preprocessor handles some of this.

All commands in CA-Clipper are defined using the #command directive and supplied in the standard header file, Std.ch, located in the \CLIP53\INCLUDE directory. The syntax rules of #command and #translate facilitate the processing of all CA-Clipper and dBASE-style commands into expressions and function calls. This provides CA-Clipper compatibility, as well as avenues of compatibility with other dialects.

When defining a command, there are several prerequisites to properly specifying the command definition. Many preprocessor commands require more than one #command directive because mutually exclusive clauses contain a keyword or argument. For example, the @...GET command has mutually exclusive VALID and RANGE clauses and is defined with a different #command rule to implement each clause.

This also occurs when a result pattern contains different expressions, functions, or parameter structures for different clauses specified for the same command (e.g., the @...SAY command). In Std.ch, there is a #command rule for @...SAY specified with the PICTURE clause and another for @...SAY specified without the PICTURE clause. Each formulation of the command is translated into a different expression. Because directives are processed in stack order, when defining more than one rule for a command, place the most general case first, followed by the more specific ones. This ensures that the proper rule will match the command specified in the program (.prg) file.

For more information and a general discussion of commands, refer to the "Basic Concepts" chapter in the *Programming and Utilities Guide*.

Match Pattern

The <*matchPattern*> portion of a translation directive is the pattern the input text must match. A match pattern is made from one or more of the following components, which the preprocessor tries to match against input text in a specific way:

- ***Literal values*** are actual characters that appear in the match pattern. These characters must appear in the input text, exactly as specified to activate the translation directive.

- ***Words*** are keywords and valid identifiers that are compared according to the dBASE convention (case-insensitive, first four letters mandatory, etc.). The match pattern must start with a Word.

  #xcommand and #xtranslate can recognize keywords of more than four significant letters.

- ***Match markers*** are label and optional symbols delimited by angle brackets (<>) that provide a substitute (idMarker) to be used in the <resultPattern> and identify the clause for which it is a substitute. Marker names are identifiers and must, therefore, follow the CA-Clipper identifier naming conventions. In short, the name must start with an alphabetic or underscore character, which may be followed by alphanumeric or underscore characters.

This table describes all match marker forms:

Match Markers

Match Marker	Name
<idMarker>	Regular match marker
<idMarker,...>	List match marker
<idMarker:word list>	Restricted match marker
<*idMarker*>	Wild match marker
<(idMarker)>	Extended Expression match marker

- ***Regular match marker:*** Matches the next legal expression in the input text. The regular match marker, a simple label, is the most general and, therefore, the most likely match marker to use for a command argument. Because of its generality, it is used with the regular result marker, all of the stringify result markers, and the blockify result marker.

- ***List match marker:*** Matches a comma-separated list of legal expressions. If no input text matches the match marker, the specified marker name contains nothing. You must take care in making list specifications because extra commas will cause unpredictable and unexpected results.

  The list match marker defines command clauses that have lists as arguments. Typically these are FIELDS clauses or expression lists used by database commands. When there is a match for a list match marker, the list is usually written to the result text using either the normal or smart stringify result marker. Often, lists are written as literal arrays by enclosing the result marker in curly ({ }) braces.

- ***Restricted match marker:*** Matches input text to one of the words in a comma-separated list. If the input text does not match at least one of the words, the match fails and the marker name contains nothing.

  A restricted match marker is generally used with the logify result marker to write a logical value into the result text. If there is a match for the restricted match marker, the corresponding logify result marker writes true (.T.) to the result text; otherwise, it writes false (.F.). This is particularly useful when defining optional clauses that consist of a command keyword with no accompanying argument. Std.ch implements the REST clause of database commands using this form.

- ***Wild match marker:*** Matches any input text from the current position to the end of a statement. Wild match markers generally match input that may not be a legal expression, such as #command NOTE <*x*> in Std.ch, gather the input text to the end of the statement, and write it to the result text using one of the stringify result markers.

- ***Extended expression match marker:*** Matches a regular or extended expression, including a file name or path specification. It is used with the smart stringify result marker to ensure that extended expressions will not get stringified, while normal, unquoted string file specifications will.

- ***Optional match clauses*** are portions of the match pattern enclosed in square brackets ([ ]). They specify a portion of the match pattern that may be absent from the input text. An optional clause may contain any of the components allowed within a <*matchPattern*>, including other optional clauses.

  Optional match clauses may appear anywhere and in any order in the match pattern and still match input text. Each match clause may appear only once in the input text. There are two types of optional match clauses: one is a keyword followed by match marker, and the other is a keyword by itself. These two types of optional match clauses can match all of the traditional command clauses typical of the CA-Clipper command set.

  Optional match clauses are defined with a regular or list match marker to match input text if the clause consists of an argument or a keyword followed by an argument (see the INDEX clause of the USE command in Std.ch). If the optional match clause consists of a keyword by itself, it is matched with a restricted match marker (see the EXCLUSIVE or SHARED clause of the USE command in Std.ch).

  In any match pattern, you may not specify adjacent optional match clauses consisting solely of match markers, without generating a compiler error. You may repeat an optional clause any number of times in the input text, as long as it is not adjacent to any other optional clause. To write a repeated match clause to the result text, use repeating result clauses in the <*resultPattern*> definition.

Result Pattern

The *<resultPattern>* portion of a translation directive is the text the preprocessor will produce if a piece of input text matches the *<matchPattern>*. *<resultPattern>* is made from one or more of the following components:

- ***Literal tokens*** are actual characters that are written directly to the result text.
- ***Words*** are CA-Clipper keywords and identifiers that are written directly to the result text.
- ***Result markers:*** refer directly to a match marker name. Input text matched by the match marker is written to the result text via the result marker.

  This table lists the Result marker forms:

Result Markers

Result Marker	Name
<idMarker>	Regular result marker
#<idMarker>	Dumb stringify result marker
<"idMarker">	Normal stringify result marker
<(idMarker)>	Smart stringify result marker
<{idMarker}>	Blockify result marker
<.idMarker.>	Logify result marker

- ***Regular result marker:*** Writes the matched input text to the result text, or nothing if no input text is matched. Use this, the most general result marker, unless you have special requirements. You can use it with any of the match markers, but it almost always is used with the regular match marker.
- ***Dumb stringify result marker:*** Stringifies the matched input text and writes it to the result text. If no input text is matched, it writes a null string (""). If the matched input text is a list matched by a list match marker, this result marker stringifies the entire list and writes it to the result text.

This result marker writes output to result text where a string is always required. This is generally the case for commands where a command or clause argument is specified as a literal value but the result text must always be written as a string even if the argument is not specified.

– ***Normal stringify result marker:*** Stringifies the matched input text and writes it to the result text. If no input text is matched, it writes nothing to the result text. If the matched input text is a list matched by a list match marker, this result marker stringifies each element in the list and writes it to the result text.

  The normal stringify result marker is most often used with the blockify result marker to compile an expression while saving a text image of the expression (See the SET FILTER condition and the INDEX key expression in Std.ch).

– ***Smart stringify result marker:*** Stringifies matched input text only if source text is enclosed in parentheses. If no input text matched, it writes nothing to the result text. If the matched input text is a list matched by a list match marker, this result marker stringifies each element in the list (using the same stringify rule) and writes it to the result text.

  The smart stringify result marker is designed specifically to support extended expressions for commands other than SETs with <*xlToggle*> arguments. Extended expressions are command syntax elements that can be specified as literal text or as an expression if enclosed in parentheses. The <*xcDatabase*> argument of the USE command is a typical example. For instance, if the matched input for the <*xcDatabase*> argument is the word Customer, it is written to the result text as the string "Customer," but the expression (cPath + cDatafile) would be written to the result text unchanged (i.e., without quotes).

– ***Blockify result marker:*** Writes matched input text as a code block without any arguments to the result text. For example, the input text x + 3 would be written to the result text as {| | x + 3}. If no input text is matched, it writes nothing to the result text. If the matched input text is a list matched by a list match marker, this result marker blockifies each element in the list.

The blockify result marker used with the regular and list match markers matches various kinds of expressions and writes them as code blocks to the result text. Remember that a code block is a piece of compiled code to execute sometime later. This is important when defining commands that evaluate expressions more than once per invocation. When defining a command, you can use code blocks to pass an expression to a function and procedure as data rather than as the result of an evaluation. This allows the target routine to evaluate the expression whenever necessary.

In Std.ch, the blockify result marker defines database commands where an expression is evaluated for each record. Commonly, these are field or expression lists, FOR and WHILE conditions, or key expressions for commands that perform actions based on key values.

- ***Logify result marker:*** Writes true (.T.) to the result text if any input text is matched; otherwise, it writes false (.F.) to the result text. This result marker does not write the input text itself to the result text.

  The logify result marker is generally used with the restricted match marker to write true (.T.) to the result text if an optional clause is specified with no argument; otherwise, it writes false (.F.). In Std.ch, this formulation defines the EXCLUSIVE and SHARED clauses of the USE command.

- ***Repeating result clauses*** are portions of the *<resultPattern>* enclosed by square brackets ([ ]). The text within a repeating clause is written to the result text as many times as it has input text for any or all result markers within the clause. If there is no matching input text, the repeating clause is not written to the result text. Repeating clauses, however, cannot be nested. If you need to nest repeating clauses, you probably need an additional #command rule for the current command.

  Repeating clauses are the result pattern part of the #command facility that create optional clauses which have arguments. You can match input text with any match marker other than the restricted match marker and write to the result text with any of the corresponding result markers. Typical examples of this facility are the definitions for the STORE and REPLACE commands in Std.ch.

Notes

- ***Less than operator:*** If you specify the less than operator (<) in the *<resultPattern>* expression, you must precede it with the escape character (\).
- ***Multistatement lines:*** You can specify more than one statement as a part of the result pattern by separating each statement with a semicolon. If you specify adjacent statements on two separate lines, the first statement must be followed by two semicolons.

Examples

These examples encompass many of the basic techniques you can use when defining commands with the #command and #translate directives. In general, these examples are based on standard commands defined in Std.ch. Note, however, the functions specified in the example result patterns are not the actual functions found in Std.ch, but fictitious functions specified for illustration only.

- This example defines the @...BOX command using regular match markers with regular result markers:

```
#command  @ <top>, <left>, <bottom>, <right> BOX ;
       <boxstring>;
=>;
       CmdBox( <top>, <left>, <bottom>, ;
       <right>,<boxstring> )
```

- This example uses a list match marker with a regular result marker to define the ? command:

```
#command ? [<list,...>] => QOUT(<list>)
```

- This example uses a restricted match marker with a logify result marker to implement an optional clause for a command definition. In this example, if the ADDITIVE clause is specified, the logify result marker writes true (.T.) to the result text; otherwise, it writes false (.F.):

```
#command RESTORE FROM <file> [<add: ADDITIVE>];
=>;
       CmdRestore( <(file)>, <.add.> )
```

- This example uses a list match marker with a smart stringify result marker to write to the result text the list of fields specified as the argument of a FIELDS clause. In this example, the field list is written as an array with each field name as an element of the array:

```
#command COPY TO <file> [FIELDS <fields,...>];
=>;
      CmdCopyAll( <(file)>, { <(fields)> } )
```

- These examples use the wild match marker to define a command that writes nothing to the result text. Do this when attempting to compile unmodified code developed in another dialect:

```
#command SET ECHO <*text*>     =>
#command SET TALK <*text*>     =>
```

- These examples use wild match markers with dumb stringify result markers to match command arguments specified as literals, then write them to the result text as strings in all cases:

```
#command SET PATH TO <*path*>  =>  ;
   SET( _SET_PATH, #<path> )
#command SET COLOR TO <*spec*> =>  SETCOLOR( #<spec> )
```

- These examples use a normal result marker with the blockify result marker to both compile an expression and save the text version of it for later use:

```
#command SET FILTER TO <xpr>;
=>;
      CmdSetFilter( <{xpr}>, <"xpr"> )

#command INDEX ON <key> TO <file>;
=>;
      CmdCreateIndex( <(file)>, <"key">, <{key}> )
```

- This example demonstrates how the smart stringify result marker implements a portion of the USE command for those arguments that can be specified as extended expressions:

```
#command USE <db> [ALIAS <a>];
=>;
      CmdOpenDbf( <(db)>, <(a)> )
```

- This example illustrates the importance of the blockify result marker for defining a database command. Here, the FOR and WHILE conditions matched in the input text are written to the result text as code blocks:

```
#command COUNT [TO <var>];
       [FOR <for>] [WHILE <while>];
       [NEXT <next>] [RECORD <rec>] [<rest:REST>] [ALL];
=>;
       <var> := 0,;
       DBEVAL( {|| <var>++}, <{for}>, <{while}>,;
          <next>, <rec>, <.rest.> )
```

- In this example the USE command again demonstrates the types of optional clauses with keywords in the match pattern. One clause is a keyword followed by a command argument, and the second is solely a keyword:

```
#command USE <db> [<new: NEW>] [ALIAS <a>] ;
       [INDEX <index,...>][<ex: EXCLUSIVE>] ;
       [<sh: SHARED>] [<ro: READONLY>];
=>;
       CmdOpenDbf(<(db)>, <(a)>, <.new.>,;
          IF(<.sh.> .OR. <.ex.>, !<.ex.>, NIL),;
             <.ro.>, {<(index)>})
```

- This example uses the STORE command definition to illustrate the relationship between an optional match clause and a repeating result clause:

```
#command STORE <value> TO <var1> [, <varN> ];
=>;
       <var1> := [ <varN> := ] <value>
```

- This example uses #translate to define a pseudofunction:

```
#translate AllTrim(<cString>) => LTRIM(RTRIM(<cString>))
```

See Also #define, #xcommand

#define directive

Define a manifest constant or pseudofunction

Syntax

```
#define <idConstant> [<resultText>]
#define <idFunction>([<arg list>]) [<exp>]
```

Arguments

<idConstant> is the name of an identifier to define.

<resultText> is the optional replacement text to substitute whenever a valid *<idConstant>* is encountered.

<idFunction> is a pseudofunction definition with an optional argument list (*<arg list>*). If you include *<arg list>*, it is delimited by parentheses (()) immediately following *<idFunction>*.

<exp> is the replacement expression to substitute when the pseudofunction is encountered. Enclose this expression in parentheses to guarantee precedence of evaluation when the pseudofunction is expanded.

Note: #define identifiers are case-sensitive, where #command and #translate identifiers are not.

Description

The #define directive defines an identifier and, optionally, associates a text replacement string. If specified, replacement text operates much like the search and replace operation of a text editor. As each source line from a program file is processed by the preprocessor, the line is scanned for identifiers. If a currently defined identifier is encountered, the replacement text is substituted in its place.

Identifiers specified with #define follow most of the identifier naming rules in CA-Clipper . Defined identifiers can contain any combination of alphabetic and numeric characters, including underscores. Defined identifiers, however, differ from other identifiers by being case-sensitive. As a convention, defined identifiers are specified in uppercase to distinguish them from other identifiers used within a program. Additionally, identifiers are specified with a one or two letter prefix to group similar identifiers together and guarantee uniqueness. Refer to one of the supplied header files in the \CLIP53\INCLUDE directory for examples.

When specified, each definition must occur on a line by itself. Unlike statements, more than one directive cannot be specified on the same source line. You may continue a definition on a subsequent line by employing a semicolon (;). Each #define directive is specified followed by one or more white space characters (spaces or tabs), a unique identifier, and optional replacement text. Definitions can be nested, allowing one identifier to define another.

A defined identifier has lexical scope like a filewide static variable. It is only valid in the program (.prg) file in which it is defined unless defined in Std.ch or the header file specified on the compiler command line with the /U option. Unlike a filewide static variable, a defined identifier is visible from the point where it is defined in the program file until it is either undefined, redefined, or the end of the program file is reached.

You can redefine or undefine existing identifiers. To redefine an identifier, specify a new #define directive with the identifier and the new replacement text as its arguments. The current definition is then overwritten with the new definition, and a compiler warning is issued in case the redefinition is inadvertent. To undefine an identifier, specify an #undef directive with the identifier as its argument. #define directives have three basic purposes:

- To define a control identifier for #ifdef and #ifndef
- To define a manifest constant—an identifier defined to represent a constant value
- To define a compiler pseudofunction

The following discussion expands these three purposes of the #define directive in your program.

Preprocessor Identifiers

The most basic #define directive defines an identifier with no replacement text. You can use this type of identifier when you need to test for the existence of an identifier with either the #ifdef or #ifndef directives. This is useful to either exclude or include code for conditional compilation. This type of identifier can also be defined using the /D compiler option from the compiler command line. See the examples below.

Manifest Constants

The second form of the #define directive assigns a name to a constant value. This form of identifier is referred to as a manifest constant. For example, you can define a manifest constant for the INKEY() code associated with a key press:

```
#define K_ESC  27
IF LASTKEY() = K_ESC
    .
    . <statements>
    .
ENDIF
```

Whenever the preprocessor encounters a manifest constant while scanning a source line, it replaces it with the specified replacement text.

Although you can accomplish this by defining a variable, there are several advantages to using a manifest constant: the compiler generates faster and more compact code for constants than for variables; and variables have memory overhead where manifest constants have no runtime overhead, thus saving memory and increasing execution speed. Furthermore, using a variable to represent a constant value is conceptually inconsistent. A variable by nature changes and a constant does not.

Use a manifest constant instead of a constant for several reasons. First, it increases readability. In the example above, the manifest constant indicates more clearly the key being represented than does the INKEY() code itself. Second, manifest constants localize the definition of constant values, thereby making changes easier to make, and increasing reliability. Third, and a side effect of the second reason, is that manifest constants isolate implementation or environment specifics when they are represented by constant values.

To further isolate the effects of change, manifest constants and other identifiers can be grouped together into header files allowing you to share identifiers between program (.prg) files, applications, and groups of programmers. Using this methodology, definitions can be standardized for use throughout a development organization. Merge header files into the current program file by using the #include directive.

For examples of header files, refer to the supplied header files in the \CLIP53\INCLUDE directory.

Compiler Pseudo-functions

In addition to defining constants as values, the #define directive can also define pseudofunctions that are resolved at compile time. A pseudofunction definition is an identifier immediately followed by an argument list, delimited by parentheses, and the replacement expression. For example:

```
#define AREA(nLength, nWidth)     (nLength * nWidth)
#define SETVAR(x, y)              (x := y)
#define MAX(x, y)                 (IF(x > y, x, y))
```

Pseudofunctions differ from manifest constants by supporting arguments. Whenever the preprocessor scans a source line and encounters a function call that matches the pseudofunction definition, it substitutes the function call with the replacement expression. The arguments of the function call are transported into the replacement expression by the names specified in the argument list of the identifier definition. When the replacement expression is substituted for the pseudofunction, names in the replacement expression are replaced with argument text. For example, the following invocations,

```
? AREA(10, 12)
SETVAR(nValue, 10)
? MAX(10, 9)
```

are replaced by :

```
? (10 * 12)
nValue := 10
? (IF(10 > 9, 10, 9)
```

It is important when defining pseudofunctions, that you enclose the result expression in parentheses to enforce the proper order of evaluation. This is particularly important for numeric expressions. In pseudofunctions, you must specify all arguments. If the arguments are not specified, the function call is not expanded as a pseudofunction and exits the preprocessor to the compiler as encountered.

Pseudofunctions do not entail the overhead of a function call and are, therefore, generally faster. They also use less memory. Pseudofunctions, however, are more difficult to debug within the debugger, have a scope different from declared functions and procedures, do not allow skipped arguments, and are case-sensitive.

You can avoid some of these deficiencies by defining a pseudofunction using the #translate directive. #translate pseudofunctions are not case-sensitive, allow optional arguments, and obey the dBASE four-letter rule. See the #translate directive reference in this chapter for more information.

Examples

- In this example a manifest constant conditionally controls the compilation of debugging code:

```
#define DEBUG
.
. <statements>
.
#ifdef DEBUG
   Assert(FILE("System.dbf"))
#endif
```

- This example defines a manifest constant and substitutes it for an INKEY() value:

```
#define K_ESC     27
.
. <statements>
.
IF INKEY() != K_ESC
   DoIt()
ELSE
   StopIt()
ENDIF
```

- This example defines pseudofunctions for the standard CA-Clipper functions, MAX() and ALLTRIM():

```
#define MAX(arg1, arg2)     (IF(arg1 > arg2, ;
   arg1, arg2))
#define ALLTRIM(cString)    (RTRIM(LTRIM(cString)))
.
. <statements>
.
? MAX(1, 2)
? ALLTRIM("  Hello  ")
```

See Also #command, #ifdef, #ifndef, #undef, #xcommand

#error directive

Generate a compiler error and display a message

Syntax

```
#error [<messageText>]
```

Arguments

<messageText> is the text of the message to be displayed. *<messageText>* is a literal character string—do not enclose the message in quotations unless you want them to appear as part of the display.

Description

#error causes the compiler to generate error number C2074. If the *<messageText>* parameter is specified, an error message is displayed.

Examples

- This example displays an error message based on whether or not a NETWORK identifier was defined:

```
#ifdef NETWORK
    #error Network version not implemented.
#endif
```

See Also

#command, #ifdef, #ifndef

#ifdef directive

Compile a section of code if an identifier is defined

Syntax

```
#ifdef <identifier>
   <statements>...
[#else]
   <statements>...
#endif
```

Arguments

<identifier> is the name of a definition whose existence is being verified.

Description

#ifdef...#endif lets you perform a conditional compilation. It does this by identifying a section of source code to be compiled if the specified *<identifier>* is defined. The *<identifier>* can be defined using either the #define directive or the /D compiler option which lets you define an identifier or manifest constant from the compiler command line.

The #else directive specifies the code to compile if *<identifier>* is undefined. The #endif terminates the conditional compilation block.

Conditional compilation is particularly useful when maintaining many different versions of the same program. For example, the demo code and full system code could be included in the same program file and controlled by a single #define statement.

Examples

- This code fragment is a general skeleton for conditional compilation with #ifdef:

```
#define DEMO
.
. <statements>
.
#ifdef DEMO
   <demo specific statements>
#endif
```

- This example controls conditional compilation with an identifier defined on the compiler command line with the /D option.

  In DOS:

```
C>CLIPPER Myfile /DDEBUG
```

  In the program (.prg) file:

```
#ifdef DEBUG
   Assert(<some condition>)
#endif
```

- This example defines a manifest constant to one value if it does not exist and redefines it to another if it exists:

```
#ifdef M_MARGIN
   #undef  M_MARGIN
   #define M_MARGIN  15
#else
   #define M_MARGIN  10
#endif
```

See Also

#define, #ifndef

#ifndef directive

Compile a section of code if an identifier is undefined

Syntax

```
#ifndef <identifier>
   <statements>...
[#else]
   <statements>...
#endif
```

Arguments

<identifier> is the name of a definition whose absence is being verified.

Description

#ifndef...#endif lets you perform conditional compilation by identifying a section of source code to compile if the specified *<identifier>* is undefined.

The #else directive specifies the code to compile if *<identifier>* is defined. The #endif terminates the conditional compilation block.

Examples

- This code fragment is a general skeleton for conditional compilation with #ifndef:

```
#define DEBUG
.
. <statements>
.
#ifndef DEBUG
   <optimized version of code>
#else
   <debugging version of code>
#endif
```

- This example compiles a section of code if a specific identifier is undefined.

  In DOS:

```
C>CLIPPER Myfile
```

  In the program (.prg) file:

```
#ifndef NODEBUG
   Assert(<some condition>)
#endif
```

- This example overrides a default definition in the program (.prg) file using a manifest constant defined on the compiler command line with the /D option.

  In DOS:

```
C>CLIPPER Myfile /DM_MARGIN=10
```

  In the program (.prg) file:

```
#ifndef M_MARGIN
   #define M_MARGIN  15
#endif
```

See Also #define, #ifdef

#include directive

Include a file into the current source file

Syntax

```
#include "<headerFileSpec>"
```

Arguments

<headerFileSpec> specifies the name of another source file to include in the current source file. As indicated in the syntax, the name must be enclosed in double quotation marks.

<headerFileSpec> may contain an explicit path and file name as well as a file extension. If, however, no path is specified, the preprocessor searches the following places:

- Source file directory
- Directories supplied with the /I option
- Directories specified in the INCLUDE environment variable

#include directives may be nested up to 15 levels deep—that is, a file that has been included may contain #include directives, up to 15 levels.

Description

#include inserts the contents of the specified file in place of the #include directive in the source file. By convention, the file inserted is referred to as a header file. Header files should contain only preprocessor directives and external declarations. By convention CA-Clipper header files have a .ch extension.

When deciding where to locate your header files, you have two basic choices. You can place them in the source file directory where they are local to the current system; or, you can make them globally available by placing them in the directory specified in the INCLUDE environment variable. A list of one or more directories can be specified.

Header files overcome the one major drawback of defining constants or inline functions—the #define directive only affects the file in which it is contained. This means that every program which needs access to these statements must have a list of directives at the top. The solution to this problem is to place #define statements in a separate file and use the #include directive to tell the preprocessor to include that file before compiling.

For example, suppose the file "Inkey.ch" contains a list of #define directives assigning key values to constants. Instead of including these directives at the top of each program file (.prg) requiring access to them, you can simply place the following line at the top of each program file:

```
#include "Inkey.ch"
```

This causes the preprocessor to look for Inkey.ch and place all the directives contained within it at the top of this program.

Another advantage of using the #include directive is that all the #define statements are contained in one file. If any modifications to these statements are necessary, only the #include file need be altered; the program itself remains untouched.

Note that the scope of definitions within an included header file is the current program file unless the header file is included on the compiler command line with the /U option. In this case, the scope is all the program files compiled in the current invocation of the compiler.

Notes

- ***Supplied header files:*** CA-Clipper provides a number of header files containing manifest constants for common operations. Refer to \CLIP53\INCLUDE for more information.

- ***Std.ch—the standard header file:*** Std.ch is the standard header file provided with CA-Clipper. Its default location is \CLIP53\INCLUDE. Std.ch contains the definitions of all CA-Clipper commands and the standard functions specified as pseudofunctions. It is strongly recommended that no changes be made to Std.ch. If changes are desired, it is advisable to copy Std.ch to a new name, make the changes, and compile with /U.

  This header file differs somewhat from a header file you might #include in that everything defined in Std.ch, with #define, #translate, or #command, has a scope of the entire compile rather than the current source file.

Examples

- This example uses #include to insert Inkey.ch, a file of common keyboard definitions, into a key exception handler called by an interface function:

```
#include "Inkey.ch"

FUNCTION GetEvent()
   LOCAL nKey, nResult
   nKey = INKEY(0)
   DO CASE
   CASE nKey = K_F10
      nResult := DoMenu("Browse")
   CASE nKey = K_ESC
      nResult := DoQuit()
      .
      . <statements>
      .
   CASE nKey = K_CTRL_RIGHT
      nResult := DoNextRec()
   ENDCASE
   RETURN nResult
```

See Also

#command, #define

#stdout directive

Send literal text to the standard output device

Syntax

```
#stdout [<messageText>]
```

Arguments

<messageText> is the text of the message to display. *<messageTest>* is a literal character string. Do not enclose the message in quotation marks unless you want them to appear as part of the display.

Description

#stdout causes the compiler to output the literal text to the standard output device (stdout) during compilation. If *<messageText>* is not specified, a carriage return/line feed pair echoes to stdout.

Warning! *Manifest constants are not translated in #stdout. Implementation is identical to #error with the following exceptions: output is written to STDOUT and no compiler error is generated.*

Examples

This example demonstrates use of #stdout:

```
#ifdef DEBUG
   #stdout Compiling debugging version...
#endif

PROCEDURE Main()

? "Hello world"
RETURN

#stdout End of "Hello World" program
```

See Also

#error

#undef directive

Remove a #define macro definition

Syntax

```
#undef <identifier>
```

Arguments

<identifier> is the name of the manifest constant or pseudofunction to remove.

Description

#undef removes an identifier defined with the #define directive. After an #undef, the specified identifier becomes undefined. Use #undef to remove an identifier before you redefine it with #define, preventing the compiler warning that occurs when an existing identifier is redefined. Also, use #undef to make conditional compilation specific to certain sections of a program.

Examples

- To define and then undefine a manifest constant and a pseudofunction:

```
#define K_ESC  27
#define MAX(x, y) IF(x > y, x, y)
.
. <statements>
.
#undef K_ESC
#undef MAX
```

- To use #undef to undefine an identifier before redefining it:

```
#define DEBUG
.
. <statements>
.
#undef DEBUG
#define DEBUG  .T.
```

- To undefine an identifier if it exists, and otherwise define it for later portions of the program file:

```
#ifdef TEST
   #undef TEST
#else
   #define TEST
#endif
```

See Also

#define, #ifdef, #ifndef

#xcommand | #xtranslate directive

Specify a user-defined command or translation directive

Syntax

```
#xcommand   <matchPattern> => <resultPattern>
#xtranslate <matchPattern> => <resultPattern>
```

Arguments

<matchPattern> is the pattern to match in the input text.

<resultPattern> is the text produced if a piece of input text matches the *<matchPattern>*.

Description

The #xcommand and #xtranslate directives work like #command and #translate except that they overcome the dBASE keyword length limitation. They are significant beyond the first four letters, limited only by available memory. All other rules apply.

See Also

#command

$ operator

Substring comparison—binary (Relational)

Syntax

```
<cString1> $ <cString2>
```

Type

Character, memo

Operands

<cString1> is a character or memo value that is searched for in *<cString2>*.

<cString2> is a character or memo value within which *<cString1>* is sought.

Description

The $ operator is a binary relational operator that performs a case-sensitive substring search and returns true (.T.) if *<cString1>* is found within *<cString2>*.

Examples

- This example illustrates the case-sensitivity of the substring operator ($):

```
? "A" $ "ABC"            // Result: .T.
? "a" $ "ABC"            // Result: .F.
```

See Also <, <=, <>, = (equality), ==, >, >=

% operator

Modulus—binary (Mathematical)

Syntax

```
<nNumber1> % <nNumber2>
```

Type

Numeric

Operands

<nNumber1> is the dividend of the division operation.

<nNumber2> is the divisor of the division operation.

Description

% returns a number representing the remainder of *<nNumber1>* divided by *<nNumber2>*.

Notes

- Seeking the modulus of any dividend using a zero as the divisor causes a runtime error. In versions of CA-Clipper prior to Summer '87, a modulus operation with a zero divisor returned zero.

Examples

- This example shows modulus results using different operands:

```
?  3 %  0           // Result: Runtime error
?  3 % -2           // Result:  1
? -3 %  2           // Result: -1
? -3 %  0           // Result: Runtime error
? -1 %  3           // Result: -1
? -2 %  3           // Result: -2
?  2 % -3           // Result:  2
?  1 % -3           // Result:  1
```

See Also

*, **, +, -, /, = (compound assign), MOD()*, SET FIXED

& operator

Macro evaluation—unary (Special)

Syntax

```
&<cMacroVar>[.]
&(<cMacroExp>)
```

Operands

<cMacroVar> can be any character variable. The period (.) is the macro terminator and indicates the end of the macro variable and distinguishes the macro variable from adjacent text in the statement.

<cMacroExp> is a character expression enclosed in parentheses. In this instance, the expression is evaluated first, and the macro operation is performed on the resulting character value. This allows the contents of fields and array elements to be compiled and run.

Description

The macro operator in CA-Clipper is a special operator that allows runtime compilation of expressions and text substitution within strings. Whenever the macro operator (&) is encountered, the operand is submitted to a special runtime compiler (the macro compiler) that compiles expressions, but not statements or commands.

Text Substitution

Whenever a reference to a private or public macro variable, embedded in a character string, is encountered, the variable reference is replaced by the content of the macro variable. For example,

```
cMacro := "there"
? "Hello &cMacro"          // Result: Hello there
```

If you specify a macro expression (e.g., &(cMacro1 + cMacro2)), and the macro variable is a local, static, field variable, or an array element, it is treated as literal text and not expanded.

Compile and Run

When a macro variable or expression specified within an expression is encountered, it is treated like an expression, with the macro symbol behaving as the compile and run operator. If the macro is specified as a macro variable,

```
cMacro := "DTOC(DATE())"
? &cMacro
```

the macro compiler compiles, then executes the content of the macro variable. The compiled code is then discarded.

If you specify an expression enclosed in parentheses and prefaced by the macro operator (&),

```
? &(INDEXKEY(0))
```

the expression is evaluated and the resulting character string is compiled and run as a macro variable.

Using the macro operator, you can compile a character string containing a code block definition:

```
bBlock := &("{ |exp| QOUT(exp) }")
```

The run portion of the operation returns the code block as a value. You may then use the code block by invoking it with the EVAL() function. This is especially significant in activations that involve extensive looping through user-defined conditions (operations that in earlier versions of CA-Clipper required macro expansion). In those versions, the macro expression was compiled and run for each iteration of the loop. With the combination of a macro expansion and a code block EVAL(), the compilation is performed once at compile time, and the EVAL() merely executes the code block each time through the loop:

```
EVAL(bBlock, DATE())
```

The time savings at runtime can be enormous.

Notes

- ***Command keywords:*** You cannot use the macro operator (&) to substitute or compile command keywords. However, you can redefine command keywords by modifying the command definition in Std.ch, overriding an existing command definition with a new definition using the #command directive, or redefining a command keyword using the #translate directive. In any case, you may redefine a command keyword only at compile time, not at runtime.

- ***Command arguments:*** In prior versions of CA-Clipper as well as in other dialects, you could use macro variables as the arguments of commands requiring literal text values. These included all file command arguments and SET commands with toggle arguments. In these instances, you can now use an extended expression enclosed in parentheses in place of the literal argument. For example,

  ```
  xcDatabase = "Invoices"
  USE &xcDatabase.
  ```

  can be replaced with:

  ```
  xcDatabase = "Invoices"
  USE (xcDatabase)
  ```

It is important to use extended expressions if you are using local and static variables. Generally, commands are preprocessed into function calls with command arguments translated into function arguments as valid CA-Clipper values. File names in file commands, for instance, are stringified using the smart stringify result marker and passed as arguments to the functions that actually perform the desired actions. If you specify a literal or macro value as the command argument, it is stringified. If, however, the argument is an extended expression, it is written to the result text exactly as specified. This example,

```
#command RENAME <xcOld> TO <xcNew>;
=>;
      FRENAME( <(xcOld)>, <(xcNew)> )
//
RENAME &xcOld TO &xcNew
RENAME (xcOld) TO (xcNew)
```

is written to the result text as this:

```
FRENAME( "&xcOld", "&xcNew" )
FRENAME( xcOld, xcNew )
```

when preprocessed. When the macro variables are stringified, the macro variable names are hidden in the string and not compiled. Later, at runtime, they are substituted into the string and passed as arguments to the FRENAME() function. This precludes local and static macro variables since the names of the variables are not present at runtime to be substituted. Public and private variables, however, behave as expected.

- ***Lists as arguments of commands:*** The macro operator (&) will not fully substitute or compile a list as an argument of most commands. In particular, these are commands where an argument list is preprocessed into an array or a code block. Instances of this are arguments of the FIELDS clause and SET INDEX. An exception is the SET COLOR command which preprocesses the list of colors into a single character string and passes it to the SETCOLOR() function.

  In any case, list arguments should always be specified as extended expressions with each list argument specified:

```
LOCAL xcIndex := { "Ntx1", "Ntx2" }
SET INDEX TO (xcIndex[1]), (xcIndex[2])
```

- ***Arrays:*** You can use the macro operator (&) with arrays and array elements. However, because of the increased power of CA-Clipper arrays, you may find less need to use the macro operator (&) to make variable references to arrays. You can now assign array references to variables, return array references from user-defined functions, and nest array references within other arrays. You may also create arrays by specifying literal arrays or using the ARRAY() function.

  You can, therefore, make references to arrays and array elements using both macro variables and macro expressions with the restriction that you cannot make the subscript references in a PRIVATE or PUBLIC statement. Also, you cannot specify the macro operator (&) in a declaration statement, such as a LOCAL or STATIC statement. Attempting this will generate a fatal compiler error.

  This example references array elements using macro variables:

```
cName := "aArray"
nElements := 5
cNameElement := "aArray[1]"
//
PRIVATE &cName.[nElements]     // Creates "array" with 5
                               // elements
&cNameElement. := 100          // Assigns 100 to element 1
&cName.[3] := "abc"            // Assigns "abc" to element 3
```

  You can successfully apply a macro operator (&) to an array element if the reference is made using a macro expression. A macro variable reference, however, will generate a runtime error. For example, the following lists the values of all fields of the current record:

```
USE Customer NEW
aStruc := DBSTRUCT()
//
FOR nField := 1 TO LEN(aStruc)
   ? &(aStruc[nField, 1])
NEXT
```

- ***Code blocks:*** You can apply the macro operator (&) to a macro variable or expression in a code block in most cases. There is a restriction when the macro variable or macro expression contains a declared variable. A runtime error occurs if you specify a complex expression (an expression that contains an operator and one or more operands) that includes the macro operator (&) within a code block.

  This has important implications for the use of local and static variables in the conditional clauses of commands, since these clauses are blockified as they are written to the result text during preprocessing. This applies to all FOR and WHILE clauses, the SET FILTER command, and the SET RELATION linking expression. The general workaround is to gather the entire expression into a single macro variable then apply the macro operator (&) to the variable.

- ***Macro conditions:*** When using the macro operator (&) to specify conditional clauses of database commands such as FOR or WHILE clauses, there are some restrictions based on the expression's complexity and size:

  - The maximum string size the macro compiler can process is 254 characters.

  - There is a limit to the complexity of conditions (the more complex, the fewer the number of conditions you can specify).

- ***Procedures and functions:*** You can reference procedure and function calls using macro variables and expressions. With DO, the macro variable reference to the procedure can be all or part of the procedure name. With a call to a function (built-in or user-defined), the macro variable reference must include the function name and all of its arguments.

  In CA-Clipper, because of the added facility code blocks, all invocations of procedures and functions using the macro operator should be converted to the evaluation of code blocks. This code fragment:

  ```
  cProc := "AcctsRpt"
  .
  .
  .
  DO &cProc
  ```

  can be replaced with:

  ```
  bProc := &( "{ || AcctsRpt() }" )
  .
  .
  .
  EVAL(bProc)
  ```

  The advantage of a code block over a macro evaluation is that the result of the compilation of a string containing a code block can be saved and, therefore, need only be compiled once. Macro evaluations compile each time they are referenced.

- ***References into overlays:*** You must declare procedures and user-defined functions that are used in macro expressions and variables but not referenced elsewhere as EXTERNAL, or the linker will not include them into the executable (.EXE) file.

- ***TEXT...ENDTEXT:*** Macro variables referenced within a TEXT...ENDTEXT construct are expanded. Note that a field cannot be expanded, so you must first assign the field value to a memory variable then reference the memory variable as a macro variable within the TEXT...ENDTEXT. For example:

  ```
  USE Customer NEW
  myVar := Customer->CustName
  TEXT
  This is text with a macro &myVar
  ENDTEXT
  ```

- ***Nested macros:*** The processing of macro variables and expressions in CA-Clipper permits nested macro definitions. For example, after assigning a macro variable to another macro variable, the original macro variable can be expanded resulting in the expansion of the second macro variable and evaluation of its contents:

```
cOne = "&cTwo"              // expand cTwo
cTwo = "cThree"             // yielding "cThree"
cThree = "hello"
//
? &cOne                     // Result: "hello"
```

See Also ( ) operator

( ) operator

Function or grouping indicator (Special)

Description

Parentheses (()) in expressions are used to group certain operations for readability or to force a particular evaluation order. Parentheses also identify a function call.

When specifying the grouping operator, the item that falls within the parentheses must be a valid expression. Subexpressions may be further grouped.

For function calls, a valid function name must precede the left parenthesis, and the function arguments, if any, must be contained within the parentheses.

Examples

- This example changes the default order of expression evaluation:

```
? 5 * (10 + 6) / 2          // Result: 40
```

- The next example shows a function call:

```
x := SQRT(100)
```

See Also

& operator

* operator

Multiplication—binary (Mathematical)

Syntax

<nNumber1> * *<nNumber2>*

Type

Numeric

Operands

<nNumber1> and ***<nNumber2>*** are the two numeric expressions to multiply.

Description

The multiplication operator (*) is a binary operator that returns a numeric value from its operation on *<nNumber1>* and *<nNumber2>*.

Examples

- This example shows multiplication results using different operands:

```
?  3 *  0          // Result:  0
?  3 * -2          // Result: -6
? -3 *  2          // Result: -6
? -3 *  0          // Result:  0
? -1 *  3          // Result: -3
? -2 * -3          // Result:  6
?  2 * -3          // Result: -6
?  1 * -3          // Result: -3
```

See Also

%, **, +, -, /, = (compound assign), SET DECIMALS, SET FIXED

** operator

Exponentiation—binary (Mathematical)

Syntax

```
<nNumber1> ** <nNumber2>
<nNumber1> ^  <nNumber2>
```

Type

Numeric

Operands

<nNumber1> is the numeric value to raise to a power.

<nNumber2> is the power to raise *<nNumber1>*.

Description

The exponentiation operator (**) is a binary operator that raises *<nNumber1>* to the power of *<nNumber2>*.

Examples

- This example shows exponentiation results to 16 decimal places, using different operands:

```
SET DECIMALS TO 16
?  3 **  0          // Result:  1.000000000
?  3 **  1          // Result:  3.000000000
?  3 ** -2          // Result:  0.111111111
? -3 **  2          // Result:  9.000000000
? -3 **  0          // Result:  1.000000000
? -1 **  3          // Result: -1.000000000
? -2 **  3          // Result: -8.000000000
?  2 ** -3          // Result:  0.125000000
?  1 ** -3          // Result:  1.000000000
```

See Also

%, *, +, -, /, = (compound assign), SET DECIMALS, SET FIXED

+ operator

Addition, unary positive, concatenation (Mathematical, Character)

Syntax

```
<nNumber1> + <nNumber2>        (addition)
<dDate>    + <nNumber>         (addition)
<cString1> + <cString2>        (concatenation)
```

Type

Character, date, memo, numeric

Operands

<nNumber1> is a numeric value to increment by ***<nNumber2>***.

<dDate> is a date value to increment by ***<nNumber>*** days.

<cString2> is a character string to join to the end of ***<cString1>***.

Description

The (+) operator performs a number of different operations depending on the data types of the operands:

- ***Unary positive sign (numeric):*** A numeric expression prefaced with the plus (+) operator performs no operation on the operand except to enforce a higher level of precedence than other numeric operations (except the unary minus).

- ***Binary addition sign (numeric, date):*** If both operands are numeric, *<nNumber2>* is added to *<nNumber1>* and the result is returned as a numeric value. If either operand is date data type and the other operand numeric data type, the *<nNumber>* is added as days to the *<dDate>* and a date value is returned.

- ***Concatenation (character, memo):*** If both operands are character, *<cString2>* (the right operand) is concatenated to *<cString1>* (the left operand) returning a character string.

Examples

- These examples illustrate the various forms of the + operator:

```
// Binary addition (numeric)
? 1 + 1                                // Result: 2
? 1 + 0                                // Result: 1
? 0 + 1                                // Result: 1

// Binary addition (date)
? CTOD("12/12/88") + 12          // Result: 12/24/88
? 12 + CTOD("12/12/88")          // Result: 12/24/88

// Concatenation (character)
? "Hi " + "there"                // Result: Hi there
```

See Also

%, *, **, -, /, = (compound assign)

++ operator

Increment—unary (Mathematical)

Syntax

```
++<idVar>         (prefix increment)
<idVar>++         (postfix increment)
```

Type

Date, numeric

Operands

<idVar> is any valid CA-Clipper identifier including a field variable. If *<idVar>* is a field, it must either be prefaced with an alias or declared with the FIELD statement.

The prefix increment can only be performed on an initialized value of numeric or date data type.

Description

The increment operator (++) increases the value of its operand by one. This operator adds one to the value of *<idVar>* and assigns the new value to *<idVar>*.

The ++ operator can appear before or after *<idVar>*. Specifying the operator before *<idVar>* increments and assigns the value before *<idVar>* is used. This is called prefix notation, and it is the most common usage. Specifying the operator after *<idVar>*, postfix notation, increments and assigns the value after *<idVar>* is used. Stated differently, postfix notation delays the assignment portion of the operation until the rest of the expression is evaluated, and prefix notation gives the assignment precedence over all other operations in the expression.

If the reference to *<idVar>* is ambiguous (i.e., not declared at compile time and not explicitly qualified with an alias), *<idVar>* is always assumed to be MEMVAR. You can assign field variables by declaring the field variable name in a FIELD statement or referring to the field name prefaced by the FIELD-> alias or the name of the work area.

Examples

- This code uses the prefix increment operator in an assignment statement. Therefore, both variables have the same value when queried:

```
nValue := 0
nNewValue := ++nValue
? nNewValue              // Result:   1
? nValue                 // Result:   1
```

- In this example, the postfix increment operator increases the first operand of the multiplication operation by one, making its value 11; however, the assignment of this new value to the *nValue* variable will not take place until the entire expression is evaluated. Therefore, its value is still 10 when the multiplication operation occurs, and the result of 11 * 10 is 110. Finally, when *nValue* is queried again after the expression is evaluated, the postfix increment assignment is reflected in its new value, 11.

```
nValue := 10
? nValue++ * nValue      // Result: 110
? nValue                 // Result:  11
```

See Also +, --, :=, = (compound assign)

- operator

Subtraction, unary negative, concatenation (Mathematical, Character)

Syntax

```
<nNumber1> - <nNumber2>        (subtraction)
<dDate1>   - <dDate2>          (subtraction)
<dDate>    - <nNumber>         (subtraction)
<cString1> - <cString2>        (concatenation)
```

Type

Character, date, memo, numeric

Operands

<nNumber1> is a numeric value to decrement by ***<nNumber2>***.

<dDate1> is a date value to decrement by ***<dDate2>***.

<dDate> is a date value to decrement by ***<nNumber>*** days.

<cString2> is a character string to join to the end of ***<cString1>*** after trimming the trailing blanks from *<cString1>*.

Description

The (-) operator performs different operations depending on the data types of the operands:

- ***Unary negative sign (numeric):*** A numeric expression prefaced with the minus (-) operator performs the equivalent of multiplying the operand by (-1), returning a negative numeric value.

- ***Binary subtraction (numeric, date):*** If both operands are numeric data type, <*nNumber2*> is subtracted from <*nNumber1*> and the result is returned as a numeric value. If both operands are date data type, <*dDate2*> is subtracted from <*dDate1*> and the result is returned as a numeric value representing the number of days between the two dates. If the left operand is date data type and the right operand numeric data type, the <*nNumber*> is subtracted as days from <*dDate*> and a date value is returned. If the order of operands is reversed, a runtime error occurs.

- ***Concatenation (character, memo):*** If both operands are character data type, <*cString2*> is joined to <*cString1*>, returning a character string. Trailing blanks are trimmed from <*cString1*> and joined to the end of the return string.

Examples

- These examples illustrate the various forms of the - operator:

```
// Binary subtraction (numeric)
? 1 - 1                               // Result: 0
? 1 - 0                               // Result: 1
? 0 - 1                               // Result: -1

// Binary subtraction (date)
? CTOD("12/12/88") - 12         // Result: 11/30/88
? 12 - CTOD("12/12/88")         // Result: Runtime error

// Concatenation (character)
? "Hi " - "There" + "*"         // Result: Hithere *
```

See Also

%, *, **, +, /, = (compound assign)

-- operator

Decrement—unary (Mathematical)

Syntax

```
--<idVar>     (prefix decrement)
<idVar>--     (postfix decrement)
```

Type

Date, numeric

Operands

<idVar> is any valid CA-Clipper identifier, including a field variable. If *<idVar>* is a field, you must reference it prefaced with an alias or declare it with the FIELD statement.

In addition, you must initialize *<idVar>* to a value before performing the decrement operation, and it must be either numeric or date data type.

Description

The decrement operator (--) decreases the value of its operand by one. This operator subtracts one from the value of *<idVar>* and assigns the new value to *<idVar>*.

The -- operator can appear before or after *<idVar>*. Specifying the operator before *<idVar>* decrements and assigns the value before the value is used. This is called prefix notation, and it is the most common usage. Specifying the operator after *<idVar>* decrements and assigns the value after it is used. This is postfix notation. Stated differently, postfix notation delays the assignment portion of the operation until the rest of the expression is evaluated, and prefix notation gives the assignment precedence over all other operations in the expression.

If the reference to *<idVar>* is ambiguous (i.e., not declared at compile time and not explicitly qualified with an alias), *<idVar>* is always assumed to be MEMVAR. You can assign field variables by declaring the field variable name in a FIELD statement or by referring to the field name prefaced by the FIELD-> alias or by the name of the work area.

Examples

- In this example of the postfix decrement operator the assignment takes place before the original variable is decremented, thus the two values are not the same when queried:

```
nValue := 1
nNewValue := nValue--
? nNewValue                    // Result:   1
? nValue                       // Result:   0
```

- In this example, the prefix decrement operator decreases the first operand of the multiplication by one, making its value 9. Then, the assignment of this new value to the *nValue* variable takes place before the rest of the expression is evaluated. Thus, the new value is used to perform the multiplication operation, and the result of 9 * 9 is 81.

```
nValue := 10
? --nValue * nValue            // Result:  81
? nValue                       // Result:   9
```

See Also

++, -, :=, = (compound assign)

-> operator

Alias assignment—binary (Special)

Syntax

```
<idAlias>-><idField>
<idAlias>->(<exp>)
(<nWorkArea>)-><idField>
(<nWorkArea>)->(<exp>)
FIELD-><idVar>
MEMVAR-><idVar>
```

Operands

<idAlias> is the name of the unselected work area to access and must refer to a work area with a database file in USE.

<nWorkArea> is the number of the unselected work area to access.

<idField> is the name of a field in the specified work area.

<exp> is an expression of any data type to be executed in the specified work area. If an expression more complicated than a single field reference is used as the second operand, the expression must be enclosed in parentheses ( ).

<idVar> is any valid CA-Clipper identifier. Depending on whether you specify FIELD or MEMVAR, the identifier will be forced to a database field or memory variable (public or private) reference.

Description

When used with an *<idAlias>* as the first operand, the alias operator (->) accesses field information or evaluates an expression in the indicated work area. The alias operator implicitly SELECTs the *<idAlias>* before evaluating the *<idField>* or *<exp>* operand. When the evaluation is complete, the original work area is SELECTed again. An alias reference can be in an expression or on a line by itself:

```
? Customer->Name
Customer->(UpdateTransaction())
```

Using the alias operator lets you:

- Access information from unselected work areas within expressions
- Access environmental information from unselected work areas
- Access information from unselected work areas in modes such as REPORT and LABEL FORMs
- Write more compact code

In addition to allowing expression and field evaluation in unselected work areas, the alias operator makes an explicit reference to a field or variable using either the FIELD or the MEMVAR keyword aliases. MEMVAR forces *<idVar>* to refer to a memory variable name, and FIELD forces it to reference a database field. These special alias identifiers allow you to avoid ambiguity when there are conflicts between field and memory variable names. Remember that a reference to a variable identifier not prefaced with an alias defaults to a field if there are both field and memory variables with the same name. To override this, use the (/V) option when compiling.

In addition to specifying the alias as an identifier, you can access the target work area using an expression that returns the work area number if the expression is enclosed by parentheses. This lets you use work area numbers as handles, which is useful for passing references to work areas without using macros, aliases, names, etc.

Examples

- This example accesses database and work area information in an unselected work area:

```
USE Customer NEW
USE Invoices NEW
? Customer->CustName              // Result: Bill Smith
? Customer->(RECNO())             // Result: 1
? Customer->(FOUND())             // Result: .F.
? Customer->(City + ", " + State + ;
    "  " + Zip)                   // Result: ShadowVille,
                                  //         CA  90415
```

- This example uses a user-defined function (MySeek()) as an operand of the alias operator for a common operation that normally requires many more statements:

```
IF Invoices->(MySeek(CustNum))
   <process customer>...
ELSE
   <process no find>...
ENDIF
RETURN

FUNCTION MySeek( cSearch )
   SEEK cSearch
RETURN (FOUND())
```

Note: This example is just an illustration of the alias operator with a user-defined function. DBSEEK() could be used instead of MySeek().

- This example explicitly references field and memory variables with the same name:

```
USE Customer NEW
MEMVAR->CustName = "Bill Smith"     // Create a memvar
                                    // CustName
LOCATE FOR MEMVAR->CustName = FIELD->CustName
```

- This example uses an expression as a work area handle to create a work area-independent database operation:

```
cTable1 := "C:Myfile.dbf"
cTable2 := "D:Myfile.dbf"
USE (cTable1) NEW
hArea1 = SELECT()
USE (cTable2) NEW
hArea2 = SELECT()
DoStuff( hArea1, hArea2 )

FUNCTION DoStuff( hArea1, hArea2 )
   LOCAL nCount, nSave
   nSave := SELECT()
   SELECT (hArea1)
   FOR nCount := 1 TO FCOUNT()
      FIELDPUT( nCount, ( hArea2 )-> ;
         ( FIELDGET( nCount )))
   NEXT
   SELECT (nSave)
   RETURN NIL
```

See Also FIELD, FIELDGET(), FIELDNAME(), FIELDPOS(), SELECT

.AND. operator

Logical AND—binary (Logical)

Syntax

```
<lCondition1> .AND. <lCondition2>
```

Type

Logical

Operands

<lCondition1> and ***<lCondition2>*** are logical expressions to AND.

Description

The .AND. operator is a binary logical operator that executes a logical AND operation using the following modified Boolean rules:

- Returns true (.T.) if both *<lCondition1>* and *<lCondition2>* evaluate to true (.T.)
- Returns false (.F.) if either *<lCondition1>* and *<lCondition2>* evaluate to false (.F.)

Warning! *In a departure from Summer '87 and other dialect behavior, CA-Clipper shortcuts the evaluation of .AND. operands. This means that <lCondition1> and <lCondition2> are both evaluated only when <lCondition1> evaluates to true (.T.). If <lCondition1> evaluates to false (.F.), the .AND. operation is false (.F.) and <lCondition2> is therefore not evaluated.*

For backward compatibility, shortcutting can be suppressed by compiling with the /Z option.

Examples

- This example shows .AND. results using different operands:

```
? .T. .AND. .T.          // Result: .T.
? .T. .AND. .F.          // Result: .F.
? .F. .AND. .T.          // Result: .F. (shortcut)
? .F. .AND. .F.          // Result: .F. (shortcut)
```

See Also

.NOT., .OR.

.NOT. operator

Logical NOT—unary (Logical)

Syntax

```
! <lCondition>
.NOT. <lCondition>
```

Type

Logical

Operands

<lCondition> is a logical expression to NOT.

Description

The .NOT. (!) operator is a unary logical operator that returns the logical inverse of *<lCondition>*.

Examples

- This example shows .NOT. results using different operands:

```
? ! (.T.)               // Result: .F.
? ! 1 > 2               // Result: .T.
? .NOT. 1 > 2           // Result: .T.
```

See Also

.AND., .OR.

.OR. operator

Logical OR—binary (Logical)

Syntax

```
<lCondition1> .OR. <lCondition2>
```

Type

Logical

Operands

<lCondition1> and ***<lCondition2>*** are logical expressions.

Description

The .OR. operator is a binary logical operator that executes a logical OR operation using the following modified Boolean rules:

- Returns true (.T.) if either *<lCondition1>* or *<lCondition2>* evaluates to true (.T.)
- Returns false (.F.) if both *<lCondition1>* and *<lCondition2>* evaluates to false (.F.)

Warning! *In a departure from the Summer '87 and other dialect behavior, CA-Clipper shortcuts the evaluation of .OR. operands. This means that <lCondition1> and <lCondition2> are both evaluated only when <lCondition1> evaluates to false (.F.). If <lCondition1> evaluates to true (.T.), the .OR. operation is true (.T.) and <lCondition2> is, therefore, not evaluated. For backward compatibility, shortcutting can be suppressed by compiling with the /Z option.*

Examples

- This example shows .OR. results using different operands:

```
? .T. .OR. .T.          // Result: .T. (shortcut)
? .T. .OR. .F.          // Result: .T. (shortcut)
? .F. .OR. .T.          // Result: .T.
? .F. .OR. .F.          // Result: .F.
```

See Also

.AND., .NOT.

/ operator

Division—binary (Mathematical)

Syntax

<nNumber1> / *<nNumber2>*

Type

Numeric

Operands

<nNumber1> is the dividend of the division operation.

<nNumber2> is the divisor of the division operation.

Description

The division operator (/) returns a numeric value from the division of *<nNumber1>* by *<nNumber2>*.

Note: Dividing *<nNumber1>* with zero value for *<nNumber2>* results in a runtime error.

Examples

- This example shows division results to 16 decimal places, using different operands:

```
SET DECIMALS TO 16
?  3 /  0         // Result: Runtime error
?  3 / -2         // Result: -1.500000000
? -3 /  2         // Result: -1.500000000
? -3 /  0         // Result: Runtime error
? -1 / -3         // Result: +0.333333333
? -2 /  3         // Result: -0.666666667
?  2 / -3         // Result: -0.666666667
?  1 / -3         // Result: -0.333333333
```

See Also

%, *, **, +, -, = (compound assign), SET DECIMALS, SET FIXED

: operator

Send—binary (Object)

Syntax

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

Type

Object

Operands

<object> is the name of the object that is to receive the message.

<message> is the name of a method or instance variable.

<argument list> is a list of parameters that are passed to the specified message. The parentheses surrounding the argument list are optional if no parameters are supplied in the message send. By convention, however, parentheses distinguish a message send with no parameters from an access to an exported instance variable.

Description

Each class defines a set of operations that can be performed on objects of that class. Perform these operations by sending a message to the object using the send operator (:).

When a message is sent to an object, the system examines the message. If the object is of a class that defines an operation for that message, the system automatically invokes a method to perform the operation on the specified object. If the class does not define a method for the specified message, a runtime error occurs.

Executing a message send produces a return value, much like a function call. The return value varies depending on the operation performed.

Examples

- In this example, myBrowse is the name of a variable that contains a TBrowse object reference. pageUp() is a method that specifies the operation to be performed. The available operations and corresponding methods vary depending on the class of the object. The Error, Get, TBColumn, and TBrowse classes are documented in this chapter.

```
myBrowse:pageUp()
```

- This example forces the checkbox state to true (.T.):

```
myCheck : Select (.T.)
```

:= operator

Inline assign—binary (Assignment)

Syntax

```
<idVar> := <exp>
```

Type

All

Operands

<idVar> is a valid variable identifier of any storage class, including a field variable. If *<idVar>* is not visible or does not exist, a private variable is created and assigned the result of *<exp>*.

<exp> is the expression whose result is assigned to *<idVar>*.

Description

The inline assignment operator (:=) assigns values of any data type to variables of any storage class. The operator evaluates *<exp>*, assigns the resulting value to *<idVar>*, and returns the resulting value as the return value of the operation. This latter behavior permits multiple assignments such as idVar1 := idVar2 := value. Unlike the simple assign operator (=), the inline assignment operator (:=) can be used anywhere an expression or constant is allowed, including variable declaration statements. With a variable declaration statement, assignment initializes a variable. Note, however, you cannot declare an array and initialize elements within the same program statement.

If the reference to *<idVar>* is ambiguous (i.e., not declared at compile time and not explicitly qualified with an alias), *<idVar>* is always assumed to be MEMVAR. At runtime, if no private or public variable exists with the specified name, a private variable is created. You can assign field variables with the := operator by declaring the field variable name in a FIELD statement or referring to the field name prefaced by the FIELD-> alias or the name of the work area.

Examples

- Several examples of inline assignment follow:

```
LOCAL nValue := 10
//
IF (dDate := (DATE() - 1000)) = CTOD("12/20/79")
//
? SQRT(nValue := (nValue ** 2))
//
cTransNo := cSortNo := (CustId + DTOC(DATE()))
```

- This last example performs multiple assignments using the inline assignment operator as you would with the STORE command. When := is used in this way, the assignments are executed from right to left. This feature is particularly useful when you need to store the same value to many different fields, possibly in different database files.

```
CustFile->CustId := TransFile-> ;
    TransNo := (CustId + DTOC(DATE())
```

See Also ++, --, = (assign), = (compound assign), STORE*

< operator

Less than—binary (Relational)

Syntax

```
<exp1> < <exp2>
```

Type

Character, date, logical, memo, numeric

Operands

<exp1> and ***<exp2>*** are expressions of the same data type to be compared.

Description

The less than (<) operator is a binary operator that compares two values of the same data type and returns true (.T.) if *<exp1>* is less than *<exp2>*.

- ***Character:*** The comparison is based on the underlying ASCII code. ASCII codes for alphabetic characters are ascending (e.g., the code for "A" is 65 and the code for "Z" is 90).
- ***Date:*** Dates are compared according to the underlying date value.
- ***Logical:*** False (.F.) is less than true (.T.).
- ***Memo:*** Treated the same as character.
- ***Numeric:*** Compared based on magnitude.

Examples

- These examples illustrate how the less than operator (<) behaves with different data types:

```
// Character
? "Z" < "A"                 // Result: .F.
? "ZA" < "A"                // Result: .F.
? "A" < "AZ"                // Result: .T.

// Date
? CTOD("12/12/88") < ;
   CTOD("12/11/88")         // Result: .F.

// Logical
? .F. < .T.                 // Result: .T.

// Numeric
? 2 < 1                     // Result: .F.
? 1 < 2                     // Result: .T.
```

See Also

$, >, <=, <>, = (equality), ==, >=

<= operator

Less than or equal—binary (Relational)

Syntax

<exp1> <= *<exp2>*

Type

Character, date, logical, memo, numeric

Operands

<exp1> and ***<exp2>*** are expressions of the same data type to compare.

Description

The less than or equal (<=) operator compares two values of the same data type and returns true (.T.) if *<exp1>* is less than or equal to *<exp2>*.

- ***Character:*** The comparison is based on the underlying ASCII code. ASCII codes for alphabetic characters are ascending (e.g., the code for "A" is 65 and the code for "Z" is 90).
- ***Date:*** Dates are compared according to the underlying date value.
- ***Logical:*** False (.F.) is less than true (.T.).
- ***Memo:*** Treated the same as character.
- ***Numeric:*** Compared based on magnitude.

Examples

- These examples illustrate how the less than or equal operator (<=) behaves with different data types:

```
// Character
? "Z" <= "A"                // Result: .F.
? "ZA" <= "A"               // Result: .F.
? "A" <= "AZ"               // Result: .T.

// Date
? CTOD("12/12/88") <= ;
   CTOD("12/11/88")         // Result: .F.

// Logical
? .F. <= .T.                // Result: .T.

// Numeric
? 2 <= 1                    // Result: .F.
? 1 <= 2                    // Result: .T.
? 1 <= 1                    // Result: .T.
```

See Also

$, <, <>, = (equality), ==, >, >=

<> != # operator

Not equal—binary (Relational)

Syntax

```
<exp1> <> <exp2>
<exp1> != <exp2>
<exp1> #  <exp2>
```

Type

Character, date, logical, memo, NIL, numeric

Operands

<exp1> and ***<exp2>*** are expressions of the same data type or NIL to be compared for inequality.

Description

The not equal (<>) operator compares two values of the same data type and returns true (.T.) if *<exp1>* is not equal to *<exp2>* according to the following rules:

- ***Character:*** The comparison is based on the underlying ASCII code and is the inverse of the equal operator (=). This means that the comparison is sensitive to the current EXACT SETting. See the examples below.
- ***Date:*** Dates are compared according to the underlying date value.
- ***Logical:*** False (.F.) is not equal to true (.T.).
- ***Memo:*** Treated the same as character.
- ***NIL:*** All values compared to NIL, other than NIL, return true (.T.).
- ***Numeric:*** Compared based on magnitude.

Examples

- These examples illustrate how the not equal operator (<>) behaves with different data types:

```
// Character
SET EXACT ON
? "123" <> "12345"          // Result: .T.
? "12345" <> "123"          // Result: .T.
? "123" <> ""               // Result: .T.
? "" <> "123"               // Result: .T.
SET EXACT OFF
? "123" <> "12345"          // Result: .T.
? "12345" <> "123"          // Result: .F.
? "123" <> ""               // Result: .F.
? "" <> "123"               // Result: .T.

// Date
? CTOD("12/12/88") <> ;
   CTOD("12/12/88")         // Result: .F.

// Logical
? .T. <> .T.                // Result: .F.
? .T. <> .F.                // Result: .T.

// NIL
? NIL <> NIL                // Result: .F.
? NIL <> 12                 // Result: .T.
? NIL <> "hello"            // Result: .T.

// Numeric
? 2 <> 1                    // Result: .T.
? 1 <> 1                    // Result: .F.
```

See Also

$, <, <=, = (equality), ==, >, >=

= (assign) operator

Simple assign—binary (Assignment)

Syntax

```
<idVar> = <exp>
```

Type

All

Operands

<idVar> is a valid variable identifier of any storage class, including a field variable. If *<idVar>* is not visible or does not exist, a private variable is created and assigned the result of *<exp>*.

<exp> is the expression whose result is assigned to *<idVar>*.

Description

The simple assignment operator (=) assigns a value to a variable. It is identical in operation to the STORE command that initializes a single variable and must be specified as a program statement. The inline assignment operator (:=) is like the = operator except that you can specify it within expressions. If you specify the simple assign operator (=) within an expression, it is interpreted as the equality (=) operator.

Note: You cannot initialize a specific variable using the simple assign operator (=) in a declaration statement. Only the inline assign (:=) operator can be used for this purpose.

If the reference to *<idVar>* is ambiguous (i.e., not declared at compile time and not explicitly qualified with an alias), *<idVar>* is always assumed to be MEMVAR. At runtime, if no private or public variable exists with the specified name, a private variable is created. To assign a field variable with the = operator, you must declare the field variable name in a FIELD statement or refer to the field name prefaced by the FIELD-> alias or the name of the work area.

Examples

- These examples are valid simple assignment statements:

```
nValue = 25
nNewValue = SQRT(nValue) ** 5
nOldValue = nValue
```

- In this example, the two lines are equivalent:

```
FIELD->CustAge = 20
REPLACE CustAge WITH 20
```

See Also

++, --, :=, = (compound assign), STORE*

= (compound assign) operator

Compound assignment—binary (Assignment)

Syntax

```
<idVar>  += <cString>     (concatenation)
<idVar>  += <nNumber>     (addition)
<idVar>  -= <cString>     (concatenation)
<idVar>  -= <nNumber>     (subtraction)
<idVar>  -= <dDate>       (subtraction)
<idVar>  *= <nNumber>     (multiplication)
<idVar>  /= <nNumber>     (division)
<idVar>  %= <nNumber>     (modulus)
<idVar>  ^= <nNumber>     (exponentiation)
```

Type

Character, date, memo, numeric

Operands

<idVar> is a variable identifier of any storage class, including a field variable. The variable must be initialized to a value before performing the operation.

If the reference to *<idVar>* is ambiguous (i.e., not declared at compile time and not explicitly qualified with an alias), *<idVar>* is assumed to be MEMVAR. You can assign field variables by declaring the field variable name in a FIELD statement or referring to the field name prefaced by the FIELD-> alias or the name of the work area.

<cString> is the character string used in the operation with *<idVar>*.

<nNumber> is the numeric expression used in the operation with *<idVar>*.

<dDate> is the date value used in the operation with *<idVar>*.

Description

In addition to the simple and inline assignment operators (= and :=), there are a series of compound assignment operators that perform an operation then assign a value. They have the form:

```
<idVar> <operator>= <exp>
```

Each compound assignment expression is equivalent to the assignment expression:

```
<idVar> := ( <idVar> <operator> <exp> )
```

For each data type that supports an operator, the compound assignment operator performs the same operation before performing the assignment operation. The following table details all of the compound operators:

Compound Operators

Operator	Defined as	Operations
a += b	a := (a + b)	concatenation, addition
a -= b	a := (a - b)	concatenation, subtraction
a *= b	a := (a * b)	multiplication
a /= b	a := (a / b)	division
a %= b	a := (a % b)	modulus
a = b	a := (a ^ b)	exponentiation

Note: The exponentiation operator (**) does not have a corresponding compound assignment operator. The exponentiation compound assignment operator is ^=.

Since the compound assignment operators are based on the inline assignment operator (:=), the operation returns the result of its operation as the value of the expression. This means you can use these operators within expressions just like the inline assignment operator (:=).

Examples

- These examples use the compound addition and concatenation assignment operator:

```
// Character (concatenation)
LOCAL cString := "Hello"
? cString += " there"        // Result: "Hello there"

// Date (addition)
LOCAL dDate := CTOD("12/12/90")
dDate += 12                  // Result: 12/24/90

// Numeric (addition)
LOCAL nValue := 10
? SQRT(nValue += 15)         // Result:  5.00
? nValue                     // Result: 25.00
```

- These examples use the compound subtraction and concatenation assignment operator:

```
// Character (concatenation)
LOCAL cString := "Hello   "
? cString -= "There"         // Result: HelloThere

// Date (subtraction)
LOCAL dDate := CTOD("12/24/90")
dDate -= 12                  // Result: 12/12/90

// Numeric (subtraction)
LOCAL nValue := 10
? newValue := (nValue -= 5) ** 2      // Result: 25
? nValue                              // Result:  5
```

- This example uses the compound multiplication assignment operator:

```
LOCAL nValue := 10
? newValue := (nValue *= 5) + 100     // Result: 150
? nValue                              // Result:  50
```

- This example uses the compound division assignment operator:

```
LOCAL nValue := 10
? newValue := (nValue /= 5) + 100     // Result: 102.00
? nValue                              // Result:   2.00
```

- This example uses the compound modulus assignment operator:

```
LOCAL nValue := 10
? newValue := (nValue %= 4) * 100     // Result: 200.00
? nValue                              // Result:   2.00
```

- This example uses the compound exponentiation assignment operator:

```
LOCAL nValue := 10
? newValue := (nValue ^= 3) + 50      // Result: 1050.00
? nValue                              // Result: 1000.00
```

See Also %, *, **, +, ++, -, --, /, :=

= (equality) operator

Equal—binary (Relational)

Syntax

```
<exp1> = <exp2>
```

Type

Character, date, logical, memo, NIL, numeric

Operands

<exp1> and ***<exp2>*** are expressions of the same data type to compare.

Description

The equal operator (=) compares two values of the same data type and returns true (.T.) if *<exp1>* is equal to *<exp2>* according to the following rules:

- ***Character:*** The comparison is based on the underlying ASCII code. ASCII codes for alphabetic characters are ascending (e.g., the code for "A" is 65 and the code for "Z" is 90).

  When EXACT is OFF, two character strings are compared according to the following rules. Assume two character strings, cLeft and cRight, where the expression to test is (cLeft = cRight):

  - If cRight is null, returns true (.T.).
  - If LEN(cRight) is greater than LEN(cLeft), returns false (.F.).
  - Compare all characters in cRight with cLeft. If all characters in cRight equal cLeft, returns true (.T.); otherwise, returns false (.F.).

  With EXACT ON, two strings must match exactly except for trailing blanks.

- ***Date:*** Dates are compared according to the underlying date value.

- ***Logical:*** True (.T.) is equal to true (.T.) and false (.F.) equal to false (.F.).

- ***Memo:*** Treated the same as character.
- ***NIL:*** True (.T.) if compared to a NIL value; false (.F.) if compared to a value of any other data type.
- ***Numeric:*** Compared based on magnitude.

Examples

- These examples illustrate how the equal operator (=) behaves with different data types:

```
// Character
SET EXACT ON
? "123" = "123  "          // Result: .T.
? " 123" = "123"           // Result: .F.
SET EXACT OFF
? "123" = "12345"          // Result: .F.
? "12345" = "123"          // Result: .T.
? "123" = ""               // Result: .T.
? "" = "123"               // Result: .F.

// Date
? CTOD("12/12/88") = ;
   CTOD("12/12/88")        // Result: .T.

// Logical
? .T. = .T.                // Result: .T.
? .F. = .T.                // Result: .F.

// NIL
? NIL = NIL                // Result: .T.
? NIL = 12                 // Result: .F.
? NIL = CTOD("")           // Result: .F.

// Numeric
? 2 = 1                    // Result: .F.
? 1 = 1                    // Result: .T.
```

See Also

$, <, <=, <>, ==, >, >=

== operator

Exactly equal—binary (Relational)

Syntax

```
<exp1> == <exp2>
```

Type

All

Operands

<exp1> and ***<exp2>*** are expressions of the same data type to be compared.

Description

The exactly equal operator (==) is a binary operator that compares two values of the same data type for exact equality depending on the data type. It returns true (.T.) if *<exp1>* is equal to *<exp2>* according to the following rules:

- ***Array:*** Compares for identity. If *<exp1>* and *<exp2>* are variable references to the same array, returns true (.T.); otherwise, returns false (.F.).
- ***Character:*** Comparison is based on the underlying ASCII code. ASCII codes for alphabetic characters are ascending (e.g., the code for "A" is 65 and the code for "Z" is 90). Unlike the relational equality operator (=) , true (.T.) is returned if *<exp1>* and *<exp2>* are exactly equal including trailing spaces; otherwise, the comparison returns false (.F.). SET EXACT has no effect.
- ***Date:*** Dates are compared according to the underlying date value; behaves the same as the relational equality operator (=).
- ***Logical:*** True (.T.) is exactly equal to true (.T.), and false (.F.) is exactly equal to false (.F.).
- ***Memo:*** Treated the same as character.
- ***NIL:*** True (.T.) if compared to a NIL value; false (.F.) if compared to a value of any other data type.
- ***Numeric:*** Compared based on magnitude; behaves the same as the relational equality operator (=).
- ***Object:*** Treated the same as array.

Examples

- These examples illustrate how the == operator behaves with different data types:

```
// Character
? "A" == "A"            // Result: .T.
? "Z" == "A"            // Result: .F.
? "A" == "A "           // Result: .F.
? "AA" == "A"           // Result: .F.

// Array or object
aOne := { 1, 2, 3 }
aTwo := { 1, 2, 3 }
aThree := aOne
? aOne == aTwo          // Result: .F.
// values within the arrays are equal, but the arrays,
// themselves, are separate and distinct
? aOne == aThree        // Result: .T.
```

See Also

$, <, <=, <>, = (equality), >, >=

> operator

Greater than—binary (Relational)

Syntax

```
<exp1> > <exp2>
```

Type

Character, date, logical, memo, numeric

Operands

<exp1> and ***<exp2>*** are expressions of any type to be compared.

Description

The greater than (>) operator compares two values of the same data type and returns true (.T.) if *<exp1>* is greater than *<exp2>*.

- ***Character:*** The comparison is based on the underlying ASCII code. ASCII codes for alphabetic characters are ascending (e.g., the code for "A" is 65 and the code for "Z" is 90).
- ***Date:*** Dates are compared according to the underlying date value.
- ***Logical:*** True (.T.) is greater than false (.F.).
- ***Memo:*** Treated the same as character.
- ***Numeric:*** Compared based on magnitude.

Examples

- These examples illustrate how the greater than operator (>) behaves with different data types:

```
// Character
? "Z" > "A"             // Result: .T.
? "AZ" > "A"            // Result: .F.
? "A" > "AZ"            // Result: .F.

// Date
? CTOD("12/12/88") > ;
   CTOD("12/11/88")     // Result: .T.

// Logical
? .T. > .F.             // Result: .T.

// Numeric
? 2 > 1                 // Result: .T.
? 1 > 2                 // Result: .F.
```

See Also $, <, <=, <>, = (equality), ==, >=

>= operator

Greater than or equal—binary (Relational)

Syntax

```
<exp1> >= <exp2>
```

Type

Character, date, logical, memo, numeric

Operands

<exp1> and ***<exp2>*** are expressions of any data type to be compared.

Description

The greater than or equal to operator (>=) is a binary operator that compares two values of the same data type and returns true (.T.) if *<exp1>* is greater than or equal to *<exp2>*.

- ***Character:*** The comparison is based on the underlying ASCII code. ASCII codes for alphabetic characters are ascending (e.g., the code for "A" is 65 and the code for "Z" is 90).
- ***Date:*** Dates are compared according to the underlying date value.
- ***Logical:*** True (.T.) is greater than false (.F.).
- ***Memo:*** Treated the same as character.
- ***Numeric:*** Compared based on magnitude.

Examples

- These examples illustrate how the greater than or equal operator (>=) behaves with different data types:

```
// Character
? "Z" >= "A"                // Result: .T.
? "AZ" >= "A"               // Result: .T.
? "A" >= "AZ"               // Result: .F.

// Date
? CTOD("12/12/88") >= ;
   CTOD("12/11/88")         // Result: .T.

// Logical
? .T. >= .F.                // Result: .T.

// Numeric
? 2 >= 1                    // Result: .T.
? 1 >= 2                    // Result: .F.
? 2 >= 2                    // Result: .T.
```

See Also

$, <, <=, <>, = (equality), ==, >

? | ?? command

Display one or more values to the console

Syntax

```
? | ?? [<exp list>]
```

Arguments

<exp list> is a list of values to display and can be any combination of data types, including memo.

If you specify no argument and use the ? command, a carriage return/linefeed is sent to the console. If you use the ?? command without arguments, nothing happens.

Description

? and ?? are console commands that display the results of one or more expressions, separated by a space, to the console. These commands are also command synonyms for the QOUT() and QQOUT() functions, respectively.

Although functionally similar, ? and ?? differ slightly. ? sends a carriage return/linefeed to the console before displaying the results of the expression list. ?? displays output at the current screen cursor or printhead position. This lets you use successive ?? commands to display output to the same line.

A ? or ?? command locates the cursor or printhead one position to the right of the last character displayed. If SET PRINTER is OFF, ROW() and COL() are updated to reflect the new cursor position. If SET PRINTER is ON, PROW() and PCOL() are updated with the new printhead position.

If output from a ? or ?? command reaches the edge of the screen as reported by MAXCOL(), it wraps to the next line. If the output reaches the bottom of the screen as reported by MAXROW(), the screen scrolls up one line.

You can echo output from the ? or ?? commands to the printer by specifying a SET PRINTER ON command before beginning output. You can echo output from both of these commands to a text file using SET ALTERNATE TO <*xcFile*> to create the file, and SET ALTERNATE ON to begin echoing to the file. Like other console commands, SET CONSOLE OFF suppresses the display to the screen without affecting output to the printer or text file.

To format any expression specified, use TRANSFORM() or a user-defined function. If you need to pad a variable length value for column alignment, use any of the PAD() functions to left-justify, right-justify, or center the value. See the examples below.

Examples

- This example prints a record from a database file using ? and ?? commands with PADR() to ensure column alignment:

```
LOCAL nPage := 0, nLine := 99
USE Salesman INDEX Salesman NEW
SET PRINTER ON
SET CONSOLE OFF
DO WHILE !EOF()
    IF nLine > 55
       IF nPage != 0
          EJECT
       ENDIF
       ? PADR("Page", LTRIM(STR(nPage++)), 72)
       ?? DTOC(DATE())
       ?
       ?
       ? PADC("Sales Listing", 79)
       ?
       nLine := 5
    ENDIF
    ? Name, Address, PADR(RTRIM(City) + "," ;
          + State, 20), ZipCode
    nLine++
    SKIP
ENDDO
SET CONSOLE ON
SET PRINTER OFF
CLOSE Salesman
```

Files

Library is CLIPPER.LIB.

See Also

@...SAY, PAD(), QOUT(), SET ALTERNATE, SET CONSOLE, SET PRINTER, TRANSFORM()

@ operator

Pass-by-reference—unary (Special)

Syntax

```
@<idVar>
```

Operands

<idVar> is any valid CA-Clipper variable identifier, other than a database field or an array element, to pass by reference. Database fields and array elements can only be passed by value.

Description

The pass-by-reference operator (@) passes variables by reference to functions or procedures invoked with function-calling syntax. It is a unary prefix operator whose operand may be any variable name.

Passing a variable by reference means that a reference to the value of the argument is passed instead of a copy of the value. The receiving parameter then refers to the same location in memory as the argument. If the called routine changes the value of the receiving parameter, it also changes the argument passed from the calling routine.

Passing a variable by value means that the argument is evaluated and its value is copied to the receiving parameter. Changes to a receiving parameter are local to the called routine and lost when the routine terminates. The default method of passing arguments is by value for all data types including references to arrays and objects.

Examples

- This example demonstrates the difference between passing a user-defined function argument by reference and by value:

```
LOCAL nNum := 1                       // Initial values
LOCAL aArr := { "a", "b", "c" }
//
CLS
// Print initial values
? VALTYPE(nNum), nNum
? VALTYPE(aArr), aArr[1], aArr[2], aArr[3]
//
Udf1(nNum)                            // Pass by value (default)
? VALTYPE(nNum), nNum                 // Result:  N, 1
//
Udf1(aArr[1])                         // Pass by value (default)
? VALTYPE(aArr), aArr[1],;
      aArr[2], aArr[3]                // Result:  A, "a" "b" "c"
//
Udf2(aArr)                            // Pass a reference to
? VALTYPE(aArr), aArr[1],;            // the array (default)
      aArr[2], aArr[3]                // A, "Oi!" "b" "c"
//
Udf1(@nNum)                           // Force pass by reference
? VALTYPE(nNum), nNum                 // Result:  N, 1000
//
Udf3(@aArr)                           // Pass array by reference
? VALTYPE(aArr)                       // Result:  N
//
RETURN NIL


FUNCTION Udf1(nParam)                 // Receive as a local
? nParam                              // parameter
nParam := 1000
? nParam
RETURN NIL
//
FUNCTION Udf2( aParam )               // Receive as a local
? VALTYPE(aParam), aParam[1]          // parameter
aParam[1] := "Oi!"
? aParam[1]
RETURN NIL
//
FUNCTION Udf3(aParam)                 // Receive as a local
? VALTYPE(aParam), aParam[1]          // parameter
aParam := 1000
? aParam
RETURN NIL
```

See Also FUNCTION, PROCEDURE

@...BOX command

Draw a box on the screen

Syntax

```
@ <nTop>, <nLeft>, <nBottom>, <nRight>
   BOX <cBoxString> [COLOR <cColorString>]
```

Arguments

<nTop>, <nLeft>, <nBottom>, <nRight> define the coordinates of the box. @...BOX draws a box using row values from zero to MAXROW(), and column values from zero to MAXCOL(). If *<nBottom>* and *<nRight>* are larger than MAXROW() and MAXCOL(), the bottom-right corner is drawn off the screen.

BOX <cBoxString> defines a string of eight border characters and a fill character. If *<cBoxString>* is specified as a single character, that character draws the whole box.

COLOR <cColorString> defines the display color of the drawn box. If not specified, the box is drawn using the standard color setting of the current system color as defined by SETCOLOR(). Note that *<cColorString>* is a character expression containing the standard color setting. If you want to specify a literal color setting, enclose it within quote marks.

Description

@...BOX draws a box on the screen using configurable border and fill characters. @...BOX draws the box using *<cBoxString>* starting from the upper left-hand corner, proceeding clockwise and filling the screen region with the ninth character. If the ninth character is not specified, the screen region within the box is not painted. Existing text and color remain unchanged.

After @...BOX executes, the cursor is located in the upper corner of the boxed region at *<nTop>* + 1 and *<nLeft>* + 1. ROW() and COL() are also updated to reflect the new cursor position.

Examples

- These examples draw two boxes using box manifest constants defined in the supplied header file, Box.ch. The first example draws a box using the specified characters for the border, but leaves all other areas of the screen intact. The second example draws the same box filling the box region with space characters.

```
#include "Box.ch"
// Draw a box with a double-line top with a
// single-line side
@ 1, 1, 22, 79 BOX B_DOUBLE_SINGLE
// Draw the same box filling the box region with
// spaces
@ 1, 1, 22, 79 BOX B_DOUBLE_SINGLE + SPACE(1)
```

Files Library is CLIPPER.LIB, header file is Box.ch.

See Also @...CLEAR, @...TO, DISPBOX(), SCROLL()

@...CLEAR command

Clear a rectangular region of the screen

Syntax

```
@ <nTop>, <nLeft> [CLEAR
   [TO <nBottom>, <nRight>]]
   [DOUBLE] [COLOR <cColor>]
```

Arguments

<nTop> and ***<nLeft>*** define the upper-left corner coordinate.

TO <nBottom>, <nRight> defines the lower-right corner coordinates of the screen region to CLEAR. If the TO clause is not specified, these coordinates default to MAXROW() and MAXCOL().

Description

@...CLEAR erases a rectangular region of the screen by filling the specified region with space characters using the current standard color setting. After @...CLEAR erases the designated region, the cursor is located in the upper corner of the region at *<nTop>* + 1 and *<nLeft>* + 1. ROW() and COL() are also updated to reflect the new cursor position.

Examples

- This example erases the screen from 10, 10 to 20, 40, painting the region blue and then displaying a bright cyan box on blue:

```
SETCOLOR("BG+/B")
@ 10, 10 CLEAR TO 20, 40
@ 10, 10 TO 20, 40
```

Files

Library is CLIPPER.LIB.

See Also

@...BOX, CLEAR SCREEN, SCROLL(), SETCOLOR()

@...GET command

Create a new Get object and display it to the screen

Syntax

```
@ <nRow>, <nCol>
   [SAY <exp>
      [PICTURE <cSayPicture>]
      [COLOR <cColorString>]]
   GET <idVar>
      [PICTURE <cGetPicture>]
      [COLOR <cColorString>]
      [CAPTION<cCaption>]
      [MESSAGE <cMessage>]
      [WHEN <lPreExpression>]
      [RANGE* <dnLower>, <dnUpper>] |
      [VALID <lPostExpression>]
      [SEND <msg>]
      [GUISEND <guimsg>]
```

Arguments

<nRow> and ***<nCol>*** specify the row and column coordinates for the operation. If the SAY clause is present, *<nRow>* and *<nCol>* specify the coordinates for the SAY, and the GET is displayed to the right of the SAY output. If the SAY clause is not present, *<nRow>* and *<nCol>* directly specify the coordinates for the GET. In either case, output which extends beyond the visible extent of the display is clipped and does not appear.

SAY <exp> displays the value of *<exp>* at the specified coordinates. If you specify the PICTURE *<cSayPicture>* clause, *<exp>* is formatted according to the rules of SAY pictures.

GET <idVar> specifies the name of the variable associated with the GET. *<idVar>* may be of any storage class (if the storage class is ambiguous, FIELD is assumed). If *<idVar>* contains an array value, you must supply one or more subscripts. The current value of *<idVar>* is displayed at the GET coordinates. The value must be character, date, numeric, or logical type. Array, NIL, code block, and null string values are not permitted.

PICTURE <cGetPicture> specifies the display format and editing rules for the GET.

COLOR <cColorString> defines the color settings for the current Get object. *<cColorString>* is a character expression containing one or more color settings. You must specify literal color settings enclosed in quote marks.

If you specify two color settings, the first determines the unselected color (the display color for the Get object) and the second determines the selected color (the color when the Get object has focus). If you only specify one color, it determines both the unselected and selected colors for the Get object.

On a combined @...SAY...GET command, two COLOR clauses are required to specify colors for both the SAY and the GET: one for the SAY, and one for the GET.

CAPTION <cCaption> specifies a character string that concisely describes the GET on the screen. CAPTION differs from the SAY argument in two ways. The first is that the caption is displayed each time the GET is shown on the screen. The other difference is that the caption, along with its screen position, is maintained within the Get object. This allows the GET to receive input when the mouse's left button is clicked and its cursor is on the caption. By default, the caption appears to the right of the GET. Use the Get object's capRow or capCol variable to change the screen position of the caption. Note that if the SAY clause is used, the CAPTION clause is ignored.

When present, the **&** character specifies that the character immediately following it in the caption is the GET's accelerator key. The accelerator key provides a quick and convenient mechanism for the user to move input focus from one data input control to the GET. The user performs the selection by pressing the Alt key in combination with an accelerator key. The case of an accelerator key is ignored.

MESSAGE <cMessage> specifies a character string that is displayed on the Get system's status bar line when the GET has input focus. Typically, it describes the anticipated contents or user response of the GET. Refer to the SET MESSAGE command for details pertaining to the Get system's status bar.

WHEN <lPreExpression> specifies an expression that must be satisfied before the cursor can enter the GET during a READ. The expression can optionally be a code block. The Get object is passed as a parameter to the code block.

RANGE* <dnLower>, <dnUpper> specifies a range of allowable values for input to the GET. During a READ command, if you enter a new GET value that does not fall within the specified range, the cursor cannot leave the GET.

VALID <lPostExpression> specifies an expression that must be satisfied before the cursor can leave the GET during a READ. The expression can optionally be a code block. The Get object is passed as a parameter to the code block.

SEND <msg> sends the specified message to the Get object. *<msg>* is sent before the GET is displayed. Any message can be sent, including method calls and instance variable assignments. Method calls must include parentheses even if no arguments are passed.

GUISEND <guimsg> can be used to send a message (such as the Display() method) to a GUI object. The GUI objects available in CA-Clipper are check boxes, list boxes, push buttons, and radio button groups. To send a message to a non-GUI object, such as standard GETs and TBrowses, use the SEND clause.

Description

The @...GET command creates a new Get object, displays its value, and adds it to the array referred to by the variable *GetList*. If no variable called *GetList* has been declared or created in the current procedure, and no PRIVATE variable called *GetList* exists from a previous procedure, the system uses the predefined PUBLIC variable *GetList*. A subsequent READ command activates the GETs contained in the *GetList* array and allows the user to edit their contents.

Each Get object has an associated variable, <*idVar*>. The variable may be of any storage class, including a database field, private, public, local, or static variable. If <*idVar*> is followed by one or more subscripts, the specified array element is associated with the GET. When the Get object is created, the <*idVar*> name is stored in the Get object, along with a code block which allows the value of <*idVar*> to be retrieved or assigned during the READ.

The READ command performs a full-screen edit of the GETs in the *GetList* array. As the user moves the cursor into each GET, the value of the associated <*idVar*> is retrieved by evaluating the code block saved in the Get object. The value is converted to textual form and placed in a buffer within the Get object. This buffer is displayed on the screen, and the user is allowed to edit the text from the keyboard. When the user moves the cursor out of the GET, the updated buffer is converted back to the appropriate data type and assigned to <*idVar*>.

Automatic formatting and validation: During a READ, some formatting and edit validation is automatically performed for numeric, date, and logical values. As the user is typing, an automatic data type test is performed on each key pressed, preventing the user from entering an invalid character.

Prevalidation: The WHEN clause specifies a condition which must be satisfied before the cursor can enter the GET. During a READ, <*lPreExpression*> is evaluated whenever the user attempts to move the cursor into the GET. If it evaluates to true (.T.), the cursor can enter; otherwise, the GET is skipped.

Postvalidation: You may perform postvalidation using either the VALID or RANGE* clauses. VALID specifies a condition which must be satisfied before the cursor can leave the GET. During a READ, <*lPostExpression*> is evaluated whenever the user attempts to move the cursor out of the GET. If it evaluates to true (.T.), the cursor can leave; otherwise, the cursor remains in the GET. RANGE* specifies a range of acceptable values for numeric or date values. If the value entered by the user is not within the specified range, the cursor cannot leave the GET.

Note: You may specify either a VALID or RANGE clause, but not both.

PICTURE: When you specify the PICTURE clause for a GET, the character string specified by <*cGetPicture*> controls formatting and edit validation. The picture string controls the display format like a SAY picture. It also controls the way the user can edit the buffer. A picture string consists of two distinct parts, a function string and a template string, either or both of which may be present.

- ***Function string:*** A PICTURE function string specifies formatting or validation rules which apply to the GET's display value as a whole, rather than to particular character positions within it. The function string consists of the @ character, followed by one or more additional characters, each of which has a particular meaning (see the following table). The function string must be the first element of a PICTURE clause and cannot contain spaces. A function string may be specified alone or with a template string. If both are present, the function string must precede the template string, and the two must be separated by a single space.

GET PICTURE Format Functions

Function	Type	Action
A	C	Allows only alphabetic characters.
B	N	Displays numbers left-justified.
C	N	Displays CR after positive numbers.
D	D,N	Displays dates in SET DATE format.
E	D,N	Displays dates with day and month inverted independent of the current DATE SETting, numerics with comma and period reverse (European style).
K	ALL	Deletes default text if first key is not a cursor key.
R	C	Nontemplate characters are inserted in the display but not saved in the variable.
S<n>	C	Allows horizontal scrolling within a GET. <n> is an integer that specifies the width of the region.
X	N	Displays DB after negative numbers.
Z	N	Displays zero as blanks.
(	N	Displays negative numbers in parentheses with leading spaces.
)	N	Displays negative numbers in parentheses without leading spaces.
!	C	Converts alphabetic character to uppercase.

- ***Template string:*** A PICTURE template string specifies formatting or validation rules on a character by character basis. The template string consists of a series of characters, some of which have special meanings (see the following table). Each position in the template string corresponds to a position in the displayed GET value. Characters in the template string that do not have assigned meanings are copied verbatim into the displayed GET value. If you use the @R picture function, these characters are inserted between characters of the display value, and are automatically removed when the display value is reassigned to <*idVar*>; otherwise, they overwrite the corresponding characters of the display value and also affect the value assigned to <*idVar*>. You may specify a template string alone or with a function string. If you use both, the function string must precede the template string, and the two must be separated by a single space.

GET PICTURE Template Symbols

Template	Action
A	Allows only alphabetic characters
N	Allows only alphabetic and numeric characters
X	Allows any character
9	Allows digits for any data type including sign for numerics
#	Allows digits, signs and spaces for any data type
L	Allows only T, F, Y or N
Y	Allows only Y or N
!	Converts an alphabetic character to uppercase
$	Displays a dollar sign in place of a leading space in a numeric
*	Displays an asterisk in place of a leading space in a numeric
.	Displays a decimal point
,	Displays a comma

SCOREBOARD: If a new value is rejected because it fails the RANGE* test or because it is a malformed date value, a message appears on the screen. The message displays in the SCOREBOARD area, which you can enable or disable using the SET SCOREBOARD command.

*Exit with **Esc**:* If the user exits a GET by pressing Esc, the GET variable is restored to the value it had on entry to the GET, and the READ is terminated. No postvalidation is performed. You can enable or suppress this behavior with the SET ESCAPE command.

SET KEY procedures: The SET KEY command lets you specify a procedure to be executed whenever a specific key is pressed during a READ. After a SET KEY procedure terminates, the GET is reactivated with the cursor restored to its previous position.

Lifetime of a Get object: Get objects, like arrays, exist as long as there are active references to them somewhere in the current program. Normally, only the array in the *GetList* variable maintains a reference to the Get object; the GET is released when *GetList* is released or reassigned. The CLEAR and CLEAR GETS commands assign an empty array to *GetList*, as does the READ command unless you specify the SAVE clause.

Assignment: Each GET is associated with a variable, *<idVar>*, in the @...GET command. At various times during the editing process, *<idVar>* may be assigned the current value of the Get object's buffer. This occurs in the following instances:

- After the user presses an exit key and before the validation expression is executed
- After the user presses a SET KEY

Also, the current Get object's buffer is refreshed from *<idVar>* and redisplayed at various intervals:

- After a SET KEY procedure terminates
- After a WHEN expression is evaluated
- After a VALID expression is evaluated

This lets you explicitly assign *<idVar>* within any of these operations. See the following note for more information.

GET coordinates and display: When you create a Get object using the @...GET command, the row and column coordinates at which the GET is initially displayed are stored in the Get object. When the @...GET command executes, the new GET displays at <*nRow*> and <*nCol*>, unless you specify the SAY clause which positions the GET so there is one display column between the last character of the SAY output and the first character of the GET (or of the DELIMITERS, see below).

If SET DELIMITERS is ON when the @...GET command executes, the current DELIMITER characters display on either side of the initial GET display, and the column coordinate of the GET is adjusted accordingly. Note that the delimiters are not attributes of the Get object, but simply display as the SAY clause does.

If INTENSITY is ON, GETs initially display in the current unselected color (or the enhanced color, if no unselected color has been specified). During a READ, the active GET displays in the enhanced color, while the remaining GETs display in the unselected color. With INTENSITY OFF, all GETs display in the standard color.

When a GET displays, the width of the displayed value is determined by the length of the value in <*idVar*> or, if you specify the PICTURE clause, by the number of positions in <*cGetPicture*>. If you specify the @S function as a part of <*cGetPicture*>, the @S argument controls the width of the displayed value.

Notes

- ***WHEN and VALID:*** The expressions specified in the WHEN and VALID clauses may be of arbitrary complexity and may include calls to user-defined functions. This is useful for attaching automatic actions to the activation or deactivation of a GET.

- ***Assigning <idVar>:*** Because of the automatic refresh and display properties of a Get object while it is being READ, you can make an explicit assignment to the Get object's *<idVar>* within a WHEN or VALID expression. You can directly assign the variable by name in the validation expression or, for private, public, local, or static variables, by passing a reference to *<idVar>* to a function; the function can then assign *<idVar>* by assigning the corresponding formal parameter. If *<idVar>* is a field, it is globally visible and can be assigned by name in a function called by the validation expression.

  When including a GET in a called function, do not include an *<idVar>* with the same name as a field *<idVar>*. Field references have precedence over public variables so the public *<idVar>* will be ignored.

- ***GET specific help:*** You can use a SET KEY procedure to display help text associated with a Get object. Within the SET KEY procedure, use the READVAR() function to determine the *<idVar>* associated with the current Get object. Use this information to display the appropriate help text. Remember that when a CA-Clipper-compiled program loads, the *F1* KEY is automatically SET TO a procedure or user-defined function named Help.

- ***SET DEVICE TO PRINTER:*** SET DEVICE TO PRINTER does not direct display of a Get object under the @...GET command to the printer or file.

Examples

- This example uses the VALID clause to validate input into a GET:

```
LOCAL nNumber := 0
@ 10, 10 SAY "Enter a number:" ;
    GET nNumber VALID nNumber > 0
```

- This example demonstrates passing a code block with the VALID clause. The parameter *oGet* is the current Get object. Udf() changes the value of the GET:

```
LOCAL GetList := {}, cVar := SPACE(10)
CLS
@ 10, 10 GET cVar  VALID { |oGet| Udf1( oGet ) }
READ
.
.
.
* Udf( <oGet> ) --> .T.

FUNCTION Udf1( oGet )

IF "test" $ oGet:BUFFER          // Compare buffer contents
oGet:varPut( "new value " )      // Change contents
ENDIF

RETURN .T.
```

- This example uses the WHEN clause to prohibit entry into GETs based on the value of another GET. In this example, entering Y in the Insured field indicates the client has insurance and the user is allowed to enter insurance information. If the client does not have insurance, the cursor moves to the Accident field:

```
@ 10, 10 GET Insured PICTURE "Y"
@ 11, 10 GET InsNumber WHEN Insured
@ 12, 10 GET InsCompany WHEN Insured
@ 13, 10 GET Accident PICTURE "Y"
READ
```

- This is an example of a GET in a secondary work area:

```
USE Invoice NEW
APPEND BLANK
USE Inventory NEW
@ 1, 1 GET Invoice->CustNo
READ
```

- This example uses the @K function to suggest a default input value, but deletes it if the first key pressed is not a cursor key or *Return*:

```
LOCAL cFile := "Accounts"
@ 1, 1 SAY "Enter file" GET cFile PICTURE "@K"
READ
```

- This is an example of a nested READ using a *GetList* and lexical scoping:

```
#include "Inkey.ch"
//
// Local to this function only
LOCAL GetList := {}
LOCAL cName   := SPACE( 10 )
//
CLS
SETKEY( K_F2, { || MiscInfo() } )   // Hot key to special READ
//
// Get object added to getlist
// works on local getlist
@ 10, 10 SAY "Name" GET cName
READ
//
RETURN NIL

/***
*  MiscInfo() ---> NIL
*/FUNCTION MiscInfo()
//
LOCAL GetList    := {}              // Local to this
LOCAL cExtraInfo := SPACE( 30 )     // function only
//
// Get object added to getlist
// works on local getlist
@ 12, 10 SAY "Note: " GET cExtraInfo
READ
//
RETURN NIL
```

Files Library is CLIPPER.LIB.

See Also ? | ??, @...SAY, CLEAR, COL(), PCOL(), PROW(), QOUT(). READ, ROW(), SET BELL, SET CONFIRM, SET DELIMITERS, SET DEVICE, SET INTENSITY, SETCOLOR(), TRANSFORM()

@...GET CHECKBOX command

Create a new check box Get object and display it to the screen

Syntax

```
@ <nRow>, <nCol>
   GET <lVar>
   CHECKBOX
      [CAPTION<cCaption>]
      [MESSAGE <cMessage>]
      [WHEN <lPreExpression>]
      [VALID <lPostExpression>]
      [COLOR <cColorString>]
      [FOCUS <fblock>
      [STATE <bBlock>]
      [STYLE <cStyle>]
      [SEND <msg>]
      [GUISEND <guimsg>]
      [BITMAPS <aBitmaps>]
```


Arguments

<nRow> and ***<nCol>*** specify the row and column coordinates for the check box and its caption. Output which extends beyond the visible extent of the display is clipped and does not appear.

GET <lVar> specifies the name of the variable associated with the GET. <l*Var*> must contain a logical value. A value of true (.T.) indicates that the check box is checked; otherwise, a value of false (.F.) indicates that it is not checked.

CAPTION <cCaption> specifies a character string that concisely describes the GET on the screen. Caption differs from the SAY argument in two ways. The first is that the caption is displayed each time the GET is shown on the screen. The other difference is that the caption, along with its screen position, is maintained within the Get object. This allows the GET to receive input when the mouse's left button is clicked and its cursor is on the caption. By default, the caption appears to the left of the GET. Use the Get object's capRow or capCol variable to change the screen position of the caption.

When present, the **&** character specifies that the character immediately following it in the caption is the check box's accelerator key. The accelerator key provides a quick and convenient mechanism for the user to move input focus from one data input control to the check box. The user performs the selection by pressing the Alt key in combination with an accelerator key. The case of an accelerator key is ignored.

MESSAGE <cMessage> specifies a character string that is displayed on the Get system's status bar line when the GET has input focus. Typically, it describes the anticipated contents or user response of the GET. Refer to the READ command for details pertaining to the Get system's status bar.

WHEN <lPreExpression> specifies an expression that must be satisfied before the cursor can enter the GET during a READ. The expression can optionally be a code block. The Get object is passed as a parameter to the code block.

VALID <lPostExpression> specifies an expression that must be satisfied before the cursor can leave the GET during a READ. The expression can optionally be a code block. The Get object is passed as a parameter to the code block.

COLOR <cColorString> defines the color settings for the check box. <*cColorString*> is a character expression containing exactly four color settings.

CheckBox Color Attributes

Position in colorSpec	Applies To	Default Value from System Color Setting
1	The check box when it does not have input focus	Unselected
2	The check box when it has input focus	Enhanced
3	The check box's caption	Standard
4	The check box caption's accelerator key	Background

Note: In graphic mode, colorSpec positions 1 and 2 have no affect and are ignored.

FOCUS <fblock> specifies a code block that is evaluated each time the check box receives focus. The code block takes no implicit arguments. Use the CheckBox:hasFocus instance variable to determine if the check box has focus.

STATE <bBlock> specifies a code block that is evaluated each time the check box's state changes. The code block takes no implicit arguments. Use the CheckBox:buffer instance variable to determine if the check box is being checked or unchecked. A value of true (.T.) indicates that it is being checked; otherwise, a value of false (.F.) indicates that it is being unchecked.

STYLE <cStyle> specifies a character string that indicates the check box's delimiter characters. The string must contain four characters. The first is the left delimiter. Its default value is the left square bracket ([) character. The second is the checked indicator. Its default value is the square root (√) character. The third is the unchecked indicator. Its default is the space character (" "). The fourth character is the right delimiter. Its default value is the right square bracket (]) character.

Note: The STYLE clause is ignored in graphic mode.

SEND <msg> sends the specified message to the Get object. *<msg>* is sent before the GET is displayed. Any message can be sent, including method calls and instance variable assignments. Method calls must include parentheses even if no arguments are passed.

GUISEND <guimsg> can be used to send a message (such as the Display() method) to a GUI object. The GUI objects available in CA-Clipper are check boxes, list boxes, push buttons, and radio button groups. To send a message to a non-GUI object, such as standard GETs and TBrowses, use the SEND clause.

BITMAPS <aBitmaps> contains an array of exactly two elements. The first element of this array is the file name of the bitmap to be displayed when the check box is selected. The second element of this array is the file name of the bitmap to be displayed when the check box is not selected.

Drive and directory names are not allowed; the file name extension is required. A bitmap file can be stored as a file on disk or in a bitmap library. If stored as a file, the file must reside in the same directory as the application. If stored in a bitmap library, the library must reside in the same directory as the application and it also must have the same name as the application with a .BML extension.

CA-Clipper will search for the file name first and, if it is not found, search in the bitmap library second. If no file is found either on disk or in the library, no bitmap will be displayed.

This argument only affects applications running in graphic mode and is ignored in text mode.

Examples

- This example demonstrates how to add a check box to a data entry screen:

```
@ 5, 20 GET lReadOnly CHECKBOX
```

- This example demonstrates a check box that has a caption and a message:

```
@ 5, 20 GET lReadOnly CHECKBOX ;
        CAPTION "&Read Only" ;
        MESSAGE "Check to open file for reading only"
```

- This example demonstrates a check box that uses the **X** character as the checked character instead of the square root (√) character.

```
@ 5, 20 GET lReadOnly CHECKBOX ;
        CAPTION "&Read Only" ;
        MESSAGE "Check to open file for reading only" ;
        STYLE "[X ]"
```

Files

Libraries are CLIPPER.LIB and LLIBG.LIB.

See Also

@...GET, Get class, CheckBox class, READ

@...GET LISTBOX command

Create a new list box Get object and display it to the screen

Syntax

```
@ <nTop>, <nLeft>, <nBottom>, <nRight>
   GET <nVar|cVar>
   LISTBOX <aList>
      [CAPTION<cCaption>]
      [MESSAGE <cMessage>]
      [WHEN <lPreExpression>]
      [VALID <lPostExpression>]
      [COLOR <cColorString>]
      [FOCUS <fblock>]
      [STATE <bBlock>]
      [DROPDOWN]
      [SCROLLBAR]
      [SEND <msg>]
      [GUISEND <guimsg>]
      [BITMAP <cBitmap>]
```

Arguments

<nTop>, <nLeft>, nBottom> and ***<nRight>*** specify the screen position for the list box and its caption. Output which extends beyond the visible extent of the display is clipped and does not appear.

GET <nVar | cVar> specifies the name of the variable associated with the GET. Its value indicates which item (if any) in the list is selected. A numeric value refers to the position in the list of the selected item. A value of 0 indicates no selected item. A character string value refers to the data or text contained within the selected item. A character string that does not refer to any item in the list indicates no selected item.

LISTBOX <aList> specifies an array that contains the items in the list. The array may be either a single- or two-dimensional array. Use a single-dimension array when the data being displayed is the same as the data being saved; otherwise, use a two-dimensional array. In this case, the data in the first element of each subarray is displayed and the data in the second element is used for determining which item is selected.

CAPTION <cCaption> specifies a character string that concisely describes the GET on the screen. Caption differs from the SAY argument in two ways. The first is that the caption is displayed each time the GET is shown on the screen. The other difference is that the caption, along with its screen position, is maintained within the Get object. This allows the GET to receive input when the mouse's left button is clicked and its cursor is on the caption. By default, the caption appears to the left of the GET. Use the Get object's capRow or capCol variable to change the screen position of the caption.

When present, the & character specifies that the character immediately following it in the caption is the list box's accelerator key. The accelerator key provides a quick and convenient mechanism for the user to move input focus from one data input control to the list box. The user performs the selection by pressing the Alt key in combination with an accelerator key. The case of an accelerator key is ignored.

MESSAGE <cMessage> specifies a character string that is displayed on the Get system's status bar line when the GET has input focus. Typically, it describes the anticipated contents or user response of the GET. Refer to the READ command for details pertaining to the Get system's status bar.

WHEN <lPreExpression> specifies an expression that must be satisfied before the cursor can enter the GET during a READ. The expression can optionally be a code block. The Get object is passed as a parameter to the code block.

VALID <lPostExpression> specifies an expression that must be satisfied before the cursor can leave the GET during a READ. The expression can optionally be a code block. The Get object is passed as a parameter to the code block.

COLOR <cColorString> defines the color settings for the list box. *<cColorString>* is a character expression containing exactly seven color settings if there is no drop-down list box, and eight if there is a drop-down list box.

ListBox Color Attributes

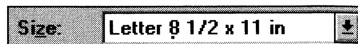
Position in colorSpec	Applies To	Default Value from System Color Setting
1	List box items that are not selected when the list does not have input focus	Unselected
2	The selected list box item when the list does not have input focus	Unselected
3	List box items that are not selected when the list has input focus	Unselected
4	The selected list box item when the list has input focus	Enhanced
5	The list box's border	Border
6	The list box's caption	Standard
7	The list box caption's accelerator key	Background
8	The list box's drop-down button	Standard

Note: In graphic mode, colorSpec position 8 has no affect and is ignored.

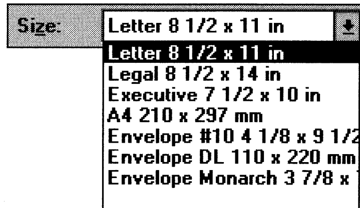
FOCUS <fblock> specifies a code block that is evaluated each time the list box receives focus. The code block takes no implicit arguments. Use the PushButton:hasFocus instance variable to determine if the push button has focus.

STATE <bBlock> specifies a code block that is evaluated immediately after the list box object's selection changes. The code block takes no implicit arguments. Use the ListBox:buffer instance variable to determine the current selection.

DROPDOWN indicates that the list box will be a drop-down list box. Drop-down list boxes are displayed differently on the screen than regular list boxes. A regular list box is always displayed regardless of whether it has input focus. A drop-down list box's display behavior is determined by whether it is closed or open. By default, a drop-down list box is closed. In this case, only the caption, the selected item, and the open button are displayed:



When open, the actual list box is displayed in addition to the caption, selected item, and open button:



Drop-down list boxes are useful in situations where screen space is limited.

SCROLLBAR indicates that the list box will be created with a scroll bar whether or not it is needed. Normally a scroll bar is not needed if the data being displayed fits inside the list box.

SEND <msg> sends the specified message to the Get object. *<msg>* is sent before the GET is displayed. Any message can be sent, including method calls and instance variable assignments. Method calls must include parentheses even if no arguments are passed.

GUISEND <guimsg> can be used to send a message (such as the display() method) to a GUI object. The GUI objects available in CA-Clipper are check boxes, list boxes, push buttons, and radio button groups. To send a message to a non-GUI object, such as standard GETs and TBrowses, use the SEND clause.

BITMAP <cBitmap> defines the bitmap that will be used as the down arrow on a drop-down list box. Drive and directory names are not allowed; the file name extension is required. A bitmap file can be stored as a file on disk or in a bitmap library. If stored as a file, the file must reside in the same directory as the application. If stored in a bitmap library, the library must reside in the same directory as the application and it also must have the same name as the application with a .BML extension.

CA-Clipper will search for the file name first and, if it is not found, search in the bitmap library second. If no file is found either on disk or in the library, no bitmap will be displayed.

Examples

- This example demonstrates how to add a list box to a data entry screen. It utilizes the position-oriented method of selection:

```
nColor := 1 //default to the first item.
@ 5, 20, 8, 28 GET nColor LISTBOX { "Red", Green", "Blue" }
```

- This example demonstrates a list box that has a caption and a message. It utilizes the data-oriented method of selection:

```
cColor := "Red" //default to red.
@ 5, 20, 8, 28 GET cColor LISTBOX { "Red", Green", "Blue" };
        CAPTION "&Color" ;
        MESSAGE "Select the background color"
```

- This example demonstrates a list box that utilizes a two-dimensional array:

```
cState := "NY"

@ 5, 20, 10, 30 GET cState LISTBOX { { "Alabama", "AL" }, ;
                            { "Alaska", AK" }    , ;
                            { "Arizona", "AZ" }, ;
                            { etc...          } } ;
        CAPTION "&State" ;
        MESSAGE "Select the client's state"
```

Files Library is CLIPPER.LIB.

See Also @...GET, Get class, ListBox class, READ

@...GET PUSHBUTTON command

Create a new push button Get object and display it to the screen

Syntax

```
@ <nRow>, <nCol>
   GET <lVar>
   PUSHBUTTON
      [CAPTION<cCaption>]
      [MESSAGE <cMessage>]
      [WHEN <lPreExpression>]
      [VALID <lPostExpression>]
      [COLOR <cColorString>]
      [FOCUS <fblock>
      [STATE <bBlock>]
      [STYLE <cStyle>]
      [SEND <msg>]
      [GUISEND <guimsg>]
      [SIZE X <nSizeX> Y <nSizeY>]
      [CAPOFF X <nCapXOff> Y <nCapYOff>]
      [BITMAP <cBitmap>]
      [BMPOFF X <nBmpXOff> Y <nBmpYOff>]
```

Arguments

<nRow> and ***<nCol>*** specify the row and column coordinates for the push button and its caption. Output which extends beyond the visible extent of the display is clipped and does not appear.

GET <lVar> specifies the name of the variable associated with the GET. <l*Var*> may contain a value of any type, but upon return from the READ, will always contain logical false (.F.). The purpose of *<lVar>* is to provide a place holder in the *GetList* for the push button.

CAPTION <cCaption> specifies a character string that concisely describes the GET on the screen. Caption differs from the SAY argument in two ways. The first is that the caption is displayed each time the GET is shown on the screen. The other difference is that the caption, along with its screen position, is maintained within the Get object. This allows the GET to receive input when the mouse's left button is clicked and its cursor is on the caption. By default, the caption appears within the push button's border. Use the Get object's capRow or capCol variable to change the screen position of the caption.

When present, the & character specifies that the character immediately following it in the caption is the push button's accelerator key. The accelerator key provides a quick and convenient mechanism for the user to move input focus from one data input control to the push button. The case of an accelerator key is ignored.

MESSAGE **<*cMessage*>** specifies a character string that is displayed on the Get system's status bar line when the GET has input focus. Typically, it describes the anticipated contents or user response of the GET. Refer to the READ command for details pertaining to the Get system's status bar.

WHEN **<*lPreExpression*>** specifies an expression that must be satisfied before the cursor can enter the GET during a READ. The expression can optionally be a code block. The Get object is passed as a parameter to the code block.

VALID **<*lPostExpression*>** specifies an expression that must be satisfied before the cursor can leave the GET during a READ. The expression can optionally be a code block. The Get object is passed as a parameter to the code block.

COLOR **<*cColorString*>** defines the color settings for the push button. <*cColorString*> is a character expression containing exactly four color settings.

PushButton Color Attributes

Position in colorSpec	Applies To	Default Value from System Color Setting
1	The push button when it does not have input focus	Unselected
2	The push button when it has input focus and is not pressed	Enhanced
3	The push button when it has input focus and is pressed	Standard
4	The push button caption's accelerator key	Background

Note: The background colors of the PushButton Color Attributes are ignored in graphic mode.

FOCUS <***fblock***> specifies a code block that is evaluated each time the push button recieves focus. The code block takes no implicit arguments. Use the PushButton:hasFocus instance variable to determine if the push button has focus.

STATE **<*bBlock*>** specifies a code block that is evaluated each time the push button object's state changes. The code block takes no implicit arguments. Use the PushButton:buffer instance variable to determine if the push button is pressed or released. A value of true (.T.) indicates that it is being pressed; otherwise, a value of false (.F.) indicates that it is being released.

STYLE <cStyle> specifies a character string that indicates the delimiter characters that are used by the push button's display() method. When specified, the string must contain either zero, two, or eight characters. The default is two characters. The first is the left delimiter. Its default value is the less than (<) character. The second character is the right delimiter. Its default value is the greater than (>) character.

When the style string is empty, the button has no delimiters. When the string length is two, the button has left and right delimiters and occupies one row on the screen. The first character is the left delimiter. The second character is the right delimiter. When the string length is eight, the button is contained within a box that occupies three rows on the screen.

Standard Box Types

Constant	**Description**
B_SINGLE	Single-line box
B_DOUBLE	Double-line box
B_SINGLE_DOUBLE	Single-line top/bottom, double-line sides
B_DOUBLE_SINGLE	Double-line top/bottom, single-line sides

Box.ch contains manifest constants for the PushButton:style value.

The default style for the @...GET PUSHBUTTON is "<>".

Note: The STYLE clause is ignored in graphic mode.

SEND <msg> sends the specified message to the Get object. *<msg>* is sent before the GET is displayed. Any message can be sent, including method calls and instance variable assignments. Method calls must include parentheses even if no arguments are passed.

GUISEND <guimsg> can be used to send a message (such as the display() method) to a GUI object. The GUI objects available in CA-Clipper are check boxes, list boxes, push buttons, and radio button groups. To send a message to a non-GUI object, such as standard GETs and TBrowses, use the SEND clause.

SIZE X <nSizeX> Y <nSizeY> defines the size of the button to be displayed. The coordinates are in pixels. *<nSizeX>* represents the number of pixels in the x direction (horizontally) and *<nSizeY>* represents the number of pixels in the y direction (vertically). This argument only affects applications running in graphic mode and is ignored in text mode.

CAPOFF X <nCapXOff> Y <nCapYOff> defines the offset where the caption is displayed. The coordinates are in pixels. *<nCapXOff>* represents the number of pixels in the x direction (horizontally) from the left edge of the button where the caption will be displayed. *<nCapYOff>* represents the number of pixels in the y direction (vertically) from the top edge of the button where the caption will be displayed. If the CAPOFF clause is not supplied, the caption will be centered on the button. This argument only affects applications running in graphic mode and is ignored in text mode.

BITMAP <cBitmap> defines a bitmap file to be displayed on the button. Drive and directory names are not allowed; the file name extension is required. A bitmap file can be stored as a file on disk or in a bitmap library. If stored as a file, the file must reside in the same directory as the application. If stored in a bitmap library, the library must reside in the same directory as the application and it also must have the same name as the application with a .BML extension.

CA-Clipper will search for the file name first and, if it is not found, search in the bitmap library second. If no file is found either on disk or in the library, no bitmap will be displayed.

BMPOFF X <nBmpXOff> Y <nBmpYOff> defines the offset where the bitmap is displayed. The coordinates are in pixels. *<nCapXOff>* represents the number of pixels in the x direction (horizontally) from the left edge where the bitmap will be displayed. *<nCapYOff>* represents the number of pixels in the y direction (vertically) from the top edge where the bitmap will be displayed. If the BMPOFF clause is not supplied, the bitmap will be placed at the upper-left corner of the button. This argument only affects applications running in graphic mode and is ignored in text mode.

Examples

- This example demonstrates how to add a push button to a data entry screen:

```
@ 5, 20 GET lCancel PUSHBUTTON ;
        CAPTION "&Cancel"
        STATE { || ReadKill( .T. ) }
```

Files

Libraries are CLIPPER.LIB and LLIBG.LIB.

See Also

@...GET, Get class, PushButton class, READ

@...GET RADIOGROUP command

Create a new radio button group Get object and display it to the screen

Syntax

```
@ <nTop>, <nLeft>, <nBottom>, <nRight>
   GET <nVar|cVar>
   RADIOGROUP <aGroup>
      [CAPTION<cCaption>]
      [MESSAGE <cMessage>]
      [COLOR <cColorString>]
      [FOCUS] <fblock>
      [WHEN <lPreExpression>]
      [VALID <lPostExpression>]
      [SEND <msg>]
      [GUISEND <guimsg>]
```

Arguments

<nTop>, <nLeft>, nBottom> and ***<nRight>*** specify the screen position for the radio button group and its caption. Output which extends beyond the visible extent of the display is clipped and does not appear.

GET <nVar | cVar> specifies the name of the variable associated with the GET. Its value indicates which radio button (if any) in the group is selected. A numeric value refers to the position in the group of the selected button. A value of 0 indicates no selected button. A character string value refers to the data or text contained within the selected button. A character string that does not refer to any item in the list indicates no selected item.

RADIOGROUP *<aGroup>* specifies an array of RadioButto objects.

CAPTION <cCaption> specifies a character string that concisely describes the GET on the screen. Caption differs from the SAY argument in two ways. The first is that the caption is displayed each time the GET is shown on the screen. The other difference is that the caption, along with its screen position is maintained within the Get object. This allows the GET to receive input when the mouse's left button is clicked and its cursor is on the caption. By default, the caption appears in the top-left border of the GET. Use the Get object's capRow or capCol variable to change the screen position of the caption.

When present, the & character specifies that the character immediately following it in the caption is the radio group's accelerator key. The accelerator key provides a quick and convenient mechanism for the user to move input focus from one data input control to the radio group. The case of an accelerator key is ignored.

MESSAGE <cMessage> specifies a character string that is displayed on the Get system's status bar line when the GET has input focus. Typically, it describes the anticipated contents or user response of the GET. Refer to the READ command for details pertaining to the Get system's status bar.

COLOR <cColorString> defines the color settings for the radio button group. *<cColorString>* is a character expression containing exactly three color settings.

RadioGroup Color Attributes

Position in colorSpec	Applies To	Default Value from System Color Setting
1	The radio group's border	Border
2	The radio group's caption	Standard
3	The radio group caption's accelerator key	Background

FOCUS <fblock> specifies a code block that is evaluated each time the radio button group receives focus. The code block takes no implicit arguments. Use the RadioGroup:hasFocus instance variable to determine if the radio group has focus.

WHEN <lPreExpression> specifies an expression that must be satisfied before the cursor can enter the GET during a READ. The expression can optionally be a code block. The Get object is passed as a parameter to the code block.

VALID <lPostExpression> specifies an expression that must be satisfied before the cursor can leave the GET during a READ. The expression can optionally be a code block. The Get object is passed as a parameter to the code block.

Note: There is no STATE clause for the @...GET RADIOGROUP command. Instead each radio button in the group can have its own sBlock instance variable. See sBlock under RadioButto class for more details.

SEND <msg> sends the specified message to the Get object. *<msg>* is sent before the GET is displayed. Any message can be sent, including method calls and instance variable assignments. Method calls must include parentheses even if no arguments are passed.

GUISEND <guimsg> can be used to send a message (such as the display() method) to a GUI object. The GUI objects available in CA-Clipper are check boxes, list boxes, push buttons, and radio button groups. To send a message to a non-GUI object, such as standard GETs and TBrowses, use the SEND clause.

Examples

- This example demonstrates how to add a radio button group to a data entry screen. It utilizes a position-oriented method of selection:

```
nColor := 1 //default to the first item.
aGroup := ARRAY( 3 )
aGroup[ 1 ] := RadioButto( 6, 22, "&Red" )
aGroup[ 2 ] := RadioButto( 7, 22, "&Green" )
aGroup[ 3 ] := RadioButto( 8, 22, "&Blue" )
@ 5, 20, 9, 40 GET nColor RADIOGROUP aGroup
```

The nColor variable will return a position, in this case 1, 2, or 3.

- This example demonstrates a radio button group that utilizes a data-oriented method of selection:

```
cColor := "R" //default to red.
aGroup := ARRAY( 3 )
aGroup[ 1 ] := RadioButto( 6, 22, "&Red", "R" )
aGroup[ 2 ] := RadioButto( 7, 22, "&Green", "G" )
aGroup[ 3 ] := RadioButto( 8, 22, "&Blue", "B" )
@ 5, 15, 9, 40 GET cColor RADIOGROUP aGroup
```

The nColor variable will return data, in this case "R", "G", or "B".

Files Library is CLIPPER.LIB.

See Also @...GET, Get class, RadioButto class, RadioGroup class, READ

@...GET TBROWSE command

Create a new TBrowse Get object and display it to the screen

Syntax

```
@ <nTop>, <nLeft>, <nBottom>, <nRight>
   GET <idVar>
   TBROWSE <oBrowse>
      [MESSAGE <cMessage>]
      [WHEN <lPreExpression>]
      [VALID <lPostExpression>]
      [SEND <msg>]
      [GUISEND <guimsg>]
```

Arguments

<nTop>, <nLeft>, <nBottom> and ***<nRight>*** specify the screen position for the TBrowse object. Output which extends beyond the visible extent of the display is clipped and does not appear.

GET <idVar> specifies the name of the variable associated with the GET. Its value is neither queried nor modified by the Get system. The GET is merely a mechanism used for integrating a TBrowse object into the @...GET/READ system.

TBROWSE <oTBrowse> specifies a previously defined TBrowse object.

MESSAGE <cMessage> specifies a character string that is displayed on the Get system's status bar line when the GET has input focus. Typically, it describes the anticipated contents or user response of the GET. Refer to the READ command for details pertaining to the Get system's status bar.

WHEN <lPreExpression> specifies an expression that must be satisfied before the cursor can enter the GET during a READ. The expression can optionally be a code block. The Get object is passed as a parameter to the code block.

VALID <lPostExpression> specifies an expression that must be satisfied before the cursor can leave the GET during a READ. The expression can optionally be a code block. The Get object is passed as a parameter to the code block.

SEND <msg> sends the specified message to the Get object. *<msg>* is sent before the GET is displayed. Any message can be sent, including method calls and instance variable assignments. Method calls must include parentheses even if no arguments are passed.

GUISEND <guimsg> can be used to send a message (such as the display() method) to a GUI object. The GUI objects available in CA-Clipper are check boxes, list boxes, push buttons, and radio button groups. To send a message to a non-GUI object, such as standard GETs and TBrowses, use the SEND clause.

Examples

- This example demonstrates how to add a TBrowse object to a data entry screen:

```
oTB := TBrowseDB( 10, 10, 15, 40 )
oTB:AddColumn(tbColumnNew("Last Name",{||Customer->lName}))
oTB:AddColumn(tbColumnNew("First Name",{||Customer->fName}))
oTB:AddColumn(tbColumnNew("Phone",{||Customer->Phone}))
uDummy := NIL
@ 10, 10, 15, 40 GET uDummy TBROWSE oTB
```

Files

Library is CLIPPER.LIB.

See Also

@...GET, Get class, TBrowse class, READ

@...PROMPT command

Paint a menu item and define a message

Syntax

```
@ <nRow>, <nCol> PROMPT <cMenuItem>
   [MESSAGE <cExpression>]
```

Arguments

<nRow> and ***<nCol>*** are the row and column coordinates of the menu item display. Row values can range from zero to MAXROW(), and column values can range from zero to MAXCOL().

PROMPT <cMenuItem> is the menu item string to display.

MESSAGE <cExpression> defines the message to display each time the current menu item is highlighted. A code block evaluating to a character expression may optionally be used.

Description

@...PROMPT is the display portion of the CA-Clipper lightbar menu system. Each @...PROMPT command paints a menu item in the current standard color and defines an associated MESSAGE to be displayed on the line specified by SET MESSAGE. The lightbar menu is then invoked with MENU TO. You can specify menu items in any order and configuration of row and column position. MENU TO, however, navigates the current list of menu items in the order they were defined. You can define up to 4096 menu items for each menu.

After each @...PROMPT command, the cursor is located one column position to the right of the last menu item character and ROW() and COL() are updated to reflect the new cursor position. This lets you use ROW() and COL() to specify consecutive menu item positions relative to the first one painted. See the example below.

Examples

- This example displays a lightbar menu with the associated messages displayed on the next line. When the user presses Return, the position of the item in the list of menu items is assigned to nChoice:

```
LOCAL nChoice := 1
SET WRAP ON
SET MESSAGE TO 2
@ 1, 3 PROMPT "File" MESSAGE "Access data files"
@ ROW(), COL() + 2 PROMPT "Edit" ;
   MESSAGE "Edit current record"
MENU TO nChoice
```

- This example shows how to pass the MESSAGE clause a code block which calls a user-defined function that displays a message in a different color:

```
SET COLOR TO gr+/b,r+/n
SET MESSAGE TO 23     // This is necessary.
CLEAR SCREEN

@ 3,4 PROMPT "one "  MESSAGE {||UDF("Message One  ")}
@ 4,4 PROMPT "two "  MESSAGE {||UDF("Message Two  ")}
@ 5,4 PROMPT "three" MESSAGE {||UDF("Message Three")}

MENU TO test

FUNCTION UDF(cParm)
cOldColor := SETCOLOR("w+/n")
@ 22,1 SAY cParm      // Must be different row than the
                      // SET MESSAGE TO nRow
SETCOLOR(cOldColor)
RETURN ""             // Character string must be returned
```

Files Library is CLIPPER.LIB.

See Also ACHOICE(), MENU TO, SET MESSAGE, SET WRAP, SETCOLOR()

@...SAY command

Display data at a specified screen or printer row and column

Syntax

```
@ <nRow>, <nCol>
   SAY <exp> [PICTURE <cSayPicture>]
   [COLOR <cColorString>]
```

Arguments

<nRow> and ***<nCol>*** are the row and column coordinates of the display. Row values can range from zero to a maximum of MAXROW(), if the current DEVICE is the SCREEN, or 32,766, if the current DEVICE is the PRINTER. Also, column values can range from zero to a maximum of MAXCOL() or 32,766 if the current DEVICE is the PRINTER.

SAY <exp> displays the result of a character, date, logical, or numeric expression to the current DEVICE.

PICTURE <cSayPicture> defines the formatting control for the display of *<exp>*. CA-Clipper provides two mechanisms, functions and templates, to control formatting. Functions apply to an entire SAY, while templates format characters position by position.

COLOR <cColorString> defines the display color of *<exp>*. If not specified, *<exp>* displays in the standard color as defined by SETCOLOR(). *<cColorString>* is a character expression containing the standard color setting. If you specify a literal color setting, it must be enclosed in quote marks.

On a combined @...SAY...GET command, two COLOR clauses are required to specify colors for both the SAY and the GET: one for the SAY and one for the GET.

Description

@...SAY is a full-screen command that outputs the results of <*exp*> to either the screen or the printer at the specified row and column coordinates. It can optionally format output using the PICTURE clause. @...SAY creates data entry screens or reports that can be sent to the screen or printer.

When an @...SAY command executes, the output from <*exp*> is sent to the current device defined with SET DEVICE. The current DEVICE can be the SCREEN or PRINTER. Unlike console commands, @...SAY output to the printer is not echoed to the screen and SET CONSOLE has no effect on @...SAY output to the screen.

If the current DEVICE is the SCREEN (the system default), @...SAY displays output to the screen leaving the cursor one column position to the right of the last character displayed. ROW() and COL() are then updated with this position. Output that displays off the screen, as defined by MAXROW() and MAXCOL(), is clipped and the cursor is positioned beyond the visible screen. All @...SAY output displays are in standard color. Refer to the SETCOLOR() reference in this chapter for more information on color.

If the current DEVICE is set to PRINTER, the display is directed to the printer at the specified <*nRow*> and <*nCol*> position. If the current MARGIN value is greater than zero, it is added to <*nCol*> first. The printhead is then advanced one column position to the right of the last character output and PROW() and PCOL() are updated. @...SAY commands to the printer behave differently from those to the screen if output is addressed to a printer row or column position less than the current PROW() and PCOL() values:

- If <*nRow*> is less than PROW(), an automatic EJECT (CHR(12)) is sent to the printer followed by the number of linefeed characters (CHR(10)) required to position the printhead on <*nRow*> on the following page
- If <*nCol*> including the SET MARGIN value is less than PCOL(), a carriage return character (CHR(13)) and the number of spaces required to position <*exp*> at <*nCol*> are sent to the printer

To override this behavior and send control codes to the printer, or for any other reason, you can use SETPRC() to reset PROW() and PCOL() to new values. See the SETPRC() function reference for more information.

If the current DEVICE is the PRINTER, redirect output from @...SAY commands to a file using the SET PRINTER TO <*xcFile*> command.

@...SAY command output can be formatted using the PICTURE clause with a <*cSayPicture*>. This performs the same action as the TRANSFORM() function. A <*cSayPicture*> may consist of a function and/or a template. A PICTURE function imposes a rule on the entire @...SAY output. A PICTURE template defines the length of the @...SAY output and the formatting rule for each position within the output.

- ***Function string:*** A PICTURE function string specifies formatting rules which apply to the SAY's entire display value, rather than to particular character positions within it. The function string consists of the @ character, followed by one or more additional characters, each of which has a particular meaning (see table below). The function string must not contain spaces. A function string may be specified alone or with a template string. If both are present, the function string must precede the template string, and the two must be separated by a single space.

SAY and TRANSFORM() PICTURE Format Functions

Function	Action
B	Displays numbers left-justified
C	Displays CR after positive numbers
D	Displays dates in SET DATE format
E	Displays dates and numbers in British format
R	Nontemplate characters are inserted
X	Displays DB after negative numbers
Z	Displays zeros as blanks
(	Encloses negative numbers in parentheses
!	Converts alphabetic characters to uppercase

- ***Template string:*** A PICTURE template string specifies formatting rules on a character-by-character basis. The template string consists of a series of characters, some of which have special meanings (see table below). Each position in the template string corresponds to a position in the displayed SAY value. Characters in the template string that do not have assigned meanings are copied verbatim into the displayed SAY value. If you use the @R picture function, characters without special PICTURE template string meaning are inserted between characters of the display value; otherwise, they overwrite the corresponding characters of the display value. You may specify a template string alone or with a function string. If both are present, the function string must precede the template string, and the two must be separated by a single space.

SAY and TRANSFORM() Template Symbols

Template	Action
A,N,X,9,#	Displays digits for any data type
L	Displays logicals as "T" or "F"
Y	Displays logicals as "Y" or "N"
!	Converts alphabetic characters to uppercase
$	Displays a dollar sign in place of a leading space in a number
*	Displays an asterisk in place of a leading space in a number
.	Specifies a decimal point position
,	Specifies a comma position

Examples

- This example uses an @...SAY with a PICTURE clause to display formatted output:

```
nNetIncome = 7125.50
nNetLoss = -125.50
cPhone = "2134567890"
cName = "Kate Mystic"
//
@ 1, 1 SAY nNetIncome PICTURE "@E 9,999.99"
// Result: 7.125,50

@ 2, 1 SAY nNetLoss PICTURE "@)"
// Result: (125.50)

@ 3, 1 SAY cPhone PICTURE "@R (999)999-9999"
// Result: (213)456-7890

@ 4, 1 SAY cName PICTURE "@!"
// Result: KATE MYSTIC
```

- This example is a small label printing program that uses SET DEVICE to direct output to the printer and SETPRC() to suppress automatic EJECTs:

```
USE Salesman INDEX Salesman NEW
SET DEVICE TO PRINTER
DO WHILE !EOF()                        // Print all records
   @ 2, 5 SAY RTRIM(FirstName) + ", " + LastName
   @ 3, 5 SAY Street
   @ 4, 5 SAY RTRIM(City) + ", " + State + "  " + ;
            PostalCode
   @ 6, 0 SAY SPACE(1)                 // Move to label bottom
   SETPRC(0, 0)                        // Suppress page eject
   SKIP                                // Next record
ENDDO
SET DEVICE TO SCREEN
CLOSE Salesman
```

Files

Library is CLIPPER.LIB.

See Also

?|??, @...GET, CLEAR, COL(), PCOL(), PROW(), QOUT(), ROW(), SET DEVICE, SET PRINTER, SETCOLOR(), SETPRC(), TRANSFORM()

@...TO command

Draw a single- or double-line box

Syntax

```
@ <nTop>, <nLeft>
   TO <nBottom>, <nRight> [DOUBLE] [COLOR
      <cColorString>]
```

Arguments

<nTop>*, *<nLeft>*, *<nBottom>, and ***<nRight>*** define the coordinates of the box. @...TO draws the box using row values from zero to MAXROW() and column values from zero to MAXCOL(). *<nBottom>* and *<nRight>* can be larger than the screen size, but output is clipped at MAXROW() and MAXCOL().

DOUBLE draws the box with a double line. If not specified, the box is drawn with a single line.

COLOR <cColorString> defines the display color of the drawn box. If not specified, the box is drawn using the standard color setting of the current system color as defined by SETCOLOR(). Note that *<cColorString>* is a character expression containing the standard color setting. If you specify a literal color setting, enclose it within quote marks.

Description

@...TO draws a single- or double-line box on the screen. If *<nTop>* and *<nBottom>* are the same, a horizontal line is drawn. If *<nLeft>* and *<nRight>* are the same, a vertical line is drawn.

After @...TO finishes drawing, the cursor is located in the upper-left corner of the boxed region at *<nTop>* + 1 and *<nLeft>* + 1. ROW() and COL() are also updated to reflect the new cursor position.

@...TO is like @...BOX except that @...BOX lets you define the characters of the box and supports a fill character. @...TO, however, is recommended for portability since it does not require the specification of hardware-dependent graphics characters.

Examples

- This example erases a region of the screen, then draws a box of the same size:

```
@ 10, 10 CLEAR TO 20, 40
@ 10, 10 TO 20, 40 DOUBLE COLOR "BG+/B"
```

Files

Library is CLIPPER.LIB.

See Also

@...BOX, @...CLEAR, DISPBOX()

[ ] operator

Array element indicator (Special)

Syntax

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

Operands

<aArray> is an expression that returns a reference to an array. This is generally a variable identifier or instance variable.

<nSubscript> is a numeric expression that addresses an individual element in the specified array or subarray. Each subscript corresponds to a dimension of the array.

Description

The subscript operator ([]) specifies a single array element. The name of a previously declared array must precede the left bracket and the array element subscript must appear as a numeric expression within the brackets. You can make array element references using Pascal or C-style syntax.

Examples

- This example accesses each element in a two-dimensional array of known dimensions:

```
LOCAL i, j
FOR i := 1 TO 5
    FOR j := 1 TO 10
        ? aOne[i, j]
    NEXT
NEXT
```

- These examples specify an <*aArray*> expression:

```
LOCAL aArray := { 1, 2, 3, 4, 5 }
//
? ArrayFunc()[2]                        // Result: 2
? { {1, 2}, {3, 4} }[1][2]              // Result: 2
? aArray[5]                             // Result: 5

FUNCTION ArrayFunc
    STATIC aArray := { 1, 2 }
    RETURN aArray
```

- This example queries and assigns a static array encapsulated within a function definition:

```
? ArrayFunc()[1]                        // Result: 1
ArrayFunc()[1] := 10
? ArrayFunc()[1]                        // Result: 10

FUNCTION ArrayFunc
    STATIC aArray := { 1, 2 }
    RETURN aArray
```

See Also ARRAY(), LOCAL, PRIVATE, PUBLIC, STATIC

{ } operator

Literal array and code block delimiters (Special)

Syntax

```
{ <exp list> }                      (literal array)
{ |<param list>| <exp list> }  (code block definition)
```

Operands

<exp list> is a list of expressions of any type. If the item is a literal array definition, it can contain another literal array definition.

<param list> is a list of variables to receive parameters passed to a code block from an invocation of the EVAL() function. The parameter list is comma-separated and must be enclosed by vertical bars (| |). Variables specified in this list are declared local to the code block and are visible only within the code block definition.

Description

Curly braces ({}) delimit references to literal arrays or code blocks. If the reference is a literal array, you can use them to create an array in either an assignment or a variable declaration statement. If the reference is a variable declaration, the array can contain expressions of any kind as elements, unless STATIC is the declaration statement. In this case, the literal array can only contain constant values.

Examples

- This example uses literal arrays in declaration statements to create a variable and initialize it with an array reference:

```
LOCAL aPerson := { "Name", "Address", DATE() }
STATIC aNumList := { 1, 2, 3, 4, 5, 6, 7, 8 }
```

- This example creates a multidimensional literal array:

```
aMulti := { {1, 2, 3}, {"A", "B", "C"} }
```

- This example uses curly braces to formulate a simple code block with a single parameter:

```
LOCAL bSayBlock
bSayBlock := { |x| QOUT(x) }
EVAL(bSayBlock, 10)              // Result: 10
```

See Also

EVAL(), LOCAL, PRIVATE, PUBLIC, STATIC

AADD() function

Add a new element to the end of an array

Syntax

```
AADD(<aTarget>, <expValue>) → Value
```

Arguments

<*aTarget*> is the array to which a new element is to be added.

<*expValue*> is the value assigned to the new element.

Returns

AADD() evaluates <*expValue*> and returns its value. If <*expValue*> is not specified, AADD() returns NIL.

Description

AADD() is an array function that increases the actual length of the target array by one. The newly created array element is assigned the value specified by <*expValue*>.

AADD() is used to dynamically grow an array. It is useful for building dynamic lists or queues. A good example of this is the *GetList* array used by the Get system to hold Get objects. After a READ or CLEAR GETS, *GetList* becomes an empty array. Each time you execute an @...GET command, the Get system uses AADD() to add a new element to the end of the *GetList* array, and then assigns a new Get object to the new element.

AADD() is similar to ASIZE() but only adds one element at a time; ASIZE() can grow or shrink an array to a specified size. AADD(), however, has the advantage that it can assign a value to the new element, while ASIZE() cannot. AADD() may also seem similar to AINS(), but they are different: AINS() moves elements within an array, but it does not change the array's length.

Note: If <*expValue*> is another array, the new element in the target array will contain a reference to the array specified by <*expValue*>.

Examples

- These examples demonstrate the effects of multiple invocations of AADD() to an array:

```
aArray := {}                // Result: aArray is an empty array
AADD(aArray, 5)             // Result: aArray is { 5 }
AADD(aArray, 10)            // Result: aArray is { 5, 10 }
AADD(aArray, { 12, 10 })    // Result: aArray is
                            // { 5, 10, { 12, 10 } }
```

Files Library is CLIPPER.LIB.

See Also AINS(), ASIZE()

ABS() function

Return the absolute value of a numeric expression

Syntax

```
ABS(<nExp>) → nPositive
```

Arguments

<nExp> is the numeric expression to be evaluated.

Returns

ABS() returns a number representing the absolute value of its argument. The return value is a positive number or zero.

Description

ABS() is a numeric function that determines the magnitude of a numeric value independent of its sign. It lets you, for example, obtain the difference between two numbers as a positive value without knowing in advance which of the two is larger.

As a formalism, ABS(x) is defined in terms of its argument, x, as follows: if x >= 0, ABS(x) returns x; otherwise, ABS(x) returns the negation of x.

Examples

- These examples show typical results from ABS():

```
nNum1 := 100
nNum2 := 150
? nNum1 - nNum2                 // Result: -50
? ABS(nNum1 - nNum2)            // Result: 50
? ABS(nNum2 - nNum1)            // Result: 50
? ABS(-12)                      // Result: 12
? ABS(0)                        // Result: 0
```

Files

Library is CLIPPER.LIB.

*ACCEPT command

Place keyboard input into a memory variable

Syntax

```
ACCEPT [<expPrompt>] TO <idVar>
```

Arguments

<*expPrompt*> is an optional prompt displayed before the input area. The prompt can be an expression of any data type.

<*idVar*> is the variable that will hold input from the keyboard. If the specified <*idVar*> does not exist or is not visible, a private variable is created.

Description

ACCEPT is a console command and wait state that takes input from the keyboard and assigns it as a character string to the specified variable. When ACCEPT is executed, it first performs a carriage return/linefeed, displays the prompt, and then begins taking characters from the keyboard at the first character position following the prompt. You may input up to 255 characters. When input reaches the edge of the screen, as defined by MAXCOL(), the cursor moves to the next line.

ACCEPT supports only two editing keys: Backspace and Return. Esc is not supported. Backspace deletes the last character typed. Return confirms entry and is the only key that can terminate an ACCEPT. If Return is the only key pressed, ACCEPT assigns a null value ("") to <*idVar*>.

Examples

- This example uses ACCEPT to get keyboard input from the user:

```
LOCAL cVar
ACCEPT "Enter a value: " TO cVar
IF cVar == ""
   ? "User pressed Return"
ELSE
   ? "User input:", cVar
ENDIF
```

Files

Library is CLIPPER.LIB.

See Also

@...GET, @...SAY, INKEY(), INPUT, KEYBOARD, WAIT*

ACHOICE() function

Execute a pop-up menu

Syntax

```
ACHOICE(<nTop>, <nLeft>, <nBottom>, <nRight>,
    <acMenuItems>,
    [<alSelectableItems> | <lSelectableItems>],
    [<cUserFunction>],
    [<nInitialItem>],
    [<nWindowRow>]) → nPosition
```

Arguments

<nTop>, <nLeft> and ***<nBottom>, <nRight>*** are the upper-left and lower-right window coordinates. Row values can range from zero to MAXROW() and column values can range from zero to MAXCOL().

<acMenuItems> is an array of character strings to display as the menu items. The individual menu items are later identified by their numeric positions in this array.

<alSelectableItems> is a parallel array of logical values—one element for each item in *<acMenuItems>*—that specify the selectable menu items. Elements can be logical values or character strings. ACHOICE() will not permit a null string and stops displaying if it encounters one. If the element is a character string, it is evaluated as a macro expression which should evaluate to a logical data type. In either case, a value of false (.F.) means that the corresponding menu item is not available, and a value of true (.T.) means that it is available. If you specify *<lSelectableItems>* instead of an array, false (.F.) makes all menu items unavailable and true (.T.) makes all menu items available. By default, all menu items are available for selection.

<cUserFunction> is the name of a user-defined function that executes when an unrecognizable key is pressed. Specify the function name as a character expression without parentheses or arguments. Note that the behavior of ACHOICE() is affected by the presence of this argument. Refer to the discussion below for further information.

<nInitialItem> is the position in the *<acMenuItems>* array of the item that will be highlighted when the menu is initially displayed. If you specify an unavailable menu item or no argument at all, the initial menu item is the first selectable item in the array.

<nWindowRow> is the number of the window row on which the initial menu item will appear. Row numbering begins with zero. By default, the initial menu item appears as close to the top of the window as possible, without leaving any empty rows at the bottom. Thus, if there are enough menu items following the initial one to fill up the window, the initial form will appear on the first row (row zero) of the menu. This function argument is used to control the initial menu appearance when there are more menu items than will fit in the window.

As with all functions, optional arguments are omitted by using a comma instead of the actual argument.

Returns

ACHOICE() returns the numeric position in the *<acMenuItems>* array of the menu item selected. If the selection process is aborted, ACHOICE() returns zero.

Description

ACHOICE() is a user interface function that can create various kinds of pop-up menus. Each menu uses an array of character strings as menu items and a parallel array of logical values to determine whether items are selectable. When you invoke ACHOICE(), the list of menu items is displayed within the specified window coordinates. When the user presses Return, the current item is selected, and ACHOICE() returns the position of the menu item in *<acMenuItems>*. When the user presses Esc, ACHOICE() aborts and returns zero.

The menu items scroll if the number of items in *<acMenuItems>* exceeds the number of rows in the menu window, and the user attempts to move the highlight beyond the top or bottom of the menu window. Note that the highlight does not wrap when you reach the top or bottom of the list of items. Pressing the first letter does, however, wrap the highlight within the set of items whose first letter matches the key you press.

- ***Navigating the menu:*** ACHOICE() has two modes depending on whether the <*cUserFunction*> argument is specified. If it is not specified the following navigation keys are active:

ACHOICE() Keys (No User Function)

Key	Action
Up arrow	Go to previous item
Down arrow	Go to next item
Home	Go to first item in menu
End	Go to last item in menu
Ctrl+Home	Go to first item in window
Ctrl+End	Go to last item in window
PgUp	Go to previous page
PgDn	Go to next page
Ctrl+PgUp	Go to the first item in menu
Ctrl+PgDn	Go to last item in menu
Return	Select current item
Esc	Abort selection
Left arrow	Abort selection
Right arrow	Abort selection
First Letter	Go to next item beginning with first letter

- ***Color:*** Menu items are displayed in standard color, the highlight in enhanced color, and the unavailable items in the unselected color. For example, the following color statement

```
SETCOLOR("W+/N, BG+/B, , , W/N")
```

displays a menu that is bright white on black, the highlight is bright cyan on blue, and the unavailable menu items are dim white on black.

- ***User function:*** Like the other user interface functions, ACHOICE() supports a user function. The user function is specified when you want to nest ACHOICE() invocations to create hierarchical menus or to redefine keys.

  When a user function is specified, ACHOICE() processes only a limited set of keys automatically. These are listed in the following table. All other keys generate a key exception which passes control to the user function for handling. Control is also passed to the user function when ACHOICE() goes idle (i.e., when there are no more keys to process).

  ACHOICE() Keys (User Function Specified)

Key	Action
Uparrow	Go to previous item
Dnarrow	Go to next item
Ctrl+Home	Go to first item in window
Ctrl+End	Go to last item in window
PgUp	Go to previous page
PgDn	Go to next page
Ctrl+PgUp	Go to the first item in menu
Ctrl+PgDn	Go to last item in menu

  When ACHOICE() executes the user function, it automatically passes the following three parameters:

  – The current ACHOICE() mode

  – The current element in the array of items

  – The relative row position within the menu window

  The mode indicates the current state of ACHOICE() depending on the key pressed and the action taken by ACHOICE() prior to executing the user function. The mode parameter has the following possible values:

  ACHOICE() Modes

Mode	Achoice.ch	Description
0	AC_IDLE	Idle
1	AC_HITTOP	Attempt to cursor past top of list
2	AC_HITBOTTOM	Attempt to cursor past bottom of list
3	AC_EXCEPT	Keystroke exceptions
4	AC_NOITEM	No selectable items

After the user function has performed whatever operations are appropriate to the ACHOICE() mode or LASTKEY(), it must RETURN a value requesting ACHOICE() to perform an operation from the following set of actions:

ACHOICE() User Function Return Values

Value	Achoice.ch	Action
0	AC_ABORT	Abort selection
1	AC_SELECT	Make selection
2	AC_CONT	Continue ACHOICE()
3	AC_GOTO	Go to the next item whose first character matches the key pressed

Examples

- This example uses two literal arrays to specify the menu items and selection criteria. After the menu is displayed and the user makes a selection, the name of the selected menu item is displayed:

```
acMenuItems := {"One", "Two", "-------", "Three"}
alSelectableItems := {.T., .T., .F., .T.}
nPosition := ACHOICE(10, 10, 12, 15, acMenuItems,;
                        alSelectableItems)
? acMenuItems[nPosition]
```

- This example declares an array of menu items and supplies a user-defined function which displays a message with each highlighted choice:

```
#include "Achoice.ch"
#include "Inkey.ch"

PROCEDURE Main()

    LOCAL acMenuItems[4], cUserFunction, nRetVal
    LOCAL nKey, nPos

    acMenuItems[1] := "Add"
    acMenuItems[2] := "Edit"
    acMenuItems[3] := "Delete"
    acMenuItems[4] := "Update"

    CLS

    nPos := ACHOICE( 10, 10, 13, 15, acMenuItems,;
                  .T., "cUserFunction" )
```

```
    DO CASE
    CASE nPos == 1
       //  Put ADD routine here
    CASE nPos == 2
       //  Put EDIT routine here
    CASE nPos == 3
       //  Put DELETE routine here
    CASE nPos ==4
       //  Put UPDATE routine here
    ENDCASE

RETURN


FUNCTION cUserFunction( nMode, nCurElement, nRowPos )

    LOCAL nRetVal := AC_CONT     // Default, Continue
    LOCAL nKey := LASTKEY()

    DO CASE
// After all pending keys are processed, display message
    CASE nMode == AC_IDLE
    DO CASE
       CASE nCurElement == 1
          @ 22, 5 SAY " Adding   "
       CASE nCurElement == 2
          @ 22, 5 SAY " Editing  "
       CASE nCurElement ==  3
          @ 22, 5 SAY " Deleting "
       CASE nCurElement ==  4
          @ 22, 5 SAY " Updating "
    ENDCASE

       nRetVal := AC_CONT           // Continue ACHOICE()

    CASE nMode == AC_HITTOP         // Attempt to go past Top
       TONE( 100, 3 )
    CASE nMode == AC_HITBOTTOM      // Attempt to go past
                                    // Bottom
       TONE( 100, 3 )

    CASE nMode == AC_EXCEPT         // Key Exception
       DO CASE
       CASE nKey == K_RETURN        // If RETURN key, select
          nRetVal := AC_SELECT
       CASE nKey == K_ESC           // If ESCAPE key, abort
          nRetVal := AC_ABORT
       OTHERWISE
             nRetVal := AC_GOTO     // Otherwise, go to item
       ENDCASE
    ENDCASE

RETURN nRetVal
```

- The next example declares the arrays, specifies a selection condition for one of the menu items, and supplies a user function:

```
EXTERNAL UPDATED
//
FUNCTION MyMenu
   LOCAL acMenuItems[4], alSelectableItems[4],;
         cUserFunction := "DoIt"
   //
   acMenuItems[1] := "Add Record"
   acMenuItems[2] := "Edit Record"
   acMenuItems[3] := "Delete Record"
   acMenuItems[4] := "Update Record"
   //
   alSelectableItems[1] := .T.
   alSelectableItems[2] := .T.
   alSelectableItems[3] := .T.
   alSelectableItems[4] := "!UPDATED()"
   // Selection condition

   RETURN ACHOICE(10, 10, 12, 15, acMenuItems,;
      alSelectableItems, cUserFunction)
```

- This example uses two arrays to specify menu items and corresponding action blocks. After the menu is displayed and the user makes a selection, the ACHOICE() return value is used to determine which action block of the *aActionItems* array is evaluated:

```
PROCEDURE Main()
   LOCAL nChoice
   LOCAL aMenuItems := { "Add Record    ", ;
                         "Edit Record   ", ;
                         "Delete Record", ;
                         "Update Record"   }

   LOCAL aActionItems := { {|| AddFunc()  }, ;
                           {|| EditFunc() }, ;
                           {|| DelFunc()  }, ;
                           {|| UpdFunc()  }  }

   nChoice := ACHOICE( 10, 10, 13, 22, aMenuItems )

   IF nChoice == 0
      QUIT     // ESCAPE was pressed
   ENDIF

   EVAL( aActionItems[nChoice] )

RETURN
```

Files Library is EXTEND.LIB, header files are Achoice.ch and Inkey.ch.

See Also MENU TO, SET COLOR*

ACLONE() function

Duplicate a nested or multidimensional array

Syntax

```
ACLONE(<aSource>) --> aDuplicate
```

Arguments

<*aSource*> is the array to be duplicated.

Returns

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

Description

ACLONE() is an array function that creates a complete duplicate of the <*aSource*> array. If <*aSource*> contains subarrays, ACLONE() creates matching subarrays and fills them with copies of the values in the <*aSource*> subarrays. ACLONE() is similar to ACOPY(), but ACOPY() does not duplicate nested arrays.

Examples

- This example creates an array then duplicates it using ACLONE(). The first array is then altered, but the duplicate copy is unaffected:

```
LOCAL aOne, aTwo
aOne := { 1, 2, 3 }         // Result: aOne is {1, 2, 3}
aTwo := ACLONE(aOne)        // Result: aTwo is {1, 2, 3}
aOne[1] := 99               // Result: aOne is {99, 2, 3}
                            // aTwo is still {1, 2, 3}
```

Files Library is CLIPPER.LIB.

See Also ACOPY(), ADEL(), AINS(), ASIZE()

ACOPY() function

Copy elements from one array to another

Syntax

```
ACOPY(<aSource>, <aTarget>,
   [<nStart>], [<nCount>], [<nTargetPos>]) → aTarget
```

Arguments

<aSource> is the array to copy elements from.

<aTarget> is the array to copy elements to.

<nStart> is the starting element position in the *<aSource>* array. If not specified, the default value is one.

<nCount> is the number of elements to copy from the *<aSource>* array beginning at the *<nStart>* position. If *<nCount>* is not specified, all elements in *<aSource>* beginning with the starting element are copied.

<nTargetPos> is the starting element position in the *<aTarget>* array to receive elements from *<aSource>*. If not specified, the default value is one.

Returns

ACOPY() returns a reference to the target array, *<aTarget>*.

Description

ACOPY() is an array function that copies elements from the *<aSource>* array to the *<aTarget>* array. The *<aTarget>* array must already exist and be large enough to hold the copied elements. If the *<aSource>* array has more elements, some elements will not be copied.

ACOPY() copies values of all data types including NIL and code blocks. If an element of the *<aSource>* array is a subarray, the corresponding element in the *<aTarget>* array will contain a reference to the subarray. Thus, ACOPY() will not create a complete duplicate of a multidimensional array. To do this, use the ACLONE() function.

Examples

- This example creates two arrays, each filled with a value. The first two elements from the source array are then copied into the target array:

```
LOCAL nCount := 2, nStart := 1, aOne, aTwo
aOne := { 1, 1, 1 }
aTwo := { 2, 2, 2 }
ACOPY(aOne, aTwo, nStart, nCount)
// Result: aTwo is now { 1, 1, 2 }
```

Files

Library is CLIPPER.LIB.

See Also

ACLONE(), ADEL(), AEVAL(), AFILL(), AINS(), ASORT()

ADEL() function

Delete an array element

Syntax

```
ADEL(<aTarget>, <nPosition>) → aTarget
```

Arguments

<aTarget> is the array to delete an element from.

<nPosition> is the position of the target array element to be deleted.

Returns

ADEL() returns a reference to the target array, *<aTarget>*.

Description

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

Warning! CA-Clipper *implements multidimensional arrays by nesting arrays within other arrays. If the <aTarget> array is a multidimensional array, ADEL() can delete an entire subarray specified by <nPosition>, causing <aTarget> to describe an array with a different structure than the original.*

Examples

- This example creates a constant array of three elements, and then deletes the second element. The third element is moved up one position, and the new third element is assigned a NIL:

```
LOCAL aArray
aArray := { 1, 2, 3 }       // Result: aArray is
                            // now { 1, 2, 3 }
ADEL(aArray, 2)             // Result: aArray is
                            // now { 1, 3, NIL }
```

Files

Library is CLIPPER.LIB.

See Also

ACOPY(), AFILL(), AINS()

ADIR()* function

Fill a series of arrays with directory information

Syntax

```
ADIR([<cFilespec>],
    [<aFilenames>],
    [<aSizes>],
    [<aDates>],
    [<aTimes>],
    [<aAttributes>]) → nFiles
```

Arguments

<cFilespec> is the path specification of files to include in the scan of the DEFAULT directory. It is a standard file specification that can include the wildcard characters * and ?, as well as a drive and path reference. If omitted, the default specification is *.*.

<aFilenames> is the array to fill with the file names matching *<cFilespec>*. Each element contains the file name and extension as a character string in all uppercase letters.

<aSizes> is the array to fill with the sizes of the corresponding files in the *<aFilenames>* array. Each element is a numeric data type.

<aDates> is the array to fill with the dates of the corresponding files in the *<aFilenames>* array. Each element is a date data type.

<aTimes> is the array to fill with the times of the corresponding files in the *<aFilenames>* array. Each element filled contains a character string of the form: *hh:mm:ss*.

<aAttributes> is the array to fill with attributes of the corresponding files in the *<aFilenames>* array. Each element is a character string. If *<aAttributes>* is specified, hidden, system, and directory files are included as well as normal files. If *<aAttributes>* is not specified, only normal files are included.

Returns

ADIR() returns the number of files matching the directory skeleton described in *<cFilespec>*.

Description

ADIR() is an array function that performs two basic operations. First, it returns the number of files matching the file specification. Second, it fills a series of arrays with file names, sizes, dates, times, and attributes.

ADIR() is a compatibility function and therefore not recommended. It is superseded by the DIRECTORY() function which returns all file information in a multidimensional array.

Notes

- ***Directories:*** If you specify the <*aAttributes*> argument and <*cFilespec*> is *.*, directories will be included in <*aFilenames*>. In the <*aAttributes*> array, directories are indicated with an attribute value of "D". If ADIR() is executed within a subdirectory, the first two entries of the <*aFilenames*> array are "." and "..", the parent and current directory aliases. The date and time of last update are reported for directories, but the size of a directory is always zero.

Examples

- This example creates an array to hold the names of all .txt files in the current DEFAULT directory, then uses AEVAL() to list them to the console:

```
LOCAL aFiles[ADIR("*.TXT")]
ADIR("*.TXT", aFiles)
AEVAL(aFiles, { |element| QOUT(element) })
```

Files Library is EXTEND.LIB.

See Also ACHOICE(), AEVAL(), ASCAN(), ASORT(), DIRECTORY(), LEN()

AEVAL() function

Execute a code block for each element in an array

Syntax

```
AEVAL(<aArray>, <bBlock>,
   [<nStart>], [<nCount>]) → aArray
```

Arguments

<aArray> is the array to traverse.

<bBlock> is a code block to execute for each element encountered.

<nStart> is the starting element. If not specified, the default is element one.

<nCount> is the number of elements to process from *<nStart>*. If not specified, the default is all elements to the end of the array.

Returns

AEVAL() returns a reference to *<aArray>*.

Description

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

AEVAL() makes no assumptions about the contents of the array elements it is passing to the block. It is assumed that the supplied block knows what type of data will be in each element.

AEVAL() is similar to DBEVAL() which applies a block to each record of a database file. Like DBEVAL(), AEVAL() can be used as a primitive for the construction of iteration commands for both simple and complex array structures.

Refer to the Code Blocks section in the "Basic Concepts" chapter of the *Programming and Utilities Guide* for more information on the theory and syntax of code blocks.

Examples

- This example uses AEVAL() to display an array of file names and file sizes returned from the DIRECTORY() function:

```
#include "Directry.ch"
//
LOCAL aFiles := DIRECTORY("*.dbf"), nTotal := 0
AEVAL(aFiles,;
    { | aDbfFile |;
       QOUT(PADR(aDbfFile[F_NAME], 10), aDbfFile[F_SIZE]),;
       nTotal += aDbfFile[F_SIZE]);
    } )
//
?
? "Total Bytes:", nTotal
```

- This example uses AEVAL() to build a list consisting of selected items from a multidimensional array:

```
#include "Directry.ch"
//
LOCAL aFiles := DIRECTORY("*.dbf"), aNames := {}
AEVAL(aFiles,;
    { | file | AADD(aNames, file[F_NAME]) };
    )
```

- This example changes the contents of the array element depending on a condition. Notice the use of the codeblock parameters:

```
LOCAL aArray[6]
AFILL(aArray,"old")
AEVAL(aArray,;
{|cValue,nIndex| IF(cValue == "old",;
                   aArray[nIndex] := "new",)})
```

Files Library is CLIPPER.LIB.

See Also DBEVAL(), EVAL(), QOUT()

AFIELDS()* function

Fill arrays with the structure of the current database file

Syntax

```
AFIELDS([<aFieldNames>], [<aTypes>],
    [<aWidths>], [<aDecimals>]) → nFields
```

Arguments

<aFieldNames> is the array to fill with field names. Each element is a character string.

<aTypes> is the array to fill with the type of fields in *<aFieldNames>*. Each element is a character string.

<aWidths> is the array to fill with the widths of fields in *<aFieldNames>*. Each element is numeric data type.

<aDecimals> is the array to fill with the number of decimals defined for fields in *<aFieldNames>*. Each element is numeric data type. If the field type is not numeric, the *<aDecimals>* element is zero.

Returns

AFIELDS() returns the number of fields or the length of the shortest array argument, whichever is less. If no arguments are specified, or if there is no file in USE in the current work area, AFIELDS() returns zero.

Description

AFIELDS() is an array function that fills a series of arrays (structure attribute arrays) with the structure of the database file currently open, one element in each array per field. AFIELDS() works like ADIR(), filling a series of existing arrays with information. To use AFIELDS(), you must first create the arrays to hold the database structure information, each with the same number of elements as the number of fields (i.e. FCOUNT()). Once the structure attribute arrays are created, you can then invoke AFIELDS() to fill the structure arrays with information about each field.

By default, AFIELDS() operates on the currently selected work area. It can operate on an unselected work area if you specify it within an aliased expression (see example below).

AFIELDS() is a compatibility function and, therefore, is not recommended. It is superseded by DBSTRUCT(), which does not require the existence of any arrays prior to invocation and returns a multidimensional array containing the current database file structure.

Examples

- This example demonstrates how AFIELDS() and ACHOICE() can be used together to create a fields picklist:

```
USE Sales NEW
PRIVATE aFieldNames[FCOUNT()]
AFIELDS(aFieldNames)
@ 1, 0 TO 10, 10 DOUBLE
nChoice := ACHOICE(2, 1, 9, 9, aFieldNames)
@ 12, 0 SAY IF(nChoice != 0, aFieldNames[nChoice],;
                "None selected")
RETURN
```

- This example uses AFIELDS() with an aliased expression to fill arrays with the structure of Sales.dbf, open in an unselected work area:

```
LOCAL aFieldNames, aTypes, aWidths, aDecimals
USE Sales NEW
USE Customer NEW
//
aFieldNames := Sales->(ARRAY(FCOUNT()))
aTypes := Sales->(ARRAY(FCOUNT()))
aWidths := Sales->(ARRAY(FCOUNT()))
aDecimals := Sales->(ARRAY(FCOUNT()))
//
Sales->(AFIELDS(aFieldNames, aTypes, ;
         aWidths, aDecimals))
```

Files

Library is EXTEND.LIB.

See Also

ACHOICE(), ADIR(), AEVAL(), ASCAN(), DBCREATE(), DBSTRUCT(), FCOUNT(), FIELDNAME(), LEN(), TYPE()

AFILL() function

Fill an array with a specified value

Syntax

```
AFILL(<aTarget>, <expValue>,
   [<nStart>], [<nCount>]) → aTarget
```

Arguments

<aTarget> is the array to be filled.

<expValue> is the value to be placed in each array element. It can be an expression of any valid data type.

<nStart> is the position of the first element to be filled. If this argument is omitted, the default value is one.

<nCount> is the number of elements to be filled starting with element *<nStart>*. If this argument is omitted, elements are filled from the starting element position to the end of the array.

Returns

AFILL() returns a reference to *<aTarget>*.

Description

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

Warning! *AFILL() cannot be used to fill multidimensional arrays. CA-Clipper implements multidimensional arrays by nesting arrays within other arrays. Using AFILL() with a multidimensional array will overwrite subarrays used for the other dimensions of the array.*

Examples

- This example, creates a three-element array. The array is then filled with the logical value, (.F.). Finally, elements in positions two and three are assigned the new value of true (.T.):

```
LOCAL aLogic[3]
// Result: aLogic is { NIL, NIL, NIL }

AFILL(aLogic, .F.)
// Result: aLogic is { .F., .F., .F. }

AFILL(aLogic, .T., 2, 2)
// Result: aLogic is { .F., .T., .T. }
```

Files

Library is CLIPPER.LIB.

See Also

AADD(), AEVAL(), DBSTRUCT(), DIRECTORY()

AINS() function

Insert a NIL element into an array

Syntax

```
AINS(<aTarget>, <nPosition>) → aTarget
```

Arguments

<aTarget> is the array into which a new element will be inserted.

<nPosition> is the position at which the new element will be inserted.

Returns

AINS() returns a reference to the target array, *<aTarget>*.

Description

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

Warning! *AINS() must be used carefully with multidimensional arrays. Multidimensional arrays in CA-Clipper are implemented by nesting arrays within other arrays. Using AINS() in a multidimensional array discards the last element in the specified target array which, if it is an array element, will cause one or more dimensions to be lost. To insert a new dimension into an array, first add a new element to the end of the array using AADD() or ASIZE() before using AINS().*

Examples

- This example demonstrates the effect of using AINS() on an array:

```
LOCAL aArray
aArray := { 1, 2, 3 }         // Result: aArray is
                              // now { 1, 2, 3 }
AINS(aArray, 2)               // Result: aArray is
                              // now { 1, NIL, 2 }
```

Files Library is CLIPPER.LIB.

See Also AADD(), ACOPY(), ADEL(), AEVAL(), AFILL(), ASIZE()

ALERT() function

Display a simple modal dialog box

Syntax

```
ALERT( <cMessage>, [<aOptions>] ) → nChoice
```

Arguments

<cMessage> is the message text displayed and centered in the alert box. If the message contains one or more semicolons, the text after the semicolons is centered on succeeding lines in the dialog box.

<aOptions> defines a list of up to 4 possible responses to the dialog box.

Returns

ALERT() returns a numeric value indicating which option was chosen. If the *Esc* key is pressed, the value returned is zero.

Description

The ALERT() function creates a simple modal dialog. It is useful in error handlers and other "pause" functions. The user can respond by moving a highlight bar and pressing the Return or SpaceBar keys, or by pressing the key corresponding to the first letter of the option. If *<aOptions>* is not supplied, a single "OK" option is presented.

ALERT() is sensitive to the presence or absence of the CA-Clipper full-screen I/O system. If the full-screen system is not present, ALERT() uses standard I/O to display the message and options tty-style (i.e., 80-column, without word wrap, each line ended with carriage return/linefeed).

Examples

- This example demonstrates use of an alert dialog box. First, the array of options is defined, the ALERT() function gets the user's selection, and finally, the user's choice is handled with a DO CASE...ENDCASE control structure:

```
#define AL_SAVE      1
#define AL_CANCEL    2
#define AL_CONT      3

// Define an array of options
aOptions := {"Save", "Don't Save", "Continue"}

// Display the dialog box and get the user's selection
nChoice  := ALERT("File has changed...", aOptions)

// Handle the user's request
DO CASE
CASE nChoice == AL_SAVE
    ? "Save"
CASE nChoice == AL_CANCEL
    ? "Don't Save"
CASE nChoice == AL_CONT
    ? "Continue"
OTHERWISE
    ? "Escape"
ENDCASE
//
RETURN
```

Files Library is LLIBG.LIB.

See Also @...PROMPT, MENU TO

ALIAS() function

Return a specified work area alias

Syntax

```
ALIAS([<nWorkArea>]) → cAlias
```

Arguments

<nWorkArea> is any work area number.

Returns

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

Description

ALIAS() is a database function that determines the alias of a specified work area. An alias is the name assigned to a work area when a database file is USEd. The actual name assigned is either the name of the database file, or a name explicitly assigned with the ALIAS clause of the USE command.

ALIAS() is the inverse of the SELECT() function. ALIAS() returns the alias name given the work area number, and SELECT() returns the work area number given the alias name.

- This example returns the name of the previously selected work area:

```
USE File1 NEW ALIAS Test1
nOldArea := SELECT()
USE File2 NEW ALIAS Test2
? ALIAS( nOldArea )          // Returns Test1
```

Files

Library is CLIPPER.LIB.

See Also

SELECT, SELECT(), USE

ALLTRIM() function

Remove leading and trailing spaces from a character string

Syntax

```
ALLTRIM(<cString>) → cTrimString
```

Arguments

<cString> is the character expression to be trimmed.

Returns

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

Description

ALLTRIM() is a character function that removes both leading and trailing spaces from a string. It is related to LTRIM() and RTRIM() which remove leading and trailing spaces, respectively. The inverse of ALLTRIM(), LTRIM(), and RTRIM() are the PADC(), PADL(), and PADR() functions which center, left-justify, or right-justify character strings by padding them with fill characters.

Notes

- ***Space characters:*** The ALLTRIM() function treats carriage returns, line feeds, and tabs as space characters and removes these as well.

Examples

- This example creates a string with both leading and trailing spaces, and then trims them with ALLTRIM():

```
cString := SPACE(10) + "string" + SPACE(10)
? LEN(cString)                    // Result: 26
? LEN(ALLTRIM(cString))           // Result: 6
```

Files Library is EXTEND.LIB.

See Also LTRIM(), PAD(), RTRIM(), TRIM()

ALTD() function

Invoke the CA-Clipper debugger

Syntax

```
ALTD([<nAction>]) → NIL
```

Arguments

<nAction> defines what action ALTD() performs when invoked. The following is a complete list of *<nAction>* values and their actions:

ALTD() Actions

Argument	Action
None	Invokes the debugger if it is enabled
0	Disables Alt+D
1	Enables Alt+D
Other	No action

Returns

ALTD() always returns NIL.

Description

ALTD() performs differently depending on its argument as shown in the table above. For more information on using the debugger, refer to "CA-Clipper Debugger-CLD.LIB" chapter in the *Programming and Utilities Guide*. Also refer to the "Debugging Your Applications" chapter in the *Workbench User Guide.*

Examples

- This example demonstrates a series of manifest constants that can be used as arguments for ALTD() before invoking the debugger programmatically:

```
#define ALTD_DISABLE    0
#define ALTD_ENABLE     1
//
ALTD(ALTD_ENABLE)
```

Files

Library is CLIPPER.LIB.

See Also

SET ESCAPE, SETCANCEL()

A

ANNOUNCE statement

Declare a module identifier

Syntax

```
ANNOUNCE <idModule>
```

Arguments

<idModule> is a module identifier name.

Description

ANNOUNCE is a declaration statement that defines a module identifier. A linker may use this identifier later to satisfy pending module REQUESTs. ANNOUNCE and REQUEST provide a mechanism for managing application modules (.prg files).

Specify ANNOUNCE statements prior to any executable statements in a program file. A source (.prg) file can only have one module identifier; all subsequent ANNOUNCE declarations produce a compiler warning and will be ignored. Module identifiers must be unique and should not duplicate the name of any procedures or user-defined functions in a source (.prg) file.

Examples

- This example illustrates the ANNOUNCE declaration:

  ```
  ANNOUNCE CustomInit

  INIT PROCEDURE MyInit
     ? "Hypothetical Industries, Inc."
     RETURN
  ```

  The above program module, *CustomInit*, should be compiled with the /N option. Subsequently, the program is addressed in the source code of another program module through use of the REQUEST statement,

  ```
  REQUEST CustomInit
  ```

  which causes the module *CustomInit* to be linked into the resultant executable (.EXE) file.

See Also

REQUEST

APPEND BLANK command

Add a new record to the current database file

Syntax

```
APPEND BLANK
```

Description

APPEND BLANK is a database command that adds a new record to the end of the current database file and then makes it the current record. The new field values are initialized to the empty values for each data type: character fields are assigned with spaces; numeric fields are assigned zero; logical fields are assigned false (.F.); date fields are assigned CTOD(""); and memo fields are left empty.

If operating under a network with the current database file shared, APPEND BLANK attempts to add and then lock a new record. If another user has locked the database file with FLOCK() or locked LASTREC() + 1 with RLOCK(), NETERR() returns true (.T.). Note that a newly APPENDed record remains locked until you lock another record or perform an UNLOCK. APPEND BLANK does not release an FLOCK() set by the current user.

Examples

- This example attempts to add a record to a shared database file and uses NETERR() to test whether the operation succeeded:

```
USE Sales SHARED NEW
.
. <statements>
.
APPEND BLANK
IF !NETERR()
    <update empty record>...
ELSE
    ? "Append operation failed"
    BREAK
ENDIF
```

Files

Library is CLIPPER.LIB.

See Also

APPEND FROM, FLOCK(), NETERR(), RLOCK()

APPEND FROM command

Import records from a database (.dbf) file or ASCII text file

Syntax

```
APPEND FROM <xcFile>
   [FIELDS <idField list>]
   [<scope>] [WHILE <lCondition>] [FOR <lCondition>]
   [SDF | DELIMITED [WITH BLANK | <xcDelimiter>] |
   [VIA <xcDriver>]]
```

Arguments

FROM <xcFile> specifies the name of the source file. You can specify *<xcFile>* either as a literal file name or as a character expression enclosed in parentheses. If a file extension is not specified, .dbf is the default input file type. If SDF or DELIMITED is specified, the file extension is assumed to be .txt unless otherwise specified.

FIELDS <idField list> specifies the list of fields to copy from *<xcFile>*. The default is all fields.

<scope> is the portion of the source database file to APPEND FROM. NEXT *<n>* APPENDs the first *<n>* records. RECORD *<n>* APPENDs only record number *<n>* from *<xcFile>*. The default scope is ALL records in *<xcFile>*.

WHILE <lCondition> specifies the set of records meeting the condition from the first record in the source file until the condition fails.

FOR <lCondition> specifies the conditional set of records to APPEND FROM within the given scope.

SDF identifies a System Data Format ASCII file. Records and fields are fixed length.

DELIMITED identifies an ASCII text file where character fields are enclosed in double quotation marks (the default delimiter). Note that delimiters are not required and CA-Clipper correctly APPENDs character fields not enclosed in them. Fields and records are variable length.

DELIMITED WITH BLANK identifies an ASCII text file in which fields are separated by one space and character fields are not enclosed in delimiters.

DELIMITED WITH <xcDelimiter> identifies a delimited ASCII text file where character fields are enclosed using the specified delimiter. You can specify *<xcDelimiter>* as a literal character or as a character expression enclosed in parentheses.

See the tables below for more information regarding the format specification requirements for ASCII text files that you want to APPEND using these arguments.

VIA <xcDriver> specifies the replaceable database driver (RDD) to use to import the desired data. *<cDriver>* is the name of the RDD specified as a character expression. If *<cDriver>* is specified as a literal value, it must be enclosed in quotes.

If the VIA clause is omitted, APPEND FROM uses the driver in the current work area. If you specify the VIA clause, you must REQUEST the appropriate RDDs to be linked into the application.

Warning! *If the DELIMITED WITH clause is specified on a COPY TO or APPEND FROM command line, it must be the last clause specified.*

Description

APPEND FROM adds records to the current database file from an ASCII text file or another database file. Only fields with the same names and types are APPENDed. Fields with the same name from both the current database file and *<xcFile>* must be the same data type. If they are not, a runtime error occurs when the APPEND FROM command is invoked.

Any date information in *<xcFile>* must be in the format yyyymmdd to be properly APPENDed.

In a network environment, APPEND FROM does not require that the current database file be USEed EXCLUSIVEly or locked with FLOCK() to perform its operation. As each record is added, CA-Clipper automatically arbitrates contention for the new record.

When you invoke APPEND FROM, CA-Clipper attempts to open *<xcFile>* as shared and read-only. If access is denied, APPEND FROM terminates with a runtime error. Refer to the "Network Programming" chapter in the *Programming and Utilities Guide* for more information. No error is raised if you attempt to open a .dbf file that is already open.

This table shows the format specifications for SDF text files:

SDF Text File Format Specifications

File Element	Format
Character fields	Padded with trailing blanks
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Padded with leading blanks or zeros
Field separator	None
Record separator	Carriage return/linefeed
End of file marker	1A hex or CHR(26)

This table shows the format specifications for DELIMITED and DELIMITED WITH <*xcDelimiter*> ASCII text files:

DELIMITED Text File Format Specifications

File Element	Format
Character fields	May be delimited, with trailing blanks truncated
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Leading zeros may be truncated
Field separator	Comma
Record separator	Carriage return/linefeed
End of file marker	1A hex or CHR(26)

This table shows the format specifications for DELIMITED WITH BLANK ASCII text files:

DELIMITED WITH BLANK Text File Format Specifications

File Element	Format
Character fields	Not delimited, trailing blanks may be truncated
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Leading zeros may be truncated
Field separator	Single blank space
Record separator	Carriage return/linefeed
End of file marker	1A hex or CHR(26)

Notes

- ***Deleted records:*** If DELETED is OFF, deleted records in <*xcFile*> are APPENDed to the current database file and retain their deleted status. If DELETED is ON, however, none of the deleted <*xcFile*> records are APPENDed.

- ***Unmatched field widths:*** If a field in the current database file is a character type and has a field length greater than the incoming <*xcFile*> data, CA-Clipper pads the <*xcFile*> data with blanks. If the current field is a character data type and its field length is less than the incoming <*xcFile*> data, the <*xcFile*> data is truncated to fit. If the current field is a numeric type and the incoming <*xcFile*> data has more digits than the current field length, a runtime error occurs.

Examples

- This example demonstrates an APPEND FROM command using a FIELDS list and a FOR condition:

```
USE Sales NEW
APPEND FROM BranchFile FIELDS Branch, Salesman, Amount;
   FOR Branch = 100
```

- This example demonstrates how a <*scope*> can be specified to import a particular record from another database file:

```
APPEND RECORD 5 FROM Temp
```

Files Library is CLIPPER.LIB.

See Also COPY TO

ARRAY() function

Create an uninitialized array of specified length

Syntax

```
ARRAY(<nElements> [, <nElements>...]) aArray
```

Arguments

<nElements> is the number of elements in the specified dimension. The maximum number of elements in a dimension is 4096. Arrays in CA-Clipper can have an unlimited number of dimensions.

Returns

ARRAY() returns an array of specified dimensions.

Description

ARRAY() is an array function that returns an uninitialized array with the specified number of elements and dimensions. If more than one *<nElements>* argument is specified, a multidimensional array is created with the number of dimensions equal to the number of *<nElements>* arguments specified. Any *<nElements>* that is itself an array creates a nested array.

In CA-Clipper, there are several ways to create an array. You can declare an array using a declaration statement such as LOCAL or STATIC; you can create an array using a PRIVATE or PUBLIC statement; you can assign a literal array to an existing variable; or you can use the ARRAY() function. ARRAY() has the advantage that it can create arrays within expressions or code blocks.

Examples

- This example creates a one-dimensional array of five elements using the ARRAY() function, and then shows the equivalent action by assigning a literal array of NIL values:

```
aArray := ARRAY(5)
aArray := { NIL, NIL, NIL, NIL, NIL }
```

- This example shows three different statements which create the same multidimensional array:

```
aArray := ARRAY(3, 2)
aArray := { {NIL, NIL}, {NIL, NIL}, {NIL, NIL} }
aArray := { ARRAY(2), ARRAY(2), ARRAY(2) }
```

- This example creates a nested, multidimensional array:

```
aArray := ARRAY(3, {NIL,NIL})
```

Files

Library is CLIPPER.LIB.

See Also

AADD(), ACLONE(), ACOPY(), ADEL(), AEVAL(), AFILL(), AINS(), ASCAN(), ASIZE(), ASORT(), LOCAL, PRIVATE, PUBLIC, STATIC

ASC() function

Convert a character to its ASCII value

Syntax

```
ASC(<cExp>) → nCode
```

Arguments

<cExp> is the character expression to be converted to a number.

Returns

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

Description

ASC() is a character conversion function that returns the ASCII value of the leftmost character in a character string. ASC() is used primarily on expressions requiring numeric calculations on the ASCII value of a character. CHR() and ASC() are inverse functions.

Examples

- These examples illustrate various results of ASC():

```
? ASC("A")                    // Result: 65
? ASC("Apple")                // Result: 65
? ASC("a")                    // Result: 97
? ASC("Z") - ASC("A")         // Result: 25
? ASC("")                     // Result: 0
```

Files Library is CLIPPER.LIB.

See Also CHR(), INKEY(), STR(), VAL()

ASCAN() function

Scan an array for a value or until a block returns true (.T.)

Syntax

```
ASCAN(<aTarget>, <expSearch>,
   [<nStart>], [<nCount>]) → nStoppedAt
```

Arguments

<*aTarget*> is the array to be scanned.

<*expSearch*> is either a simple value to scan for, or a code block. If <*expSearch*> is a simple value it can be character, date, logical, or numeric type.

<*nStart*> is the starting element of the scan. If this argument is not specified, the default starting position is one.

<*nCount*> is the number of elements to scan from the starting position. If this argument is not specified, all elements from the starting element to the end of the array are scanned.

Returns

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

Description

ASCAN() is an array function that scans an array for a specified value and operates like SEEK when searching for a simple value. The <*expSearch*> value is compared to the target array element beginning with the leftmost character in the target element and proceeding until there are no more characters left in <*expSearch*>. If there is no match, ASCAN() proceeds to the next element in the array.

Since ASCAN() uses the equal operator (=) for comparisons, it is sensitive to the status of EXACT. If EXACT is ON, the target array element must be exactly equal to the result of <*expSearch*> to match.

If the <*expSearch*> argument is a code block, ASCAN() scans the <*aTarget*> array executing the block for each element accessed. As each element is encountered, ASCAN() passes the element's value as an argument to the code block, and then performs an EVAL() on the block. The scanning operation stops when the code block returns true (.T.), or ASCAN() reaches the last element in the array.

Examples

- This example demonstrates scanning a three-element array using simple values and a code block as search criteria. The code block criteria shows how to perform a case-insensitive search:

```
aArray := { "Tom", "Mary", "Sue" }
? ASCAN(aArray, "Mary")              // Result: 2
? ASCAN(aArray, "mary")              // Result: 0
//
? ASCAN(aArray, { |x| UPPER(x) ;
      == "MARY" })                   // Result: 2
```

- This example demonstrates scanning for multiple instances of a search argument after a match is found:

```
LOCAL aArray := { "Tom", "Mary", "Sue",;
                   "Mary" }, nStart := 1
//
// Get last array element position
nAtEnd := LEN(aArray)
DO WHILE (nPos := ASCAN(aArray, "Mary", ;
             nStart)) > 0
   ? nPos, aArray[nPos]
   //
   // Get new starting position and test
   // boundary condition
   IF (nStart := ++nPos) > nAtEnd
      EXIT
   ENDIF
ENDDO
```

- This example scans a two-dimensional array using a code block. Note that the parameter *aVal* in the code block is an array:

```
LOCAL aArr:={}
CLS
AADD(aArr,{"one","two"})
AADD(aArr,{"three","four"})
AADD(aArr,{"five","six"})
? ASCAN(aArr, {|aVal| aVal[2] == "four"})      // Returns 2
```

Files Library is CLIPPER.LIB.

See Also AEVAL(), EVAL()

ASIZE() function

Grow or shrink an array

Syntax

```
ASIZE(<aTarget>, <nLength>) → aTarget
```

Arguments

<aTarget> is the array to grow or shrink.

<nLength> is the new size of the array.

Returns

ASIZE() returns a reference to the target array, *<aTarget>*.

Description

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

ASIZE() is similar to AADD() which adds a single new element to the end of an array and optionally assigns a new value at the same time. Note that ASIZE() is different from AINS() and ADEL(), which do not actually change the array's length.

Note: ASIZE() only supports single-dimensional arrays.

Examples

- These examples demonstrate adding new elements and deleting existing elements:

```
aArray := { 1 }          // Result: aArray is { 1 }
ASIZE(aArray, 3)         // Result: aArray is { 1, NIL, NIL }
ASIZE(aArray, 1)         // Result: aArray is { 1 }
```

Files Library is CLIPPER.LIB.

See Also AADD(), ADEL(), AFILL(), AINS()

ASORT() function

Sort an array

Syntax

```
ASORT(<aTarget>, [<nStart>],
   [<nCount>], [<bOrder>]) → aTarget
```

Arguments

<*aTarget*> is the array to be sorted.

<*nStart*> is the first element of the sort. If not specified, the default starting position is one.

<*nCount*> is the number of elements to be sorted. If not specified, all elements in the array beginning with the starting element are sorted.

<*bOrder*> is an optional code block used to determine sorting order. If not specified, the default order is ascending.

Returns

ASORT() returns a reference to the <*aTarget*> array.

Description

ASORT() is an array function that sorts all or part of an array containing elements of a single data type. Data types that can be sorted include character, date, logical, and numeric.

If the <*bOrder*> argument is not specified, the default order is ascending. Elements with low values are sorted toward the top of the array (first element), while elements with high values are sorted toward the bottom of the array (last element).

If the <*bOrder*> block argument is specified, it is used to determine the sorting order. Each time the block is evaluated, two elements from the target array are passed as block parameters. The block must return true (.T.) if the elements are in sorted order. This facility can be used to create a descending or dictionary order sort. See the examples below.

When sorted, character strings are ordered in ASCII sequence; logical values are sorted with false (.F.) as the low value; date values are sorted chronologically; and numeric values are sorted by magnitude.

Notes

- ASORT() is only guaranteed to produce sorted output (as defined by the block), not to preserve any existing natural order in the process.
- Because CA-Clipper implements multidimensional arrays by nesting subarrays within other arrays, ASORT() will not directly sort a multidimensional array. To sort a nested array, you must supply a code block which properly handles the subarrays.

Examples

- This example creates an array of five unsorted elements, sorts the array in ascending order, then sorts the array in descending order using a code block:

```
aArray := { 3, 5, 1, 2, 4 }
ASORT(aArray)
// Result: { 1, 2, 3, 4, 5 }

ASORT(aArray,,, { |x, y| x > y })
// Result: { 5, 4, 3, 2, 1 }
```

- This example sorts an array of character strings in ascending order, independent of case. It does this by using a code block that converts the elements to uppercase before they are compared:

```
aArray := { "Fred", Kate", "ALVIN", "friend" }
ASORT(aArray,,, { |x, y| UPPER(x) < UPPER(y) })
```

- This example sorts a nested array using the second element of each subarray:

```
aKids := { {"Mary", 14}, {"Joe", 23}, {"Art", 16} }
aSortKids := ASORT(aKids,,, { |x, y| x[2] < y[2] })
```

Result:

```
{ {"Mary", 14}, {"Art", 16}, {"Joe", 23} }
```

Files Library is EXTEND.LIB.

See Also ASCAN(), EVAL(), SORT

AT() function

Return the position of a substring within a character string

Syntax

```
AT(<cSearch>, <cTarget>) → nPosition
```

Arguments

<cSearch> is the character substring to be searched for.

<cTarget> is the character string to be searched.

Returns

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

Description

AT() is a character function used to determine the position of the first occurrence of a character substring within another string. If you only need to know whether a substring exists within another string, use the $ operator. To find the last instance of a substring within a string, use RAT().

Examples

- These examples show typical use of AT():

```
? AT("a", "abcde")                        // Result: 1
? AT("bcd", "abcde")                      // Result: 2
? AT("a", "bcde")                         // Result: 0
```

- This example splits a character string based on the position of a comma within the target string:

```
cTarget := "Langtree, Lilly"
? SUBSTR(cTarget, 1, AT(",", cTarget) - 1)
// Result: Langtree

? SUBSTR(cTarget, AT(",", cTarget) + 2)
// Result: Lilly
```

Files Library is CLIPPER.LIB.

See Also RAT(), STRTRAN(), SUBSTR()

ATAIL() function

Return the highest numbered element of an array

Syntax

```
ATAIL(<aArray>) → Element
```

Arguments

<*aArray*> is the array.

Returns

ATAIL() returns either a value or a reference to an array or object. The array is not changed.

Description

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

Examples

- The following example creates a literal array and returns that last element of the array:

```
aArray := {"a", "b", "c", "d"}
? ATAIL(aArray)                    // Result: d
```

Files

Library is CLIPPER.LIB.

See Also

LEN()

AVERAGE command

Average numeric expressions in the current work area

Syntax

```
AVERAGE <nExp list> TO <idVar list>
   [<scope>] [WHILE <lCondition>] [FOR <lCondition>]
```

Arguments

<nExp list> is a list of the numeric values to AVERAGE for each record processed.

TO <idVar list> identifies a list of receiving variables which will contain the average results. Variables that either do not exist or are not visible are created as private variables. *<idVar list>* must contain the same number of elements as *<nExp list>*.

<scope> defines the portion of the current database file to AVERAGE. The default scope is ALL.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to AVERAGE within the given scope.

Description

AVERAGE calculates the average of one or more numeric expressions to variables for a range of records in the current database file. Zero values are counted in the AVERAGE unless explicitly ruled out with a FOR condition.

Examples

- This example averages a single numeric field using a condition to select a subset of records from the database file:

```
USE Sales NEW
AVERAGE Amount TO nAvgAmount FOR Branch = "100"
```

- This example finds the average date for a range of dates:

```
AVERAGE (SaleDate - CTOD("00/00/00")) ;
   TO nAvgDays FOR !EMPTY(SaleDate)
dAvgDate := CTOD("00/00/00") + nAvgDays
```

Files Library is CLIPPER.LIB.

See Also DBEVAL(), SUM, TOTAL

BEGIN SEQUENCE statement

Define a sequence of statements for a BREAK

Syntax

```
BEGIN SEQUENCE
   <statements>...
[BREAK [<exp>]]
   <statements>...
[RECOVER [USING <idVar>]]
   <statements>...
END [SEQUENCE]
```

Arguments

BREAK <exp> branches execution to the statement immediately following the nearest RECOVER statement if one is specified or the nearest END SEQUENCE statement. *<exp>* is the value returned into the *<idVar>* specified in the USING clause of the RECOVER statement.

RECOVER USING <idVar> defines a recover point in the SEQUENCE construct where control branches after a BREAK statement. If USING *<idVar>* clause is specified, *<idVar>* receives the value returned by the BREAK statement. In general, this is an error object.

END defines the end point of the SEQUENCE control structure. If no RECOVER statement is specified, control branches to the first statement following the END statement after a BREAK.

Description

BEGIN SEQUENCE...END is a control structure used for exception and runtime error handling. It delimits a block of statements, including invoked procedures and user-defined functions. When a BREAK is encountered anywhere in a block of statements following the BEGIN SEQUENCE statement up to the corresponding RECOVER statement, control branches to the program statement immediately following the RECOVER statement. If a RECOVER statement is not specified, control branches to the statement following the END statement, terminating the SEQUENCE. If control reaches a RECOVER statement without encountering a BREAK, it branches to the statement following the corresponding END.

The RECOVER statement optionally receives a parameter passed by a BREAK statement that is specified with a return value. This is usually an error object, generated and returned by the current error handling block defined by ERRORBLOCK(). If an error object is returned, it can be sent messages to query information about the error. With this information, a runtime error can be handled within the context of the operation rather than in the current runtime error handler. See the example below.

Within a SEQUENCE construct there are some restrictions on what statements are allowed between the BEGIN SEQUENCE and RECOVER statements. You cannot RETURN, LOOP, or EXIT between a BEGIN SEQUENCE and RECOVER statement. From within the RECOVER statement block, however, you can LOOP, EXIT, BREAK, or RETURN since the SEQUENCE is essentially completed at that point. Using LOOP from within the RECOVER statement block is useful for re-executing the SEQUENCE statement block. See the example below.

SEQUENCE constructs are quite flexible. They can be nested and more than one can be defined in the same procedure or user-defined function. If more than one SEQUENCE construct is specified, each SEQUENCE should delimit one discrete operation.

For more information on Error objects, refer to the Error class in this chapter.

Examples

- This code fragment demonstrates a SEQUENCE construct in which the BREAK occurs within the current procedure:

```
BEGIN SEQUENCE
   <statements>...
   IF lBreakCond
      BREAK
   ENDIF
RECOVER
   <recovery statements>...
END

<recovery statements>...
```

- This example demonstrates an error handler returning an Error object to the variable specified in the USING clause of the RECOVER statement:

```
LOCAL objLocal, bLastHandler
//
// Save current and set new error handler
bLastHandler := ERRORBLOCK({ |objErr| ;
      MyHandler(objErr, .T.) })
//
BEGIN SEQUENCE
   .
   . <operation that might fail>
   .
RECOVER USING objLocal
   //
   // Send messages to objLocal and handle the error
   ? "Error: "
   IF objLocal:genCode != 0
      ?? objLocal:description
   ENDIF
   .
   .
   .
END
//
// Restore previous error handler
ERRORBLOCK( bLastHandler )

FUNCTION MyHandler( objError, lLocalHandler )
   //
   // Handle locally returning the error object
   IF lLocalHandler
      BREAK objError
   ENDIF
   .
   . <other statements to handle the error>
   .
   RETURN NIL
```

- This example re-executes a SEQUENCE statement block by LOOPing from within the RECOVER statement block:

```
DO WHILE .T.
   BEGIN SEQUENCE
      .
      . <operation that may fail>
      .
   RECOVER
      IF PrintRecover()
         LOOP     // Repeat the SEQUENCE statement block
      ENDIF
   END
   EXIT           // Escape from the operation
ENDDO
```

See Also Error class, ERRORBLOCK(), RETURN

BIN2I() function

Convert a 16-bit signed integer to a numeric value

Syntax

```
BIN2I(<cSignedInt>) → nNumber
```

Arguments

<cSignedInt> is a character string in the form of a 16-bit signed integer number—least significant byte first. Only the first two characters are used by the function; all others are ignored.

Returns

BIN2I() returns an integer numeric value.

Description

BIN2I() is a low-level file function that is used with FREAD() to convert a two-byte character string formatted as a signed integer to a CA-Clipper numeric data type. This is most useful when you are reading foreign file types and want to read numeric data in its native format.

Examples

- This example opens a database file using low-level file functions and reads the date of last update (bytes 1-3). The result is the same as with LUPDATE():

```
#include "Fileio.ch"
//
nHandle := FOPEN("Sales.dbf", FO_READ)
//
// Point to byte 1 in the file
FSEEK(nHandle, 1, FS_SET)
//
// Read date of last update
nYear := BIN2I(FREADSTR(nHandle, 1) + CHR(0))
nMonth := BIN2I(FREADSTR(nHandle, 1) + CHR(0))
nDay := BIN2I(FREADSTR(nHandle, 1) + CHR(0))
//
? LTRIM(STR(nMonth)), LTRIM(STR(nDay)), LTRIM(STR(nYear))
FCLOSE(nHandle)
```

Files

Library is EXTEND.LIB, source file is SOURCE\SAMPLE\EXAMPLEA.ASM

See Also

BIN2L(), BIN2W(), FREAD(), FREADSTR(), I2BIN(), L2BIN()

BIN2L() function

Convert a 32-bit signed integer to a numeric value

Syntax

BIN2L(<*cSignedInt*>) → *nNumber*

Arguments

<*cSignedInt*> is a character string in the form of a 32-bit signed integer number—least significant byte first. Only the first four characters are used by the function; all others are ignored.

Returns

BIN2L() returns an integer numeric value.

Description

BIN2L() is a low-level file function that is used with FREAD() to convert a four-byte character string formatted as a signed integer to a CA-Clipper numeric data type. This is most useful when you are reading foreign file types and want to read numeric data in its native format.

Examples

- This example opens a database file using low-level file functions and reads the number of records (bytes 4-7). The result is the same as with LASTREC():

```
#include "Fileio.ch"
//
nHandle := FOPEN("Sales.dbf", FO_READ)
// Note:  Sales.dbf contains 84 records
//
// Point to byte 4
FSEEK(nHandle, 4, FS_SET)
//
// Read the number of records
cRecords := SPACE(4)
FREAD(nHandle, @cRecords, 4)
//
? LTRIM(STR(BIN2L(cRecords)))       // Result: 84
FCLOSE(nHandle)
```

Files

Library is EXTEND.LIB, source file is SOURCE\SAMPLE\EXAMPLEA.ASM

See Also

BIN2I(), BIN2W(), FREAD(), FREADSTR(), I2BIN(), L2BIN()

BIN2W() function

Convert a 16-bit unsigned integer to a numeric value

Syntax

```
BIN2W(<cUnsignedInt>) → nNumber
```

Arguments

<cUnsignedInt> is a character string in the form of a 16-bit unsigned integer number—least significant byte first. Only the first two characters are used by the function; all others are ignored.

Returns

BIN2W() returns an integer numeric value.

Description

BIN2W() is a low-level file function that is used with FREAD() to convert a two-byte character string formatted as an unsigned integer to a CA-Clipper numeric data type. This is most useful when you are reading from a binary file and want to read data in its native format.

Examples

- This example opens a database file using low-level file functions and reads the number of bytes per record (bytes 10-11). The result is the same as with RECSIZE():

```
#include "Fileio.ch"
//
nHandle := FOPEN("Sales.dbf", FO_READ)
// Note:  The length of a record in Sales.dbf is 124
//
// Point to byte 10, the first record size byte
FSEEK(nHandle, 10, FS_SET)
//
// Read record size
cRecSize := SPACE(2)
FREAD(nHandle, @cRecSize, 2)
//
? LTRIM(STR(BIN2W(cRecSize)))          // Result: 124
FCLOSE(nHandle)
```

Files

Library is EXTEND.LIB, source file is SOURCE\SAMPLE\EXAMPLEA.ASM

See Also

BIN2I(), BIN2L(), FREAD(), FREADSTR(), I2BIN(), L2BIN()

BLOBDIRECTEXPORT() function

Export the contents of a binary large object (BLOB) pointer to a file

Note: A BLOB file (.dbv or .fpt) is used for storing memo field information as an alternative to the standard .dbt file mechanism supported by some RDDs. It is a more powerful and efficient mechanism for storing and retrieving large amounts of data than using .dbt files. CA-Clipper supplies the DBFCDX driver, which uses the BLOB file storage mechanism by default, and the DBFMEMO driver, which you can use as a driver with a named super RDD (see MEMOSETSUPER()). Refer to the RDD Features in the "Replaceable Database Driver Architecture" chapter of the *Drivers Guide* for further information on using this driver.

Syntax

```
BLOBDIRECTEXPORT(<nPointer>, <cTargetFile>,
   [<nMode>])    → lSuccess
```

Arguments

<nPointer> is a pointer to the BLOB data. This pointer can be obtained using BLOBDIRECTPUT(), BLOBDIRECTEXPORT(), or DBFIELDINFO(DBS_BLOB_POINTER, *<nFieldPos>*).

<cTargetFile> is the name of the target file where the BLOB data will be written, including an optional drive, directory, and extension. See SETDEFAULT() and SETPATH() for file searching and creation rules. No default extension is assumed.

If *<cTargetFile>* does not exist, it is created. If it exists, this function attempts to open the file in exclusive mode and, if successful, the file is written to without warning or error. If access is denied because another process is using the file, for example, NETERR() is set to true (.T.).

<nMode> is a constant defining the copy mode, as shown in the table below:

Copy Mode Constants

Constant	Description
BLOB_EXPORT_APPEND	Appends to the file
BLOB_EXPORT_OVERWRITE	Overwrites the file (this is the default)

Returns

BLOBDIRECTEXPORT() returns true (.T.) if successful; otherwise, it returns false (.F.).

Description

By default, BLOBDIRECTEXPORT() operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression.

Examples

- This example extracts an array of pointers from the BLOB file's root area, then uses one of the pointers to export a picture to a file:

```
FUNCTION PUTPIX()
   LOCAL cPixFile
   LOCAL nPointer
   LOCAL aBLOBPtrs

   cPixFile := "Picture.gif"

   // Customer database with a picture of
   // each customer stored in a field called Pix
   USE Customer NEW VIA "DBFCDX"

   // Assumes that the program previously
   // stored an array of direct BLOB pointers
   // into the root area of the BLOB file.
   // The picture that we want is assumed to
   // be the second array element.
   aBLOBPtrs := BLOBROOTGET()
   nPointer := aBLOBPtrs[2]

   // Export picture pointed to by nPointer to a file
   IF !BLOBDirectExport(nPointer, cPixFile, ;
      BLOB_EXPORT_OVERWRITE)
         Alert("Export of picture " + cPixFile + ";
            failed!")
   ELSE
      // Code for displaying picture would go here
   ENDIF
```

Files

Library is CLIPPER.LIB.

See Also

BLOBDIRECTIMPORT(), BLOBEXPORT()

BLOBDIRECTGET() function

Retrieve data stored in a BLOB file without referencing a specific field

Syntax

```
BLOBDIRECTGET(<nPointer>, [<nStart>], [<nCount>])
    → expBLOB
```

Arguments

<nPointer> is a pointer to the BLOB data. This pointer can be obtained using BLOBDIRECTPUT(), BLOBDIRECTIMPORT(), or DBFIELDINFO(DBS_BLOB_POINTER, <*nFieldPos*>).

<nStart> is the starting position in <*nPointer*>. If <*nStart*> is positive, the starting position is relative to the leftmost character in <*nPointer*>. If <*nStart*> is negative, it is relative to the rightmost character in <*nPointer*>. If <*nStart*> is omitted, it is assumed to be 1.

<nCount> is the number of bytes of data to retrieve beginning at <*nStart*>. If <*nCount*> is larger than the amount of data stored, excess data is ignored. If omitted, BLOBDIRECTGET() retrieves to the end of the data.

Note: <*nStart*> and <*nCount*> apply to string data only. They are ignored for any other data types.

Returns

BLOBDIRECTGET() returns the data retrieved from the BLOB file. The data type of the return value depends on the actual data stored. Use VALTYPE() to determine the data type.

Description

BLOBDIRECTGET() retrieves data stored in a BLOB file without the need to reference a particular field in the database file. It is particularly useful when accessing data that is larger than 64 KB (such as memo fields created with the BLOBIMPORT() function).

By default, this function operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression.

Examples

- This example illustrates storing setup information in a BLOB file, then selectively retrieving the stored information:

```
FUNCTION PutSettings(aColors,;
               aPaths, ;
               aPassWords)

   LOCAL aSettings

   RDDSETDEFAULT ( "DBFCDX" )
   MEMOSETSUPER ( "DBFCDX" )

   USE Setup NEW via "DBFMEMO"

   aSettings := {}
   AADD(aSettings, BLOBDIRECTPUT(0, aColors))
   AADD(aSettings, BLOBDIRECTPUT(0, aPaths))
   AADD(aSettings, BLOBDIRECTPUT(0, aPassWords))

   BLOBROOTPUT(aSettings)

   CLOSE

FUNCTION GETCOLORS()
   LOCAL aSettings
   LOCAL aColors

   USE Setup NEW VIA "DBFMEMO"

   aSettings := BLOBROOTGET()
   aColors := BLOBDIRECTGET(aSettings[1])

   CLOSE

   RETURN aColors
```

Files

Library is CLIPPER.LIB.

See Also

BLOBDIRECTPUT(), BLOBGET(), BLOBIMPORT(), DBFIELDINFO(), VALTYPE()

BLOBDIRECTIMPORT() function

Import a file into a BLOB file and return a pointer to the data

Syntax

```
BLOBDIRECTIMPORT(<nOldPointer>, <cSourceFile>)
    → nNewPointer
```

Arguments

<nOldPointer> is a pointer to the BLOB data which will be released after the import. This pointer can be obtained using BLOBDIRECTPUT(), BLOBDIRECTIMPORT(), or DBFIELDINFO(DBS_BLOB_POINTER, *<nFieldPos>*). Passing zero (0) disables the release of data.

Important! *If specified, BLOBDIRECTIMPORT() releases the space associated with <nOldPointer> for reuse by other data. Therefore, it is illegal to use <nOldPointer> with any of the BLOB functions after passing it as an argument to this function. Use the function's return value to refer to the newly stored data.*

<cSourceFile> is the name of the file from which to read the BLOB data, including an optional drive, directory, and extension. See SETDEFAULT() and SETPATH() for file searching and creation rules. No default extension is assumed.

This function attempts to open *<cSourceFile>* in shared mode. If the file does not exist, a runtime error is raised. If the file is successfully opened, the operation proceeds. If access is denied because another process has exclusive use of the file, for example, NETERR() is set to true (.T.).

Note: There are no restrictions on the size of *<cSourceFile>* except that you must have enough disk space to make the copy.

Returns

BLOBDIRECTIMPORT() returns a numeric pointer to the BLOB image stored in <*cSourceFile*>.

Description

BLOBDIRECTIMPORT() provides a mechanism for copying the contents of a file into a BLOB file. By default, this function operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression.

BLOBDIRECTIMPORT() is used in conjunction with BLOBDIRECTEXPORT() to transfer data back and forth between external files and BLOB files. You can use BLOBDIRECTIMPORT() with a variety of file types, including graphic images, word processor files, and printer fonts. These two functions are excellent for creating databases for documents, graphics, sounds, etc.

Important! *After importing a file with BLOBDIRECTIMPORT(), nNewPointer, the return value, is the only way to access the data from the BLOB file. It is up to you to provide permanent storage for this reference (see example below).*

Note: DBFIELDINFO(DBS_BLOB_TYPE, <*nFieldPos*>) will return "C" (string) for any memo field created using BLOBDIRECTIMPORT().

Examples

- This example imports a bitmap (.bmp) file to be part of an array of startup data. The data, stored in the root area of the BLOB file, could then be used to display the application's startup screen:

```
FUNCTION PUTPIX()
   LOCAL cBMPFile
   LOCAL aSettings

   cBMPFile := "Logo.bmp"
   aSettings := {}

   // Customer database where startup parameters
   // are stored for convenience
   USE Customer NEW VIA "DBFMEMO"

   // Get default path settings
   AADD(aSettings, STARTPATHS())

   // Get default color settings
   AADD(aSettings, DEFAULTCOLORS())

   // Get company logo for display at startup.
   // There is nothing to free because this
   // is the first time importing.
   nPointer := BLOBDIRECTIMPORT(0, cBMPFile)
   AADD(aSettings, nPointer)

   // Store the settings in the root area of
   // the customer.fpt file

   BLOBROOTPUT(aSettings)
```

Files Library is CLIPPER.LIB.

See Also BLOBDIRECTEXPORT(), BLOBIMPORT()

BLOBDIRECTPUT() function

Put data in a BLOB file without referencing a specific field

Syntax

```
BLOBDIRECTPUT(<nOldPointer>, <uBLOB>) → nNewPointer
```

Arguments

<nOldPointer> is a reference to previously stored BLOB data. This reference can be obtained using BLOBDIRECTPUT(), BLOBDIRECTIMPORT(), or DBFIELDINFO(DBS_BLOB_POINTER, *<nFieldPos>*). If other than zero (0), the data referenced by *<nOldPointer>* is replaced by *<uBLOB>*; otherwise, *<uBLOB>* is added to the current contents of the BLOB file.

Important! *If specified, BLOBDIRECTPUT() releases the space associated with <nOldPointer> for reuse by other data. Therefore, it is illegal to use <nOldPointer> with any of the BLOB functions after passing it as an argument to this function. Use the function's return value to refer to the newly stored data.*

<uBLOB> is the data you want to put into the BLOB file. *<uBLOB>* can be any CA-Clipper data type except code block or an object.

Returns

BLOBDIRECTPUT() returns a numeric pointer to the *<uBLOB>* data.

Description

BLOBDIRECTPUT() stores variable length BLOB data without creating a link with a particular memo field in a database file. After adding data to a BLOB file using BLOBDIRECTPUT(), you should store the function's return value, as this is the only way to access the data from the BLOB file. It is up to you, the developer, to provide permanent storage for this reference (see BLOBROOTPUT()).

By default, this function operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression.

Examples

- This example illustrates storing setup information in a BLOB file, then selectively retrieving the stored information:

```
FUNCTION PutSettings(aColors,;
   aPaths, aPassWords)

   LOCAL aSettings

   USE Setup NEW VIA "DBFMEMO"

   aSettings := {}
   AADD(aSettings, BLOBDIRECTPUT(0, aColors))
   AADD(aSettings, BLOBDIRECTPUT(0, aPaths))
   AADD(aSettings, BLOBDIRECTPUT(0, aPassWords))

   BLOBROOTPUT(aSettings)

   CLOSE

FUNCTION GETCOLORS()
   LOCAL aSettings
   LOCAL aColors

   USE Setup NEW VIA "DBFMEMO"

   aSettings := BLOBROOTGET()
   aColors := BLOBDIRECTGET(aSettings[1])

   CLOSE

   RETURN aColors
```

Files

Library is CLIPPER.LIB.

See Also

BLOBDIRECTGET(), BLOBEXPORT(), BLOBGET(), BLOBIMPORT(), BLOBROOTPUT(), DBFIELDINFO(), FIELDGET()

BLOBEXPORT() function

Copy the contents of a BLOB, identified by its memo field number, to a file

Syntax

```
BLOBEXPORT(<nFieldPos>, <cTargetFile>, [<nMode>])
    → lSuccess
```

Arguments

<nFieldPos> is the position of the field in the database file structure.

<cTargetFile> is the name of the target file where the BLOB data will be written, including an optional drive, directory, and extension. See SETDEFAULT() and SETPATH() for file searching and creation rules. No default extension is assumed.

If *<cTargetFile>* does not exist, it is created. If it exists, this function attempts to open the file in exclusive mode and, if successful, the file is written to without warning or error. If access is denied because another process is using the file, for example, NETERR() is set to true (.T.).

<nMode> is a constant defining the copy mode, as shown in the table below:

Field Information Type Constants

Constant	Description
BLOB_EXPORT_APPEND	Number of decimal places for the field
BLOB_EXPORT_OVERWRITE	Length of the field

Returns

BLOBEXPORT() returns true (.T.) if successful; otherwise, it returns false (.F.).

Description

By default, this function operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression.

Examples

- This example exports the contents of a field that stores a picture to a graphic interchange format (.gif) file, so that the file can be programmatically displayed:

```
FUNCTION SHOWPIX()
   LOCAL cPixFile := "Picture.gif"
   LOCAL nPos

   // Customer database with a picture of each
   // customer stored in a field called Pix
   USE Customer NEW VIA "DBFCDX"
   nPos := FIELDPOS("Pix")

   // Export the BLOB file's data
   // for the current Pix field
   IF !BLOBEXPORT(nPos, cPixFile,;
      BLOB_EXPORT_OVERWRITE)
         ALERT("Export of picture " + cPixFile + ";
            failed!")
   ELSE
      // Code for displaying picture would go here
   ENDIF
```

Files

Library is CLIPPER.LIB.

See Also

BLOBDIRECTEXPORT(), BLOBIMPORT()

BLOBGET() function

Get the contents of a BLOB, identified by its memo field number

Syntax

```
BLOBGET(<nFieldPos>, [<nStart>], [<nCount>]) → uBLOB
```

Arguments

<nFieldPos> is the position of the field in the database file structure.

<nStart> is the starting position in the memo field of the BLOB data. If *<nStart>* is positive, the starting position is relative to the leftmost character in *<nFieldPos>*. If *<nStart>* is negative, it is relative to the rightmost character in *<nFieldPos>*. If *<nStart>* is omitted, it is assumed to be 1.

<nCount> is the number of bytes of data to retrieve beginning at *<nStart>*. If *<nCount>* is larger than the amount of data stored, excess data is ignored. If omitted, BLOBGET() retrieves to the end of the data.

Note: *<nStart>* and *<nCount>* apply to string data only. They are ignored for any other data types.

Returns

BLOBGET() returns the BLOB data retrieved from the memo field. The data type of the return value depends on the actual data stored. Use VALTYPE() to determine the data type. If the indicated field is not a memo field, BLOBGET() returns NIL.

Description

BLOBGET() is very similar to FIELDGET(). However, because string type variables cannot be larger than 64 KB, FIELDGET() will raise a runtime error when attempting to retrieve memo fields of this magnitude or greater.

BLOBGET() will also raise an error if you attempt to retrieve a field greater than this magnitude; however, you can retrieve any subset of the BLOB data by using an *<nCount>* less than 64 KB.

Note: BLOB data less than 64 KB can be retrieved from a memo field using standard means (for example, referring to the field by name in an expression or using the FIELDGET() function).

By default, this function operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression.

Examples

- This example imports information from a word processing document into a field, then uses BLOBGET() to extract the first 25 characters of the field:

```
FUNCTION GETFIRST25()
    LOCAL nPos
    LOCAL cStr

    USE customer NEW VIA "DBFCDX"

    // Field that contains word processor
    // documentation
    nPos := FieldPos("WP_DOC")

       // Import a file (can be larger than 64 KB), then
    // obtain the first 25 characters to show to the
    // user
    IF BLOBImport(nPos, "c:\application\temp.doc")
       cStr := BLOBGet(nPos, 1, 25)
    ELSE
       cStr := "Error: could not import file!"
    ENDIF

CLOSE

RETURN cStr
```

See Also

BLOBDIRECTGET(), BLOBDIRECTPUT(), BLOBIMPORT(), FIELDGET()

BLOBIMPORT() function

Read the contents of a file as a BLOB, identified by a memo field number

Syntax

```
BLOBIMPORT(<nFieldPos>, <cSourceFile>) → lSuccess
```

Arguments

<nFieldPos> is the position of the field in the database file structure.

<cSourceFile> is the name of the file from which to read the BLOB data, including an optional drive, directory, and extension. See SETDEFAULT() and SETPATH() for file searching and creation rules. No default extension is assumed.

This function attempts to open *<cSourceFile>* in shared mode. If the file does not exist, a runtime error is raised. If the file is successfully opened, the operation proceeds. If access is denied because another process has exclusive use of the file, for example, NETERR() is set to true (.T.).

Note: There are no restrictions on the size of *<cSourceFile>* except that you must have enough disk space to make the copy.

Returns

BLOBIMPORT() returns true (.T.) if successful; otherwise, it returns false (.F.).

Description

BLOBIMPORT() provides a mechanism for copying the contents of a file into a memo field as BLOB data. By default, this function operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression.

BLOBIMPORT() is used in conjunction with BLOBEXPORT() to transfer BLOB data back and forth between files and memo fields. You can use BLOBIMPORT() with a variety of file types, including graphic images, word processor files, and printer fonts. These two functions are excellent for creating databases for documents, graphics, sounds, etc.

Note: DBFIELDINFO(DBS_BLOB_TYPE, *<nFieldPos>*) will return "C" (string) for any memo field created using BLOBIMPORT().

Examples

- This example imports information from a word processing document into a field, and then uses BLOBGET() to extract the first 25 characters of the field:

```
FUNCTION POPULATE()
    USE customer NEW VIA "DBFCDX"

    // Construct unique file name based on last
    // name and id
    DO WHILE .NOT. EOF()
       GetPix("Pix", Substr(LastName, 1, 4) + CustID)
       Customer->DBSKIP()
    ENDDO
FUNCTION GetPix(cPixField, cPixFile)
    LOCAL nPos

    nPos := FieldPos(cPixField)

    // Import the picture file into indicated field
    IF !BLOBImport(nPos, cPixFile)
       Alert("Import of picture " + cPixFile + ";
       failed!")
    ENDIF
```

See Also

BLOBEXPORT(), DBFIELDINFO()

BLOBROOTGET() function

Retrieve the data from the root area of a BLOB file

Syntax

```
BLOBROOTGET() → uBLOB
```

Returns

BLOBROOTGET() returns the data retrieved from the root of the BLOB file. The data type of the return value depends on the actual data stored. Use VALTYPE() or USUALTYPE() to determine the data type. Note that BLOBROOTGET() returns NIL if the root reference has never been written to with BLOBROOTPUT().

Description

BLOBROOTGET() allows the retrieval of a BLOB from the root of a BLOB file in a work area. By default, this function operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression.

Note: Because the root data does not reference a particular record in the database file, the DBRLOCK() will not protect this root storage reference. Therefore, if the database file is opened in shared mode, you should use BLOBROOTLOCK() before calling BLOBROOTGET().

Examples

- This example uses BLOBROOTGET() to read system settings from a BLOB file into an array, and then demonstrates how to allow the user to modify the settings and restore them in the BLOB file:

```
FUNCTION UPDATESETTINGS()
   LOCAL aSettings

   USE customer NEW SHARED VIA "DBFCDX"

   IF BLOBROOTLOCK()
      // Get any existing settings
      aSettings := BLOBROOTGET()

      IF Empty(aSettings)
         // This function would populate aSettings
         // with default data
         aSettings := GETDEFAULTSETTINGS()
      ENDIF

      // This function would allow the user to
      // modify the settings.
      IF ModifySettings(aSettings)
         // Finally, store the settings
         BLOBRootPut(aSettings)
      ENDIF
      BLOBROOTUNLOCK()
   ELSE
      aSettings := {}
      Alert("Could not obtain a lock on the root;
         area")
   ENDIF

   CLOSE

   RETURN aSettings
```

See Also BLOBROOTLOCK(), BLOBROOTPUT(), DBRLOCK(), DBUSEAREA()

BLOBROOTLOCK() function

Obtain a lock on the root area of a BLOB file

Syntax

```
BLOBROOTLOCK() → lSuccess
```

Returns

BLOBROOTLOCK() returns true (.T.) if successful; otherwise, it returns false (.F.).

Description

Use BLOBROOTLOCK() when accessing the database file in shared mode to obtain a lock on the root area of a BLOB file for reading from or writing to the root area.

By default, this function operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression.

Examples

- This example illustrates how to properly lock and unlock the root area of a BLOB file for a database file opened in shared mode:

```
FUNCTION GETSETTINGS()
   LOCAL aCustSettings

   // Open a customer file in shared mode
   USE customer NEW SHARED DBFCDX

   IF BLOBROOTLOCK()
      aCustSettings := BLOBROOTGET()
      BLOBROOTUNLOCK()
   ELSE
      Alert("Could not obtain root lock of Customer;
         file")
   ENDIF

   CLOSE

   RETURN aCustSettings
```

See Also

BLOBROOTGET(), BLOBROOTPUT(), BLOBROOTUNLOCK(), DBRLOCK(), DBUSEAREA()

BLOBROOTPUT() function

Store data in the root area of a BLOB file

Syntax

```
BLOBROOTPUT(<uBLOB>) → lSuccess
```

Arguments

<uBLOB> is the data you want to put into the BLOB file's root area. *<uBLOB>* can be any CA-Clipper usual data type except code block and object.

Returns

BLOBROOTPUT() returns true (.T.) if successful; otherwise, it returns false (.F.).

Description

BLOBROOTPUT() allows the storage of one—and only one—piece of data to a BLOB file's root area (there is no size limitation on this one piece of data). After storing the new data, BLOBROOTPUT() releases the space associated with any data previously stored in the BLOB file's root area.

By default, this function operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression.

Note: Because the root data does not reference a particular record in the database file, the DBRLOCK() will not protect this root storage reference. Therefore, if the database file is opened in shared mode, you should use BLOBROOTLOCK() before calling BLOBROOTPUT().

B

Examples

- This example uses BLOBROOTPUT() to store system settings to a BLOB file after modification:

```
FUNCTION UPDATESETTINGS()
   LOCAL aSettings

   USE customer NEW SHARED VIA "DBFCDX"

   IF BLOBROOTLOCK()
      // Get any existing settings
      aSettings := BLOBROOTGET()

      IF Empty(aSettings)
         // This function would populate aSettings
         // with default data
         aSettings := GETDEFAULTSETTINGS()
      ENDIF

      // This function would allow the user to
      // modify the settings.
      IF ModifySettings(aSettings)
         // Finally, store the settings
         BLOBRootPut(aSettings)
      ENDIF
      BLOBROOTUNLOCK()
   ELSE
      aSettings := {}
      Alert("Could not obtain a lock on the root;
         area")
   ENDIF

   CLOSE

   RETURN aSettings
```

See Also

BLOBROOTGET(), BLOBROOTLOCK()

BLOBROOTUNLOCK() function

Release the lock on a BLOB file's root area

Syntax

```
BLOBROOTUNLOCK() → NIL
```

Description

Use BLOBROOTUNLOCK() to release a lock previously obtained using BLOBROOTLOCK().

By default, this function operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression.

Note: The only functions that require the use of BLOBROOTLOCK() or BLOBROOTUNLOCK() are BLOBROOTGET() and BLOBROOTPUT().

Examples

- This example illustrates how to properly lock and unlock the root area of a BLOB file for a database file opened in shared mode:

```
FUNCTION GETSETTINGS()
   LOCAL aCustSettings

   // Open a customer file in shared mode
   USE customer NEW SHARED VIA DBFCDX

   IF BLOBROOTLOCK()
      aCustSettings := BLOBROOTGET()
      BLOBROOTUNLOCK()
   ELSE
      Alert("Could not obtain root lock of Customer;
         file")
   ENDIF

   CLOSE

   RETURN aCustSettings
```

See Also

BLOBROOTLOCK()

BOF() function

Determine when beginning of file is encountered

Syntax

```
BOF() → lBoundary
```

Returns

BOF() returns true (.T.) after an attempt to SKIP backward beyond the first logical record in a database file; otherwise, it returns false (.F.). If there is no database file open in the current work area, BOF() returns false (.F.). If the current database file contains no records, BOF() returns true (.T.).

Description

BOF() is a database function used to test for a boundary condition when you are moving the record pointer backward through a database file using the SKIP command. A simple usage example is a descending order record list with an ascending order index file. A more sophisticated example is a screen paging routine that pages forward or backward through the current database file based on the key the user presses. When the user attempts to page backward, you would use BOF() to test for a beginning of file condition before using the SKIP command to move the record pointer and repaint the screen.

Once BOF() is set to true (.T.), it retains its value until there is another attempt to move the record pointer.

By default, BOF() operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression (see example below).

The SKIP command is the only record movement command that can set BOF() to true (.T.).

Examples

- This example demonstrates BOF() by attempting to move the record pointer before the first record:

```
USE Sales NEW
? RECNO(), BOF()            // Result: 1 .F.
SKIP -1
? RECNO(), BOF()            // Result: 1 .T.
```

- This example uses aliased expressions to query the value of BOF() in unselected work areas:

```
USE Sales NEW
USE Customer NEW
USE Invoices NEW
? Sales->(BOF()), Customer->(BOF())
```

Files Library is CLIPPER.LIB.

See Also EOF(), SKIP

BREAK() function

Branch out of a BEGIN SEQUENCE...END construct

Syntax

```
BREAK(<exp>) → NIL
```

Arguments

<exp> is the value passed to the RECOVER clause, if any. Note that *<exp>* is not optional. NIL may be specified if there is no break value.

Returns

BREAK() always returns NIL.

Description

The BREAK() function is identical in functionality to the BREAK statement. The function must be executed during a SEQUENCE. BREAK() has the advantage that, as an expression, it can be executed from a code block.

Examples

- This example illustrates exiting a SEQUENCE from a code block:

```
bSave := ERRORBLOCK( {|x| BREAK(x)} )

BEGIN SEQUENCE
   .
   .
   .
RECOVER USING objError
   .
   .
   .
END

ERRORBLOCK(bSave)
```

Files Library is CLIPPER.LIB.

See Also BEGIN SEQUENCE

BROWSE()* function

Browse records within a window

Syntax

```
BROWSE([<nTop>], [<nLeft>],
    [<nBottom>], [<nRight>]) lSuccess
```

Arguments

<nTop>*, *<nLeft>*, *<nBottom>, and ***<nRight>*** define the window coordinates. If not specified, the default window coordinates are 1, 0 to MAXROW(), and MAXCOL().

Returns

BROWSE() returns false (.F.) if there is no database file in use; otherwise, it returns true (.T.).

Description

BROWSE() is a user interface function that invokes a general purpose table-oriented browser and editor for records in the current work area. For a list of the navigation keys which are used by BROWSE(), refer to the DBEDIT() function. Note that BROWSE() is a compatibility function. DBEDIT() should be used in its place. For a more complicated BROWSE(), TBROWSE() should be used.

Notes

- ***Status line:*** BROWSE() supports a status line in the upper right corner of the browse window indicating one of the following:

BROWSE() Status Line Messages

Message	Meaning
<new>	Append mode
<bof>	Top of file
<delete>	Current record is deleted
Record	Record number display

- BROWSE() has the following three modes:

  – ***Browsing:*** This is the default mode of BROWSE(). Pressing any DBEDIT() navigation key moves the highlight to a new column or row.

  – ***Field edit:*** Pressing Return on any field enters field edit using a GET. Pressing Return terminates the edit mode, saving the changes. Esc terminates without saving changes. Since the field edit mode uses GET, all navigation and editing keys are READ keys.

  – ***Append:*** GOing BOTTOM with Ctrl+PgDn and then pressing Down arrow enters append mode with the indicating message "<new>" on the status line. A new blank record is then inserted. Pressing Up arrow terminates the append mode, saving the new record if data has been entered. If no data has been entered, the new record is not saved.

Examples

- This is an example of browsing a file:

```
USE File1 NEW
BROWSE()
```

Files

Library is EXTEND.LIB, source file is SOURCE\SAMPLE\BROWSE.PRG

See Also

DBEDIT()*

CALL* command

Execute a C or Assembler procedure

Syntax

```
CALL <idProcedure> [WITH <exp list>]
```

Arguments

<idProcedure> is the name of the external procedure to CALL.

WITH <exp list> is an optional list of up to seven expressions of any data type to pass to the external procedure.

Description

CALL executes a separately compiled or assembled procedure. The procedure must be defined as FAR and end with a FAR return instruction. Place parameters on the stack using the C parameter passing convention. Each parameter consists of a FAR (four-byte) pointer to the actual parameter value. When necessary you may use the WORD() function to pass a two-byte binary value in the WITH expression. The DX:BX and ES:BX registers also contain a copy of the first four bytes of parameter information.

The procedure must preserve the BP, SS, SI, DI, ES, and DS registers as well as clear the direction flag.

CALL is a compatibility command and therefore not recommended. It is superseded by the Extend system which provides functions for passing data to and from CA-Clipper.

Notes

- ***Character strings:*** Pass a character argument as a FAR pointer to a null-terminated string (a string with a 00 hex byte at the end).
- ***Numeric values:*** Pass each numeric argument as a FAR pointer to an eight-byte IEEE floating point value. To pass a parameter as an integer, use the WORD() function. The WORD() function converts the numeric value to a two-byte binary integer, and passes the integer value directly rather than through a pointer. Note that WORD() will not work for values outside of the ±32,767 range since these values cannot be accurately represented as two-byte integers.

- ***Date values:*** Pass each date argument as a FAR pointer to a four-byte (long) integer containing a Julian day number.
- ***Logical values:*** Pass each logical argument as a FAR pointer to a two-byte binary integer containing zero for false (.F.) and one for true (.T.).
- ***Compiling and linking:*** CALLed programs must conform to the following rules:
  - Procedures must be in INTEL 8086 relocatable object file format with the .OBJ file extension.
  - Procedures must follow the C calling and parameter passing conventions.
  - Procedures must be available to the linker at link time, along with the library of the source compiler. You will need runtime support for any language other than assembly language. See your compiler manual for further information.
- ***Screen position:*** When using a CALL statement to access a C or Assembler routine, the cursor is set to the current screen position within the C or Assembler routine.
- ***Microsoft C:*** Microsoft C versions 5.0 and above place a leading underscore on function names when they are compiled. To call them, therefore, you must CALL _<function>.
- ***dBASE III PLUS:*** To convert a dBASE III PLUS load module to a CA-Clipper-compatible module, add the following statements to your .asm file:

  ```
  PUBLIC <proc>
  ```

  and

  ```
  push ds
  mov ds, dx
  ```

Warning! *Modifying the parameter values may produce incorrect or unexpected results and, therefore, is strongly discouraged.*

Files Library is CLIPPER.LIB.

See Also WORD()*

CANCEL* command

Terminate program processing

Syntax

```
CANCEL* | QUIT
```

Description

CANCEL and QUIT both terminate the current program, closing all open files, and returning control to the operating system. You can use either command from anywhere in a CA-Clipper program system. A RETURN executed at the highest level procedure performs the same action. Note that CANCEL is a compatibility command. QUIT should be used in its place.

Notes

- ***Return code:*** When a CA-Clipper program terminates, the return code is set to 1 if the process ends with a fatal error. If the process ends normally, the return code is set to 0 or the last ERRORLEVEL() set in the program.

Files

Library is CLIPPER.LIB.

See Also

ERRORLEVEL(), QUIT, RETURN

CDOW() function

Convert a date value to a character day of the week

Syntax

```
CDOW(<dExp>) → cDayName
```

Arguments

<dExp> is the date value to convert.

Returns

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

Description

CDOW() is a date conversion function used in formatting date displays for reports, labels, and screens.

Examples

- These examples illustrate CDOW():

```
? DATE()                       // Result: 09/01/90
? CDOW(DATE())                 // Result: Friday
? CDOW(DATE() + 7)             // Result: Friday
? CDOW(CTOD("06/12/90"))       // Result: Tuesday
```

Files

Library is CLIPPER.LIB.

See Also

CTOD(), DATE(), DAY(), DOW()

CheckBox class

Create check boxes, which are controls that can be toggled on or off by a user

Class Function

```
CheckBox( <nRow>, <nColumn>,
   [,<cCaption>] ) → oCheckBox
```

Arguments

<***nRow***> is a numeric value that indicates the screen row of the check box.

<***nColumn***> is a numeric value that indicates the screen column of the check box.

<***nCaption***> is an optional character string that describes the check box on the screen. If omitted, the default is an empty string.

Returns

Returns a CheckBox object when all of the required arguments are present; otherwise, CheckBox() returns NIL.

Description

Check boxes present a choice to the user which can be either on or off. When a check box is clicked, its state is toggled (as indicated by an **X** in the box) between *checked* (on) and *unchecked* (off).

The CheckBox class has been designed to be easily integrated into the standard CA-Clipper GET/READ system in addition to providing the necessary functionality to be utilized on its own.

Exported Instance Variables

`bitmaps` (Assignable)

Contains an array of exactly two elements that indicates the bitmap files to be displayed. The first element indicates the file name of the bitmap to be displayed when the check box is selected. The second element indicates the file name of the bitmap to be displayed when the check box is not selected.

Drive and directory names are not allowed; the file name extension is required. A bitmap file can be stored as a file on disk or in a bitmap library. If stored as a file, the file must reside in the same directory as the application. If stored in a bitmap library, the library must reside in the same directory as the application and it also must have the same name as the application with a .bml extension.

CA-Clipper will search for the file name first, and if it is not found, then search in the bitmap library. If no file is found either on disk or in the library, no bitmap will be displayed.

If this instance variable is not used, and the application is running in graphic mode, the files CHECK_F.BMU and CHECK_E.BMU will be used for the selected bitmap and unselected bitmap, respectively.

This instance variable only affects applications running in graphic mode and is ignored in text mode.

`buffer`

Contains a logical value that indicates whether the check box is checked or unchecked. A value of true (.T.)indicates that it is checked and a value of false (.F.) indicates that it is not checked.

`capCol` `(Assignable)`

Contains a numeric value that indicates the screen column where the check box's caption is displayed.

`capRow` `(Assignable)`

Contains a numeric value that indicates the screen row where the check box's caption is displayed.

`caption` `(Assignable)`

Contains an optional character string that concisely describes the check box on the screen. If omitted, the default is an empty string.

When present, the & character specifies that the character immediately following it in the caption is the check box's accelerator key. The accelerator key provides a quick and convenient mechanism for the user to move input focus from one data input control to the check box. The user performs the selection by pressing the Alt key in combination with an accelerator key. The case of an accelerator key is ignored.

`cargo` (Assignable)

Contains a value of any type that is ignored by the CheckBox object. CheckBox:cargo is provided as a user-definable slot allowing arbitrary information to be attached to a CheckBox object and retrieved later.

`col` (Assignable)

Contains a numeric value that indicates the screen column where the check box is displayed.

`colorSpec` (Assignable)

Contains a character string that indicates the color attributes that are used by the check box's Display() method. The string must contain four color specifiers.

Note: In graphic mode, colorSpec positions 1 and 2 have no affect and are ignored.

CheckBox Color Attributes

Position in colorSpec	Applies To	Default Value from System Color setting
1	The check box when it does not have input focus	Unselected
2	The check box when it has input focus	Enhanced
3	The check box's caption	Standard
4	The check box caption's accelerator key	Background

Note: The colors available to a DOS application are more limited than those for a Windows application. The only colors available to you here are those listed in the drop-down list box of the Workbench Properties window for that item.

`fBlock` (Assignable)

Contains an optional code block that, when present, is evaluated each time the CheckBox object receives or loses input focus. The code block takes no implicit arguments. Use CheckBox:hasFocus to determine if the check box is receiving or losing input focus. A value of true (.T.) indicates that it is receiving input focus; otherwise, a value of false (.F.) indicates that it is losing input focus.

This code block is included in the CheckBox class to provide a method of indicating when an input focus change event has occurred. The name "fBlock" refers to focus block.

`hasFocus`

Contains a logical value that indicates whether the CheckBox object has input focus. CheckBox:hasFocus contains true (.T.) if it has input focus; otherwise, it contains false (.F.).

`message` (Assignable)

Contains a character string that describes the check box. It is displayed on the screen's status bar line.

`row` (Assignable)

Contains a numeric value that indicates the screen row where the check box is displayed.

`sBlock` (Assignable)

Contains an optional code block that, when present, is evaluated each time the CheckBox object's state changes. The code block takes no implicit arguments. Use the buffer instance variable to determine if the check box is being checked or unchecked. A value of true (.T.) indicates that it is being checked; otherwise, a value of false (.F.) indicates that it is being unchecked.

This code block is included in the CheckBox class to provide a method of indicating when a state change event has occurred. The name "sBlock" refers to state block.

`style` (Assignable)

Contains a character string that indicates the delimiter characters that are used by the check box's display() method. The string must contain four characters. The first is the left delimiter. Its default value is the left square bracket ([) character. The second is the checked indicator. Its default value is the square root (√) character. The third is the unchecked indicator. Its default is the space character (" "). The fourth character is the right delimiter. Its default value is the right square bracket (]) character.

Note: The style instance variable is ignored in graphic mode.

`typeOut`

Contains the logical value false (.F.). CheckBox:typeOut never changes. It is not used by the CheckBox object and is only provided for compatibility with the other GUI control classes.

Exported Methods

```
<oCheckBox>:display() → self
```

display() is a method of the CheckBox class that is used for showing a check box and its caption on the screen. display() uses the values of the following instance variables to correctly show the check box in its current context in addition to providing maximum flexibility in the manner a check box appears on the screen: buffer, caption, capCol, capRow, col, colorSpec, hasFocus, row, and style.

```
<oCheckBox>:hitTest( <nMouseRow>, <nMouseCol> )
    → nHitStatus
```

<nMouseRow> is a numeric value that indicates the current screen row position of the mouse cursor.

<nMouseCol> is a numeric value that indicates the current screen column position of the mouse cursor.

Returns a numeric value that indicates the relationship of the mouse cursor with the check box. hitTest() is a method of the CheckBox class that is used for determining if the mouse cursor is within the region of the screen that the check box or its caption occupies.

Applicable Hit Test Return Values

Value	Constant	Description
0	HTNOWHERE	The mouse cursor is not within the region of the screen that the check box occupies
-1025	HTCAPTION	The mouse cursor is on the check box's caption
-2049	HTCLIENT	The mouse cursor is on the check box

Button.ch contains manifest constants for the hitTest() return value.

```
<oCheckBox>:killFocus() → self
```

killFocus() is a method of the CheckBox class that is used for taking input focus away from a CheckBox object. Upon receiving this message, the CheckBox object redisplays itself and, if present, evaluates the code block within its fBlock variable.

This message is meaningful only when the CheckBox object has input focus.

<*oCheckBox*>:select([<*lNewState*>]) → *self*

<*lNewState*> is a logical value that indicates whether the check box should be checked or not. Set to true (.T.) to check the box or false to uncheck the box. If omitted, the check box state will toggle to its opposing state.

select() is a method of the CheckBox class that is used for changing the state of a check box. Its state is typically changed when the space bar is pressed or the mouse's left button is pressed when its cursor is within the check box's region of the screen.

<*oCheckBox*>:setFocus() → *self*

setFocus() is a method of the CheckBox class that is used for giving focus to a CheckBox object. Upon receiving this message, the CheckBox object redisplays itself and if present, evaluates the code block within its fBlock variable.

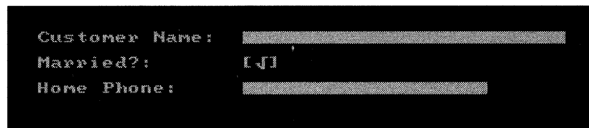
This message is meaningful only when the CheckBox object does not have input focus.

Examples

- This example creates and integrates a check box within a *Get List* and activates it by performing a READ:

```
STORE SPACE (25) TO Name
STORE .T. TO Married
STORE SPACE (19) TO Phone

CLS

@  5,10 SAY "Customer Name:   " GET Name
@  7,10 SAY "Married?:        " GET Married CHECKBOX
@  9,10 SAY "Home Phone:      " GET Phone

READ

? ALLTRIM (Name) + " IS " + IIF(Married,"","NOT") + "Married."
WAIT
```

Result:



CHR() function

Convert an ASCII code to a character value

Syntax

```
CHR(<nCode>) → cChar
```

Arguments

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

Returns

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

Description

CHR() is a numeric conversion function that converts an ASCII code to a character. It is the inverse of ASC(). CHR() serves a number of common tasks including:

- Sending control codes and graphics characters to the screen or printer
- Ringing the bell
- Converting INKEY() return values to characters
- Stuffing the keyboard buffer

Notes

- ***The null character:*** CHR(0) has a length of one (1) and is treated like any other character. This lets you send it to any device or file, including a database file.

Examples

- These examples illustrate CHR() with various arguments:

```
? CHR(72)                   // Result: H
? CHR(ASC("A") + 32)        // Result: a
? CHR(7)                    // Result: bell sounds
```

- These lines of code show the difference between a null string and the null character:

```
? LEN("")                   // Result: 0
? LEN(CHR(0))               // Result: 1
```

Files

Library is CLIPPER.LIB.

See Also

ASC(), INKEY()

CLEAR ALL* command

Close files and release public and private variables

Syntax

```
CLEAR ALL
```

Description

CLEAR ALL releases all public and private variables, closes all open databases and related files in all active work areas, and SELECTs work area 1. Related files are index, alternate, and memo files. CLEAR ALL, however, does not release local or static variables.

CLEAR ALL is a compatibility command—its use is not recommended. Its usage in CA-Clipper is superseded by the command or function that performs the specific action you need. You can close files associated with work areas with one of the various forms of the CLOSE command. You can release private and public variables using the RELEASE command although explicitly releasing variables is discouraged in most instances. For more information on the scope and lifetime of variables, refer to the "Basic Concepts" chapter in the *Programming and Utilities Guide*.

Files

Library is CLIPPER.LIB.

See Also

CLEAR GETS, CLEAR MEMORY, CLOSE, RELEASE

CLEAR GETS command

Release Get objects from the current *GetList* array

Syntax

```
CLEAR GETS
```

Description

CLEAR GETS explicitly releases all Get objects in the current and visible *GetList* array, and terminates the calling READ, releasing any remaining objects in the calling READ, if executed within a SET KEY procedure or a user-defined function invoked by a VALID clause. CLEAR GETS releases Get objects by assigning an empty array to the variable *GetList*. *GetList* is the name of the variable used to hold an array of Get objects for subsequent READ commands. There are two other mechanisms that automatically release Get objects: CLEAR specified without the SCREEN clause, and READ specified without the SAVE clause.

CLEAR GETS has two basic uses. First, it can be used to terminate a READ from a SET KEY procedure or VALID user-defined function. Second, it can be used to delete Get objects from the *GetList* array when you have not executed a READ or you have saved the Get objects by using READ SAVE.

Files Library is CLIPPER.LIB.

See Also @...CLEAR, @...GET, CLOSE, READ, RELEASE, SET TYPEAHEAD

CLEAR MEMORY command

Release all public and private variables

Syntax

```
CLEAR MEMORY
```

Description

CLEAR MEMORY deletes both public and private memory variables from the memory variable table. It operates in contrast to RELEASE ALL, which does not actually delete public and private memory variables but assigns NIL to those whose scope is the current procedure. CLEAR MEMORY is the only way to delete all public memory variables from current memory. Local and static variables, however, are unaffected by CLEAR MEMORY.

For more information on variables, refer to the Variables section of the "Basic Concepts" chapter in the *Programming and Utilities Guide*.

Files Library is CLIPPER.LIB.

See Also RELEASE

CLEAR SCREEN command

Clear the screen and return the cursor home

Syntax

```
CLEAR [SCREEN] | CLS
```

Arguments

SCREEN suppresses the automatic releasing of Get objects from the current and visible *GetList* array when the screen is CLEARed.

Description

CLEAR is a full-screen command that erases the screen, releases pending GETs, and positions the cursor at row and column zero. If the SCREEN clause is specified, Get objects are not released.

CLS is a synonym for CLEAR SCREEN.

Notes

- ***SET KEY and VALID:*** If you are editing GETs, executing a CLEAR within a SET KEY procedure or within a VALID user-defined function will abort the active READ when control returns. To clear the screen without CLEARing GETs, use either the CLEAR SCREEN or CLS commands.

Files Library is CLIPPER.LIB.

See Also @...CLEAR, CLEAR GETS, SCROLL()

CLEAR TYPEAHEAD command

Empty the keyboard buffer

Syntax

```
CLEAR TYPEAHEAD
```

Description

CLEAR TYPEAHEAD is a keyboard command that clears all pending keys from the CA-Clipper keyboard buffer. This is useful in user interface procedures or user-defined functions to guarantee that keys processed from the keyboard buffer are appropriate to the current activity and not pending from a previous activity. User functions called by ACHOICE() and DBEDIT() are especially sensitive to such keys.

Note that both the SET TYPEAHEAD and KEYBOARD commands also clear the keyboard buffer.

Examples

- This example empties the keyboard buffer before invoking DBEDIT(), guaranteeing that any pending keys will not be executed:

```
CLEAR TYPEAHEAD
DBEDIT()
```

Files Library is CLIPPER.LIB.

See Also KEYBOARD, SET TYPEAHEAD

CLOSE command

Close a specific set of files

Syntax

```
CLOSE [<idAlias> | ALL | ALTERNATE | DATABASES |
   FORMAT |INDEXES]
```

Arguments

<idAlias> specifies the work area where all files are to be closed.

ALL closes alternate, database, and index files in all work areas, releasing all active filters, relations, and format definitions.

ALTERNATE closes the currently open alternate file, performing the same action as SET ALTERNATE TO with no argument.

DATABASES closes all open databases, memo and associated index files in all work areas, and releases all active filters and relations. It does not, however, have any effect on the active format.

FORMAT releases the current format, performing the same action as SET FORMAT TO with no argument.

INDEXES closes all index files open in the current work area.

Description

CLOSE is a general purpose command that closes various types of CA-Clipper files depending on the specified option. CLOSE with no option closes the current database and associated files, the same as USE with no arguments.

In CA-Clipper, a number of other commands also close files including:

- QUIT
- CANCEL*
- RETURN from the highest level procedure
- CLEAR ALL*
- USE with no argument

Files

Library is CLIPPER.LIB.

See Also

QUIT, RETURN, SET ALTERNATE, SET INDEX, USE

CMONTH() function

Convert a date to a character month name

Syntax

```
CMONTH(<dDate>) → cMonth
```

Arguments

<dDate> is the date value to convert.

Returns

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

Description

CMONTH() is a date conversion function useful for creating formatted date strings that can be used in reports, labels, or screens.

Examples

- These examples illustrate CMONTH():

```
? CMONTH(DATE())                    // Result: September
? CMONTH(DATE() + 45)               // Result: October
? CMONTH(CTOD("12/01/94"))          // Result: December
? SUBSTR(CMONTH(DATE()), 1, 3) +;
   STR(DAY(DATE()))           // Result: Sep 1
```

Files

Library is CLIPPER.LIB.

See Also

CDOW(), DATE(), DAY(), MONTH(), YEAR()

COL() function

Return the screen cursor column position

Syntax

```
COL() → nCol
```

Returns

COL() returns an integer numeric value. The range of the return value is zero to MAXCOL().

Description

COL() is a screen function that returns the current column position of the cursor. The value of COL() changes whenever the cursor position changes on the screen. Both console and full-screen commands can change the cursor position. In addition, COL() is automatically set to zero whenever a CLEAR, CLEAR SCREEN, or CLS command is executed.

Use COL() to position the cursor to a column relative to the current column. It is generally used in combination with ROW() and all variations of the @ command. In particular, use COL() and ROW() to create screen position-independent procedures or functions that pass the upper-left row and column as parameters.

If DEVICE is SET TO PRINTER, all the output of @...SAY commands is directed to the printer and PROW() and PCOL() are updated instead of ROW() and COL(). Use these functions when you need to determine the position of the printhead.

Examples

- This example displays a Customer name beginning at column 10. The customer's account status is then displayed to the right of the last character of the customer name using COL():

  ```
  USE Sales NEW
  CLS
  @ 1, 10 SAY "Customer Name: " + TRIM(Customer)
  @ ROW(), COL() + 1 SAY "Account status: " + Status
  ```

Files

Library is CLIPPER.LIB.

See Also

@...CLEAR, @...GET, @...SAY, CLEAR, MAXCOL(), PCOL(), PROW(), QOUT(), ROW()

COLORSELECT() function

Activate attribute in current color settings

Syntax

```
COLORSELECT(<nColorIndex>) → NIL
```

Returns

Always returns NIL.

Arguments

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

Description

COLORSELECT() activates the specified color pair from the current list of color attributes (established by SETCOLOR()). Manifest constants for *<nColorIndex>* are defined in Color.ch.

Color.ch Constants

Constant	Value
CLR_STANDARD	0
CLR_ENHANCED	1
CLR_BORDER	2
CLR_BACKGROUND	3
CLR_UNSELECTED	4

COLORSELECT() does not alter the current SET COLOR setting.

This table describes the scope of the CA-Clipper color settings affected by SETCOLOR():

Color Settings

Setting	Scope
Standard	All screen output commands and functions
Enhanced	GETs and selection highlights
Border	Screen border (not supported on EGA and VGA monitors)
Background	Not supported
Unselected	Unselected GETs

Examples

- This example demonstrates use of COLORSELECT() with the Color.ch manifest constants:

```
USE Sales NEW
? SETCOLOR()                   // displays "W/B,N/B,W/N,W/N,W/N"
                               // in white on blue

COLORSELECT(CLR_ENHANCED)      // enhanced is active color pair
? "I'm black and blue"         // displayed in black on blue
COLORSELECT(CLR_STANDARD)      // restore standard color
```

Files Library is CLIPPER.LIB, header file is Color.ch.

See Also SETCOLOR(), SET COLOR

COMMIT command

Perform a solid-disk write for all active work areas

Syntax

```
COMMIT
```

Description

COMMIT is a database command that flushes CA-Clipper buffers and performs a solid-disk write for all work areas with open database and index files. The solid-disk write capability is available under DOS version 3.3 and above. Under DOS 3.2 or less, COMMIT flushes CA-Clipper buffers to DOS.

In a network environment, issuing a GO TO RECNO() or a SKIP0 will flush CA-Clipper's database and index buffers, but only a COMMIT will flush the buffers and perform a solid-disk write. Thus to insure updates are visible to other processes, you must issue a COMMIT after all database update commands (e.g., APPEND, REPLACE). To insure data integrity, COMMIT should be issued before an UNLOCK operation. Refer to the "Network Programming" chapter in the *Programming and Utilities Guide* for more information on update visibility.

Notes

- COMMIT uses DOS interrupt 21h function 68h to perform the solid-disk write. It is up to the network operating system to properly implement this request. Check with the network vendor to see if this is supported.

Examples

- In this example, COMMIT forces a write to disk after a series of memory variables are assigned to field variables:

```
USE Sales EXCLUSIVE NEW
MEMVAR->Name := Sales->Name
MEMVAR->Amount := Sales->Amount
//
@ 10, 10 GET MEMVAR->Name
@ 11, 10 GET MEMVAR->Amount
READ
//
IF UPDATED()
    APPEND BLANK
    REPLACE Sales->Name WITH MEMVAR->Name
    REPLACE Sales->Amount WITH MEMVAR->Amount
    COMMIT
ENDIF
```

Files

Library is CLIPPER.LIB.

See Also

DBCOMMIT(), DBCOMMITALL(), GO, SKIP

CONTINUE command

Resume a pending LOCATE

Syntax

```
CONTINUE
```

Description

CONTINUE is a database command that searches from the current record position for the next record meeting the most recent LOCATE condition executed in the current work area. It terminates when a match is found or end of file is encountered. If CONTINUE is successful, the matching record becomes the current record and FOUND() returns true (.T.); if unsuccessful, FOUND() returns false (.F.).

Each work area may have an active LOCATE condition. In CA-Clipper, a LOCATE condition remains pending until a new LOCATE condition is specified. No other commands release the condition.

Notes

- ***Scope and WHILE condition:*** Note that the scope and WHILE condition of the initial LOCATE are ignored; only the FOR condition is used with CONTINUE. If you are using a LOCATE with a WHILE condition and want to continue the search for a matching record, use SKIP and then repeat the original LOCATE statement adding REST as the scope.

Examples

- This example scans records in Sales.dbf for a particular salesman and displays a running total sales amounts:

```
LOCAL nRunTotal := 0
USE Sales NEW
LOCATE FOR Sales->Salesman = "1002"
DO WHILE FOUND()
    ? Sales->Salesname, nRunTotal += Sales->Amount
    CONTINUE
ENDDO
```

- This example demonstrates how to continue if the pending LOCATE scope contains a WHILE condition:

```
LOCAL nRunTotal := 0
USE Sales INDEX Salesman NEW
SEEK "1002"
LOCATE REST WHILE Sales->Salesman = "1002";
       FOR Sales->Amount > 5000
DO WHILE FOUND()
    ? Sales->Salesname, nRunTotal += Sales->Amount
    SKIP
    LOCATE REST WHILE Sales->Salesman = "1002";
       FOR Sales->Amount > 5000
ENDDO
```

Files Library is CLIPPER.LIB.

See Also EOF(), FOUND(), LOCATE, SEEK

COPY FILE command

Copy a file to a new file or to a device

Syntax

```
COPY FILE <xcSourceFile> TO <xcTargetFile>|<xcDevice>
```

Arguments

<*xcSourceFile*> is the name of the source file to copy including the extension.

<*xcTargetFile*> is the name of the target file including the extension.

Both arguments can be specified as literal file names or as character expressions enclosed in parentheses. COPY FILE supplies no default extensions.

<*xcDevice*> is the name of the device where all subsequent output will be sent. You can specify a device name as a literal character string or a character expression enclosed in parentheses. Additionally, a device can be either local or network. If you COPY TO a non-existing device, you create a file with the name of the device. When specifying device names, do not use a trailing colon.

Description

COPY FILE is a file command that copies files to and from the CA-Clipper default drive and directory unless you specify a drive and/or path. If the <*xcTargetFile*> exists, it is overwritten without warning or error.

Examples

- This example copies a file to a new file and then tests for the existence of the new file:

```
COPY FILE Test.prg TO Real.prg
? FILE("Real.prg")                    // Result: .T.
```

Files Library is EXTEND.LIB.

See Also COPY TO, RENAME, SET DEFAULT

C

COPY STRUCTURE command

Copy the current .dbf structure to a new database (.dbf) file

Syntax

```
COPY STRUCTURE [FIELDS <idField list>]
   TO <xcDatabase>
```

Arguments

FIELDS <idField list> defines the set of fields to copy to the new database structure in the order specified. The default is all fields.

TO <xcDatabase> is the name of the target database file and can be specified either as a literal database file name or as a character expression enclosed in parentheses. The default extension is .dbf unless another is specified.

Description

COPY STRUCTURE is a database command that creates an empty database file with field definitions from the current database file. If *<xcDatabase>* exists, it is overwritten.

COPY STRUCTURE creates empty structures that can be used to archive records from the current database file or to create a temporary database file for data entry.

Examples

- In this example, COPY STRUCTURE creates a temporary file. After the user enters data into the temporary file, the master database file is updated with the new information:

```
USE Sales NEW
COPY STRUCTURE TO Temp
USE Temp NEW
lMore := .T.
DO WHILE lMore
   APPEND BLANK
   @ 10, 10 GET Temp->Salesman
   @ 11, 11 GET Temp->Amount
   READ
   IF UPDATED()
      SELECT Sales
      APPEND BLANK
      REPLACE Sales->Salesman WITH Temp->Salesman
      REPLACE Sales->Amount WITH Temp->Amount
      SELECT Temp
      ZAP
   ELSE
      lMore := .F.
   ENDIF
ENDDO
CLOSE DATABASES
```

Files Library is CLIPPER.LIB.

See Also COPY STRUCTURE EXTENDED, CREATE

COPY STRUCTURE EXTENDED command

Copy field definitions to a .dbf file

Syntax

```
COPY STRUCTURE EXTENDED
   TO <xcExtendedDatabase>
```

Arguments

TO <xcExtendedDatabase> is the name of the target structure extended database file. This argument can be specified either as a literal database file name or as a character expression enclosed in parentheses.

Description

COPY STRUCTURE EXTENDED creates a database file whose contents is the structure of the current database file with a record for the definition of each field. The structure extended database file has the following structure:

Structure of an Extended File

Field	Name	Type	Length	Decimals
1	Field_name	Character	10	
2	Field_type	Character	1	
3	Field_len	Numeric	3	0
4	Field_dec	Numeric	4	0

Used in application programs, COPY STRUCTURE EXTENDED permits you to create or modify the structure of a database file programmatically. To create a new database file from the structure extended file, use CREATE FROM. If you need an empty structure extended file, use CREATE.

Notes

- ***Character field lengths greater than 255:*** In CA-Clipper, the maximum character field length is 64K. For compatibility reasons, field lengths greater than 255 are represented as a combination of the Field_dec and Field_len fields. After COPYing STRUCTURE EXTENDED, you can use the following formula to determine the length of any character field:

```
nFieldLen := IF(Field_type = "C" .AND. ;
      Field_dec != 0, Field_dec * 256 + ;
      Field_len, Field_len)
```

Examples

- This example creates Struc.dbf from Sales.dbf as a structure extended file and then lists the contents of Struc.dbf to illustrate the typical layout of field definitions:

```
USE Sales NEW
COPY STRUCTURE EXTENDED TO Struc
USE Struc NEW
LIST Field_name, Field_type, Field_len, Field_dec
```

Result:

```
1 BRANCH        C     3     0
2 SALESMAN      C     4     0
3 CUSTOMER      C     4     0
4 PRODUCT       C     25    0
5 AMOUNT        N     8     2
6 NOTES         C     0     125
// Field length is 32,000 characters
```

Files Library is CLIPPER.LIB.

See Also COPY STRUCTURE, CREATE, CREATE FROM, FIELDNAME(), TYPE()

COPY TO command

Export records to a new database (.dbf) file or ASCII text file

Syntax

```
COPY [FIELDS <idField list>] TO <xcFile>
   [<scope>] [WHILE <lCondition>] [FOR <lCondition>]
   [SDF | DELIMITED [WITH BLANK | <xcDelimiter>] |
   [VIA <xcDriver>]]
```

Arguments

FIELDS <idField list> specifies the list of fields to copy to the target file. The default is all fields.

TO <xcFile> specifies the name of the target file. The file name can be specified either as a literal file name or as a character expression enclosed in parentheses. If SDF or DELIMITED is specified, .txt is the default extension. Otherwise, .dbf is the default extension.

<scope> defines the portion of the current database file to COPY. The default is ALL records.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to copy within the given scope.

SDF specifies the output file type as a System Data Format ASCII text file. Records and fields are fixed length.

DELIMITED specifies the output file type as a delimited ASCII text file where character fields are enclosed in double quotation marks (the default delimiter). Records and fields are variable length.

DELIMITED WITH BLANK identifies an ASCII text file in which fields are separated by one space and character fields have no delimiters.

DELIMITED WITH <xcDelimiter> identifies a delimited ASCII text file where character fields are enclosed using the specified delimiter. *<xcDelimiter>* can be specified either as a literal character or as a character expression enclosed in parentheses.

See the following tables for more information regarding the format specifications for ASCII text files created using these arguments.

VIA <xcDriver> specifies the replaceable database driver (RDD) to use to create the resulting copy. *<cDriver>* is name of the RDD specified as a character expression. If *<cDriver>* is specified as a literal value, it must be enclosed in quotes.

If the VIA clause is omitted, COPY TO uses the driver in the current work area. If you specify the VIA clause, you must REQUEST the appropriate RDDs to be linked into the application.

Note: If the DELIMITED WITH clause is specified on a COPY or APPEND command, it must be the last clause specified.

Description

COPY TO is a database command that copies all or part of the current database file to a new file. Records contained in the active database file are copied unless limited by a *<scope>*, a FOR | WHILE clause, or a filter.

If DELETED is OFF, deleted records in the source file are copied to *<xcFile>* where they retain their deleted status. If DELETED is ON, however, no deleted records are copied. Similarly, if a FILTER has been SET, invisible records are not copied.

Records are copied in controlling index order if there is an index open in the current work area and SET ORDER is not zero. Otherwise, records are copied in natural order.

In a network environment, CA-Clipper opens the target database file EXCLUSIVEly before the COPY TO operation begins. Refer to the "Network Programming" chapter in the *Programming and Utilities Guide* for more information.

This table shows the format specifications for SDF text files:

SDF Text File Format Specifications

File Element	Format
Character fields	Padded with trailing blanks
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Padded with leading blanks for zeros
Field separator	None
Record separator	Carriage return/linefeed
End of file marker	1A hex or CHR(26)

This table shows the format specifications for DELIMITED and DELIMITED WITH <*xcDelimiter*> ASCII text files:

DELIMITED Text File Format Specifications

File Element	Format
Character fields	Delimited, with trailing blanks truncated
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Leading zeros truncated
Field separator	Comma
Record separator	Carriage return/linefeed
End of file marker	1A hex or CHR(26)

This table shows the format specifications for DELIMITED WITH BLANK ASCII text files:

DELIMITED WITH BLANK Text File Format Specifications

File Element	Format
Character fields	Not delimited, trailing blanks truncated
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Leading zeros truncated
Field separator	Single blank space
Record separator	Carriage return/linefeed
End of file marker	1A hex or CHR(26)

Examples

- This example demonstrates copying to another database file:

```
USE Sales NEW
COPY TO Temp
```

- This example demonstrates the layout of a DELIMITED file:

```
COPY NEXT 1 TO Temp DELIMITED
TYPE Temp.txt
```

Result:

```
"Character",12.00,19890801,T
```

- This example demonstrates the layout of an SDF file with four fields, one for each data type:

```
USE Testdata NEW
COPY NEXT 1 TO Temp SDF
TYPE Temp.txt
```

Result:

```
Character     12.0019890801T
```

- This example demonstrates the layout of a DELIMITED file WITH a different delimiter:

```
COPY NEXT 1 TO Temp DELIMITED WITH '
TYPE Temp.txt
```

Result:

```
'Character',12.00,19890801,T
```

Files Library is CLIPPER.LIB.

See Also APPEND FROM, COPY FILE, COPY STRUCTURE, SET DELETED

COUNT command

Tally records to a variable

Syntax

```
COUNT TO <idVar>
    [<scope>] [WHILE <lCondition>] [FOR <lCondition>]
```

Arguments

TO <idVar> identifies the variable that holds the COUNT result. A variable that either does not exist or is invisible is created as a private variable whose scope is the current procedure.

<scope> is the portion of the current database file to COUNT. The default is ALL records.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to COUNT within the given scope.

Description

COUNT tallies the number of records from the current work area that match the specified record scope and conditions. The result is then placed in the specified variable. *<idVar>* can be a variable of any storage class including a field.

Examples

- This example demonstrates a COUNT of Branches in Sales.dbf:

```
USE Sales NEW
COUNT TO nBranchCnt FOR Branch = 100
? nBranchCnt                          // Result: 4
```

- This example tallies the number of records in Sales.dbf whose Branch has the value of 100 and assigns the result to the Count field in Branch.dbf for branch 100:

```
USE Branch INDEX Branch NEW
SEEK 100
USE Sales INDEX SalesBranch NEW
SEEK 100
COUNT TO Branch->Count WHILE Branch = 100
```

Files Library is CLIPPER.LIB.

See Also AVERAGE, DBEVAL(), SUM, TOTAL

CREATE command

Create an empty structure extended (.dbf) file

Syntax

```
CREATE <xcExtendedDatabase>
```

Arguments

<xcExtendedDatabase> is the name of the empty structure extended database file. This argument can be specified either as a literal database file name or as a character expression enclosed in parentheses. If no extension is specified, .dbf is the default extension.

Description

CREATE produces an empty structure extended database file with the following structure:

Structure of an Extended File

Field	Name	Type	Length	Decimals
1	Field_name	Character	10	
2	Field_type	Character	1	
3	Field_len	Numeric	3	0
4	Field_dec	Numeric	4	0

Like COPY STRUCTURE EXTENDED, CREATE can be used in conjunction with CREATE FROM to form a new database file. Unlike COPY STRUCTURE EXTENDED, CREATE produces an empty database file and does not require the presence of another database file to create it.

<xcExtendedDatabase> is automatically opened in the current work area after it is created.

Examples

- This example creates a new structure extended file, places the definition of one field into it, and then CREATEs a new database file FROM the extended structure:

```
CREATE TempStru
APPEND BLANK
REPLACE Field_name WITH "Name",;
   Field_type WITH "C",;
   Field_len WITH 25,;
   Field_dec WITH 0
CLOSE
CREATE NewFile FROM TempStru
```

Files

Library is CLIPPER.LIB.

See Also

COPY STRUCTURE EXTENDED, CREATE FROM

CREATE FROM command

Create a new .dbf file from a structure extended file

Syntax

```
CREATE <xcDatabase> FROM <xcExtendedDatabase> [NEW]
    [ALIAS <xcAlias>] [VIA <cDriver>]
```

Arguments

<xcDatabase> is the name of the new database file to create from the structure extended file.

<xcExtendedDatabase> is the name of a structure extended file to use as the structure definition for the new database file.

Both of these arguments can be specified either as literal database file names or as character expressions enclosed in parentheses. If an extension is not specified, the default is .dbf.

NEW opens *<xcDatabase>* in the next available work area making it the current work area. If this clause is not specified, *<xcDatabase>* is opened in the current work area.

ALIAS <xcAlias> is the name to associate with the work area when *<xcDatabase>* is opened. You may specify the alias name as a literal name or as a character expression enclosed in parentheses. A valid *<xcAlias>* may be any legal identifier (i.e., it must begin with an alphabetic character and may contain numeric or alphabetic characters and the underscore). Within a single application, CA-Clipper will not accept duplicate aliases. If this clause is not specified, the alias defaults to the database file name.

VIA <cDriver> specifies the replaceable database driver (RDD) to use to process the current work area. *<cDriver>* is the name of the RDD specified as a character expression. If you specify *<cDriver>* as a literal value, you must enclose it in quotes.

Description

CREATE FROM produces a new database file with the field definitions taken from the contents of a structure extended file. To qualify as a structure extended file, a database file must contain the following four fields:

Structure of an Extended File

Field	Name	Type	Length	Decimals
1	Field_name	Character	10	
2	Field_type	Character	1	
3	Field_len	Numeric	3	0
4	Field_dec	Numeric	4	0

<*xcDatabase*> is automatically opened in the current work area after it is created.

Notes

- ***Data dictionaries:*** For data dictionary applications, you can have any number of other fields within the structure extended file to describe the extended field attributes. You may, for example, want to have fields to describe such field attributes as a description, key flag, label, color, picture, and a validation expression for the VALID clause. When you CREATE FROM, CA-Clipper creates the new database file from the required fields only, ignoring all other fields in the extended structure. Moreover, CA-Clipper is not sensitive to the order of the required fields.

- ***Character field lengths greater than 255:*** There is one method for creating a character field with a length greater than 255 digits:

  - Specify the field length using both the Field_len and Field_dec fields according to the following formulation:

    ```
    FIELD->Field_len := <nFieldLength> % 256
    FIELD->Field_dec := INT(<nFieldLength> / 256)
    ```

Examples

- This example is a procedure that simulates an interactive CREATE utility:

```
CreateDatabase("NewFile")
RETURN

FUNCTION CreateDatabase( cNewDbf )
   CREATE TmpExt          // Create empty structure extended
   USE TmpExt
   lMore := .T.
   DO WHILE lMore         // Input new field definitions
      APPEND BLANK
      CLEAR
      @ 5, 0 SAY "Name.....: " GET Field_name
      @ 6, 0 SAY "Type.....: " GET Field_type
      @ 7, 0 SAY "Length...: " GET Field_len
      @ 8, 0 SAY "Decimals.: " GET Field_dec
      READ
      lMore := (!EMPTY(Field_name))
   ENDDO

   // Remove all blank records
   DELETE ALL FOR EMPTY(Field_name)
   PACK
   CLOSE

// Create new database file
   CREATE (cNewDbf) FROM TmpExt
   ERASE TmpExt.dbf
   RETURN NIL
```

- This example creates a new definition in a structure extended file for a character field with a length of 4000 characters:

```
APPEND BLANK
REPLACE Field_name WITH "Notes",;
   Field_type WITH "C",;
   Field_len  WITH 4000 % 256,;
   Field_dec  WITH INT(4000 / 256)
```

Files Library is CLIPPER.LIB.

See Also COPY STRUCTURE, COPY STRUCTURE EXTENDED, CREATE

CTOD() function

Convert a date string to a date value

Syntax

```
CTOD(<cDate>) → dDate
```

Arguments

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

Returns

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

Description

CTOD() is a character conversion function that converts a character string to a date. To initialize an empty date for date entry, specify <*cDate*> as a null string (""), SPACE(8), or " / / ".

CTOD() is used whenever you need a literal date value. Some examples are:

- Initializing a variable to a date value
- Specifying a literal date string as an argument of a RANGE clause of @...GET
- Specifying a literal date string in order to perform date arithmetic
- Comparing the result of a date expression to a literal date string
- REPLACEing a date field with a literal date string

CTOD() is the inverse of DTOC() which converts a date value to a character string in the format specified by SET DATE and SET CENTURY. DTOS() also converts a date value to a character string in the form yyyymmdd.

Examples

- This example uses CTOD() to initialize two date variables, using one as a GET and the other for RANGE validation:

```
SET CENTURY ON
dBegin := CTOD("01-26-1876")
dCurrent := CTOD("")
@ 10, 10 SAY "Enter date:" GET dCurrent ;
      RANGE dBegin, DATE()
READ
```

- This example uses CTOD() to create a date value within a FOR condition:

```
USE Inventory NEW
REPLACE ALL Inventory->Price WITH ;
   Inventory->Price * 1.1 FOR ;
   Inventory->InvDate < CTOD("10/10/90")
```

Files Library is CLIPPER.LIB.

See Also DATE(), DTOC(), DTOS(), SET CENTURY, SET DATE, SET EPOCH

CURDIR() function

Return the current DOS directory

Syntax

```
CURDIR([<cDrivespec>]) → cDirectory
```

Arguments

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

Returns

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

If an error occurs, or the current directory of the specified drive is the root directory, CURDIR() returns a null string ("").

Description

CURDIR() is an environment function that gives you the name of the current DOS directory, ignoring the SET DEFAULT and SET PATH settings.

Examples

- These examples illustrate various CURDIR() results:

```
? CURDIR("E:")     // Result: null string--root directory
? CURDIR("C")      // Result: CLIP53\SOURCE
? CURDIR("C:")     // Result: CLIP53\SOURCE
? CURDIR()         // Result: null string--root directory
? CURDIR("A")      // Result: null string--drive not ready
```

- This example changes the current DOS directory to a new value if it does not match a specified directory:

```
IF CURDIR("C:") != "CLIP53\SOURCE"
   RUN CD \CLIP53\SOURCE
ENDIF
```

Files

Library is EXTEND.LIB, source file is SOURCE\SAMPLE\EXAMPLEA.ASM

See Also

FILE()

DATE() function

Return the system date as a date value

Syntax

```
DATE() → dSystem
```

Returns

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

Description

DATE() is a date function that provides a means of initializing memory variables to the current date, comparing other date values to the current date, and performing date arithmetic relative to the current date.

The display format for dates is controlled by the SET DATE command. The default format is *mm/dd/yy*.

Examples

- These examples show the DATE() function used in various ways:

```
? DATE()                   // Result: 09/01/90
? DATE() + 30              // Result: 10/01/90
? DATE() - 30              // Result: 08/02/90
dDate := DATE()
? CMONTH(dDate)            // Result: September
```

Files Library is CLIPPER.LIB.

See Also CTOD(), DTOC(), DTOS(), SET CENTURY, SET DATE

DAY() function

Return the day of the month as a numeric value

Syntax

```
DAY(<dDate>) → nDay
```

Arguments

<dDate> is a date value to convert.

Returns

DAY() returns a number in the range of zero to 31 as an integer numeric value. If the month is February, leap years are considered. If the date argument is February 29 and the year is not a leap year, DAY() returns zero. If the date argument is empty, DAY() returns zero.

Description

DAY() is a date conversion function used to convert a date value to the day of a month. This function is used in combination with CMONTH() and YEAR() to format dates. In addition, it is often used in various date calculations.

Examples

- These examples show the DAY() function used several ways:

```
? DATE()                          // Result: 09/01/90
? DAY(DATE())                     // Result: 1
? DAY(DATE()) + 1                 // Result: 2
? DAY(CTOD("12/01/94"))           // Result: 1
```

- This example uses DAY() in combination with CMONTH() and YEAR() to format a date value:

```
? CMONTH(DATE()) + STR(DAY(DATE())) +;
   "," + STR(YEAR(DATE()))    // Result: June 15, 1990
```

Files

Library is CLIPPER.LIB.

See Also

CDOW(), CMONTH(), DOW(), MONTH(), STR(), YEAR()

DBAPPEND() function

Append a new record to the database open in the current work area

Syntax

```
DBAPPEND([<lReleaseRecLocks>]) → NIL
```

Arguments

<lReleaseRecLocks> is a logical data type that if true (.T.), clears all pending record locks, then appends the next record. If *<lReleaseRecLocks>* is false (.F.), all pending record locks are maintained and the new record is added to the end of the Lock List. The default value of *<lReleaseRecLocks>* is true (.T.).

Returns

DBAPPEND() always returns NIL.

Description

DBAPPEND() is a database function that lets you add records to the current database. The enhancement to this function lets you maintain multiple record locks during an append.

DBAPPEND() without a parameter as in earlier versions of CA-Clipper, clears all pending record locks prior to an append. This is the same as DBAPPEND(.T.).

Examples

- This example appends a blank record to the database, Sales, without releasing the record locks in the current Lock List, and then checks for a network error:

```
USE Sales NEW
SET INDEX TO Sales
DBAPPEND(.F.)
IF NETERR()
   ? "A network error has occurred!"
ENDIF
```

See Also

APPEND BLANK

DBCLEARFILTER() function

Clear a filter condition

Syntax

```
DBCLEARFILTER() → NIL
```

Returns

DBCLEARFILTER() always returns NIL.

Description

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

DBCLEARFILTER() performs the same function as the standard SET FILTER command with no expression specified. For more information, refer to the SET FILTER command.

Notes

- ***Logical records:*** DBCLEARFILTER() affects the logical visibility of records in the current work area. For more information, refer to DBSETFILTER() and the SET FILTER command.

Examples

- The following example sets a filter, lists data as filtered, and then clears the filter:

```
USE Employee NEW
DBSETFILTER( {|| Age < 40}, "Age < 40" )
LIST Employee->Name
DBCLEARFILTER()
```

Files Library is CLIPPER.LIB.

See Also DBFILTER(), DBSETFILTER(), SET DELETED, SET FILTER

DBCLEARINDEX() function

Close all indexes for the current work area

Syntax

```
DBCLEARINDEX() → NIL
```

Returns

DBCLEARINDEX() always returns NIL.

Description

DBCLEARINDEX() closes any active indexes for the current work area. Any pending index updates are written and the index files are closed.

DBCLEARINDEX() performs the same function as the standard SET INDEX command with no indexes specified. For more information, refer to the SET INDEX command.

Examples

- The following example clears index files if any are set:

```
cFirst := "Winston"
DBUSEAREA( .T., "DBFNTX", "Sales", "Sales", .T. )
DBSETINDEX( "FIRSTNAM" )
DBSETINDEX( "LASTNAME" )
//
IF INDEXORD() > 0                  // is there an index?
   DBCLEARINDEX()                  // clear index files
ELSE
   COPY TO FILE TEMP SDF           // copy to SDF in natural
ENDIF                              // order
```

Files

Library is CLIPPER.LIB.

See Also

DBCREATEINDEX(), DBREINDEX(), DBSETINDEX(), DBSETORDER(), SET INDEX

DBCLEARRELATION() function

Clear active relations

Syntax

```
DBCLEARRELATION() → NIL
```

Returns

DBCLEARRELATION() always returns NIL.

Description

DBCLEARRELATION() clears any active relations for the current work area.

DBCLEARRELATION() performs the same function as the standard SET RELATION TO command with no clauses specified. For more information, refer to the SET RELATION command.

Examples

- The following example sets a relation, lists data, and then clears the relation:

```
USE Employee NEW
USE Department NEW INDEX Dept
//
SELECT Employee
DBSETRELATION("Department", ;
    {|| Employee->Dept}, "Employee->Dept")
LIST Employee->Name, Department->Name
DBCLEARRELATION()
```

Files

Library is CLIPPER.LIB.

See Also

DBSETRELATION(), SET RELATION

DBCLOSEALL() function

Close all occupied work areas

Syntax

```
DBCLOSEALL() → NIL
```

Returns

DBCLOSEALL() always returns NIL.

Description

DBCLOSEALL() releases all occupied work areas from use. It is equivalent to calling DBCLOSEAREA() on every occupied work area. DBCLOSEALL() has the same effect as the standard CLOSE DATABASES command. For more information, refer to the USE and CLOSE commands.

Examples

- The following example closes all work areas:

```
cLast := "Winston"
DBUSEAREA( .T., "DBFNTX", "Sales", "Sales", .T. )
DBSETINDEX( "SALEFNAM" )
DBSETINDEX( "SALELNAM" )
//
DBUSEAREA( .T., "DBFNTX", "Colls", "Colls", .T. )
DBSETINDEX( "COLLFNAM" )
DBSETINDEX( "COLLLNAM" )
//
DBSELECTAREA( "Sales" )    // select "Sales" work area
//
IF ( Sales->(DBSEEK(cLast)) )
    IF Sales->( DELETED() )
       IF RLOCK()
          Sales->( DBRECALL() )
          ? "Record deleted: ", Sales( DELETED() )
       ENDIF
    ENDIF
ELSE
    ? "Not found"
ENDIF
DBCLOSEALL()               // close all work areas
```

Files

Library is CLIPPER.LIB.

See Also

CLOSE, DBCLOSEAREA(), DBUSEAREA(), SELECT, USE

DBCLOSEAREA() function

Close a work area

Syntax

```
DBCLOSEAREA() → NIL
```

Returns

DBCLOSEAREA() always returns NIL.

Description

DBCLOSEAREA() releases the current work area from use. Pending updates are written, pending locks are released, and any resources associated with the work area are closed or released. DBCLOSEAREA() is equivalent to the standard CLOSE command or the USE command with no clauses. For more information, refer to the USE and CLOSE commands.

Examples

- The following example closes a work area via an alias reference:

```
cLast := "Winston"
//
DBUSEAREA( .T., "DBFNTX", "Sales", "Sales", .T. )
DBSETINDEX( "SALEFNAM" )
DBSETINDEX( "SALELNAM" )
//
DBUSEAREA( .T., "DBFNTX", "Colls", "Colls", .T. )
DBSETINDEX( "COLLFNAM" )
DBSETINDEX( "COLLLNAM" )
//
DBSELECTAREA( "Sales" )    // select "Sales" work area
//
IF ( Sales->(DBSEEK(cLast)) )
    IF Sales->( DELETED() ) .AND. Sales->( RLOCK() )
       Sales->( DBRECALL() )
       ? "Record deleted: ", Sales( DELETED() )
    ENDIF
ELSE
    ? "Not found"
    Colls->( DBCLOSEAREA() )
ENDIF
```

Files

Library is CLIPPER.LIB.

See Also

CLOSE, DBCLOSEALL(), DBCOMMIT(), DBUSEAREA(), USE

DBCOMMIT() function

Flush pending updates

Syntax

```
DBCOMMIT() → NIL
```

Returns

DBCOMMIT() always returns NIL.

Description

DBCOMMIT() causes all updates to the current work area to be written to disk. All updated database and index buffers are written to DOS and a DOS COMMIT request is issued for the database (.dbf) file and any index files associated with the work area.

DBCOMMIT() performs the same function as the standard COMMIT command except that it operates only on the current work area. For more information, refer to the COMMIT command.

Notes

- ***Network environment:*** DBCOMMIT() makes database updates visible to other processes. To insure data integrity, issue DBCOMMIT() before an UNLOCK operation. For more information, refer to the "Network Programming" chapter in the *Programming and Utilities Guide.*
- DBCOMMIT() uses DOS interrupt 21h function 68h to perform the solid-disk write. It is up to the network operating system to properly implement this request. Check with the network vendor to see if this is supported.

Examples

- In this example, COMMIT is used to force a write to disk after a series of memory variables are assigned to field variables:

```
USE Sales EXCLUSIVE NEW
MEMVAR->Name := Sales->Name
MEMVAR->Amount := Sales->Amount
//
@ 10, 10 GET MEMVAR->Name
@ 11, 10 GET MEMVAR->Amount
READ
//
IF UPDATED()
    APPEND BLANK
    REPLACE Sales->Name WITH MEMVAR->Name
    REPLACE Sales->Amount WITH MEMVAR->Amount
    Sales->( DBCOMMIT() )
ENDIF
```

Files

Library is CLIPPER.LIB.

See Also

CLOSE, COMMIT, DBCLOSEALL(), DBCOMMITALL(), DBUNLOCK(), UNLOCK

DBCOMMITALL() function

Flush pending updates in all work areas

Syntax

```
DBCOMMITALL() → NIL
```

Returns

DBCOMMITALL() always returns NIL.

Description

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

For more information, refer to DBCOMMIT() and the COMMIT command.

Notes

- DBCOMMITALL() uses DOS interrupt 21h function 68h to perform the solid-disk write. It is up to the network operating system to properly implement this request. Check with the network vendor to see if this is supported.

Examples

- The following example writes all pending updates to disk:

```
cLast := "Winston"
//
DBUSEAREA( .T., "DBFNTX", "Sales", "Sales", .T. )
DBSETINDEX( "SALEFNAM" )
DBSETINDEX( "SALELNAM" )
//
DBUSEAREA( .T., "DBFNTX", "Colls", "Colls", .T. )
DBSETINDEX( "COLLFNAM" )
DBSETINDEX( "COLLLNAM" )

DBSELECTAREA( "Sales" )    // select "Sales" work area

IF ( Sales->(DBSEEK(cLast)) )
   IF Sales->( DELETED() ) .AND. Sales( RLOCK() )
      Sales->( DBRECALL() )
      ? "Deleted record has been recalled."
   ENDIF
ELSE
   ? "Not found"
ENDIF
//
// processing done, write updates to disk and close
DBCOMMITALL()
DBCLOSEALL()
QUIT
```

Files Library is CLIPPER.LIB.

See Also CLOSE, COMMIT, DBCLOSEALL(), DBCOMMIT(), DBUNLOCK(), UNLOCK

D

DBCREATE() function

Create a database file from a database structure array

Syntax

```
DBCREATE(<cDatabase>, <aStruct>
   [<cDriver>]) → NIL
```

Arguments

<cDatabase> is the name of the new database file, with an optional drive and directory, specified as a character string. If specified without an extension, .dbf is assumed.

<aStruct> is an array that contains the structure of *<cDatabase>* as a series of subarrays, one per field. Each subarray contains the definition of each field's attributes and has the following structure:

Field Definition Subarray

Position	Metasymbol	Dbstruct.ch
1	cName	DBS_NAME
2	cType	DBS_TYPE
3	nLength	DBS_LEN
4	nDecimals	DBS_DEC

<cDriver> specifies the replaceable database driver (RDD) to use to process the current work area. *<cDriver>* is the name of the RDD specified as a character expression. If you specify *<cDriver>* as a literal value, you must enclose it in quotes.

Returns

DBCREATE() always returns NIL.

Description

DBCREATE() is a database function that creates a database file from an array containing the structure of the file. You may create the array programmatically or by using DBSTRUCT(). DBCREATE() is similar to the CREATE FROM command which creates a new database file structure from a structure extended file. Use CREATE or COPY STRUCTURE EXTENDED commands to create a structure extended file.

Before using DBCREATE(), you must first create the <*aStruct*> array and fill it with the field definition arrays according to the structure in Field Definition Subarray table (above). There are some specific rules for creating a field definition array, including:

- Specify all field attributes with a value of the proper data type for the attribute. The decimals attribute must be specified—even for non-numeric fields. If the field does not have a decimals attribute, specify zero.

- Specify the type attribute using the first letter of the data type as a minimum. Use longer and more descriptive terms for readability. For example, both "C" and "Character" can be specified as the type attribute for character fields.

- In CA-Clipper, character fields contain up to 64,000 characters. Unlike the CREATE FROM command, DBCREATE() does not use the decimals attribute to specify the high-order part of the field length. Specify the field length directly, regardless of its magnitude.

To make references to the various elements of the field definition subarray more readable, the header file called Dbstruct.ch is supplied. It contains the #defines to assign a name to the array position for each field attribute. It is located in \CLIP53\INCLUDE.

Notes

- ***Duplicate field names:*** DBCREATE() does not check for duplicate field names. Therefore, be careful not to use the same field name twice.

- ***EG_ARG error:*** DBCREATE() generates an EG_ARG error if the file name is NIL.

Examples

- This example creates an empty array and then adds field definition subarrays using the AADD() function before creating People.dbf. You might use this technique to add field definitions to your structure array dynamically:

```
aDbf := {}
AADD(aDbf, { "Name", "C", 25, 0 })
AADD(aDbf, { "Address", "C", 1024, 0 })
AADD(aDbf, { "Phone", "N", 13, 0 })
//
DBCREATE("People", aDbf)
```

- This example performs the same types of actions but declares the structure array as a two-dimensional array, and then uses subscript addressing to specify the field definitions. It will be created using the DBFMDX RDD:

```
#include "Dbstruct.ch"
//
LOCAL aDbf[1][4]
aDbf[1][ DBS_NAME ] := "Name"
aDbf[1][ DBS_TYPE ] := "Character"
aDbf[1][ DBS_LEN ]  := 25
aDbf[1][ DBS_DEC ]  := 0
//
DBCREATE("Name", aDbf, "DBFMDX")
```

Files Library is CLIPPER.LIB, header file is Dbstruct.ch.

See Also AFIELDS()*, COPY STRUCTURE EXTENDED, CREATE FROM, DBSTRUCT()

DBCREATEINDEX() function

Create an index file

Syntax

```
DBCREATEINDEX(<cIndexName>, <cKeyExpr>,
   [<bKeyExpr>], [<lUnique>]) → NIL
```

Arguments

<cIndexName> is a character value that specifies the file name of the index file to be created.

<cKeyExpr> is a character value that expresses the index key expression in textual form.

<bKeyExpr> is a code block that expresses the index key expression in executable form.

<lUnique> is an optional logical value that specifies whether a unique index is to be created. If *<lUnique>* is omitted, the current global _SET_UNIQUE setting is used.

Returns

DBCREATEINDEX() always returns NIL.

Description

DBCREATEINDEX() creates an index for the database (.dbf) file associated with the current work area. If the work area has active indexes, they are closed. After the new index is created, it becomes the controlling index for the work area and is positioned to the first logical record.

DBCREATEINDEX() performs the same function as the standard INDEX command. For more information, refer to the INDEX command.

D

Notes

- ***Side effects:*** DBCREATEINDEX() is guaranteed to create an index that, when made active, will impose the specified logical order on the database. The key expression is not necessarily evaluated at any particular time, by any particular means, or on any particular record or series of records. If the key expression relies on information external to the database file or work area, the effect is unpredictable. If the key expression changes the state of the work area (e.g., by moving to a different record or changing the contents of a record), the effect is unpredictable.

- ***Evaluation context:*** When the key expression is evaluated, the associated work area is automatically selected as the current work area before the evaluation; the previously selected work area is automatically restored afterward.

- ***Network environment:*** DBCREATEINDEX() creates the new index for the exclusive use of the current process.

- ***Illegal expression:*** If <cKeyExpr> evaluates to an illegal expression, the code block will always evaluate to end of file (EOF).

Examples

- This example creates an index file, Name, indexed on the Name field:

```
USE Employees NEW
DBCREATEINDEX( "Name", "Name", { || Name })
```

Files Library is CLIPPER.LIB.

See Also DBCLEARINDEX(), DBREINDEX(), DBSETINDEX(), DBSETORDER(), INDEX

DBDELETE() function

Mark a record for deletion

Syntax

```
DBDELETE() → NIL
```

Returns

DBDELETE() always returns NIL.

Description

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

DBDELETE() performs the same function as the standard DELETE command with a scope of the current record. For more information, refer to the DELETE command.

Notes

- ***Logical records:*** If the global _SET_DELETED status is true (.T.), deleted records are not logically visible. That is, database operations which operate on logical records will not consider records marked for deletion. Note, however, that if _SET_DELETED is true (.T.) when the current record is marked for deletion, the record remains visible until it is no longer the current record.
- ***Network environment:*** For a shared database on a network, DBDELETE() requires the current record to be locked. For more information, refer to the "Network Programming" chapter of the *Programming and Utilities Guide.*

D

Examples

- The following example deletes a record after a successful record lock:

```
cLast := "Winston"
DBUSEAREA( .T., "DBFNTX", "Sales", "Sales", .T. )
DBSETINDEX( "LASTNAME" )
//
IF ( Sales->(DBSEEK(cLast)) )
    IF Sales->( RLOCK() )
       Sales->( DBDELETE() )
       ? "Record deleted: ", Sales->( DELETED() )
    ELSE
       ? "Unable to lock record..."
    ENDIF
ELSE
    ? "Not found"
ENDIF
```

Files Library is CLIPPER.LIB.

See Also DBRECALL(), DELETE, RECALL

DBEDIT() function

Browse records in a table layout

Syntax

```
DBEDIT([<nTop>], [<nLeft>],
   [<nBottom>], <nRight>],
   [<acColumns>],
   [<cUserFunction>],
   [<acColumnSayPictures> | <cColumnSayPicture>],
   [<acColumnHeaders> | <cColumnHeader>],
   [<acHeadingSeparators> | <cHeadingSeparator>],
   [<acColumnSeparators> | <cColumnSeparator>],
   [<acFootingSeparators> | <cFootingSeparator>],
   [<acColumnFootings> | <cColumnFooting>]) → NIL
```

Arguments

<nTop>, ***<nLeft>***, ***<nBottom>***, and ***<nRight>*** define the upper-left and lower-right coordinates of the DBEDIT() window. Row values can range from zero to MAXROW() and column positions can range from zero to MAXCOL(). If not specified, the default coordinates are 0, 0, MAXROW(), and MAXCOL().

<acColumns> is an array of character expressions containing database field names or expressions to use as column values for each row displayed. If this argument is not specified, DBEDIT() displays all fields in the current work area as columns.

<cUserFunction> is the name of a user-defined function that executes when an unrecognizable key is pressed or there are no keys pending in the keyboard buffer. Specify the function name as a character expression without parentheses or arguments. Note that the behavior of DBEDIT() is affected by the presence of this argument. Refer to the discussion below for more information.

<acColumnSayPictures> is a parallel array of picture clauses to format each column. Specifying *<cColumnSayPicture>* instead of an array displays all columns with the same format. Refer to TRANSFORM() or @...SAY for more information on pictures.

<acColumnHeaders> is a parallel array of character expressions that define the headings for each column. Specifying *<cColumnHeader>* gives the same heading for all columns. To display a multi-line heading, embed a semicolon in the heading expression where you want the string to break. If not specified, column headings are taken from the *<acColumns>* array or the field names in the current work area, if the *<acColumns>* argument is not specified.

<acHeadingSeparators> is a parallel array of character expressions that define the characters used to draw horizontal lines separating column headings from the field display area. Specifying *<cHeadingSeparator>* instead of an array uses the same heading separator for all columns. If this argument is not specified, the default separator is a double graphics line.

<acColumnSeparators> is a parallel array of character expressions that define the characters used to draw vertical lines separating the columns. Specifying *<cColumnSeparator>* instead of an array uses the same separator for all columns. If this argument is not specified, the default separator is a single graphics line.

<acFootingSeparators> is a parallel array of character expressions that define the characters used to draw horizontal lines separating column footings from the field display area. Specifying *<cFootingSeparator>* instead of an array uses the same footing separator for all columns. If this argument is not specified, there is no footing separator.

<acColumnFootings> is a parallel array of character expressions that define footings for each column. Specifying *<cColumnFooting>* instead of an array gives the same footing for all columns. To display a multi-line footing, embed a semicolon in the footing expression where you want the string to break. If this argument is not specified, there are no column footings.

Returns

DBEDIT() always returns NIL.

Description

DBEDIT() is a user interface and compatibility function that displays records from one or more work areas in a table form. The DBEDIT() window display is a grid of cells divided into columns and rows. Columns correspond to database fields and rows correspond to database records. Each column is defined by an element of the <*acColumns*> array. The display width of each column is determined by the evaluation of the column expression in <*acColumns*> array or the column picture specified in the <*acColumnSayPictures*> array.

All cursor movement keys are handled within DBEDIT(), including Page up, Page down, Home, End, the four arrow keys, and all Ctrl key combinations that produce cursor movement. The navigation keys that DBEDIT() responds to when a user function argument is not specified are listed in the Active Keys table below:

DBEDIT() Active Keys

Key	Action
Up arrow	Up one row
Down arrow	Down one row
Left arrow	Column left
Right arrow	Column right
Ctrl+Left arrow	Pan left one column
Ctrl+Right arrow	Pan right one column
Home	Leftmost current screen column
End	Rightmost current screen column
Ctrl+Home	Leftmost column
Ctrl+End	Rightmost column
PgUp	Previous screen
PgDn	Next screen
Ctrl+PgUp	First row of current column
Ctrl+PgDn	Last row of current column
Return	Terminate DBEDIT()
Esc	Terminate DBEDIT()

When the user function argument (<*cUserFunction*>) is specified, all keys indicated in the Active Keys table are active with the exception of Esc and Return. When DBEDIT() calls the user function, it automatically passes two arguments:

- The current mode passed as a numeric value
- The index of the current column in <*acColumns*> passed as a numeric value

The mode parameter indicates the current state of DBEDIT() depending on the last key executed. The possible mode values are listed in the DBEDIT() Modes table below:

DBEDIT() Modes

Status	Dbedit.ch	Description
0	DE_IDLE	Idle, any cursor movement keystrokes have been handled and no keystrokes are pending
1	DE_HITTOP	Attempt to cursor past top of file
2	DE_HITBOTTOM	Attempt to cursor past bottom of file
3	DE_EMPTY	No records in work area
4	DE_EXCEPT	Key exception

The index parameter points to the position of the current column definition in the <*acColumns*> array. If <*acColumns*> is not specified, the index parameter points to the position of the field in the current database structure. Access the field name using FIELD().

A user-defined function must return a value that indicates to DBEDIT() the action to perform. The User Function Return Values table below lists the possible return values and the corresponding actions:

DBEDIT() User Function Return Values

Value	Dbedit.ch	Description
0	DE_ABORT	Abort DBEDIT()
1	DE_CONT	Continue DBEDIT()
2	DE_REFRESH	Force reread/repaint and continue; after repaint, process keys and go to idle

A number of instances affect calls to the user function:

- A key exception occurs. This happens when DBEDIT() fetches a keystroke that it does not recognize from the keyboard. Any pending keys remain in the keyboard buffer until fetched within the user function or until DBEDIT() continues.
- DBEDIT() enters the idle mode (i.e., all pending keys have been processed). This happens when the keyboard is empty or after a screen refresh. In this instance, there is one call to the user function and then DBEDIT() waits for a key.
- Beginning or end of file is encountered. This is the same as idle. All executable keys are performed, and there is one call to the user function with the appropriate status message.

Note that when DBEDIT() is first executed, all keys pending in the keyboard buffer are executed and then DBEDIT() enters the idle mode with a user function call. If no keys are pending, the idle mode is immediate.

The user function should handle all modes and status messages received from DBEDIT().

A user-defined function must ensure that the DBEDIT() status is equivalent to DE_EXCEPT (4); otherwise, the value of LASTKEY() is meaningless and a Return value of DE_REFRESH (2) will place the application into an endless loop. For example:

```
FUNCTION DBEditFunc ( nMode, nColumnPos )
   LOCAL RetVal := DE_CONT

   IF ( nMode == DE_EXCEPT )
      IF ( LASTKEY() == K_F5 )
            RetVal := DE_REFRESH
      ENDIF
   ENDIF
RETURN(  RetVal )
```

DBEDIT() is fully re-entrant, which means you can make nested calls to it. Using this feature, you can have multiple browse windows on the screen at the same time.

DBEDIT() is a compatibility function and, therefore, no longer recommended as a programmable browse facility. As such, it is superseded by the TBrowse object class. For more information, refer to TBrowse class in this chapter.

Examples

- This example demonstrates a generic call to DBEDIT():

```
USE Names NEW
DBEDIT()
```

- This example demonstrates calling DBEDIT() with a user function:

```
#include "Dbedit.ch"
#include "Inkey.ch"

// Array must be visible to other user-defined programs in
// program

STATIC acColumns := {}

PROCEDURE Main()

    USE Names NEW
    INDEX ON Names->Lastname + Names->FirstName TO Names

    CLS

    acColumns := { "LastName", "FirstName" }

    DBEDIT( 5, 5, 20, 70, acColumns, "UserFunc" )

RETURN

FUNCTION UserFunc( nMode, nCol )
    LOCAL nKey := LASTKEY()
    LOCAL nRetVal := DE_CONT         // Default return value

    DO CASE
    CASE nMode == DE_IDLE
       nRetVal := IdleFunc()
    CASE nMode == DE_HITTOP
       TONE( 100, 3 )
    CASE nMode == DE_HITBOTTOM
       TONE( 100, 3 )
       nRetVal := AppendFunc( nKey )
    CASE nMode == DE_EMPTY
       nRetVal := EmptyFunc()
    CASE nMode == DE_EXCEPT
       nRetVal := ExceptFunc( nKey, nCol )
    OTHERWISE
       TONE( 100, 3 )
    ENDCASE

RETURN nRetVal
```

```
FUNCTION AppendFunc( nKey )
   LOCAL nRetVal := DE_CONT            // Default return value

   IF nKey == K_DOWN                   // If DOWN ARROW
      APPEND BLANK                     // Append blank record
// Note: The appended record will appear at the top of the
//       DBEDIT() screen when the database file is indexed.

      nRetVal := DE_REFRESH            // Refresh screen
   ENDIF

RETURN nRetVal

FUNCTION ExceptFunc( nKey, nCol )
   LOCAL nRetVal := DE_CONT            // Default return value

   DO CASE
   CASE nKey == K_ESC                  // If ESCAPE
      nRetVal := DE_ABORT              // Exit
   CASE nKey == K_RETURN               // If RETURN
      nRetVal := EditFunc( nCol )      // Function to edit
                                       // field

   // Toggle DELETED status
   CASE nKey == K_DEL .AND. LASTREC() != 0   // DELETE pressed
      IF DELETED()
         RECALL
      ELSE
         DELETE
      ENDIF
   OTHERWISE
         TONE( 100, 1 )
   ENDCASE

RETURN nRetVal


FUNCTION EditFunc( nCol )
   LOCAL cIndexVal          // Value of current key expression
   LOCAL nRetVal            // Return value
   LOCAL nField             // Position of current field
   LOCAL cFieldVal          // Value of current field
   LOCAL nCursSave          // Preserve state of cursor

   // This will return an error if no index is open
   cIndexVal := &( INDEXKEY(0) )

   nField := FIELDPOS( acColumns[nCol] )

   IF nField != 0
      nCursSave := SETCURSOR()            // Save state of cursor
      SETCURSOR(1)                        // Change cursor shape
      cFieldVal := FIELDGET( nField )     // Save contents
                                          // of field
      @ ROW(), COL() GET cFieldVal        // GET new value
      READ
      FIELDPUT( nField, cFieldVal )       // REPLACE with
                                          // new value
      SETCURSOR( nCursSave )              // Restore cursor
                                          // shape
   ENDIF
```

```
IF cIndexVal != &( INDEXKEY(0) )        // If key expression
                                        // changed
      nRequest := DE_REFRESH            // Refresh screen
   ELSE                                 // Otherwise
      nRequest := DE_CONT               // Continue
   ENDIF

RETURN nRequest

FUNCTION IdleFunc()
   // Idle routine
RETURN DE_CONT

FUNCTION EmptyFunc()
   // Empty Records routine
RETURN DE_CONT
```

Files Library is EXTEND.LIB, header files are Dbedit.ch and Inkey.ch.

See Also @...GET, ACHOICE(), BROWSE(), MEMOEDIT(), READ, TRANSFORM()

DBEVAL() function

Evaluate a code block for each record matching a scope and condition

Syntax

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

Arguments

<bBlock> is a code block to execute for each record processed.

<bForCondition> is an optional condition specified as a code block that is evaluated for each record in the scope. It provides the same functionality as the FOR clause of record processing commands.

<bWhileCondition> is an optional condition specified as a code block that is evaluated for each record from the current record until the condition returns false (.F.). It provides the same functionality as the WHILE clause of record processing commands.

<nNextRecords> is an optional number that specifies the number of records to process starting with the current record. It is the same as the NEXT clause.

<nRecord> is an optional record number to process. If this argument is specified, *<bBlock>* will be evaluated for the specified record. This argument is the same as the RECORD clause.

<lRest> is an optional logical value that determines whether the scope of DBEVAL() is all records, or, starting with the current record, all records to the end of file. This argument corresponds to the REST and ALL clauses of record processing commands. If true (.T.) , the scope is REST; otherwise, the scope is ALL records. If *<lRest>* is not specified the scope defaults to ALL.

Returns

DBEVAL() always returns NIL.

Description

DBEVAL() is a database function that evaluates a single block for each record within the current work area that matches a specified scope and/or condition. On each iteration, DBEVAL() evaluates the specified block. All records within the scope or matching the condition are processed until the end of file is reached.

By default, DBEVAL() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression.

DBEVAL() is similar to AEVAL() which applies a block to each element in an array. Like AEVAL(), DBEVAL() can be used as a primitive for the construction of user-defined commands that process database files. In fact, many of the standard CA-Clipper database processing commands are created using DBEVAL().

Refer to the Code Blocks section in the "Basic Concepts" chapter of the *Programming and Utilities Guide* for more information on the syntax and theory of code blocks; and refer also to the Database System section in the same chapter for information on record scoping and conditions. Also refer to the CA-Clipper standard header file, Std.ch, found in \CLIP53\INCLUDE for examples of CA-Clipper database command definitions that use DBEVAL().

Examples

- This example uses DBEVAL() to implement Count(), a user-defined function that counts the number of records in a work area matching a specified scope. The scope is passed as an array to Count(). To make the example more interesting, there is a user-defined command to create the scope array, thereby allowing you to specify the scope in a familiar form. Additionally, there is a set of manifest constants that define the attributes of the scope object.

```
// Scope command definition
#command CREATE SCOPE <aScope> [FOR <for>] ;
    [WHILE <while>] [NEXT <next>] [RECORD <rec>] ;
    [<rest:REST>] [ALL];
=>;
    <aScope> := { <{for}>, <{while}>, <next>, ;
       <rec>, <.rest.> }
//
// Scope attribute constants
#define FOR_COND     1
#define WHILE_COND      2
#define NEXT_SCOPE      3
#define REC_SCOPE    4
#define REST_SCOPE      5
//
// Create a scope and count records using it
LOCAL mySet, myCount
USE Customer NEW
CREATE SCOPE mySet FOR Customer = "Smith" WHILE ;
          Zip > "90000"
myCount := Count( mySet )
RETURN

FUNCTION Count( aScope )
    LOCAL nCount := 0
    DBEVAL( {|| nCount++},;
       aScope[ FOR_COND ],;
       aScope[ WHILE_COND ],;
       aScope[ NEXT_SCOPE ],;
       aScope[ REC_SCOPE ],;
       aScope[ REST_SCOPE ];
    )
    RETURN nCount
```

Files Library is CLIPPER.LIB

See Also AEVAL(), EVAL()

DBF()* function

Return current alias name

Syntax

```
DBF() → cAlias
```

Returns

DBF() returns the alias of the current work area as a character string. If there is no active database file in the current work area, DBF() returns a null string ("").

Description

DBF() is a compatibility function that replicates the DBF() function in dBASE III PLUS. CA-Clipper implements it by invoking the ALIAS() function without an argument.

DBF() is a compatibility function and, therefore, no longer recommended. It is superseded entirely by the ALIAS() function.

Files

Library is EXTEND.LIB, source file is SOURCE\SAMPLE\DBF.PRG

See Also

ALIAS(), USED()

DBFIELDINFO() function

Return and optionally change information about a field

Syntax

```
DBFIELDINFO(<nInfoType>,
   <nFieldPos>,
   [<expNewSetting>]) → uCurrentSetting
```

Arguments

<nInfoType> determines the type of information as specified by the constants below. Note, however, that not all constants are supported for all RDDs, nor are all constants supported by all field types. These constants are defined in the Dbstruct.ch header file, which must be included (#include) in your application.

Field Information Type Constants

Constant	Description
DBS_BLOB_LEN	Returns the storage length of the data in a BLOB (memo) file.
DBS_BLOB_OFFSET	Returns the file offset of the data in a BLOB (memo) file.
DBS_BLOB_POINTER	Returns a numeric pointer to the data in a blob file. This pointer can be used with BLOBDIRECTGET(), BLOBDIRECTIMPORT(), etc.
DBS_BLOB_TYPE	Returns the data type of a BLOB (memo) field. This is more efficient than using TYPE() or VALTYPE() since the data itself does not have to be retrieved from the BLOB file in order to determine the type.
DBS_DEC	Number of decimal places for the field.
DBS_LEN	Length of the field.
DBS_NAME	Name of the field.
DBS_TYPE	Data type of the field.

<nFieldPos> is the position of the field in the database file structure.

<expNewSetting> is reserved for CA-Clipper future use. It can be omitted or specified as NIL.

Returns

DBFIELDINFO() returns the current setting.

Description

DBFIELDINFO() retrieves information about the state of a field. By default, this function operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression.

The field information that is available is defined by the RDD.

To support RDDs for other database models (such as dictionary-based databases) that store more information about each field or column, the CA-Clipper 5.3 RDD API has been enhanced. The DBFIELDINFO() is designed to allow for additional <*nInfoType*> values that can be defined by third-party RDD developers.

Examples

- The following example uses DBFIELDINFO() to retrieve field information:

```
#include Dbstruct.ch

QOut(DBFIELDINFO(DBS_NAME, 1))   // Same as FIELDNAME(1)

FUNCTION DBOUTSTRUCT()
    LOCAL aStruct := {}
    LOCAL nFcount, i

    nFcount := FCOUNT()
    FOR i := 1 TO nFcount
       AADD(aStruct, {FIELDNAME(i), ;
                      DBFIELDINFO(DBS_TYPE, i), ;
                      DBFIELDINFO(DBS_LEN, i),  ;
                      DBFIELDINFO(DBS_DEC, i)})
    NEXT
    RETURN aStruct
```

Files Library is CLIPPER.LIB, header file is Dbstruct.ch.

See Also DBINFO(), DBORDERINFO(), DBRECORDINFO()

DBFILEGET() function

Insert the contents of a field into a file.

Syntax

```
DBFILEGET(<nFieldPos>, <cTargetFile>, <nMode>)
   → lSuccess
```

Arguments

<nFieldPos> is the position of the field in the database file structure.

<cTargetFile> is the name of the file where the field data will be written, including an optional drive, directory and extension. See SetDefault() and SetPath() for file searching and creation rules. No default extension is assumed.

If *<cTargetFile>* does not exist, it is created. If it exists, this function attempts to open the file in exclusive mode and if successful, the file is written to without warning or error. If access is denied because, for example, another proess is using the file, NetErr() is set to true (.T.).

<nMode> is a constant defining the copy mode, as shown in the table below:

DBFILEGET() Constants

Constant	Description
FILEGET_APPEND	Appends to the file.
FILEGET_OVERWRITE	Overwrites the file. This is the default.

Returns

DBFILEGET() returns true (.T.) if successful; otherwise it returns false (.F.).

Description

DBFILEGET() provides a mechanism for copying the contents of a field into a file. By default, this function operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression.

DBFILEGET() is used in conjunction with DBFILEPUT() to transfer data back and forth between files and database fields.

Examples

- This example exports the contents of a field that stores a picture to a .GIF file, so that the file can be programmatically displayed:

```
FUNCTION ShowPix()
   LOCAL cPixFile := "picture.gif"
   LOCAL nPos

   // Customer database with a picture of each
   // customer stored in a field called "Pix"
   USE customer NEW VIA "DBFCDX"
   nPos := FieldPos("Pix")

   // Export the file's data for the current Pix field
   IF ! DBFILEGET(nPos, cPixFile, FILEGET_OVERWRITE )
      Alert("Export of picture " + cPixFile +
      " failed!")

   ELSE
      // Code for displaying picture would go here
   ENDIF
```

Files Library is CLIPPER.LIB, header is Dbinfo.ch

See Also DBFILEPUT()

DBFILEPUT() function

Insert the contents of a file into a field.

Syntax

```
DBFILEPUT(<nFieldPos>, <cSourceFile>)
   → lSuccess
```

Arguments

<nFieldPos> is the position of the field in the database file structure.

<cSourceFile> is the name of the file containing the data to insert into the specified field, including an optional drive, directory and extension. See SETDEFAULT() and SETPATH() for file searching rules. No default extension is assumed.

This function attempts to open ***<cSourceFile>*** in shared mode. If the file does not exist, a runtime error is raised. If the file is successfully opened, the operation proceeds. If access is denied because, for example, another process has exclusive use of the file, NETERR() is set to true (.T.).

Note: There are no restrictions on the size of ***<cSourceFile>*** except that you must have enough disk space to make the copy.

Returns

DBFILEPUT() returns true (.T.) if successful; otherwise, it returns false (.F.).

Description

DBFILEPUT() provides a mechanism for copying the contents of a file into a field. By default, this function operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression.

DBFILEPUT() is used in conjunction with DBFILEGET() to transfer data back and forth between files and database fields. You can use DBFILEPUT() with a variety of field types, including graphics images, word processor files, and printer fonts. These two functions are excellent for creating databases of documents, graphics, sounds, etc.

Note: DBFIELDINFO (DBS_BLOB_TYPE, *<nFieldPos>*) will return "C" (string) for any memo field created using DBFILEPUT().

Examples

- This example imports information from a word processing document into a field, then uses BLOBGET() to extract the first 25 characters of the field:

```
FUNCTION Populate()
   USE customer NEW VIA "DBFCDX"
   DO WHILE .NOT. EOF()
      GetPix( "Pix", Substr(LastName, 1, 4) + CustID)
      Customer->DBSkip()
   ENDDO

FUNCTION GetPix(cPixField, cPixFile)
   LOCAL nPos
   nPos := FieldPos(cPixField)

   // Import the picture field into the indicated field
   IF ! DBFILEPUT(nPos, cPixFile)
      Alert("Import of picture " + cPixFile + ;
            " failed!")
   ENDIF
```

Files Library is CLIPPER.LIB

See Also DBFILEGET()

DBFILTER() function

Return the current filter expression as a character string

Syntax

```
DBFILTER() → cFilter
```

Returns

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

Description

DBFILTER() is a database function used to save and re-execute an active filter by returning the filter expression as a character string that can be later recompiled and executed using the macro operator (&). This function operates like the DBRELATION() and DBRSELECT() functions which save and re-execute the linking expression of a relation within a work area.

Since each work area can have an active filter, DBFILTER() can return the filter expression of any work area. This is done by referring to DBFILTER() within an aliased expression as demonstrated below.

Notes

- ***Declared variables:*** A character string returned by DBFILTER() may not operate correctly when recompiled and executed using the macro operator (&) if the original filter expression contained references to local or static variables, or otherwise depended on compile-time declarations.

Examples

- This example opens two database files, sets two filters, and then displays the filter expressions for both work areas:

```
USE Customer INDEX Customer NEW
SET FILTER TO Last = "Smith"
USE Invoices INDEX Invoices NEW
SET FILTER TO CustId = "Smi001"
SELECT Customer
//
? DBFILTER()                    // Result: Last = "Smith"
? Invoices->(DBFILTER())        // Result: Custid = "Smi001"
```

- This user-defined function, CreateQry(), uses DBFILTER() to create a memory file containing the current filter expression in the private variable cFilter:

```
FUNCTION CreateQry( cQryName )
   PRIVATE cFilter := DBFILTER()
   SAVE ALL LIKE cFilter TO (cQryName + ".qwy")
   RETURN NIL
```

- You can later RESTORE a query file with this user-defined function, SetFilter():

```
FUNCTION SetFilter()
PARAMETER cQryName
   RESTORE FROM &cQryName..qwy ADDITIVE
   SET FILTER TO &cFilter.
   RETURN NIL
```

Files Library is CLIPPER.LIB.

See Also DBRELATION(), DBRSELECT(), SET FILTER

DBGOBOTTOM() function

Move to the last logical record

Syntax

```
DBGOBOTTOM() → NIL
```

Returns

DBGOBOTTOM() always returns NIL.

Description

DBGOBOTTOM() moves to the last logical record in the current work area.

DBGOBOTTOM() performs the same function as the standard GO BOTTOM command. For more information, refer to the GO command.

Notes

- ***Logical records:*** DBGOBOTTOM() operates on logical records. If there is an active index, DBGOBOTTOM() moves to the last record in indexed order. If a filter is set, only records which meet the filter condition are considered.
- ***Controlling order:*** If more than one index is active in the work area, the operation is performed using the controlling order as set by DBSETORDER() or the SET ORDER command. For more information, refer to the SET ORDER command.
- ***Network environment:*** For a shared file on a network, moving to a different record may cause updates to the current record to become visible to other processes. For more information, refer to the "Network Programming" chapter in the *Programming and Utilities Guide*. This function will not affect the locked status of any record.

Examples

- The following example uses DBGOBOTTOM() to position the record pointer on the last logical record:

```
cLast := "Winston"
DBUSEAREA( .T., "DBFNTX", "Sales", "Sales", .T. )
DBSETINDEX( "LASTNAME" )
//
Sales->( DBGOBOTTOM() )
IF ( Sales->Last == "Winston" )
    IF RLOCK()
       Sales->( DBDELETE() )
       ? "Record deleted: ", Sales->( DELETED() )
    ELSE
       ? "Unable to lock record..."
    ENDIF
END
```

Files

Library is CLIPPER.LIB.

See Also

BOF(), DBGOTOP(), DBSEEK(), DBSKIP(), EOF(), GO

DBGOTO() function

Position record pointer to a specific identity

Syntax

```
DBGOTO(<xIdentity>) → NIL
```

Arguments

<xIdentity> is a unique value guaranteed by the structure of the data file to reference a specific item in a data source (database). In an Xbase data structure (.dbf), *<xIdentity>* is the record number. In other data formats, *<xIdentity>* is the unique primary key value. *<xIdentity>* could be an array offset or virtual memory handle if the data set is in memory instead of on disk.

Returns

DBGOTO() always returns NIL.

Description

DBGOTO() is a database function that positions the record pointer in the current work area at the specified *<xIdentity>*. In an Xbase data structure, this identity is the record number because every record, even an empty record, has a record number. In non-Xbase data structures, identity may be defined as something other than record number.

Examples

- This example saves the current record number, searches for a key, and then restores the record pointer to the saved position:

```
FUNCTION KeyExists( xKeyExpr )

   LOCAL nSavRecord := RECNO()      // Save the current record
                                    // pointer position
   LOCAL lFound

   SEEK xKeyExpr
   IF ( lFound := FOUND() )
   .
   .< statements >
   .
   ENDIF

   DBGOTO( nSavRecord )             // Restore the record
                                    // pointer position
   RETURN ( lFound )
```

See Also

BOF(), DBGOBOTTOM(), DBGOTOP(), DBSEEK(), DBSKIP(), EOF(), GO

DBGOTOP() function

Move to the first logical record

Syntax

```
DBGOTOP() → NIL
```

Returns

DBGOTOP() always returns NIL.

Description

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

DBGOTOP() performs the same function as the standard GO TOP command. For more information, refer to the GO TOP command.

Notes

- ***Logical records:*** DBGOTOP() operates on logical records. If there is an active index, DBGOTOP() moves to the first record in indexed order. If a filter is set, only records which meet the filter condition are considered.
- ***Controlling order:*** If more than one index is active in the work area, the operation is performed using the controlling order as set by DBSETORDER() or the SET ORDER command. For more information, refer to the SET ORDER command.
- ***Network environment:*** For a shared file on a network, moving to a different record may cause updates to the current record to become visible to other processes. For more information, refer to the "Network Programming" chapter in the *Programming and Utilities Guide*. This function will not affect the locked status of any record.

Examples

- This example demonstrates the typical use of DBGOTOP():

```
DBGOTOP()
WHILE ( !EOF() )
    ? FIELD->Name
    DBSKIP()
END
```

Files

Library is CLIPPER.LIB.

See Also

BOF(), DBGOBOTTOM(), DBSEEK(), DBSKIP(), EOF(), GO

DBINFO() function

Return and optionally change information about a database file opened in a work area

Syntax

```
DBINFO(<nInfoType>, [<expNewSetting>])
   → uCurrentSetting
```

Arguments

<*nInfoType*> determines the type of information, as specified by the constants below. Note, however, that not all constants are supported for all RDDs. These constants are defined in the Dbinfo.ch header file, which must be included (#include) in your application.

File Information Type Constants

Constant	Description
DBI_ALIAS	Alias name of the work area as a string.
DBI_BLOB_DIRECT_LEN	Returns the storage length of a data item in a BLOB file. <*expNewSetting*> must specify a valid BLOB pointer obtained from DBFIELDINFO (DBS_BLOB_POINTER,<*nFieldpos*>), BLOBDIRECTPUT() or BLOBDIRECTIMPORT().
DBI_BLOB_DIRECT_TYP E	Returns the data type of a data item in a BLOB file. <*expNewSetting*> must specify a valid BLOB pointer obtained from DBFIELDINFO (DBS_BLOB_POINTER,<*nFieldpos*>), BLOBDIRECTPUT(), or BLOBDIRECTIMPORT().
DBI_BLOB_INTEGRITY	Tests a BLOB file for the integrity of its internal tables and returns a logical value indicating the success, true (.T.) or failure, false (.F.) of the integrity check. If the integrity check fails, you can run DBINFO (DBI_BLOB_RECOVER) which will automatically correct the BLOB file's tables. Checking a BLOB file's integrity is a disk intensive operation and should only be performed when the file's integrity is in question.

File Information Type Constants (cont.)

Constant	Description
DBI_BLOB_OFFSET	Returns the file offset of a data item in a BLOB file. <*expNewSetting*> must specify a valid BLOB pointer obtained from DBFIELDINFO (DBS_BLOB_POINTER,<*nFieldpos*>), BLOBDIRECTPUT(), or BLOBDIRECTIMPORT().
DBI_BLOB_RECOVER	Recovers a damaged BLOB file by correcting its internal tables. You should run this function only is DBINFO (DBI_BLOB_INTEGRITY) returns false (.F.). Note that this function can only correct the BLOB file's internal tables, it cannot restore any data that may have become corrupted.
DBI_BOF	Logical value indicating the work area's beginning of file status (see BOF()).
DBI_CANPUTREC	Logical value indicating whether the work area supports putting records.
DBI_CHILDCOUNT	Number of relations set from this work area.
DBI_DB_VERSION	String containing version information of the host RDD. If the optional <*expNewSetting*> parameter is provided, and it is one (1), the result is a more detailed version of the version being returned.
DBI_DBFILTER	Filter expression as a string (see DBFILTER()).
DBI_EOF	Logical value indicating the work area's end of file status (see EOF()).
DBI_FCOUNT	Number of fields (see FCOUNT()).
DBI_FILEHANDLE	Integer representing the DOS file handle for this database file.
DBI_FOUND	Logical value indicating the success or failure of the last seek operation in the work area (see FOUND()).
DBI_FCOUNT	Number of fields (see FCOUNT()).
DBI_FULLPATH	Returns the full path name of the opened database file.
DBI_GETDELIMITER	Default delimiter.
DBI_GETHEADERSIZE	Header size of the file (see HEADER()).
DBI_GETLOCKARRAY	Array of locked records.
DBI_GETRECSIZE	Record size of the file (see RECSIZE()).

File Information Type Constants (cont.)

Constant	Description
DBI_ISDBF	Logical value indicating whether the RDD provides support for the .dbf file format.
DBI_ISFLOCK	File lock status.
DBI_LASTUPDATE	Last date on which the file was updated (see LUPDATE()).
DBI_LOCKCOUNT	Number of locked records.
DBI_LOCKOFFSET	Current locking offset as a numeric value.
DBI_MEMOBLOCKSIZE	Block size for the memo file associated with this database.
DBI_MEMOEXT	Default extension for the memo file associated with this database.
DBI_MEMOHANDLE	Integer representing the DOS file handle for the memo file associated with this database file.
DBI_RDD_VERSION	String containing version information of the RDD for this database. If the optional *<expNewSetting>* parameter is provided, and it is one (1), the result is a more detailed version of the version being returned.
DBI_SETDELIMITER	Default delimiter.
DBI_SHARED	Shared flag value.
DBI_TABLEEXT	Database file extension.
DBI_VALIDBUFFER	Logical value indicating whether the current buffer is valid.

Important! *DBI_USER is a constant that returns the minimum value that third-party RDD developers can use for defining new <nInfoType> parameters. Values less than DBI_USER are reserved for Computer Associates Development.*

<expNewSetting> is reserved for RDDs that allow the file information to be changed, in addition to being retrieved. None of the RDDs supplied with CA-Clipper support this argument. It can be omitted or specified as NIL.

Returns

DBINFO() returns the current setting if <*expNewSetting*> is not specified. If <*expNewSetting*> is specified, the previous setting is returned.

Description

DBINFO() retrieves information about a database file. By default, this function operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression.

Examples

- The following examples return work area information:

```
#include Dbinfo.ch

? DBINFO(DBI_GETHEADERSIZE)      // Same as HEADER()
? DBINFO(DBI_LASTUPDATE)         // Same as LUPDATE()
? DBINFO(DBI_GETRECSIZE)         // Same as RECSIZE()
? DBINFO(DBI_FILEHANDLE)         // DOS file handle for
                                 // the current database file
```

Files

Library is CLIPPER.LIB, header file is Dbinfo.ch.

DBORDERINFO() function

Return and optionally change information about orders and index files

Syntax

```
DBORDERINFO(<nInfoType>,   [<cIndexFile>],
   [<cOrder> | <nPosition>],
   [<expNewSetting>]) → uCurrentSetting
```

Arguments

<nInfoType> determines the type of information as specified by the constants below. Note, however, that not all constants are supported for all RDDs. These constants are defined in the Dbinfo.ch header file, which must be included (#include) in your application.

Order/Index Information Type Constants

Constant	Description
DBOI_CONDITION	For condition of the specified order as a string.
DBOI_CUSTOM	Logical flag indicating whether the specified order is custom-built (for RDDs that support custom-built orders). Note that although you can turn the custom-built flag on for a standard order by specifying true (.T.) for the *<uNewSetting>* argument, you cannot turn a custom-built order into a standard order. Specifying false (.F.) for *<uNewSetting>* is the same as not specifying the argument at all—both return the current setting.
DBOI_EXPRESSION	Order key expression of the specified order as a string.
DBOI_FILEHANDLE	Handle of the specified index file as a number.
DBOI_HPLOCKING	Logical flag indicating whether the specified index file uses the high performance index locking schema.
DBOI_INDEXEXT (DBOI_BAGEXT)	Default index file extension as a string.
DBOI_INDEXNAME (DBOI_BAGNAME)	Name of the specified index file as a string.

Order/Index Information Type Constants (cont.)

Constant	Description
DBOI_ISCOND	Logical flag that determines whether the specified order was defined using a FOR condition.
DBOI_ISDESC	Logical flag that determines if the specified order is descending. For drivers that support dynamically setting the descending flag at runtime, specify the new value as a logical, using DBORDERINFO(DBOI_ISDESC, [<*cIndexFile*>], [<*cOrder*> | <*nPosition*>], <*lNewSetting*>). The current setting is returned before it is changed.
DBOI_KEYADD	Logical flag indicating whether a key has been successfully added to the specified custom-built order.
DBOI_KEYCOUNT	Number of keys in the specified order.
DBOI_KEYDEC	Number of decimals in the key of the specified order.
DBOI_KEYDELETE	Logical flag indicating whether a key has been successfully deleted from the specified custom-built order.
DBOI_KEYGOTO	Logical flag indicating whether the record pointer has been successfully moved to a record specified by its logical record number in the controlling order.
DBOI_KEYSINCLUDED	Number of keys included in the specified order so far. This is primarily useful for conditional orders. It can be used during the status display process (with the EVAL clause of the INDEX command).
DBOI_KEYSIZE	Size of the key in the specified order as a number.
DBOI_KEYTYPE	Data type of the key in the specified order as a string.
DBOI_KEYVAL	Key value of the current record from the controlling order.
DBOI_LOCKOFFSET	Locking offset for the specified index file as a numeric value.

Order/Index Information Type Constants (cont.)

Constant	Description
DBOI_NAME	Name of the specified order as a string.
DBOI_NUMBER	Numeric position of the specified order in the order list.
DBOI_ORDERCOUNT	Number of orders in the specified index file.
DBOI_POSITION	Logical record number of the current record within the specified order.
DBOI_RECNO	Physical record number of the current record within the specified order.
DBOI_SCOPEBOTTOM	Bottom boundary of the scope (as a number) for the specified order.
DBOI_SCOPEBOTTOMCLEAR	Clears the bottom boundary of the scope for the specified order.
DBOI_SCOPETOP	Top boundary of the scope (as a number) for the specified order.
DBOI_SCOPETOPCLEAR	Clears the top boundary of the scope for the specified order.
DBOI_SETCODEBLOCK	Key for the specified order as a code block.
DBOI_SKIPUNIQUE	Logical flag indicating whether the record pointer has been successfully moved to the next or previous unique key in the controlling order.
DBOI_UNIQUE	Logical flag indicating whether the specified order has the unique attribute set.

Important! *DBOI_USER is a constant that returns the minimum value that third-party RDD developers can use for defining new <nInfoType> parameters. Values less than DBOI_USER are reserved for Computer Associates Development.*

<cIndexFile> is the name of an index file, including an optional drive and directory (no extension should be specified). Use this argument with *<cOrder>* to remove ambiguity when there are two or more orders with the same name in different index files.

<cOrder> | ***<nPosition>*** is the name of the order about which you want to obtain information or a number representing its position in the order list. For single-order index files, the order name is the eight-letter index file name. Using the order name is the preferred method since the position may be difficult to determine using multiple-order index files. Invalid values are ignored. If no index file or order is specified, the controlling order is assumed.

<expNewSetting> is reserved for RDDs that allow the file information to be changed, in addition to being retrieved. None of the RDDs supplied with CA-Clipper support this argument. It can be omitted or specified as NIL.

Returns

If *<expNewSetting>* is not specified, DBORDERINFO() returns the current setting. If *<expNewSetting>* is specified, the previous setting is returned.

Description

DBORDERINFO() retrieves information about the orders and index files. By default, DBORDERINFO() operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression.

Examples

- This example uses DBOI_NAME to save the current controlling order. After changing to a new controlling order, it uses the saved value to restore the original order:

```
#include Dbinfo.ch

USE Customer INDEX Name, Serial NEW
cOrder := DBORDERINFO(DBOI_NAME)                // Name
Customer->DBSETORDER("Serial")
? DBORDERINFO(DBOI_NAME)                        // Serial
Customer->DBSETORDER(cOrder)
? DBORDERINFO(DBOI_NAME)                        // Name
```

- This example uses aliased expressions to return the default index file extension (using DBOI_INDEXEXT) in two different work areas:

```
#include Dbinfo.ch

USE Sales INDEX All_Sales VIA "DBFCDX" NEW
USE Customer INDEX Name, Serial VIA "DBFNTX" NEW
? Sales->DBORDERINFO(DBOI_INDEXEXT)                  // .CDX
? Customer->DBORDERINFO(DBOI_INDEXEXT)               // .NTX
```

- In this example, DBORDERINFO(DBOI_INDEXEXT) checks for the existence of the Customer index file independent of the RDD linked into the current work area:

```
#include Dbinfo.ch

USE Customer NEW
IF !FILE( "Customer" + DBORDERINFO(DBOI_INDEXEXT))
      Customer->DBCREATEINDEX("Customer", "CustName",;
                              {||Customer->CustName} )
ENDIF
```

D

- This example accesses the key expression of several orders from the same index file:

```
#include Dbinfo.ch

USE Customer INDEX All_Cust VIA "DBFMDX" NEW
Customer->DBSETORDER("Serial")
? DBORDERINFO(DBOI_EXPRESSION,, "Name")
// Result: key expression for name order
? DBORDERINFO(DBOI_EXPRESSION,, "Serial")
// Result: key expression for serial order
```

- This example uses DBORDERINFO() as part of a TOTAL ON key expression. Since DBORDERINFO() returns the expression as a string, it is specified using a macro expression to force evaluation of the key expression:

```
#include Dbinfo.ch

USE Sales INDEX Salesman NEW
TOTAL ON &(DBORDERINFO(DBOI_EXPRESSION)) ;
   FIELDS SaleAmount TO Summary
```

- In this example, All_Cust.mdx contains three orders named CuAcct, CuName, CuZip. The DBOI_INDEXNAME constant is used to display the name of the index file using one of its orders:

```
#include Dbinfo.ch

USE Customer VIA "DBFNTX" NEW
Customer->DBSETINDEX("All_Cust")
? DBORDERINFO(DBOI_INDEXNAME,, "CuName")
// Returns: All_Cust
```

- The following example searches for CuName in the order list:

```
#include Dbinfo.ch

USE Customer VIA "DBFNTX" NEW
Customer->DBSETINDEX("CuAcct")
Customer->DBSETINDEX("CuName")
Customer->DBSETINDEX("CuZip")
? DBORDERINFO(DBOI_NUMBER,, "CuName")          // 2
```

- This example retrieves the FOR condition from an order:

```
#include Dbinfo.ch

USE Customer NEW
INDEX ON Customer->Acct TO Customer ;
   FOR Customer->Acct > "AZZZZZ"
? DBORDERINFO(DBOI_CONDITION,, "Customer")
// Returns: Customer->Acct > "AZZZZZ"
```

Files Library is CLIPPER.LIB, header file is Dbinfo.ch.

See Also DBFIELDINFO(), DBINFO(), DBRECORDINFO()

DBRECALL() function

Reinstate a record marked for deletion

Syntax

```
DBRECALL() → NIL
```

Returns

DBRECALL() always returns NIL.

Description

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

DBRECALL() performs the same function as the RECALL command. For more information, refer to the DELETE and RECALL commands.

Notes

- ***Logical records:*** Reinstating a deleted record affects the record's logical visibility if the global _SET_DELETED status is true (.T.). For more information, refer to the DBDELETE() function and the DELETE and RECALL commands.

- ***Network environment:*** For a shared database on a network, DBRECALL() requires the current record to be locked. For more information, refer to the "Network Programming" chapter in the *Programming and Utilities Guide*.

Examples

- The following example recalls a record if it is deleted and attempts to lock the record if successful:

```
cLast := "Winston"
DBUSEAREA( .T., "DBFNTX", "Sales", "Sales", .T. )
DBSETINDEX( "LASTNAME" )
//
IF ( Sales->(DBSEEK(cLast)) )
   IF Sales->( DELETED() )
      IF Sales( RLOCK() )
         Sales( DBRECALL() )
         ? "Record recalled"
      ELSE
         "Unable to lock record..."
      ENDIF
   ENDIF
ELSE
   ? "Not found"
ENDIF
```

Files Library is CLIPPER.LIB.

See Also DBDELETE(), DELETE, RECALL

DBRECORDINFO() function

Return and optionally change information about a record

Syntax

```
DBRECORDINFO(<nInfoType>,
    [<nRecord>],
    [<expNewSetting>]) → uCurrentSetting
```

Arguments

<nInfoType> determines the type of information, as specified by the constants below. Note, however, that not all constants are supported for all RDDs. These constants are defined in the Dbinfo.ch header file, which must be included (#include) in your application.

Record Information Type Constants

Constant	Description
DBRI_DEL	Deleted flag status of the record
DBRI_LOCK	Locked flag status of the record
DBRI_SIZE	Length of the record
DBRI_RECNO	Position of the record
DBRI_UPDAT	Updated flag status of the record

Important! *DBI_USER is a constant that returns the minimum value that third-party RDD developers can use for defining new <nInfoType> parameters. Values less than DBI_USER are reserved for Computer Associates Development.*

<nRecord> is the record to obtain information. If omitted, the current record is used.

<expNewSetting> is reserved for RDDs that allow the file information to be changed, in addition to being retrieved. None of the RDDs supplied with CA-Clipper support this argument. It can be omitted or specified as NIL.

Returns

If *<expNewSetting>* is not specified, DBRECORDINFO() returns the current setting. If *<expNewSetting>* is specified, the previous setting is returned.

D

Description

DBRECORDINFO() retrieves information about the state of a record (row). The type of information is specified by the <*nInfoType*> parameter. By default, this function operates on the currently selected record.

DBRECORDINFO() is designed to allow for additional <*nInfoType*> values that can be defined by third-party RDD developers.

Examples

- The following example uses DBRECORDINFO() to retrieve field information:

```
#include "Dbinfo.ch"

DBRECORDINFO(DBRI_SIZE)               // Same as RECSIZE()

DBRECORDINFO(DBRI_LOCK, 200)          // Is record 200 locked?

DBRECORDINFO(DBRI_DEL, 201)           // Is record 201 locked?

DBRECORDINFO(DBRI_UPDAT)              // Is the current record
                                      // updated?

DBRECORDINFO(DBRI_RECNO, 230)         // On which position is
                                      // record 230?
                                      // If no orders are active,
                                      // the position is 230;
                                      // otherwise, the relative
                                      // position within the order
                                      // will be returned.
```

Files Library is CLIPPER.LIB, header file is Dbinfo.ch.

See Also DBFIELDINFO(), DBINFO(), DBORDERINFO()

DBREINDEX() function

Recreate all active indexes for the current work area

Syntax

```
DBREINDEX() → NIL
```

Returns

DBREINDEX() always returns NIL.

Description

DBREINDEX() rebuilds all active indexes associated with the current work area. After the indexes are recreated, the work area is moved to the first logical record in the controlling order.

DBREINDEX() performs the same function as the standard REINDEX command. For more information, refer to the REINDEX command.

Examples

- The following example reindexes the work area:

```
cLast := "Winston"
DBUSEAREA( .T., "DBFNTX", "Sales", "Sales", .T. )
DBSETINDEX( "LASTNAME" )
//
IF ( Sales->(DBSEEK(cLast)) )
    IF RLOCK()
       DELETE FOR Sales->LastName == "Winston"
       Sales->( DBREINDEX() )
    ELSE
       ? "Unable to lock record..."
    ENDIF
ELSE
    ? "Not found"
ENDIF
```

Files Library is CLIPPER.LIB.

See Also DBCLEARINDEX(), DBCREATEINDEX(), DBSETINDEX(), DBSETORDER(), REINDEX

DBRELATION() function

Return the linking expression of a specified relation

Syntax

```
DBRELATION(<nRelation>) → cLinkExp
```

Arguments

<nRelation> is the position of the desired relation in the list of current work area relations. The relations are numbered according to the order in which they were defined with SET RELATION.

Returns

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

Description

DBRELATION() is a database function used with DBRSELECT() to determine the linking expression and work area of an existing relation created with the SET RELATION command.

DBRELATION() returns the linking expression defined by the TO clause. DBRSELECT() returns the work area linked as defined by the INTO clause.

By default, DBRELATION() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression (see example below).

Notes

- ***Declared variables:*** A character string returned by DBRELATION() may not operate correctly when recompiled and executed using the macro operator (&) if the original expression contained references to local or static variables, or otherwise depends on compile-time declarations.

Examples

- This example opens three database files, sets two child relations from the parent work area, and then displays the linking expression to the second child work area:

```
USE Invoices INDEX Invoices NEW
USE BackOrder INDEX BackOrder NEW
USE Customer INDEX Customer NEW
SET RELATION TO CustNum INTO Invoices, OrderNum ;
      INTO BackOrder
//
? DBRELATION(2)                // Result: OrderNum
```

- Later you can query the same linking expression from an unselected work area by using an aliased expression like this:

```
USE Archive NEW
? Customer->(DBRELATION(2))    // Result: OrderNum
```

- This example is a user-defined function, Relation(), that returns the results of both DBRELATION() and DBRSELECT() as an array:

```
FUNCTION Relation( nRelation )
   RETURN { DBRELATION(nRelation), ;
         ALIAS(DBRSELECT(nRelation)) }
```

Files Library is CLIPPER.LIB.

See Also DBFILTER(), DBRSELECT(), SET RELATION

DBRLOCK() function

Lock the record at the current or specified identity

Syntax

```
DBRLOCK([<xIdentity>]) → lSuccess
```

Arguments

<*xIdentity*> is a unique value guaranteed by the structure of the data file to reference a specific item in a data source (database). In a .dbf <*xIdentity*> is the record number. In other data formats, <*xIdentity*> is the unique primary key value.

Returns

DBRLOCK() returns *lSuccess*, a logical data type that is true (.T.) if successful, false (.F.) if unsuccessful.

Description

DBRLOCK() is a database function that locks the record identified by the value <*xIdentity*>. In Xbase, <*xIdentity*> is the record number.

If you do not specify <*xIdentity*>, all record locks are released and the current record is locked. If you specify <*xIdentity*>, DBRLOCK() attempts to lock it and, if successful, adds it to the locked record list.

Examples

- This example shows two different methods for locking multiple records:

```
FUNCTION dbRLockRange( nLo, nHi )

      LOCAL nRec
      FOR nRec := nLo TO nHi
         IF ! DBRLOCK( nRec )
            DBRUNLOCK()    // Failed - unlock everything
         ENDIF
      NEXT
   RETURN DBRLOCKLIST()    // Return array of actual locks

FUNCTION dbRLockArray( aList )

      LOCAL nElement, nLen, lRet
      lRet := .T.
      nLen := LEN( aList )
      FOR nElement := 1 TO nLen
         IF ! DBRLOCK( aList[ nElement ] )
            DBRUNLOCK()    // Failed - unlock everything
            lRet := .F.
         ENDIF
      NEXT
   RETURN DBRLOCKLIST()
```

See Also

DBUNLOCK(), DBUNLOCKALL(), FLOCK(), RLOCK(), UNLOCK

DBRLOCKLIST() function

Return an array of the current lock list

Syntax

```
DBRLOCKLIST() → aRecordLocks
```

Returns

DBRLOCKLIST() returns an array of the locked records in the current or aliased work area.

Description

DBRLOCKLIST() is a database function that returns a one-dimensional array that contains the identities of record locks active in the selected work area.

Examples

```
PROCEDURE PrintCurLocks()

   LOCAL aList
   LOCAL nSize
   LOCAL nCount

   aList := DBRLOCKLIST()
   nSize := LEN( aList )

   ? "Currently locked records: "
   FOR nCount := 1 TO nSize
      ?? aList[ nCount ]
      ?? SPACE( 1 )
   NEXT
   ?

   RETURN
```

See Also

RLOCK(), UNLOCK, DBRLOCK(), DBRUNLOCK()

DBRSELECT() function

Return the target work area number of a relation

Syntax

```
DBRSELECT(<nRelation>) → nWorkArea
```

Arguments

<nRelation> is the position of the desired relation in the list of current work area relations. The relations are numbered according to the order in which they were defined with SET RELATION.

Returns

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

Description

DBRSELECT() is a database function used in combination with DBRELATION() to determine the work area and linking expression of an existing relation created with the SET RELATION command. DBRSELECT() returns the work area defined by the INTO clause. DBRELATION() returns the linking expression defined by the TO clause. To determine the alias of the relation instead of the work area number, use the expression ALIAS(DBRSELECT(*<nRelation>*)).

By default, DBRSELECT() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression (see example below).

Examples

- This example opens three database files, sets two child relations from the parent work area, and then displays the linking expression to the second child work area, as well as the target work area of the relation:

```
USE Invoices INDEX Invoices NEW
USE BackOrder INDEX BackOrder NEW
USE Customer INDEX Customer NEW
SET RELATION TO CustNum INTO Customer, ;
      OrderNum INTO BackOrder
//
? DBRELATION(2), DBRSELECT(2)       // Result: OrderNum 3
? ALIAS(DBRSELECT(2))               // Result: BACKORDER
```

- Later, you can query the same information from an unselected work area by using an aliased expression:

```
USE Archive NEW
? Customer->(DBRELATION(2))      // Result: OrderNum
? Customer->(DBSELECT(2))        // Result: 3
```

Files Library is CLIPPER.LIB.

See Also DBFILTER(), DBRELATION(), SET RELATION

DBRUNLOCK() function

Release all or specified record locks

Syntax

```
DBRUNLOCK([<xIdentity>]) → NIL
```

Arguments

<xIdentity> is a unique value guaranteed by the structure of the data file to reference a specific item in a data source (database). In a .dbf *<xIdentity>* is the record number. In other data formats, *<xIdentity>* is the unique primary key value.

Returns

DBRUNLOCK() always returns NIL.

Description

DBRUNLOCK() is a database function that releases the lock on *<xIdentity>* and removes it from the Lock List. If *<xIdentity>* is not specified, all record locks are released.

Examples

```
PROCEDURE dbRUnlockRange( nLo, nHi )

   LOCAL nCounter

   // Unlock the records in the range from nLo to nHi
   FOR nCounter := nLo TO nHi
      DBRUNLOCK( nCounter )
   NEXT

   RETURN
```

See Also

RLOCK(), DBRLOCK(), DBRLOCKLIST()

DBSEEK() function

Move to the record having the specified key value

Syntax

```
DBSEEK(<expKey>, [<lSoftSeek>], [<lLast>]) → lFound
```

Arguments

<*expKey*> is a value of any type that specifies the key value associated with the desired record.

<*lSoftSeek*> is an optional logical value that specifies whether a soft seek is to be performed. This determines how the work area is positioned if the specified key value is not found (see below). If <*lSoftSeek*> is omitted, the current global _SET_SOFTSEEK setting is used.

<*lLast*> is specified as true (.T.) to seek the last occurrence of the specified key value. False (.F.), the default, seeks the first occurrence.

Note: This parameter is only supported for specific RDDs. DBFNTX is NOT one of them.

Returns

DBSEEK() returns true (.T.) if the specified key value was found; otherwise, it returns false (.F.).

Description

DBSEEK() moves to the first logical record whose key value is equal to <*expKey*>. If such a record is found, it becomes the current record and DBSEEK() returns true (.T.); otherwise, it returns false (.F.). The positioning of the work area is as follows: for a normal (not soft) seek, the work area is positioned to LASTREC() + 1 and EOF() returns true (.T.); for a soft seek, the work area is positioned to the first record whose key value is greater than the specified key value. If no such record exists, the work area is positioned to LASTREC() + 1 and EOF() returns true (.T.).

For a work area with no active indexes, DBSEEK() has no effect.

DBSEEK() performs the same function as the standard SEEK command. For more information, refer to the SEEK command.

Notes

- ***Logical records:*** DBSEEK() operates on logical records which are considered in indexed order. If a filter is set, only records which meet the filter condition are considered.

- ***Controlling order:*** If the work area has more than one active index, the operation is performed using the controlling order as set by DBSETORDER() or the SET ORDER command. For more information, refer to the SET ORDER command.

- ***Network environment:*** For a shared file on a network, moving to a different record may cause updates to the current record to become visible to other processes. For more information, refer to the "Network Programming" chapter in the *Programming and Utilities Guide*. This function will not affect the locked status of any record.

Examples

- In this example, DBSEEK() moves the pointer to the record in the database, Employee, in which the value in FIELD "cName" matches the entered value of cName:

```
ACCEPT "Employee name: " TO cName
IF ( Employee->(DBSEEK(cName)) )
   Employee->(VIEWRECORD())
ELSE
   ? "Not found"
END
```

Files

Library is CLIPPER.LIB.

See Also

DBGOBOTTOM(), DBGOTOP(), DBSKIP(), EOF(), FOUND(), SEEK

D

DBSELECTAREA() function

Change the current work area

Syntax

```
DBSELECTAREA(<nArea> | <cAlias>) → NIL
```

Arguments

<nArea> is a numeric value between zero and 250, inclusive, that specifies the work area being selected.

<cAlias> is a character value that specifies the alias of a currently occupied work area being selected.

Returns

DBSELECTAREA() always returns NIL.

Description

DBSELECTAREA() causes the specified work area to become the current work area. All subsequent database operations will apply to this work area unless another work area is explicitly specified for an operation. DBSELECTAREA() performs the same function as the standard SELECT command. For more information, refer to the SELECT command.

Notes

- ***Selecting zero:*** Selecting work area zero causes the lowest numbered unoccupied work area to become the current work area.
- ***Aliased expressions:*** The alias operator (->) can temporarily select a work area while an expression is evaluated and automatically restore the previously selected work area afterward. For more information, refer to the alias operator (->).

Examples

- The following example selects a work area via the alias name:

```
cLast := "Winston"
DBUSEAREA( .T., "DBFNTX", "Sales", "Sales", .T. )
DBSETINDEX( "SALEFNAM" )
DBSETINDEX( "SALELNAM" )
//
DBUSEAREA( .T., "DBFNTX", "Colls", "Colls", .T. )
DBSETINDEX( "COLLFNAM" )
DBSETINDEX( "COLLLNAM" )
//
DBSELECTAREA( "Sales" )    // select "Sales" work area
//
IF ( Sales->(DBSEEK(cLast)) )
   IF Sales->( DELETED() ) .AND. Sales->( RLOCK() )
      Sales->( DBRECALL() )
      ? "Deleted record has been recalled."
   ENDIF
ELSE
   ? "Not found"
ENDIF
```

Files Library is CLIPPER.LIB.

See Also DBUSEAREA(), SELECT, SELECT(), USE

DBSETDRIVER() function

Return the default database driver and optionally set a new driver

Syntax

```
DBSETDRIVER([<cDriver>]) → cCurrentDriver
```

Arguments

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

Returns

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

Description

DBSETDRIVER() sets the database driver to be used when activating new work areas without specifying a driver. If the specified driver is not available to the application, the call has no effect. DBSETDRIVER() returns the name of the current default driver, if any.

Examples

- This example makes the "DBFNDX" driver the default driver. If the driver is unavailable, a message is issued:

```
DBSETDRIVER("DBFNDX")
IF ( DBSETDRIVER() <> "DBFNDX" )
   ? "DBFNDX driver not available"
ENDIF
```

Files Library is CLIPPER.LIB.

See Also DBUSEAREA(), USE

DBSETFILTER() function

Set a filter condition

Syntax

```
DBSETFILTER(<bCondition>, [<cCondition>]) → NIL
```

Arguments

<*bCondition*> is a code block that expresses the filter condition in executable form.

<*cCondition*> stores the filter condition as a character string for later retrieval by the DBFILTER() function. If you omit this optional parameter, the DBFILTER() function will return an empty string for the work area.

Returns

DBSETFILTER() always returns NIL.

Description

DBSETFILTER() sets a logical filter condition for the current work area. When a filter is set, records which do not meet the filter condition are not logically visible. That is, database operations which act on logical records will not consider these records.

The filter expression supplied to DBSETFILTER() evaluates to true (.T.) if the current record meets the filter condition; otherwise, it should evaluate to false (.F.).

The filter expression may be a code block (<*bCondition*>) or both a code block and equivalent text (<*cCondition*>). If both versions are supplied, they must express the same condition. If the text version is omitted, DBFILTER() will return an empty string for the work area.

DBSETFILTER() performs the same function as the standard SET FILTER command. For more information, refer to the SET FILTER command.

Notes

- ***Logical records:*** DBSETFILTER() affects the logical visibility of records (see above).
- ***Side effects:*** Setting a filter condition is only guaranteed to restrict visibility of certain records as described above. The filter expression is not necessarily evaluated at any particular time, by any particular means, or on any particular record or series of records. If the filter expression relies on information external to the database file or work area, the effect is unpredictable. If the filter expression changes the state of the work area (e.g., by moving to a different record or changing the contents of a record), the effect is unpredictable.
- ***Evaluation context:*** When the filter expression is evaluated, the associated work area is automatically selected as the current work area before the evaluation; the previously selected work area is automatically restored afterward.

Examples

- This example limits data access to records in which the Age field value is less than 40:

```
USE Employee NEW
DBSETFILTER( {|| Age < 40}, "Age < 40" )
DBGOTOP()
```

Files

Library is CLIPPER.LIB.

See Also

DBCLEARFILTER(), DBFILTER(), SET DELETED, SET FILTER

DBSETINDEX() function

Empty orders from an order bag into the order list

Syntax

```
DBSETINDEX(<cOrderBagName>) → NIL
```

Arguments

<cOrderBagName> is the name of a disk file containing one or more orders. You may specify *<cOrderBagName>* as the file name with or without the path name or extension. If you do not include the extension as part of *<cOrderBagName>*, CA-Clipper uses the default extension of the current RDD.

Returns

DBSETINDEX() always returns NIL.

Description

DBSETINDEX() is a database function that adds the contents of an order bag into the order list of the current work area. Any orders already associated with the work area continue to be active. If the newly opened order bag is the only order associated with the work area, it becomes the controlling order; otherwise, the controlling order remains unchanged. If the order bag contains more than one order, and there are no other orders associated with the work area, the first order in the new order bag becomes the controlling order.

Note: DBSETINDEX() does not close all currently open index files.

DBSETINDEX() is a compatibility command and therefore is not recommended. It is superseded by the ORDLISTADD() function.

Examples

```
USE Customer NEW
DBSETINDEX( "Cust01" )          // Open the index Cust01
                                // in the current work area
DBSETINDEX( "Cust02" )          // Open the index Cust02
                                // leaving Cust01 open
```

See Also ORDLISTADD()

DBSETORDER() function

Set the controlling order

Syntax

```
DBSETORDER(<nOrderNum>) → NIL
```

Arguments

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

Returns

DBSETORDER() always returns NIL.

Description

DBSETORDER() controls which of the current work area's active indexes is the controlling index. The controlling index is the index which determines the logical order of records in the work area.

Active indexes are numbered from 1 to the number of active indexes, based on the order in which the indexes were opened. <*nOrderNum*> specifies the number of the desired index.

DBSETORDER() performs the same function as the standard SET ORDER command. For more information, refer to the SET ORDER command.

Notes

- ***Setting order to zero:*** Setting order to zero causes the work area to be accessed in natural (record number) order. Only the logical order of the records is affected; any open indexes continue to be active and are properly maintained.

Examples

- This example sets the second named index, Age, as the controlling index:

```
USE Employee NEW
SET INDEX TO Name, Age
DBSETORDER(2)
```

Files Library is CLIPPER.LIB.

See Also DBCLEARINDEX(), DBCREATEINDEX(), DBREINDEX(), DBSETINDEX(), SET ORDER

DBSETRELATION() function

Relate two work areas

Syntax

```
DBSETRELATION(<nArea> | <cAlias>, <bExpr>, <cExpr>)
   → NIL
```

Arguments

<*nArea*> is a numeric value that specifies the work area number of the child work area.

<*cAlias*> is a character value that specifies the alias of the child work area.

<*bExpr*> is a code block that expresses the relational expression in executable form.

<*cExpr*> is a character value that expresses the relational expression in textual form.

Returns

DBSETRELATION() always returns NIL.

Description

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

Relating work areas synchronizes the child work area with the parent work area. This is achieved by automatically repositioning the child work area whenever the parent work area moves to a new record. If there is an active index in the child work area, moving the parent work area causes an automatic SEEK operation in the child work area; the seek key is based on the expression specified by <*bExpr*> and/or <*cExpr*>. If the child work area has no active index, moving the parent work area causes an automatic GOTO in the child work area; the record number for the GOTO is based on the expression specified by <*bExpr*> and/or <*cExpr*>.

The relational expression may be a code block (<*bExpr*>) or both a code block and equivalent text (<*cExpr*>). If both versions are supplied, they must be equivalent. If the text version is omitted, DBRELATION() will return an empty string for the relation.

DBSETRELATION() performs the same function as the standard SET RELATION command with the ADDITIVE clause. For more information, refer to the SET RELATION command.

Notes

- ***Side effects:*** DBSETRELATION() is only guaranteed to synchronize the work areas as described above. The relational expression is not necessarily evaluated at any particular time, by any particular means, or on any particular record or series of records. If the relational expression relies on information external to the parent work area or its associated database file, the effect is unpredictable. If the expression changes the state of either work area (e.g., by moving to a different record or changing the contents of a record), the effect is unpredictable.

- ***Evaluation context:*** When the relational expression is evaluated, the parent work area is automatically selected as the current work area before the evaluation; the previously selected work area is automatically restored afterward.

- ***Soft seeking:*** Seek operations that occur as part of relational positioning are never soft seeks. If a relational movement is unsuccessful, the child work area is positioned to LASTREC() + 1, its FOUND() status returns false (.F.), and its EOF() status returns true (.T.).

Examples

- This example demonstrates a typical use of the DBSETRELATION() function:

```
USE Employee NEW
USE Department NEW INDEX Dept
SELECT Employee
DBSETRELATION("Department", {|| Employee->Dept},;
                "Employee->Dept")
LIST Employee->Name, Department->Name
```

Files

Library is CLIPPER.LIB.

See Also

DBCLEARRELATION(), DBRELATION(), DBRSELECT(), FOUND(), SET RELATION

DBSKIP() function

Move relative to the current record

Syntax

```
DBSKIP([<nRecords>]) → NIL
```

Arguments

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

Returns

DBSKIP() always returns NIL.

Description

DBSKIP() moves either forward or backward relative to the current record. Attempting to skip forward beyond the last record positions the work area to LASTREC() + 1 and EOF() returns true (.T.). Attempting to skip backward beyond the first record positions the work area to the first record and BOF() returns true (.T.).

DBSKIP() performs the same function as the standard SKIP command. For more information, refer to the SKIP command.

Notes

- ***Logical records:*** DBSKIP() operates on logical records. If there is an active index, records are considered in indexed order. If a filter is set, only records which meet the filter condition are considered.

- ***Controlling order:*** If the work area has more than one active index, the skip operation is performed using the controlling order as set by DBSETORDER() or the SET ORDER command. For more information, refer to the SET ORDER command.

- ***Network environment:*** For a shared file on a network, moving to a different record may cause updates to the current record to become visible to other processes. For more information, refer to the "Network Programming" chapter in the *Programming and Utilities Guide*.

Examples

- This example demonstrates a typical use of the DBSKIP() function:

```
DBGOTOP()
DO WHILE ( !EOF() )
   ? FIELD->Name
   DBSKIP()
ENDDO
```

Files

Library is CLIPPER.LIB.

See Also

BOF(), DBGOBOTTOM(), DBGOTOP(), DBSEEK(), EOF(), SKIP

DBSTRUCT() function

Create an array containing the structure of a database file

Syntax

```
DBSTRUCT() → aStruct
```

Returns

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

DBSTRUCT() Return Array

Position	Metasymbol	Dbstruct.ch
1	cName	DBS_NAME
2	cType	DBS_TYPE
3	nLength	DBS_LEN
4	nDecimals	DBS_DEC

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

Description

DBSTRUCT() is a database function that operates like COPY STRUCTURE EXTENDED by creating an array of structure information rather than a database file of structure information. There is another function, DBCREATE(), that can create a database file from the structure array.

By default, DBSTRUCT() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression as shown below.

Note, a header file, Dbstruct.ch, located in \CLIP53\INCLUDE contains a series of manifest constants for each field attribute.

Examples

- This example opens two database files and then creates an array containing the database structure using DBSTRUCT() within an aliased expression. The field names are then listed using AEVAL():

```
#include "Dbstruct.ch"
//
LOCAL aStruct
USE Customer NEW
USE Invoices NEW
//
aStruct := Customer->(DBSTRUCT())
AEVAL( aStruct, {|aField| QOUT(aField[DBS_NAME])} )
```

Files

Library is CLIPPER.LIB, header file is Dbstruct.ch.

See Also

AFIELDS()*, COPY STRUCTURE EXTENDED

DBUNLOCK() function

Release all locks for the current work area

Syntax

```
DBUNLOCK() → NIL
```

Returns

DBUNLOCK() always returns NIL.

Description

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

DBUNLOCK() performs the same function as the standard UNLOCK command. For more information, refer to the UNLOCK command.

Notes

- ***Network environment:*** Releasing locks may cause updates to the database to become visible to other processes. For more information, refer to the "Network Programming" chapter in the *Programming and Utilities Guide*.

Examples

- The following example illustrates a basic use of the DBUNLOCK() function:

```
cLast := "Winston"
USE Sales SHARED NEW VIA "DBFNTX"
DBSETINDEX( "LASTNAME" )
//
IF ( Sales->(DBSEEK(cLast)) )
    IF Sales->( RLOCK() )
       Sales->( DBDELETE() )
       ? "Record deleted: ", Sales( DELETED() )
       Sales->( DBUNLOCK() )
    ELSE
       ? "Unable to lock record..."
    ENDIF
ELSE
    ? "Not found"
ENDIF
```

Files

Library is CLIPPER.LIB.

See Also

DBUNLOCKALL(), FLOCK(), RLOCK(), UNLOCK

D

DBUNLOCKALL() function

Release all locks for all work areas

Syntax

```
DBUNLOCKALL() → NIL
```

Returns

DBUNLOCKALL() always returns NIL.

Description

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

DBUNLOCKALL() performs the same function as the UNLOCK ALL command. For more information, refer to the UNLOCK ALL command.

Examples

- The following example marks a record for deletion if an RLOCK() attempt is successful, then clears all locks in all work areas:

```
cLast := "Winston"
USE Sales SHARED NEW VIA "DBFNTX"
DBSETINDEX( "SALEFNAM" )
DBSETINDEX( "SALELNAM" )
//
USE Colls SHARED NEW VIA "DBFNTX"
DBSETINDEX( "COLLFNAM" )
DBSETINDEX( "COLLLNAM" )
//
DBSELECTAREA( "Sales" )    // select "Sales" work area
//
IF ( Colls->(DBSEEK(cLast)) )
    IF Colls->( DELETED() )
        ? "Record deleted: ", Colls->( DELETED() )
        IF Colls->( RLOCK() )
            Colls->( DBRECALL() )
            ? "Record recalled..."
        ENDIF
    ENDIF
ELSE
    ? "Not found"
    DBUNLOCKALL()              // remove all locks in
ENDIF                          // all work areas
```

Files Library is CLIPPER.LIB.

See Also DBUNLOCK(), FLOCK(), RLOCK(), UNLOCK

DBUSEAREA() function

Use a database file in a work area

Syntax

```
DBUSEAREA( [<lNewArea>], [<cDriver>], <cName>,
[<xcAlias>], [<lShared>], [<lReadonly>]) → NIL
```

Arguments

<lNewArea> is an optional logical value. A value of true (.T.) selects the lowest numbered unoccupied work area as the current work area before the use operation. If *<lNewArea>* is false (.F.) or omitted, the current work area is used; if the work area is occupied, it is closed first.

<cDriver> is an optional character value. If present, it specifies the name of the database driver which will service the work area. If *<cDriver>* is omitted, the current default driver is used (see note below).

<cName> specifies the name of the database (.dbf) file to be opened.

<xcAlias> is an optional character value. If present, it specifies the alias to be associated with the work area. The alias must constitute a valid CA-Clipper identifier. A valid *<xcAlias>* may be any legal identifier (i.e., it must begin with an alphabetic character and may contain numeric or alphabetic characters and the underscore). Within a single application, CA-Clipper will not accept duplicate aliases. If *<xcAlias>* is omitted, a default alias is constructed from *<cName>*.

<lShared> is an optional logical value. If present, it specifies whether the database (.dbf) file should be accessible to other processes on a network. A value of true (.T.) specifies that other processes should be allowed access; a value of false (.F.) specifies that the current process is to have exclusive access. If *<lShared>* is omitted, the current global _SET_EXCLUSIVE setting determines whether shared access is allowed.

<lReadonly> is an optional logical value that specifies whether updates to the work area are prohibited. A value of true (.T.) prohibits updates; a value of false (.F.) permits updates. A value of true (.T.) also permits read-only access to the specified database (.dbf) file. If *<lReadonly>* is omitted, the default value is false (.F.).

Returns

DBUSEAREA() always returns NIL.

Description

DBUSEAREA() associates the specified database (.dbf) file with the current work area. It performs the same function as the standard USE command. For more information, refer to the USE command.

Notes

- ***Current driver:*** If no driver is specified in the call to DBUSEAREA() the default driver is used. If more than one driver is available to the application, the default driver is the driver specified in the most recent call to DBSETDRIVER(). If DBSETDRIVER() has not been called, the DBFNTX driver is used. If the default driver is undetermined, DBFNTX will be used.

Examples

- This example is a typical use of the DBUSEAREA() function:

```
DBUSEAREA(.T., "DBFNDX", "Employees")
```

Files Library is CLIPPER.LIB.

See Also DBCLOSEAREA(), DBSETDRIVER(), SELECT(), SET(), USE

DECLARE* statement

Create and initialize private memory variables and arrays

Syntax

```
DECLARE <identifier> [[:= <initializer>], ... ]
```

Arguments

<identifier> is the name of a private variable or array to create. If the *<identifier>* is followed by square brackets ([ ]), it is created as an array. If the *<identifier>* is an array, the syntax for specifying the number of elements for each dimension is either array[*<nElements>*, *<nElements2>*,...] or array[*<nElements>*][*<nElements2>*]... The maximum number of elements per dimension is 4096.

<initializer> is the optional assignment of a value to a new private variable. An *<initializer>* expression for a private variable consists of the inline assignment operator (:=) followed by any valid CA-Clipper expression, including a literal array. If no explicit *<initializer>* is specified, the variable is given an initial value of NIL. In the case of an array, each element is NIL. Array identifiers, cannot be given values with an *<initializer>*.

DECLARE can create and, optionally, initialize a list of variable arrays, if definitions are separated by commas.

Description

DECLARE is a compatibility statement that is a synonym for the PRIVATE statement. Its general use is not recommended. PRIVATE should be used in all instances.

See Also PRIVATE statement

DELETE command

Mark records for deletion

Syntax

```
DELETE [<scope>] [WHILE <lCondition>]
   [FOR <lCondition>]
```

Arguments

<scope> is the portion of the current database file to DELETE. If a scope is not specified, DELETE acts only on the current record. If a conditional clause is specified, the default becomes ALL records.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to DELETE within the given scope.

Description

DELETE is a database command that tags records so they can be filtered with SET DELETED ON, queried with DELETED(), or physically removed from the database file with PACK. In addition, display commands such as LIST and DISPLAY identify deleted records with an asterisk (*) character. Once records are deleted, you can reinstate them by using RECALL. If you want to remove all records from a database file, use ZAP instead of DELETE ALL and PACK.

Any deleted record can be recalled as long as the PACK or ZAP command has *not* been issued. Once the PACK or ZAP command has been issued, the deleted data *cannot* be retrieved.

In a network environment, DELETE requires the current record be locked with RLOCK() if you are deleting a single record. If you are deleting several records, the current database file must be locked with FLOCK() or USEed EXCLUSIVEly. Refer to the "Network Programming" chapter in the *Programming and Utilities Guide* for more information.

Notes

- ***DELETE with SET DELETED ON:*** If the current record is deleted with SET DELETED ON, it will be visible until the record pointer is moved.

Examples

- This example demonstrates use of the FOR clause to mark a set of records for deletion:

```
USE Sales INDEX Salesman NEW
DELETE ALL FOR Inactive
```

Files

Library is CLIPPER.LIB.

See Also

DBEVAL(), DELETED(), FLOCK(), PACK, RECALL, RLOCK(), SET DELETED, ZAP

DELETE FILE command

Remove a file from disk

Syntax

```
DELETE FILE | ERASE <xcFile>
```

Arguments

<*xcFile*> is the name of the file to be deleted from disk and can be specified either as a literal file name or as a character expression enclosed in parentheses. You must specify the file name, including the extension, and it may optionally be preceded by a drive and/or path specification.

Description

DELETE FILE, a synonym for ERASE, is a file command that removes the specified file from disk. SET DEFAULT and SET PATH do not affect DELETE FILE. The file is deleted from disk only if found in the current DOS directory or in the directory explicitly specified as part of the file name.

Warning! *Files must be CLOSEd before deleting them. Otherwise, either a sharing violation or a date corruption on the drive may occur.*

Examples

- This example removes a specified file from disk then tests to see if the file was in fact removed:

```
? FILE("Temp.dbf")              // Result: .T.
DELETE FILE Temp.dbf
? FILE("Temp.dbf")              // Result: .F.
```

Files Library is CLIPPER.LIB.

See Also CLOSE, CURDIR(), FILE(), USE

DELETE TAG command

Delete a tag

Syntax

```
DELETE TAG <cOrderName> [IN <xcOrderBagName>]
   [, <cOrderName> [IN xcOrderBagName] list>]
```

Arguments

<cOrderName> is a character string that represents the order name.

<xcOrderBagName> is the name of a disk file containing one or more orders. You may specify *<xcOrderBagName>* as the file name with or without the path name or appropriate extension. If you do not include the extension as part of *<xcOrderBagName>*, CA-Clipper uses the default extension of the current RDD.

<cOrderName ...list> is an optional list of order and order bag name pairs, separated by commas. Any reference to *<cOrderName>* that results in either a null string ("") or spaces is ignored. You can specify each order as a literal expression or as a character expression enclosed in parentheses. If you specify no extension for the order bag name, the current database driver supplies a default extension.

Description

This command removes an order from an order bag in the current or specified work area. If you do not specify an *<xcOrderBagName>*, all orders bags are searched in the current or specified work area. The first occurrence of *<cOrderName>* is deleted. A runtime recoverable error is raised if the order is not found.

If *<cOrderName>* is the active order, the database in the current or specified work area reverts to its identity order (natural or entry) and SET FILTER scoping.

A runtime error is raised if *<xcOrderBagName>* does not exist or if it exists but does not contain *<cOrderName>*.

The active RDD determines the order capacity of an order bag. The default DBFNTX and the DBFNDX drivers only support single-order bags, while other RDDs may support multiple-order bags (e.g., the DBFCDX and DBFMDX drivers).

Examples

```
USE Customer VIA "DBFCDX" NEW
SET INDEX TO Customer

// Delete the Orders (Tags) Cust01 and Cust02 that
// exist in the index file Customer
DELETE TAG Cust01 IN Customer
DELETE TAG Cust02 IN Customer

// or
// DELETE TAG Cust01 IN Customer, Cust02 IN Customer
```

See Also

ORDDESTROY()

DELETED() function

Return the deleted status of the current record

Syntax

```
DELETED() → lDeleted
```

Returns

DELETED() returns true (.T.) if the current record is marked for deletion; otherwise, it returns false (.F.). If there is no database file in USE in the current work area, DELETED() returns false (.F.).

Description

DELETED() is a database function that determines if the current record in the active work area is marked for deletion. Since each work area with an open database file can have a current record, each work area has its own DELETED() value.

By default, DELETED() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression (see example below).

In applications, DELETED() is generally used to query the deleted status as a part of record processing conditions, or to display the deleted status as a part of screens and reports.

Examples

- This example uses DELETED() in the current and in an unselected work area:

```
USE Customer NEW
USE Sales NEW
? DELETED()                   // Result: .F.
DELETE
? DELETED()                   // Result: .T.
? Customer->(DELETED())       // Result: .F.
```

- This example uses DELETED() to display a record's deleted status in screens and reports:

```
@ 1, 65 SAY IF(DELETED(), "Inactive", "Active")
```

Files Library is CLIPPER.LIB.

See Also DELETE, PACK, RECALL, SET DELETED

DESCEND() function

Create a descending index key value

Syntax

```
DESCEND(<exp>) → ValueInverted
```

Arguments

<*exp*> is any valid expression of character, date, logical, or numeric type. Memo type is treated in the same way as character type.

Returns

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

Description

DESCEND() is a conversion function that returns the inverted form of the specified expression to be used with INDEX to create descending order indexes. Specify that part of the index expression you want to be descending as the DESCEND() argument. To subsequently perform a lookup with SEEK, specify DESCEND() in the search expression.

Notes

- The preferred way to create a descending index is to use the DESCENDing clause of the INDEX command. Using DESCENDING is the same as specifying the DESCEND() function, but without the performance penalty during index updates. If you create a DESCENDING index, you will not need to use the DESCEND() function during a SEEK. DESCENDING is an attribute of the index (.ntx) file, where it is stored and used for REINDEXing purposes.

Examples

- This example uses DESCEND() in an INDEX expression to create a descending order date index:

  ```
  USE Sales NEW
  INDEX ON DESCEND(OrdDate) TO SalesDate
  ```

  Later, use DESCEND() to SEEK on the descending index:

  ```
  SEEK DESCEND(dFindDate)
  ```

- This example illustrates how to create a descending order index using more than one data type. Here, the key is created using the concatenation of date and character fields after the appropriate type conversion has taken place. This example uses STR() instead of DTOS(), since DESCEND() of a date returns a numeric value:

  ```
  USE Sales NEW
  INDEX ON STR(DESCEND(SaleDate)) + Salesman TO LastSale
  ```

Files Library is EXTEND.LIB.

See Also FIND*, INDEX, SEEK

DEVOUT() function

Write a value to the current device

Syntax

```
DEVOUT(<exp>, [<cColorString>]) → NIL
```

Arguments

<exp> is the value to display.

<cColorString> is an optional argument that defines the display color of *<exp>*. If the current DEVICE setting is SCREEN, the output is displayed in the specified color.

If not specified, *<exp>* is displayed as the standard color of the current system color as defined by SETCOLOR(). *<cColorString>* is a character expression containing the standard color setting. If you want to specify a literal color setting, enclose it in quote marks.

Returns

DEVOUT() always returns NIL.

Description

DEVOUT() is a full-screen display function that writes the value of a single expression to the current device at the current cursor or printhead position. Use DEVOUT() with DEVPOS() in Std.ch to implement the @...SAY command.

Examples

- This example shows the relationship between the DEVOUT() function and the @...SAY command:

```
DEVPOS(10, 10)
DEVOUT("Hello there", "BG+/B"))
//
@ 10, 10 SAY "Hello there" COLOR "BG+/B"
```

Files

Library is CLIPPER.LIB.

See Also

COL(), DEVPOS(), QOUT(), ROW(), SETPOS()

DEVOUTPICT() function

Write a value to the current device using a picture clause

Syntax

```
DEVOUTPICT(<exp>, <cPicture>, [<cColorString>])
   → NIL
```

Arguments

<exp> is the value to display.

<cPicture> defines the formatting control for the display of *<exp>*. The picture specified here is the same as one used with @...SAY or TRANSFORM and can include both templates and functions.

<cColorString> is an optional argument that defines the display color of *<exp>*. If the current DEVICE is SCREEN, output displays in the specified color.

If not specified, *<exp>* displays as the standard color of the current system color as defined by SETCOLOR(). *<cColorString>* is a character expression containing the standard color setting. If you want to specify a literal color setting, it must be enclosed in quote marks.

Returns

DEVOUTPICT() always returns NIL.

Description

DEVOUTPICT() is a full-screen display function that writes the value of a single expression to the current device at the current cursor or printhead position. DEVOUTPICT() is used in combination with DEVPOS() in Std.ch to implement the @...SAY command used with a PICTURE clause.

D

Examples

- This example shows the relationship between the DEVOUTPICT() function and the @...SAY command:

```
DEVPOS(10, 10)
DEVOUTPICT("Hello there", "@!", "BG+/B"))
//
@ 10, 10 SAY "Hello there" PICTURE "@!" COLOR "BG+/B"
```

Files

Library is CLIPPER.LIB.

See Also

@...SAY, COL(), DEVOUT(), DEVPOS(), QOUT(), ROW(), SETPOS(), TRANSFORM()

DEVPOS() function

Move the cursor or printhead to a new position depending on the current device

Syntax

```
DEVPOS(<nRow>, <nCol>) → NIL
```

Arguments

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

Returns

DEVPOS() always returns NIL.

Description

DEVPOS() is an environment function that moves the screen or printhead depending on the current DEVICE. If DEVICE is SET to SCREEN, DEVPOS() behaves like SETPOS(), moves the cursor to the specified location, and updates ROW() and COL() with the new cursor position.

If DEVICE is SET to PRINTER, DEVPOS() moves the printhead instead. It does this by sending the number of linefeed and/or formfeed characters to the printer, and advancing the printhead to the new position. If the current SET MARGIN value is greater than zero, it is added to *<nCol>*. The printhead is then advanced to the specified *<nRow>* and *<nCol>* position and PROW() and PCOL() are updated. If either *<nRow>* or *<nCol>* are less than the current PROW() and PCOL() values, the printhead is moved according to the following special rules:

- If *<nRow>* is less than PROW(), an automatic EJECT (CHR(12)) is sent to the printer followed by the number of linefeed characters (CHR(10)) required to position the printhead on *<nRow>* of the following page.
- If *<nCol>* including the current SET MARGIN value is less than PCOL(), a carriage return character (CHR(13)) and the number of space characters required to position the printhead at *<nCol>* are sent to the printer.

To circumvent these rules, use SETPRC() to reset PROW() and PCOL() to new values before using DEVPOS(). See the SETPRC() discussion for more information.

If the printer is redirected to a file using the SET PRINTER command, DEVPOS() updates the file instead of the printer.

Files Library is CLIPPER.LIB.

See Also @...SAY, DEVOUT(), PCOL(), PROW(), SET DEVICE, SET PRINTER, SETPOS()

DIR* command

Display a listing of files from a specified path

Syntax

```
DIR [<xcFileSpec>]
```

Arguments

<xcFileSpec> is a template that must be matched by displayed files and can include standard wildcard (* and ?) characters. You can specify *<xcFileSpec>* as a literal file specification string or as a character expression enclosed in parentheses.

Description

DIR is a file command that displays a listing of files from the specified path in one of two formats depending on whether you specify the *<xcFileSpec>* argument. If you do not specify a path, DIR displays a standard listing of database files from the current or specified directory. The list includes the database file name, date of last update, and number of records. Including the option *<xcFileSpec>* displays a list of all files in the specified directory. This list includes the file name, extension, number of bytes, and date of last update.

If no path is specified, DIR displays files from the current DOS drive and directory unless SET DEFAULT has been used to specify a new default directory.

DIR is a compatibility command and therefore not recommended. It is superseded by the DIRECTORY() function which returns an array of file information from a wildcard pattern. Using this array, you can display file information to screen.

Notes

- ***Directory picklists:*** To present directory information to the user, use ADIR() to return one or more arrays of directory information, and ACHOICE() to present a picklist to the user.

Examples

- These examples display DOS and database files in the current directory:

```
cFilespec := "*.*"
DIR (cFilespec)          // Display all files
DIR                      // Display all (.dbf) files
DIR *.prg                // Display all program files
```

Files

Library is EXTEND.LIB.

See Also

DIRECTORY(), SET DEFAULT, DIR

DIRCHANGE () function

Change the current DOS directory

Syntax

```
DIRCHANGE(<cDir>) → nSuccess
```

Arguments

<cDir> is the name of the directory to change to, including the drive.

Returns

DIRCHANGE() returns 0 if successful; -1 if there is an argument error. Otherwise, DIRCHANGE() returns the DOS error code.

Description

DIRCHANGE() changes the current DOS directory. This function may also be used to determine whether or not a directory exists.

Examples

- The following example attempts to change to the "c:\dos" directory. If it is unsuccessful, an error message is displayed.

```
nResult :=  DIRCHANGE("c:\dos")
IF nResult != 0
    ? "Cannot change directory. "
    DO CASE
       CASE nResult == 3
          ?? "Directory does not exist."
       CASE nResult == 5
          ?? "Access to directory denied."
    END
    BREAK
ENDIF
```

You may also use something like this:

```
DIRCHANGE( "..\..\test" )
```

Files

Library is EXTEND.LIB.

See Also

CURDIR(), DIR, DIRMAKE(), DIRREMOVE(), DISKCHANGE()

DIRECTORY() function

Create an array of directory and file information

Syntax

```
DIRECTORY(<cDirSpec>, [<cAttributes>]) → aDirectory
```

Arguments

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

<cAttributes> specifies inclusion of files with special attributes in the returned information. *<cAttributes>* is a string containing one or more of the following characters:

DIRECTORY() Attributes

Attribute	Meaning
H	Include hidden files
S	Include system files
D	Include directories
V	Search for the DOS volume label and exclude all other files

Normal files are always included in the search, unless you specify V.

Returns

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

DIRECTORY() Subarray Structure

Position	Metasymbol	Directry.ch
1	cName	F_NAME
2	cSize	F_SIZE
3	dDate	F_DATE
4	cTime	F_TIME
5	cAttributes	F_ATTR

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

Description

DIRECTORY() is an environment function that returns information about files in the current or specified directory. It is similar to ADIR(), but returns a single array instead of adding values to a series of existing arrays passed by reference.

Use DIRECTORY() to perform actions on groups of files. In combination with AEVAL(), you can define a block that can be applied to all files matching the specified <*cDirSpec*>.

The header file, Directry.ch, in the \CLIP53\INCLUDE subdirectory contains #defines for the subarray subscripts, so that the references to each file subarray are more readable.

Examples

- This example creates an array of information about files in the current directory and then lists the names of the files using AEVAL() and QOUT():

```
#include "Directry.ch"
//
aDirectory := DIRECTORY("*.*", "D")
AEVAL( aDirectory, {|aFile| QOUT(aFile[F_NAME])} )
```

Files

Library is CLIPPER.LIB, header file is Directry.ch.

See Also

AEVAL(), CURDIR()

D

DIRMAKE() function

Create a directory

Syntax

```
DIRMAKE(<cNewDir>) → nSuccess
```

Arguments

<cNewDir> is the name of the directory to be created, including an optional drive. If you do not specify a drive, the current one is used.

Returns

DIRMAKE() returns 0 if successful; -1 if there is an argument error. Otherwise, DIRMAKE() returns the DOS error code.

Description

DIRMAKE() creates a specified directory. Note that first you must have sufficient rights to create a directory. To create nested subdirectories, you must create each subdirectory separately, starting from the top-level directory that you want to create (see example below.)

Examples

- This example assumes that C:\TEST exists and uses DIRMAKE() twice to create a nested subdirectory under it:

```
DIRMAKE("c:\test\one")    // Create top-most one
nResult := DIRMAKE("c:\test\one\two")
IF nResult != 0
   ? "Cannot make directory, DOS error ", nResult
   BREAK
ENDIF
```

You may also use something like this:

```
DIRMAKE( ".\test" )
```

Files Library is EXTEND.LIB.

See Also DIRCHANGE(), DIRREMOVE()

DIRREMOVE() function

Remove a directory

Syntax

```
DIRREMOVE(<cDirName>) → nSuccess
```

Arguments

<cDirName> is the name of the directory to erase, including an optional drive. If you do not specify a drive, the current one is used.

Returns

DIRREMOVE() returns 0 if successful; -1 if there is an argument error. Otherwise, DIRREMOVE returns the DOS error code.

Description

DIRREMOVE() removes a specified directory. Note that you must first have sufficient rights to delete a directory. A directory must be empty in order to be deleted. Therefore, to delete a directory that contains subdirectories, you must first delete the subdirectories (see example below).

Examples

- This example uses DIRREMOVE() to delete a subdirectory named C:\TEST\ONE, which only contains an empty subdirectory named C:\TEST\ONE\TWO:

```
DIRREMOVE("c:\test\one\two")        // First delete lowest dir
nResult := DIRREMOVE("c:\test\one")  // Then delete higher dir
IF nResult != 0
    ? "Cannot remove directory, DOS error ", siResult
    BREAK
ENDIF
```

Files

Library is EXTEND.LIB.

See Also

DIRCHANGE(), DIRMAKE()

D

DISKCHANGE() function

Change the current DOS disk drive

Syntax

```
DISKCHANGE(<cDrive>) → lSuccess
```

Arguments

<cDrive> specifies the letter of the disk drive to change to.

Returns

DISKCHANGE() returns true (.T.) if successful; otherwise, it returns false (.F.).

Examples

- This example uses DISKCHANGE() to change to drive "D":

```
IF DISKCHANGE("D:")
   ? "Successfully changed"
ELSE
   ? "Not changed"
ENDIF
```

- This example builds a string that contains all currently available drives on your system:

```
FUNCTION AllDrives()
   LOCAL wI, cDrives := ""
   FOR wI := 1 TO 26
      IF DISKCHANGE( Chr(wI + 64) )
         cDrives := cDrives + Chr(wI + 64)
      ENDIF
   NEXT
RETURN cDrives
```

Files Library is EXTEND.LIB.

See Also DIRCHANGE(), DISKNAME()

DISKNAME() function

Return the current DOS drive

Syntax

```
DISKNAME() → cDrive
```

Returns

DISKNAME() returns the letter of the current DOS drive, without a trailing colon.

Examples

- This example illustrates the relationship between DISKNAME()and DISKCHANGE() and shows that DISKNAME() is unaffected by the SET DEFAULT TO command:

```
? DISKNAME()    // C
SET DEFAULT TO A
? DISKNAME()    // C
DISKCHANGE("A")
? DISKNAME()    // A
DISKCHANGE("C")
? DISKNAME()    // C
```

Files Library is EXTEND.LIB.

See Also CURDIR(), DISKCHANGE(), ISDISK()

DISKSPACE() function

Return the space available on a specified disk

Syntax

```
DISKSPACE([<nDrive>]) → nBytes
```

Arguments

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

Returns

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

Description

DISKSPACE() is an environment function that determines the number of available bytes remaining on the specified disk drive. It is useful when COPYing or SORTing to another drive to determine if there is enough space available before initiating the operation. You may also use DISKSPACE() with RECSIZE() and RECCOUNT() to create a procedure to back up database files.

DISKSPACE() ignores the SET DEFAULT drive setting.

Examples

- This example is a user-defined function that demonstrates the use of DISKSPACE() to back up a database file to another drive:

```
FUNCTION BackUp( cTargetFile, cTargetDrive )
   LOCAL nSpaceNeeded, nTargetDrive
   //
   nSpaceNeeded := INT((RECSIZE() * ;
      LASTREC()) + HEADER() + 1)
   nTargetDrive := ASC(UPPER(cTargetDrive)) - 64
   //
   IF DISKSPACE(nTargetDrive) < nSpaceNeeded
      RETURN .F.
   ENDIF
   COPY TO (cTargetDrive + ":" + cTargetFile)
   //
   RETURN .T.
```

Files

Library is EXTEND.LIB.

See Also

LASTREC(), LUPDATE(), RECSIZE()

DISPBEGIN() function

Begin buffering screen output

Syntax

```
DISPBEGIN() → NIL
```

Returns

DISPBEGIN() always returns NIL.

Description

DISPBEGIN() is a screen function that informs the CA-Clipper display output system that the application is about to perform a series of display operations.

Use DISPBEGIN() with DISPEND() to allow the display output system to buffer display updates. Display output which occurs after DISPBEGIN() but before DISPEND() is allowed to accumulate in internal buffers. When DISPEND() executes, any pending updates appear on the physical display. This is useful in applications where complex screen displays are slow and the appearance of performance is desired.

DISPBEGIN() and DISPEND() calls are optional. They are not required for normal output.

Notes

- ***Nested calls:*** DISPBEGIN() calls are nested internally. If several DISPBEGIN() calls occur, buffering is allowed until a corresponding number of DISPEND() calls occur.

- ***Guaranteed operations:*** Display updates performed between DISPBEGIN() and DISPEND() are not guaranteed to be buffered—some updates may become visible before DISPEND() is called. However, all updates are guaranteed to be visible after the closing call to DISPEND().

- ***Terminal operations:*** Terminal input operations such as INKEY() or READ should not be performed between DISPBEGIN() and DISPEND(). Doing this may cause input or display output to be lost.

- ***Incompatible operations:*** Display output by other than the CA-Clipper display functions (e.g., by add-on libraries or by DOS via OUTSTD(), etc.) may not be compatible with DISPBEGIN() and DISPEND(). Output may be lost.

Examples

- This example buffers screen output, updates the screen, and then displays the buffered screen output:

```
DISPBEGIN()            // Start screen buffering
//
SETPOS(10, 10)
DISPOUT("A display update")
SETPOS(11, 10)
DISPOUT("Another display update")
//
DISPEND()              // Display buffered screen data
```

Files

Library is CLIPPER.LIB.

See Also

DISPEND()

DISPBOX() function

Display a box on the screen

Syntax

```
DISPBOX(<nTop>, <nLeft>, <nBottom>, <nRight>,
   [<cnBoxString>], [<cColorString>]) → NIL
```

Arguments

<nTop>*, *<nLeft>*, *<nBottom>, and ***<nRight>*** define the coordinates of the box. DISPBOX() draws a box using row values from zero to MAXROW(), and column values from zero to MAXCOL(). If *<nBottom>* and *<nRight>* are larger than MAXROW() and MAXCOL(), the bottom-right corner is drawn off the screen.

<cnBoxString> is a numeric or character expression that defines the border characters of the box. If specified as a numeric expression, a value of 1 displays a single-line box and a value of 2 displays a double-line box. All other numeric values display a single-line box.

If ***<cnBoxString>*** is a character expression, it specifies the characters to be used in drawing the box. This is a string of eight border characters and a fill character. If ***<cnBoxString>*** is specified as a single character, that character is used to draw the whole box.

If this argument is not specified, a single-line box is drawn.

<cColorString> defines the display color of the box that is drawn. If not specified, the box is drawn using the standard color setting of the current system color as defined by SETCOLOR().

Returns

DISPBOX() always returns NIL.

Description

DISPBOX() is a screen function that draws a box at the specified display coordinates in the specified color. If you specify <*cnBoxString*>, DISPBOX() draws a box on the screen using configurable border and fill characters. DISPBOX() draws the box using <*cnBoxString*> starting from the upper left-hand corner, proceeding clockwise and filling the screen region with the ninth character. If the ninth character is not specified, the screen region within the box is not painted. Existing text and color remain unchanged.

In cases where cnBoxString respects CA-Clipper conventions, the behavior of DISPBOX() is unchanged. The behavior of this function can easily be modified to take advantage of graphic mode. For example, you can replace the standard window frames using single or double lines with new graphical frames that have an impressive 3-D look. Simply replace the cBoxString parameter using the following:

```
CHR(2) + CHR(nColor+1) // draws a box of thickness 16x8x16x8
CHR(3) + CHR(nColor+1) // draws a box of thickness 8x8x8x8
CHR(4) + CHR(nColor+1) // draws a box of thickness
                       // 16x16x16x16
CHR(5) + CHR(nColor+1) // draws a box of thickness 16x8x8x8
```

Note that <nColor> is a numeric color representation. You must add 1 to this value.

In general, CHR(2) + CHR(nColor+1) can be used instead of CA-Clipper's B_SINGLE or B_DOUBLE defines.

CA-Clipper graphics comes with two #defines LLG_BOX_GRAY_STD and LLG_BOX_GRAY_SQUARE to allow gray (nColor=7) boxes of width 16x8 or 16x16.

You can completely customize the box by passing chr(1) + ... as the first parameter:

```
CHR(1)              + ;  // Box entirely defined
CHR(nBackColor+1)   + ;  // Color used as background fill
CHR(nLightColor+1)  + ;  // Color used to lighten the frame
CHR(nDarkColor+1)   + ;  // Color used to darken the frame
CHR(nWidthUp)       + ;  // Thickness of upper edge of box
CHR(nWidthRight)    + ;  // Thickness of right edge of box
CHR(nWidthDown)     + ;  // Thickness of lower edge of box
CHR(nWidthLeft)          // Thickness of left edge of box
```

After DISPBOX() executes, the cursor is located in the upper corner of the boxed region at <*nTop*> + 1 and <*nLeft*> + 1. ROW() and COL() are also updated to reflect the new cursor position.

Note that Box.ch, located in \CLIP53\INCLUDE, provides constants for various border configurations.

Notes

The number of colors available depends on the current video mode setting (SET VIDEOMODE).

Examples

- This code example displays a double-line box using a numeric value to specify the box border:

```
#define B_SINGLE  1
#define B_DOUBLE  2
//
DISPBOX(1, 1, 10, 10, B_DOUBLE, "BG+/B")
```

- This example displays a single-line top and double-line side box by specifying border characters with a manifest constant defined in Box.ch:

```
#include "Box.ch"
//
DISPBOX(1, 1, 10, 10, B_SINGLE_DOUBLE, "BG+/B")
```

- This example displays a box with a 3-D look. It can be used for graphic mode:

```
// Display a box with a 3D look of constant width 16x16x16x16
DISPBOX( nTop, nLeft, nBottom, nRight, LLG_BOX_GRAY_SQUARE )
// Write some transparent text in the 3D frame
GWRITEAT(  nLeft * GMODE()[LLG_MODE_FONT_COL] ,;
nTop   * GMODE()[LLG_MODE_FONT_ROW] ,;
"This is some Text...",;
4,;
LLG_MODE_SET; )
```

Files

Library is LLIBG.LIB, header file is Llibg.ch.

See Also

GFRAME(), GMODE(), SET VIDEOMODE

DISPCOUNT() function

Return the number of pending DISPEND() requests

Syntax

```
DISPCOUNT() → nDispCount
```

Returns

DISPCOUNT() returns the number of DISPEND() calls required to restore the original display context.

Description

You can use DISPCOUNT() to determine the current display context. CA-Clipper uses display contexts to buffer and to supervise screen output operations.

Each call to DISPBEGIN() defines a new display context. Output to the display context is suppressed until a matching DISPEND() statement executes.

Since you may nest DISPBEGIN() calls, use DISPCOUNT() to determine whether there are pending screen refresh requests.

Examples

- This example saves the setting of DISPCOUNT(), then releases all pending display contexts before writing to the screen:

```
PROCEDURE ForceDisplay(cExp)
   LOCAL nSavCount

nSavCount := DISPCOUNT()

//  Discard pending display contexts
DO WHILE ( DISPCOUNT() > 0)
   DISPEND()
ENDDO

DISPOUT(cExp)

//  "Rewind" the current display context
DO WHILE (DISPCCOUNT() < nSavCount )
   DISPBEGIN()
ENDDO

RETURN
```

Files

Library is CLIPPER.LIB.

See Also

DISPBEGIN(), DISPEND()

DISPEND() function

Display buffered screen updates

Syntax

```
DISPEND() → NIL
```

Returns

DISPEND() always returns NIL.

Description

DISPEND() is a screen function that informs the CA-Clipper display output system that the application has finished performing a series of display operations.

DISPEND() is used with DISPBEGIN() so the display output system can buffer display updates. This can be important for applications in which complex screen displays are slow and the appearance of performance is desired.

Examples

- This example buffers screen output, updates the screen, and then displays the buffered screen output:

```
DISPBEGIN()             // Start screen buffering
//
SETPOS(10, 10)
DISPOUT("A display update")
SETPOS(11, 10)
DISPOUT("Another display update")
//
DISPEND()               // Display buffered screen data
```

Files Library is CLIPPER.LIB.

See Also DISPBEGIN()

D

DISPLAY command

Display records to the console

Syntax

```
DISPLAY <exp list>
   [TO PRINTER] [TO FILE <xcFile>]
   [<scope>] [WHILE <lCondition>]
   [FOR <lCondition>] [OFF]
```

Arguments

<exp list> is the list of values to display for each record processed.

TO PRINTER echoes output to the printer.

TO FILE <xcFile> echoes output to the indicated file which can be specified either as a literal file name or as a character expression enclosed in parentheses. If an extension is not specified, .txt is added.

<scope> is the portion of the current database file to DISPLAY. The default is the current record, or NEXT 1. If a condition is specified, the scope becomes ALL.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to DISPLAY within the given scope.

OFF suppresses the display of the record number.

Description

DISPLAY is a database command that sequentially accesses records in the current work area, sending the results of the *<exp list>* to the console in a tabular format with each column separated by a space. The command does not display column headers or pause at predetermined intervals. DISPLAY is identical to LIST with the exception that its default scope is NEXT 1 rather than ALL.

When invoked, output is sent to the screen and optionally to the printer and/or a file. To suppress output to the screen while printing or echoing output to a file, SET CONSOLE OFF before the DISPLAY command line.

Notes

- ***Interrupting output:*** To let the user interrupt the processing of a DISPLAY command, using the INKEY() function, add a test for the interrupt key press to the FOR condition. See the example below.

- ***Printer margin:*** Since DISPLAY is a console command, it honors the current SET MARGIN for output echoed to the printer.

Examples

- This example illustrates a simple DISPLAY, and a conditional DISPLAY to the printer:

```
USE Sales NEW
DISPLAY DATE(), TIME(), Branch
DISPLAY Branch, Salesman FOR Amount > 500 TO PRINTER
```

- This example interrupts a DISPLAY using INKEY() to test whether the user pressed the Esc key:

```
#define K_ESC  27
USE Sales INDEX SalesMan NEW
DISPLAY Branch, Salesman, Amount WHILE ;
   INKEY() != K_ESC
```

Files

Library is CLIPPER.LIB.

See Also

DBEVAL(), INKEY(), LIST, SET MARGIN

DISPOUT() function

Write a value to the display

Syntax

```
DISPOUT(<exp>, [<cColorString>]) → NIL
```

Arguments

<exp> is the value to display.

<cColorString> is an optional argument that defines the display color of *<exp>*. If unspecified, *<exp>* is displayed as the standard color of the current system color as defined by SETCOLOR(). *<cColorString>* is a character expression containing the standard color setting. You can specify a literal color setting, if you enclose it in quote marks.

Returns

DISPOUT() always returns NIL.

Description

DISPOUT() is a simple output function that writes the value of a single expression to the display at the current cursor position. This function ignores the SET DEVICE setting; output always goes to the screen. You can only use this function within a procedure or function.

Examples

- This example performs screen output at a specified location in different colors. Note how the cursor position is saved and restored using ROW(), COL(), and SETPOS():

```
PROCEDURE Showit
   LOCAL nRow, nCol
   ? nCol := COL()              // save original
   ?? nRow := ROW()             // cursor position

   INKEY(2)

   SETPOS(nRow, nCol)
   DISPOUT("This is a test of DISPOUT()")
   ? COL()                      // display current
   ?? ROW()                     // cursor position

   INKEY(2)

   SETPOS(nRow, nCol)
   DISPOUT(space(26))           // clear original position
   SET DEVICE TO PRINTER        // ignores SET DEVICE
   SETPOS(nRow, nCol)           // display at
   DISPOUT("          all through")
                                // original position

   RETURN
```

Files Library is CLIPPER.LIB.

See Also COL(), OUTSTD(), QOUT(), ROW(), SETCOLOR(), SETPOS()

DO* statement

Call a procedure

Syntax

```
DO <idProcedure> [WITH <argument list>]
```

Arguments

<idProcedure> is the name of the procedure or user-defined function to be executed.

WITH <argument list> specifies up to 128 arguments, separated by commas, to pass to *<idProcedure>*. Each argument may be a single variable, field, array, array element, expression, or object. Arguments can be skipped or left off the end of the list.

Description

The DO statement calls a procedure or user-defined function, optionally passing arguments to the called routine. It performs the same action as a user-defined function or procedure specified on a line by itself with the exception that variables other than field variables are passed by reference as the default. In order to pass a field variable as an argument, enclose it in parentheses, unless you declare it with the FIELD statement or with an alias.

In CA-Clipper, the number of specified arguments need not match the number of specified parameters in the called procedure. If the number of arguments is less than the number of parameters, the parameter variables with no corresponding arguments are initialized with a NIL value when the procedure is called. If the number of arguments is greater than the number of parameters, they are ignored.

Also, skipping an argument within the *<argument list>* by leaving an empty spot next to the comma initializes the corresponding argument to NIL. To detect the position of the last argument passed in the *<argument list>*, use PCOUNT(). To detect a skipped argument, compare the receiving parameter to NIL.

In addition to calling a procedure or user-defined function, DO also has an effect on compilation if you compile the current program file without the /M option. If the CA-Clipper compiler encounters a DO statement and the specified procedure has not already been compiled, the compiler searches the current directory for a .prg file with the same name and compiles it. If the file with the same name as the procedure is not found, the called procedure is assumed to be external, and a reference is added to the object (.OBJ) file. At link time, the linker will search other object files and libraries for this external reference.

In CA-Clipper, DO is a compatibility statement and therefore not recommended. Calling a procedure or function on a line by itself is the preferred method. Since this preferred calling convention normally passes parameters by value, you must preface an argument with the pass-by-reference operator (@) in order to pass by reference. If you are using DO to make a procedure call more readable, a user-defined command, specified with the #command directive, can provide greater readability without sacrificing the safety of variables passed as parameters.

For more information on passing parameters refer to the Functions and Procedures section of the "Basic Concepts" chapter in the *Programming and Utilities Guide*.

Examples

- This example executes a procedure with no parameters:

```
DO AcctsRpt
AcctsRpt()                          // Preferred method
```

- This example executes a procedure passing two constants:

```
DO QtrRpt WITH "2nd", "Sales Division"
QtrRpt("2nd", "Sales Division")     // Preferred method
```

- In this example, a procedure is executed with the first argument passed by value and the second passed by reference:

```
nNumber := 12
DO YearRpt WITH nNumber + 12, nNumber
YearRpt(nNumber + 12, @nNumber)     // Preferred method
```

- Here, a procedure is invoked with skipped arguments embedded in the list of arguments:

```
DO DisplayWindow WITH ,,,,"My Window"
DisplayWindow(,,,,"My Window")      // Preferred method
```

See Also FUNCTION, LOCAL, PARAMETERS, PRIVATE, PROCEDURE, PUBLIC, RETURN, FIELD

D

DO CASE statement

Execute one of several alternative blocks of statements

Syntax

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

Arguments

CASE <lCondition> defines a block of statements to be executed if *<lCondition>* evaluates to true (.T.).

OTHERWISE defines a block of statements to be executed if none of the specified CASE conditions evaluates to true (.T.).

Description

DO CASE...ENDCASE is a control structure that executes one of several blocks of statements depending on which of the associated conditions is true (.T.). It works by branching execution to the statements following the first CASE *<lCondition>* that evaluates to true (.T.). Execution continues until the next CASE, OTHERWISE, or ENDCASE is encountered. Control then branches to the first statement following the next ENDCASE statement.

If none of the CASE conditions evaluates to true (.T.), the statements following the OTHERWISE statement are executed up to the matching ENDCASE statement. If an OTHERWISE statement is omitted, control branches to the first statement following the matching ENDCASE statement.

You may nest any number of statements, including other control structures (i.e., DO WHILE and FOR), within a single DO CASE structure. In addition, within a single DO CASE structure, there is no fixed limit on the number of CASE statements that a DO CASE structure may contain.

DO CASE...ENDCASE is identical to IF...ELSEIF...ENDIF with neither syntax having a specific advantage over the other.

Examples

- This example uses DO CASE in a menu structure to branch control based on user selection:

```
@ 3, 25 PROMPT "First choice"
@ 4, 25 PROMPT "Second choice"
MENU TO nChoice
//
DO CASE
CASE nChoice = 0
    RETURN
CASE nChoice = 1
    ChoiceOne()
CASE nChoice = 2
    ChoiceTwo()
ENDCASE
```

See Also BEGIN SEQUENCE, DO WHILE, FOR, IF, IF()

DO WHILE statement

Execute a loop while a condition is true (.T.)

Syntax

```
[DO] WHILE <lCondition>
   <statements>...
   [EXIT]
   <statements>...
   [LOOP]
   <statements>...
END[DO]
```

Arguments

<lCondition> is the logical control expression for the DO WHILE loop.

EXIT unconditionally branches control from within a DO WHILE or FOR...NEXT structure to the statement immediately following the corresponding ENDDO or NEXT statement.

LOOP branches control to the most recently executed DO WHILE or FOR statement.

Description

DO WHILE...ENDDO is a control structure that executes a block of statements repetitively, as long as *<lCondition>* evaluates to true (.T.). When the condition evaluates to true (.T.), control passes into the structure and proceeds until an EXIT, LOOP, or ENDDO is encountered. ENDDO returns control to the DO WHILE statement and the process repeats itself. If an EXIT statement is encountered, control branches to the nearest ENDDO or NEXT statement. If a LOOP statement is encountered, control branches to the nearest DO WHILE or FOR statement. If the condition evaluates to false (.F.), the DO WHILE construct terminates and control passes to the statement immediately following the ENDDO.

Use EXIT to terminate a DO WHILE structure based on a condition other than the DO WHILE condition. LOOP, by contrast, prevents execution of statements within a DO WHILE based on an intermediate condition, and returns to the most recent DO WHILE statement.

DO WHILE constructs may be nested within any other control structures to any depth. The only requirement is that each control structure be properly nested.

Examples

- This example demonstrates a typical control structure for a simple grouped report:

```
LOCAL cOldSalesman, nTotalAmount
USE Sales INDEX Salesman NEW
DO WHILE .NOT. EOF()
    cOldSalesman := Sales->Salesman
    nTotalAmount := 0
    DO WHILE cOldSalesman = Sales->Salesman ;
        .AND. (.NOT. EOF())
        ? Sales->Salesman, Sales->Amount
        nTotalAmount := nTotalAmount + Sales->Amount
        SKIP
    ENDDO
    ? "Total: ", nTotalAmount, "for", cOldSalesman
ENDDO
CLOSE Sales
```

- This code fragment demonstrates how LOOP can be used to provide an intermediate processing condition:

```
DO WHILE <lCondition>
    <initial processing>...
    IF <intermediate condition>
        LOOP
    ENDIF
    <continued processing>...
ENDDO
```

- This example demonstrates the use of DO WHILE to emulate a repeat until looping construct:

```
LOCAL lMore := .T.
DO WHILE lMore
    <statements>...
    lMore := (<lCondition>)
ENDDO
```

- This example uses a DO WHILE loop to move sequentially through a database file:

```
DO WHILE .NOT. EOF()
    <statements>...
    SKIP
ENDDO
```

See Also

BEGIN SEQUENCE, DBEVAL(), DO CASE, FOR, IF, IF(), RETURN

DOSERROR() function

Return the last DOS error number

Syntax

```
DOSERROR([<nNewOsCode>]) → nOsCode
```

Arguments

<nNewOsCode>, if specified, alters the value returned by DOSERROR(). The value must be a numeric value that reflects a DOS error number.

Returns

DOSERROR() returns the DOS error number as an integer numeric value.

Description

DOSERROR() is an error function that returns the last DOS error code associated with an activation of the runtime error block. When a runtime error occurs, the DOSERROR() function is set to the current DOS error if the operation has an associated DOS error. The function value is retained until another runtime error occurs. If the failed operation has no associated DOS error, the DOSERROR() returns zero. With low-level file functions, FERROR() returns the same value as DOSERROR().

Through use of the optional <*nNewOsCode*>, you may customize to the reporting activation the returned value for any DOS error.

For a complete list of DOS error numbers and accompanying descriptions, refer to the *Error Messages and Appendices Guide*.

Files Library is CLIPPER.LIB.

See Also ERRORBLOCK(), FERROR()

D

DOW() function

Convert a date value to a numeric day of the week

Syntax

```
DOW(<dDate>) → nDay
```

Arguments

<*dDate*> is a date value to convert.

Returns

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

Description

DOW() is a date conversion function that converts a date value to a number identifying the day of the week. It is useful when you want date calculations on a weekly basis. DOW() is similar to CDOW(), which returns the day of week as a character string instead of a number.

Examples

- These examples illustrate CDOW() and its relationship to DOW():

```
? DATE()                          // Result: 09/01/89
? DOW(DATE())                     // Result: 3
? CDOW(DATE())                    // Result: Tuesday
? DOW(DATE() - 2)                 // Result: 1
? CDOW(DATE() - 2)                // Result: Sunday
```

- This is a user-defined function that uses DOW() to calculate the date of last Monday from any other date:

```
FUNCTION LastMonday(dDate)
   RETURN (dDate - DOW(dDate) + 2)
```

Files

Library is CLIPPER.LIB.

See Also

CDOW(), DATE(), DAY()

DTOC() function

Convert a date value to a character string

Syntax

```
DTOC(<dDate>) → cDate
```

Arguments

<dDate> is the date value to convert.

Returns

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

Description

DTOC() is a date conversion function used for formatting purposes when you want to display the date in the SET DATE format and when a character expression is required (in a LABEL FORM, for example). If you need a specialized date format, you can use TRANSFORM() or a custom expression.

If you are INDEXing a date in combination with a character string, use DTOS() instead of DTOC() to convert the date value to a character string.

Examples

- These examples show general uses of DTOC():

```
? DATE()                    // Result: 09/01/90
? DTOC(DATE())              // Result: 09/01/90
? "Today is " + DTOC(DATE())
                            // Result: Today is 09/01/90
```

Files

Library is CLIPPER.LIB.

See Also

CTOD(), DATE(), DTOS(), SET CENTURY, SET DATE, TRANSFORM()

DTOS() function

Convert a date value to a character string formatted as yyyymmdd

Syntax

```
DTOS(<dDate>) → cDate
```

Arguments

<dDate> is the date value to convert.

Returns

DTOS() returns a character string eight characters long in the format *yyyymmdd*. When *<dDate>* is a null date (CTOD("")), DTOS() returns a string of eight spaces. The return value is not affected by the current date format.

Description

DTOS() is a date conversion function that is used when creating index expressions consisting of a date value and a character expression. DTOS() converts a date value to a character string that can be concatenated to any other character expression. The return value is structured to preserve date order (year, month, and day).

Examples

- These examples illustrate DTOS() in conjunction with several other functions:

```
? DATE()                         // Result: 09/01/90
? DTOS(DATE())                   // Result: 19900901
? LEN(DTOS(CTOD("")))            // Result: 8
```

- This example demonstrates how to create an index with a compound date and character key using DTOS():

```
USE Sales NEW
INDEX ON DTOS(Date) + Salesman TO DateName
```

Files

Library is CLIPPER.LIB.

See Also

CTOD(), DATE(), DTOC(), INDEX

EJECT command

Advance the printhead to top of form

Syntax

```
EJECT
```

Description

EJECT is a printing command that sends a formfeed character (CHR(12)) to the printer and sets the PCOL() and PROW() values to zero. If you address a printer row less than the last row position since an EJECT or SETPRC() was executed, CA-Clipper automatically performs an EJECT. Because of this, your printing logic must proceed sequentially from left to right down the page. If you need to reset the internal printer row and column values to zero without sending a formfeed, use SETPRC().

Examples

- This example prints a simple list report and uses EJECT to advance to a new page when the line counter reaches the maximum number of lines to print per page:

```
LOCAL nLine := 99, nPage := 0
USE Sales NEW
SET PRINTER ON
SET CONSOLE OFF
DO WHILE !EOF()
    IF nLine > 55
       EJECT
       ? "Page " + LTRIM(STR(++nPage, 3))
       ? "Date " + CTOD(DATE())
       ?
       ? "Salesman", "Amount"
       ?
       nLine := 6
    ENDIF
    ? Sales->Salesman, Sales->Amount
    nLine++
    SKIP
ENDDO
SET PRINTER OFF
SET CONSOLE ON
CLOSE
```

Files

Library is CLIPPER.LIB.

See Also

ISPRINTER(), PCOL(), PROW(), SET CONSOLE, SET DEVICE, SET PRINTER, SETPRC()

EMPTY() function

Determine if the result of an expression is empty

Syntax

```
EMPTY(<exp>) → lEmpty
```

Arguments

<exp> is an expression of any data type.

Returns

EMPTY() returns true (.T.) if the expression results in an empty value; otherwise, it returns false (.F.). The criteria for determining whether a value is considered empty depends on the data type of *<exp>* according to the following rules:

List of Empty Values

Data Type	Contents
Array	Zero-length
Character	Spaces, tabs, CR/LF, or ("")
Numeric	0
Date	Null(CTOD(""))
Logical	False (.F.)
Memo	Same as character
NIL	NIL

Description

The EMPTY() function has a number of uses. You can use it to determine if a user entered a value into a Get object before committing changes to a database file. It can also determine whether a formal parameter is NIL or unsupplied. In addition, it can test an array for zero-length.

Notes

- ***Space characters:*** The EMPTY() function treats carriage returns, line feeds, and tabs as space characters and removes these as well.

Examples

- These examples illustrate use of EMPTY() against several different data types:

```
? EMPTY(SPACE(5)), EMPTY("")        // Result: .T. .T.
? EMPTY(0), EMPTY(CTOD(""))         // Result: .T. .T.
? EMPTY(.F.), EMPTY(NIL)            // Result: .T. .T.
```

- This example uses EMPTY() to determine whether the user entered a value into the first Get object before writing the new value to the database file:

```
LOCAL cCust := SPACE(15), nAmount := 0
USE Sales NEW
@ 10, 10 GET cCust
@ 11, 10 GET nAmount PICTURE "999.99"
READ
//
IF !EMPTY(cCust)
   APPEND BLANK
   REPLACE Sales->Cust WITH cCust, Sales->Amount ;
      WITH nAmount
ENDIF
```

- This example uses EMPTY() as part of the VALID clause to force the user to enter data into the current Get object:

```
LOCAL cCode := SPACE(5)
@ 2, 5 SAY "Enter code" GET cCode VALID !EMPTY(cCode)
READ
```

Files Library is CLIPPER.LIB.

See Also LEN()

EOF() function

Determine when end of file is encountered

Syntax

```
EOF() → lBoundary
```

Returns

EOF() returns true (.T.) when an attempt is made to move the record pointer beyond the last logical record in a database file; otherwise, it returns false (.F.). If there is no database file open in the current work area, EOF() returns false (.F.). If the current database file contains no records, EOF() returns true (.T.).

Description

EOF() is a database function used to test for an end of file boundary condition when the record pointer is moving forward through a database file. Any command that can move the record pointer can set EOF().

The most typical application is as a part of the *<lCondition>* argument of a DO WHILE construct that sequentially processes records in a database file. Here *<lCondition>* would include a test for .NOT. EOF(), forcing the DO WHILE loop to terminate when EOF() returns true (.T.).

EOF() and FOUND() are often used interchangeably to test whether a SEEK, FIND, or LOCATE command failed. With these commands, however, FOUND() is preferred.

When EOF() returns true (.T.), the record pointer is positioned at LASTREC() + 1 regardless of whether there is an active SET FILTER or SET DELETED is ON. Further attempts to move the record pointer forward return the same result without error. Once EOF() is set to true (.T.), it retains its value until there is another attempt to move the record pointer.

By default, EOF() operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression (see the following example).

Examples

- This example demonstrates EOF() by deliberately moving the record pointer beyond the last record:

```
USE Sales
GO BOTTOM
? EOF()             // Result: .F.
SKIP
? EOF()             // Result: .T.
```

- This example uses aliased expressions to query the value of EOF() in unselected work areas:

```
USE Sales NEW
USE Customer NEW
? Sales->(EOF())
? Customer->(EOF())
```

- This example illustrates how EOF() can be used as part of a condition for sequential database file operations:

```
USE Sales INDEX CustNum NEW
DO WHILE !EOF()
   nOldCust := Sales->CustNum
   nTotalAmount := 0
   DO WHILE nOldCust = Sales->CustNum .AND. (!EOF())
      ? Sales->CustNum, Sales->Description, ;
            Sales->SaleAmount
      nTotalAmount += Sales->SaleAmount
      SKIP
   ENDDO
   ? "Total amount: ", nTotalAmount
ENDDO
```

Files Library is CLIPPER.LIB.

See Also BOF(), DO WHILE, FOUND(), GO, LASTREC(), LOCATE, RECNO(), SEEK, SKIP

ERASE command

Remove a file from disk

Syntax

```
ERASE | DELETE FILE <xcFile>
```

Arguments

<xcFile> is the name of the file to be deleted from disk and can be specified either as a literal file name or as a character expression enclosed in parentheses. The file name must be fully specified including the extension, and it may optionally be preceded by a drive and/or path specification.

Description

ERASE is a file command that removes a specified file from disk. SET DEFAULT and SET PATH do not affect ERASE. The file is deleted from disk only if found in the current DOS directory or in the directory explicitly specified as part of the file name.

Warning! *Files must be CLOSEd before being ERASEd. Otherwise, either a sharing violation or a date corruption on the drive may occur.*

Examples

- This example removes a specified file from disk and then tests to see if the file was in fact removed:

```
? FILE("Temp.dbf")          // Result: .T.
ERASE Temp.dbf
? FILE("Temp.dbf")          // Result: .F.
```

Files Library is CLIPPER.LIB.

See Also CLOSE, CURDIR(), FERASE(), FILE(), FRENAME(), RENAME, USE

Error class

Provide objects containing information about runtime errors

Class Function

`ErrorNew() → oError`

Returns

ErrorNew() returns a new Error object.

Description

An Error object is a simple object that contains information pertaining to a runtime error. Error objects have no methods, only exported instance variables. When a runtime error occurs, CA-Clipper creates a new Error object and passes it as an argument to the error handler block specified with the ERRORBLOCK() function. Within the error handler, the Error object can then be queried to determine the nature of the error condition.

Error objects can also be returned to the RECOVER statement of a BEGIN SEQUENCE construct with a BREAK statement. Here, the error object can be queried for local error handling. For more detailed information and examples refer to the "Error Handling Strategies" chapter in the *Programming and Utilities Guide.*

Exported Instance Variables

`args` (Assignable)

Contains an array of the arguments supplied to an operator or function when an argument error occurs. For other types of errors, Error:args contains a NIL value.

`canDefault` (Assignable)

Contains a logical value indicating whether the subsystem can perform default error recovery for the error condition. A value of true (.T.) indicates that default recovery is available. Availability of default handling and the actual default recovery strategy depends on the subsystem and the error condition. The minimum action is simply to ignore the error condition.

Default recovery is requested by returning false (.F.) from the error block invoked to handle the error. Note that Error:canDefault is never true (.T.) if Error:canSubstitute is true (.T.).

`canRetry` (Assignable)

Contains a logical value indicating whether the subsystem can retry the operation that caused the error condition. A value of true (.T.) indicates that a retry is possible. Retry may or may not be available, depending on the subsystem and the error condition.

Retry is requested by returning true (.T.) from the error block invoked to handle the error. Note that Error:canRetry never contains true (.T.) if Error:canSubstitute contains true (.T.).

`canSubstitute` (Assignable)

Contains a logical value indicating whether a new result can be substituted for the operation that produced the error condition. Argument errors and certain other simple errors allow the error handler to substitute a new result value for the failed operation. A value of true (.T.) means that substitution is possible.

The substitution is performed by returning the new result value from the code block invoked to handle the error. Note that Error:canSubstitute is never true (.T.) if either Error:canDefault or Error:canRetry is true (.T.).

`cargo` (Assignable)

Contains a value of any data type unused by the Error system. It is provided as a user-definable slot, allowing arbitrary information to be attached to an Error object and retrieved later.

`description` (Assignable)

Contains a character string that describes the error condition. A zero-length string indicates that the subsystem does not provide a printable description for the error. If Error:genCode is not zero, a printable description is always available.

`filename` (Assignable)

Contains a character value representing the name originally used to open the file associated with the error condition. A zero-length string indicates either that the error condition is not associated with a particular file or that the subsystem does not retain file name information.

`genCode` (Assignable)

Contains an integer numeric value representing a CA-Clipper generic error code. Generic error codes allow default handling of similar errors from different subsystems. A value of zero indicates that the error condition is specific to the subsystem and does not correspond to any of the generic error codes. The header file, Error.ch, provides a set of manifest constants for generic error codes.

`operation` (Assignable)

Contains a character string that describes the operation being attempted when the error occurred. For operators and functions, Error:operation contains the name of the operator or function. For undefined variables or functions, it contains the name of the variable or function. A zero-length string indicates that the subsystem does not provide a printable description of the operation.

`osCode` (Assignable)

Contains an integer numeric value representing the operating system error code associated with the error condition. A value of zero indicates that the error condition was not caused by an error from the operating system. When Error:osCode is set to a value other than zero, DOSERROR() is updated with the same value.

Error:osCode properly reflects the DOS extended error code for file errors. This allows proper distinction between errors which result from sharing violations (e.g., opening EXCLUSIVE when another process has already opened the file) and access violations (e.g., opening read/write when the file is marked read-only).

For a list of DOS error codes refer to the *Error Messages and Appendices Guide*.

`severity` (Assignable)

Contains a numeric value indicating the severity of the error condition. Four standard values are defined in Error.ch:

Error:severity Values

Error.ch	Meaning
ES_WHOCARES	The condition does not represent a failure; the error is informational.
ES_WARNING	The condition does not prevent further operations, but may result in a more serious error later.
ES_ERROR	The condition prevents further operations without corrective action of some kind.
ES_CATASTROPHIC	The condition requires immediate termination of the application.

Note that the CA-Clipper runtime support code only generates errors with severities of ES_WARNING or ES_ERROR.

`subCode` (Assignable)

Contains an integer numeric value representing a subsystem-specific error code. A value of zero indicates that the subsystem does not assign any particular number to the error condition.

`subSystem` (Assignable)

Contains a character string representing the name of the subsystem generating the error. For errors with basic CA-Clipper operators and functions, the subsystem name "BASE" is given. For errors generated by a database driver, Error:subSystem contains the name of the database driver.

`tries` (Assignable)

Contains an integer numeric value representing the number of times the failed operation has been attempted. When Error:canRetry is true (.T.), Error:tries can be used to limit the number of retry attempts. A value of zero indicates that the subsystem does not track the number of times the operation has been tried.

Examples

- This example demonstrates how a file open operation might be handled in an error handler replicating the default CA-Clipper behavior. When, for example, an attempt to open a database file with a USE command fails, control returns to the statement following the offending command:

```
#include "Error.ch"
#command RETRY      => RETURN (.T.)    // Retry operation
#command RESUME     => RETURN (.F.)    // Default recovery
//

FUNCTION MyError( <oError> )
    //
    // Handle file open error
    IF <oError>:genCode == EG_OPEN .AND.;
       <oError>:canDefault .AND.;
       NETERR()
       //
       RESUME
    ENDIF
    .
    . <other error statements>
    .
    RETURN NIL
```

- This example retries an operation within an error handler a specified number of times:

```
#include "Error.ch"
#command RETRY      => RETURN (.T.)    // Retry operation
#command RESUME     => RETURN (.F.)    // Default recovery
//

FUNCTION MyError( <oError> )
    //
    // Handle printer not ready error
    IF <oError>:genCode == EG_PRINT .AND.;
       <oError>:canRetry .AND.;
       <oError>:tries < 25
       //
       RETRY
    ENDIF
    .
    . <other error statements>
    .
    RETURN NIL
```

- This code fragment returns an error object from an error handler to the RECOVER statement for further processing:

```
LOCAL objLocal, bLastHandler
//
// Save current and set new error handler
bLastHandler := ERRORBLOCK({   |oErr| ;
                    MyHandler(oErr, .T.)})
//
BEGIN SEQUENCE
    .
    . <operation that might fail>
    .
RECOVER USING objLocal
    .
    . <send messages to objLocal and handle the error>
    .
END
//
// Restore previous error handler
ERRORBLOCK( bLastHandler )

FUNCTION MyHandler( <oError>, lLocalHandler )
   //
   // Handle locally returning the error object
   IF lLocalHandler
      BREAK <oError>
   ENDIF
   .
   . <other statements to handle the error>
   .
   RETURN NIL
```

Files Header file is Error.ch, default error handler is in Errorsys.prg.

See Also BEGIN SEQUENCE, DOSERROR(), ERRORBLOCK(), NETERR()

ERRORBLOCK() function

Post a code block to execute when a runtime error occurs

Syntax

```
ERRORBLOCK([<bErrorHandler>]) → bCurrentErrorHandler
```

Arguments

<*bErrorHandler*> is the code block to execute whenever a runtime error occurs. When evaluated, the <*bErrorHandler*> is passed an error object as an argument by the system.

Returns

ERRORBLOCK() returns the current error handling code block. If no error handling block has been posted since the program was invoked, ERRORBLOCK() returns the default error handling block.

Description

ERRORBLOCK() is an error function that defines an error handler to execute whenever a runtime error occurs. Specify the error handler as a code block with the following form,

```
{ |<oError>| <expression list>,... }
```

where <*oError*> is an error object containing information about the error. Within the code block, messages can be sent to the error object to obtain information about the error. Returning true (.T.) from the error handling block retries the failed operation and false (.F.) resumes processing.

The error handling code block can be specified either as a list of expressions or as a call to a user-defined function. A call to a user-defined function is more useful since you can use CA-Clipper control statements instead of expressions. This is particularly the case if there is a BEGIN SEQUENCE pending and you want to BREAK to the nearest RECOVER statement.

As this implies, error handling blocks can be used in combination with BEGIN SEQUENCE...END control structures. Within an error handling block, you handle device, low-level, and common errors that have a general recovery mechanism. If the operation needs specific error handling, define a BEGIN SEQUENCE then BREAK to the RECOVER statement, returning the error object for local processing. See the example below.

If no <*bErrorHandler*> has been specified using ERRORBLOCK() and a runtime error occurs, the default error handling block is evaluated. This error handler displays a descriptive message to the screen, sets the ERRORLEVEL() to 1, then QUITs the program.

Since ERRORBLOCK() returns the current error handling block, it is possible to specify an error handling block for an operation saving the current error handling block, then restore it after the operation has completed. Also, error handlers specified as code blocks, can be passed to procedures and user-defined functions, and RETURNed as values.

For more information on the structure and operations of error objects, refer to the Error class entry in this chapter and the "Error Handling Strategies" chapter in the *Programming and Utilities Guide*.

Examples

- This code fragment posts, and then calls an error handling block when there is an error within a BEGIN SEQUENCE construct:

```
LOCAL bErrorHandler, bLastHandler, objErr
bErrorHandler := { |oError| ;
      MyErrorHandler(oError) }
//
// Save current handler
bLastHandler := ERRORBLOCK(bErrorHandler)
//
BEGIN SEQUENCE
    .
    . <operation statements>
    .
// Receive error object from BREAK
RECOVER USING oErrorInfo
    .
    . <recovery statements>
    .
END
ERRORBLOCK(bLastHandler)      // Restore handler
RETURN

FUNCTION MyErrorHandler( oError )
   //
   BREAK oError       // Return error object to RECOVER
   RETURN NIL
```

Files Library is CLIPPER.LIB.

See Also BEGIN SEQUENCE

ERRORLEVEL() function

Set the CA-Clipper return code

Syntax

```
ERRORLEVEL([<nNewReturnCode>]) → nCurrentReturnCode
```

Arguments

<*nNewReturnCode*> is the new return code setting. This can be a value between zero and 255. The default value at startup is zero. If not specified, ERRORLEVEL() reports the current setting without assigning a new value.

Returns

ERRORLEVEL() returns the current CA-Clipper exit code as a numeric value, if one has been set using ERRORLEVEL() with an argument; otherwise, it returns zero.

Description

ERRORLEVEL() is a dual purpose environment function. It returns the current CA-Clipper return code and optionally sets a new return code. The return code is a value set by a child process so the parent process can test the termination state of the child process. Typically, the parent process is DOS and the child process is an application program. Retrieve a return code with the DOS ERRORLEVEL command or INT 21 Function 4Dh.

When a CA-Clipper program terminates, the return code is set to 1 if the process ends with a fatal error. If the process ends normally, the return code is set to zero, or the last ERRORLEVEL() set in the program.

Typically, you would set a return code with ERRORLEVEL() to indicate an error state to the program that invoked the current CA-Clipper program. In most cases this is the application batch file. Here you would test the return code using the DOS ERRORLEVEL command. Refer to your DOS manual for more information.

Notes

- ERRORLEVEL() is not updated after a RUN command terminates. To obtain the return code of the invoked program, you must create an assembler or C routine that queries the child process return code using INT 21 Function 4Dh. Refer to your DOS documentation for more information.

Examples

- This example saves the current CA-Clipper return code, then sets a new value:

```
nOldCode := ERRORLEVEL()        // Get current error level
ERRORLEVEL(1)                   // Set new error level
```

- This example uses ERRORLEVEL() to set a return code that can be tested by the parent process:

```
#define ERR_FILE_MISSING      255
#define ERR_POST_INCOMPLETE   254
//
IF !FILE("Sysfile.dbf")
   @ 0, 0
   @ 1, 0
   @ 0, 0 SAY "Fatal error: System ;
            file is missing...quitting"
   ERRORLEVEL(ERR_FILE_MISSING)
   QUIT
ENDIF
```

Files Library is CLIPPER.LIB.

See Also QUIT

EVAL() function

Evaluate a code block

Syntax

```
EVAL(<bBlock>, [<BlockArg list>]) → LastBlockValue
```

Arguments

<bBlock> is the code block to be evaluated.

<BlockArg list> is a list of arguments to send to the code block before it is evaluated.

Returns

EVAL() returns the value of the last expression within the block. A code block can return a value of any type.

Description

EVAL() is a code block function. It is the most basic code block evaluation facility in the CA-Clipper system. A code block is a special data value that refers to a piece of compiled program code. For more information on code blocks, refer to the "Basic Concepts" chapter in the *Programming and Utilities Guide*.

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

The CA-Clipper compiler usually compiles a code block at compile time. There are, however, occasions at runtime when you may need to compile a code block from a character string. You can do this by using the macro operator (&).

EVAL() is often used to create iteration functions. These are functions that apply a block to each member of a data structure. AEVAL(), ASORT(), ASCAN(), and DBEVAL() are iteration functions (e.g., AEVAL() applies a block to each element within an array).

Examples

- This example creates a code block that increments a number, and then evaluates it:

```
bBlock := { |nArg| nArg + 1 }
? EVAL(bBlock, 1)                    // Result: 2
```

- This example demonstrates compiling a code block at runtime using the macro operator (&):

```
// Compile a string to a block
bBlock := &("{ |nArg| nArg + 1 }")

// Evaluate the block
? EVAL(bBlock, 1)                    // Result: 2
```

Files Library is CLIPPER.LIB.

See Also AEVAL(), ASCAN(), ASORT(), DBEVAL()

EXIT PROCEDURE statement

Declare an exit procedure

Syntax

```
EXIT PROCEDURE <idProcedure>
   [FIELD <idField list> [IN <idAlias>]]
   [LOCAL <identifier> [[:= <initializer>]]]
   [MEMVAR <identifer list>]
   .
   . <executable statements>
   .
   [RETURN]
```

Arguments

EXIT PROCEDURE declares a procedure that will be executed on program termination.

<idProcedure> is the name of the exit procedure to declare. Exit procedure names can be any length, but only the first 10 characters are significant. Names may not begin with an underscore but can contain any combination of characters, numbers, or underscores.

FIELD declares a list of identifiers to use as field names whenever encountered. If the IN clause is specified, referring to the declared name includes an implicit reference to the specified alias.

LOCAL declares and optionally initializes a list of variables or arrays whose visibility and lifetime is the current procedure.

MEMVAR declares a list of identifiers to use as private or public memory variables or arrays whenever encountered.

RETURN passes control to the next exit procedure or to the operating system, if no other exit procedures are pending.

Description

The EXIT PROCEDURE statement declares a procedure that will be executed upon program termination. EXIT procedures are called after the last executable statement in a CA-Clipper application has completed. EXIT PROCEDUREs can be used to perform common housekeeping tasks such as saving configuration settings to a file, closing a log file, or concluding a communications session.

The visibility of exit procedures is restricted to the system; therefore, it is not possible to call an EXIT PROCEDURE from a procedure or user-defined function. Exit procedures do not receive parameters.

Once the last executable statement has completed, control passes from one EXIT PROCEDURE to the next until all procedures in the exit list have been called. Control then passes to the operating system.

The ANNOUNCE statement declares a module identifier for a source (.prg) file. Once declared, EXIT PROCEDUREs are referenced with this module identifier. An application may use any number of exit procedures by explicitly REQUESTing their module identifiers.

The EXIT PROCEDUREs requested for an application are collectively referred to as the exit list. There is no mandatory execution order of procedures in the exit list; however, if an EXIT PROCEDURE is declared in the same source (.prg) file as the application's primary routine (root), it is guaranteed to be the first exit procedure called.

Termination of a given CA-Clipper application can be attributed to any of the following:

- RETURNing from the primary (root) routine
- The QUIT command
- Issuing a BREAK without an enclosing BEGIN SEQUENCE...END
- Unrecoverable error

Execution of an EXIT PROCEDURE cannot be guaranteed when the system encounters an unrecoverable error. If an error is raised during an exit procedure, the system returns to DOS. Pending exit procedures are not called.

Examples

- This example illustrates construction of a simple timing mechanism using INIT and EXIT PROCEDUREs:

```
// prints the amount of time required to read,
// sort, and display a list of file names.

ANNOUNCE MySystem

STATIC nStart

PROCEDURE Main()
   AEVAL( ASORT( DIRECTORY( "*.*" ) ),;
      { | aFileInfo | QOUT( aFileInfo[ 1 ] ) } )
   RETURN

INIT PROCEDURE MyInit()
   nStart := SECONDS()
   RETURN

EXIT PROCEDURE MyExit()
   ?
   ? "Elapsed Time: "
   ?? SECONDS() - nStart
   RETURN
```

See Also

ANNOUNCE, REQUEST, INIT PROCEDURE

EXP() function

Calculate e**x

Syntax

```
EXP(<nExponent>) → nAntilogarithm
```

Arguments

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

Returns

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

Description

EXP() is a mathematical function that calculates the value, y, (the antilogarithm) of the following equation,

```
e**x = y
```

where e is the base of natural logarithms (2.71828...) and x is *<nExponent>*. The maximum value of *<nExponent>* is 45 before a numeric overflow occurs. EXP() and LOG() are inverse functions.

The number of decimal places displayed is determined solely by SET DECIMALS regardless of the current SET FIXED value.

Examples

- This example demonstrates several invocations of EXP():

```
? EXP(1)                        // Result: 2.72
SET DECIMALS TO 10
? EXP(1)                        // Result: 2.7182818285
? LOG(EXP(1))                   // Result: 1.0000000000
```

Files

Library is CLIPPER.LIB.

See Also

LOG(), SET DECIMALS, SET FIXED

EXTERNAL* statement

Declare a list of procedure or user-defined function names to the linker

Syntax

```
EXTERNAL <idProcedure list>
```

Arguments

<idProcedure list> is the list of procedures, user-defined functions, or format procedures to add to the list of routines that will be linked into the current executable (.EXE) file.

Description

EXTERNAL is a declaration statement that specifies uncoded references to the linker. Like all other declaration statements, an EXTERNAL statement must be specified before any executable statements in either the program file, or a procedure or user-defined function definition.

During the compilation of CA-Clipper source code, all explicit references to procedures and user-defined functions are made to the linker. In some instances, there may be no references made to procedure or user-defined function names until runtime. EXTERNAL resolves this by forcing the named procedures or user-defined functions to be linked even if they are not explicitly referenced in the source file. This is important in several instances:

- Procedures, user-defined functions, or formats referenced with macro expressions or variables
- Procedures and user-defined functions used in REPORT and LABEL FORMs and not referenced in the source code

- User-defined functions used in index keys and not referenced in the source code
- ACHOICE(), DBEDIT(), or MEMOEDIT() user functions

To group common EXTERNAL declarations together, place them in a header file and then include (#include) the header file into each program (.prg) file that might indirectly use them.

EXTERNAL is a compatibility statement and therefore not recommended. It is superseded by the REQUEST statement that defines a list of module identifiers to the linker.

Examples

- These examples are equivalent header files consisting of common EXTERNAL references for REPORT FORMs:

```
// Externals.ch
EXTERNAL HARDCR
EXTERNAL TONE
EXTERNAL MEMOTRAN
EXTERNAL STRTRAN

// Externals.ch
EXTERNAL HARDCR, TONE, MEMOTRAN, STRTRAN
```

See Also

#include, REQUEST

FCLOSE() function

Close an open binary file and write DOS buffers to disk

Syntax

```
FCLOSE(<nHandle>) → lError
```

Arguments

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

Returns

FCLOSE() returns false (.F.) if an error occurs while writing; otherwise, it returns true (.T.).

Description

FCLOSE() is a low-level file function that closes binary files and forces the associated DOS buffers to be written to disk. If the operation fails, FCLOSE() returns false (.F.). FERROR() can then be used to determine the reason for the failure. For example, attempting to use FCLOSE() with an invalid handle returns false (.F.), and FERROR() returns DOS error 6, invalid handle. See FERROR() for a complete list of error numbers.

Warning! *This function allows low-level access to DOS files and devices. It should be used with extreme care and requires a thorough knowledge of the operating system.*

Examples

- This example uses FCLOSE() to close a newly created binary file and displays an error message if the close fails:

```
#include "Fileio.ch"
//
nHandle := FCREATE("Testfile", FC_NORMAL)
IF !FCLOSE(nHandle)
   ? "Error closing file, error number: ", FERROR()
ENDIF
```

Files

Library is CLIPPER.LIB.

See Also

FCREATE(), FERROR(), FOPEN()

FCOUNT() function

Return the number of fields in the current .dbf file

Syntax

```
FCOUNT() → nFields
```

Returns

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

Description

FCOUNT() is a database function. It is useful in applications containing data-independent programs that can operate on any database file. These include generalized import/export and reporting programs. Typically, you use FCOUNT() to establish the upper limit of a FOR...NEXT or DO WHILE loop that processes a single field at a time.

By default, FCOUNT() operates on the currently selected work area.

Examples

- This example illustrates FCOUNT(), returning the number of fields in the current and an unselected work area:

```
USE Sales NEW
USE Customer NEW
? FCOUNT()                          // Result: 5
? Sales->(FCOUNT())                 // Result: 8
```

- This example uses FCOUNT() to DECLARE an array with field information:

```
LOCAL aFields := ARRAY(FCOUNT())
AFIELDS(aFields)
```

- This example uses FCOUNT() as the upper boundary of a FOR loop that processes the list of current work area fields:

```
LOCAL nField
USE Sales NEW
FOR nField := 1 TO FCOUNT()
   ? FIELD(nField)
NEXT
```

Files Library is CLIPPER.LIB.

See Also AFIELDS()*, FIELDNAME(), TYPE()

FCREATE() function

Create and/or truncate a binary file to zero-length

Syntax

```
FCREATE(<cFile>, [<nAttribute>]) → nHandle
```

Arguments

<cFile> is the name of the file to create. If the file already exists, its length is truncated to zero without warning.

<nAttribute> is one of the binary file attributes shown in the table below. If this argument is omitted, the default value is zero.

Binary File Attributes

Value	Fileio.ch	Attribute	Description
0	FC_NORMAL	Normal	Create normal read/write file (default)
1	FC_READONLY	Read-only	Create read-only file
2	FC_HIDDEN	Hidden	Create hidden file
4	FC_SYSTEM	System	Create system file

Returns

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

Description

FCREATE() is a low-level file function that either creates a new file or opens and truncates an existing file. If *<cFile>* does not exist, it is created and opened for writing. If it does exist and can be opened for writing, it is truncated to zero-length. If it cannot be opened for writing, FCREATE() returns -1 and FERROR() returns the appropriate error value.

When FCREATE() successfully creates a new file, the file is left open in compatibility sharing mode and read/write access mode. The file attribute specified by the *<nAttribute>* argument is applied to the new file when it is closed, allowing writing to a newly created read-only file. For a list of access modes, see FOPEN().

Since a file handle is required in order to identify an open file to other file functions, always assign the return value from FCREATE() to a variable for later use.

Like other file functions, FCREATE() does not use either the DEFAULT or PATH settings for its operation. Instead, it writes to the current DOS directory unless a path is explicitly stated.

Warning! *This function allows low-level access to DOS files and devices. It should be used with extreme care and requires a thorough knowledge of the operating system.*

Examples

- This example creates a file called Testfile and opens it for reading and writing:

```
#include "Fileio.ch"
//
IF (nHandle := FCREATE("Testfile", FC_NORMAL)) == -1
    ? "File cannot be created:", FERROR()
    BREAK
ELSE
    FWRITE(nHandle, "Hello there")
    FCLOSE(nHandle)
ENDIF
```

Files

Library is CLIPPER.LIB, header file is Fileio.ch.

See Also

FCLOSE(), FERROR(), FOPEN()

FERASE() function

Delete a file from disk

Syntax

```
FERASE(<cFile>) → nSuccess
```

Arguments

<cFile> is the name of the file to be deleted from disk, including extension, optionally preceded by a drive and/or path specification.

Returns

FERASE() returns -1 if the operation fails and zero if it succeeds. In the case of a failure, FERROR() can be used to determine the nature of the error.

Description

FERASE() is a file function that deletes a specified file from disk. FERASE() is the same as the ERASE command but returns a value and can be specified within an expression. When FERASE() is called, *<cFile>* is deleted from disk only if found in the current DOS directory or in the directory explicitly specified as part of the file name. Like the other file functions and commands, FERASE() does not use either SET DEFAULT or SET PATH to locate *<cFile>*.

Warning! *Files must be CLOSEd before removing them with FERASE().*

Examples

- This example deletes a set of files matching a wildcard pattern:

```
#include "Directry.ch"
AEVAL(DIRECTORY("*.BAK"), { |aFile| ;
   FERASE(aFile[F_NAME]) })
```

- This example erases a file and displays a message if the operation fails:

```
IF FERASE("AFile.txt") == -1
   ? "File erase error:", FERROR()
   BREAK
ENDIF
```

Files Library is CLIPPER.LIB.

See Also CLOSE, ERASE, FERROR(), FRENAME(), RENAME

FERROR() function

Test for errors after a binary file operation

Syntax

```
FERROR() → nErrorCode
```

Returns

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

FERROR() Return Values

Error	Meaning
0	Successful
2	File not found
3	Path not found
4	Too many files open
5	Access denied
6	Invalid handle
8	Insufficient memory
15	Invalid drive specified
19	Attempted to write to a write-protected disk
21	Drive not ready
23	Data CRC error
29	Write fault
30	Read fault
32	Sharing violation
33	Lock Violation

Description

FERROR() is a low-level file function that indicates a DOS error after a file function is used. These functions include FCLOSE(), FCREATE(), FERASE(), FOPEN(), FREAD(), FREADSTR(), and FRENAME(). FERROR() retains its value until the next execution of a file function.

Warning! *This function allows low-level access to DOS files and devices. It should be used with extreme care and requires a thorough knowledge of the operating system.*

Examples

- This example tests FERROR() after the creation of a binary file and displays an error message if the create fails:

```
#include "Fileio.ch"
//
nHandle := FCREATE("Temp.txt", FC_NORMAL)
IF FERROR() != 0
   ? "Cannot create file, DOS error ", FERROR()
ENDIF
```

Files

Library is CLIPPER.LIB.

See Also

DOSERROR(), FCLOSE(), FCREATE(), FERASE(), FOPEN(), FREAD(), FREADSTR(), FRENAME(), FWRITE()

FIELD statement

Declare database field names

Syntax

```
FIELD <idField list> [IN <idAlias>]
```

Arguments

<idField list> is a list of names to declare as fields to the compiler.

IN <idAlias> specifies an alias to assume when there are unaliased references to the names specified in the *<idField list>*. Unaliased references to variables in *<idField list>* are treated as if they are preceded by the special field alias (FIELD->).

Description

The FIELD statement declares the names of database fields to the compiler, and optionally supplies an implicit alias for each name. This allows the compiler to resolve references to variables with no explicit alias--by implicitly assuming the specified *<idAlias>*. Only explicit, unaliased references to the specified fields in the *<idField list>* are affected. The FIELD statement, like all declarations, has no effect on references made within macro expressions or variables.

The FIELD statement neither opens a database file nor verifies the existence of the specified fields. It is useful primarily to ensure correct references to fields whose accessibility is known at runtime. Attempting to access the fields when the associated database is not in USE will cause an error.

The scope of the FIELD declaration is the procedure or function in which it occurs, or the entire program (.prg) file if the declaration precedes all PROCEDURE or FUNCTION declarations and the /N compiler option is specified.

FIELD statements, like other declarations, must precede any executable statements in the procedure or function definition or the program (.prg) file if the declaration has filewide scope.

FIELD used with the /W compiler option performs compile-time checking for undeclared variables.

For more information on variable declarations and scoping, refer to the Variables section in the "Basic Concepts" chapter of the *Programming and Utilities Guide.*

Examples

- This user-defined function includes statements to declare database field names in both the current and Employee work areas:

```
FUNCTION DisplayRecord
   FIELD CustNo, OrderNo, Salesman
   FIELD EmpName, EmpCode IN Employee
   USE Employee NEW
   USE Orders NEW
   //
   ? CustNo              // Refers to Orders->CustNo
   ? EmpName             // Refers to Employee->EmpName
   //
   CLOSE Orders
   CLOSE Employee
   RETURN NIL
```

See Also

FUNCTION, LOCAL, MEMVAR, PROCEDURE, STATIC

FIELDBLOCK() function

Return a SET-GET code block for a given field

Syntax

```
FIELDBLOCK(<cFieldName>) → bFieldBlock
```

Arguments

<cFieldName> is the name of the field to which the SET-GET block will refer.

Returns

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

Description

FIELDBLOCK() is a database function that builds a code block. When executed with an argument, the code block created by this function assigns the value of the argument to *<cFieldName>*. When executed without an argument, the code block retrieves the value of *<cFieldName>*.

Note that the specified field variable may not exist when the code block is created, but must exist before the code block is executed.

Notes

- ***Work area:*** The code block returned by FIELDBLOCK() sets or gets the value of the specified field in whatever work area is current when the block is run. For example, given work areas 1 and 2, both containing field FName:

  ```
  SELECT 1
  FName:= "Kate"
  SELECT 2
  FName := "Cindy"
  bFName := FIELDBLOCK("FName")
  SELECT 1
  ? EVAL(bFName)              // Result: "Kate"
  SELECT 2
  ? EVAL(bFName)              // Result: "Cindy"
  ```

  The function FIELDWBLOCK() provides a SET-GET block for a field in a specific work area.

Examples

- This example compares FIELDBLOCK() to a code block created using the macro operator. Note that using FIELDBLOCK() avoids the speed and size overhead of the macro operator:

```
// Set-Get block defined using macro operator
bSetGet := &( "{ |setVal| IF( setVal == NIL,;
                FName, FName := setVal ) }" )
// Set-Get block defined using FIELDBLOCK()

// bSetGet created here is the functional
// equivalent of bSetGet above
bSetGet := FIELDBLOCK("FName")
```

Files

Library is CLIPPER.LIB.

See Also

FIELDWBLOCK(), MEMVARBLOCK()

FIELDGET() function

Retrieve the value of a field using the ordinal position of the field in the database structure

Syntax

FIELDGET(<*nField*>) → *ValueField*

Arguments

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

Returns

FIELDGET() returns the value of the specified field. If <*nField*> does not correspond to the position of any field in the current database file, FIELDGET() returns NIL.

Description

FIELDGET() is a database function that retrieves the value of a field using its position within the database file structure rather than its field name. Within generic database service functions this allows, among other things the retrieval of field values without use of the macro operator.

Examples

- This example compares FIELDGET() to functionally equivalent code that uses the macro operator to retrieve the value of a field:

```
LOCAL nField := 1, FName, FVal
USE Customer NEW
//
// Using macro operator
FName := FIELD( nField )          // Get field name
FVal := &FName                    // Get field value
// Using FIELDGET()
FVal := FIELDGET( nField )        // Get field value
```

Files Library is CLIPPER.LIB.

See Also FIELDPUT()

FIELDNAME()/FIELD() function

Return a field name from the current database (.dbf) file

Syntax

```
FIELDNAME/FIELD(<nPosition>) → cFieldName
```

Arguments

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

Returns

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

Description

FIELDNAME() is a database function that returns a field name using an index to the position of the field name in the database structure. Use it in data-independent applications where the field name is unknown. If information for more than one field is required, use AFIELDS() to create an array of field information or COPY STRUCTURE EXTENDED to create a database of field information.

If you need additional database file structure information, use TYPE() and LEN(). To obtain the number of decimal places for a numeric field, use the following expression:

```
LEN(SUBSTR(STR(<idField>), RAT(".", ;
            STR(<idField>)) + 1))
```

By default, FIELDNAME() operates on the currently selected work area as shown in the example below.

Examples

- These examples illustrate FIELDNAME() used with several other functions:

```
USE Sales
? FIELDNAME(1)                 // Result: BRANCH
? FCOUNT()                     // Result: 5
? LEN(FIELDNAME(0))            // Result: 0
? LEN(FIELDNAME(40))           // Result: 0
```

- This example uses FIELDNAME() to list the name and type of each field in Customer.dbf:

```
USE Customer NEW
FOR nField := 1 TO FCOUNT()
    ? PADR(FIELDNAME(nField), 10),;
           VALTYPE(&(FIELDNAME(nField)))
NEXT
```

- This example accesses fields in unselected work areas using aliased expressions:

```
USE Sales NEW
USE Customer NEW
USE Invoices NEW
//
? Sales->(FIELDNAME(1))              // Result: SALENUM
? Customer->(FIELDNAME(1))           // Result: CUSTNUM
```

Files Library is CLIPPER.LIB.

See Also AFIELDS()*, COPY STRUCTURE EXTENDED, DBSTRUCT(), FCOUNT(), LASTREC(), LEN(), TYPE(), VALTYPE()

FIELDPOS() function

Return the position of a field in a work area

Syntax

```
FIELDPOS(<cFieldName>) → nFieldPos
```

Arguments

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

Returns

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

Description

FIELDPOS() is a database function that is the inverse of the FIELDNAME() function. FIELDPOS() is most often used with the FIELDPUT() and FIELDGET() functions.

FIELDPOS() return the names of fields in any unselected work area by referring to the function using an aliased expression. See the example below.

Examples

- This example demonstrates a typical specification of the FIELDPOS() function:

```
USE Customer NEW
? FIELDPOS("Name")                       // Result: 1
? FIELDGET(FIELDPOS("Name"))             // Result: Kate
```

- This example uses FIELDPOS() to return the position of a specified field in a unselected work area:

```
USE Customer NEW
USE Invoices NEW
? Customer->(FIELDPOS("Name"))           // Result: 1
? Customer->(FIELDGET(FIELDPOS("Name")))
                                         // Result: Kate
```

Files

Library is CLIPPER.LIB.

See Also

FIELDGET(), FIELDPUT()

FIELDPUT() function

Set the value of a field variable using the ordinal position of the field in the database structure

Syntax

```
FIELDPUT(<nField>, <expAssign>) → ValueAssigned
```

Arguments

<*nField*> is the ordinal position of the field in the current database file.

<*expAssign*> is the value to assign to the given field. The data type of this expression must match the data type of the designated field variable.

Returns

FIELDPUT() returns the value assigned to the designated field. If <*nField*> does not correspond to the position of any field in the current database file, FIELDPUT() returns NIL.

Description

FIELDPUT() is a database function that assigns <*expAssign*> to the field at ordinal position <*nField*> in the current work area. This function allows you to set the value of a field using its position within the database file structure rather than its field name. Within generic database service functions this allows, among other things, the setting of field values without use of the macro operator.

Examples

- This example compares FIELDPUT() to functionally equivalent code that uses the macro operator to set the value of a field:

```
// Using macro operator
FName := FIELD(nField)           // Get field name
FIELD->&FName := FVal            // Set field value
// Using FIELDPUT()
FIELDPUT(nField, FVal)           // Set field value
```

Files

Library is CLIPPER.LIB.

See Also

FIELDGET()

FIELDWBLOCK() function

Return a SET-GET code block for a field in a given work area

Syntax

```
FIELDWBLOCK(<cFieldName>, <nWorkArea>)
    → bFieldWBlock
```

Arguments

<cFieldName> is the name of the field specified as a character string.

<nWorkArea> is the work area number where the field resides specified as a numeric value.

Returns

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

Description

FIELDWBLOCK() is a database function that builds a code block. When evaluated with the EVAL() function, the code block first selects the designated *<nWorkArea>*. If an argument was passed, the code block then assigns the value of the argument to *<cFieldName>*. If no argument was passed, the code block retrieves the value of *<cFieldName>*. The original work area is then reselected before the code block returns control.

Note that the specified field variable may not exist when the code block is created but must exist before the code block is executed.

Notes

- FIELDWBLOCK() is similar to FIELDBLOCK(), except that FIELDBLOCK() incorporates a fixed work area into the SET-GET block.

Examples

- This example compares FIELDWBLOCK() to a code block created using the macro operator. Note that using FIELDWBLOCK() avoids the speed and size overhead of the macro operator:

```
// Set-Get block for work area 1 defined with
// macro operator
bSetGet := &( "{ |setVal| IF( setVal == NIL, ;
    1->FName, 1->FName := setVal ) }" )
// Set-Get block defined using FIELDWBLOCK()

// bSetGet created here is the functional
// equivalent of bSetGet above
bSetGet := FIELDWBLOCK("FName", 1)
```

Files

Library is CLIPPER.LIB.

See Also

FIELDBLOCK(), MEMVARBLOCK()

FILE() function

Determine if files exist in the CA-Clipper default directory or path

Syntax

```
FILE(<cFilespec>) → lExists
```

Arguments

<cFilespec> is in the current CA-Clipper default directory and path. It is a standard file specification that can include the wildcard characters * and ? as well as a drive and path reference. Explicit references to a file must also include an extension.

Returns

FILE() returns true (.T.) if there is a match for any file matching the *<cFilespec>* pattern; otherwise, it returns false (.F.).

Description

FILE() is an environment function that determines whether any file matching a file specification pattern is found. FILE() searches the specified directory if a path is explicitly specified.

If a path is not specified, FILE() searches the current CA-Clipper default directory and then the CA-Clipper path. In no case is the DOS path searched. Note also that FILE() does not recognize hidden or system files in its search.

Examples

- In this example FILE() attempts to find Sales.dbf in other than the current CA-Clipper default:

```
? FILE("Sales.dbf")                // Result: .F.
? FILE("\APPS\DBF\Sales.dbf")      // Result: .T.
//
SET PATH TO \APPS\DBF
? FILE("Sales.dbf")                // Result: .T.
//
SET PATH TO
SET DEFAULT TO \APPS\DBF\
? FILE("Sales.dbf")                // Result: .T.
? FILE("*.dbf")                    // Result: .T.
```

Files

Library is CLIPPER.LIB.

See Also

SET DEFAULT, SET PATH

FIND* command

Search an index for a specified key value

Syntax

```
FIND <xcSearchString>
```

Arguments

<xcSearchString> is part or all of the index key of a record to search for, and can be specified either as a literal string or as a character expression enclosed in parentheses. If an expression is specified instead of a literal string, FIND operates the same as SEEK.

Description

FIND is a database command that searches an index for the first key matching the specified character string and positions the record pointer to the corresponding record.

If SOFTSEEK is OFF and FIND does not find a record, the record pointer is positioned to LASTREC() + 1, EOF() returns true (.T.), and FOUND() returns false (.F.).

If SOFTSEEK is ON, the record pointer is positioned to the record with the first key value greater than the search argument and FOUND() returns false (.F.). In this case, EOF() returns true (.T.) only if there are no keys in the index greater than the search argument.

FIND is a compatibility command and therefore not recommended. Its usage is superseded entirely by the SEEK command.

Examples

- These examples show simple FIND results:

```
USE Sales INDEX Branch NEW
FIND ("500")
? FOUND(), EOF(), RECNO()          // Result: .F. .T. 85
FIND "200"
? FOUND(), EOF(), RECNO()          // Result: .T. .F. 5
FIND "100"
? FOUND(), EOF(), RECNO()          // Result: .T. .F. 1
```

Files

Library is CLIPPER.LIB.

See Also

EOF(), FOUND(), RECNO(), SEEK, SET INDEX, SET ORDER
SET SOFTSEEK

FKLABEL()* function

Return function key name

Syntax

```
FKLABEL(<nFunctionKey>) → cKeyLabel
```

Returns

FKLABEL() returns a character string representing the name of the function key specified by the numeric argument, *<nFunctionKey>*. If this argument is less than one or greater than 40, the function returns a null ("") string.

Description

FKLABEL() is a compatibility function used to replicate the FKLABEL() function in dBASE III PLUS. As a general principle, the use of this function is not recommended and not needed in CA-Clipper. The function keys are labeled Fn, where n ranges from one to 40 and corresponds directly to the FKLABEL() argument.

Files

Library is EXTEND.LIB, source file is SOURCE\SAMPLE\FKLABEL.PRG.

See Also

SET FUNCTION

FKMAX()* function

Return number of function keys as a constant

Syntax

```
FKMAX() → nFunctionKeys
```

Description

FKMAX() is a compatibility function used to replicate the FKMAX() function in dBASE III PLUS. As a general principle, the use of this function is not recommended and not needed in CA-Clipper. It simply returns a constant value of 40.

Files

Library is EXTEND.LIB, source file is SOURCE\SAMPLE\FKMAX.PRG.

See Also

#define, SET FUNCTION

FLOCK() function

Lock an open and shared database file

Syntax

```
FLOCK() → lSuccess
```

Returns

FLOCK() returns true (.T.) if an attempt to lock a database file in USE in the current work area succeeds; otherwise, it returns false (.F.). For more information on file locking, refer to the "Network Programming" chapter in the *Programming and Utilities Guide*.

Description

FLOCK() is a database function used in network environments to lock an open and shared database file, preventing other users from updating the file until the lock is released. Records in the locked file are accessible for read-only operations.

FLOCK() is related to USE...EXCLUSIVE and RLOCK(). USE...EXCLUSIVE opens a database file so that no other user can open the same file at the same time and is the most restrictive locking mechanism in CA-Clipper. RLOCK() is the least restrictive and attempts to place an update lock on a shared record, precluding other users from updating the current record. FLOCK() falls in the middle.

FLOCK() is used for operations that access the entire database file. Typically, these are commands that update the file with a scope or a condition such as DELETE or REPLACE ALL. The following is a list of such commands:

Commands That Require an FLOCK()

Command	Mode
APPEND FROM	FLOCK() or USE...EXCLUSIVE
DELETE (multiple records)	FLOCK() or USE...EXCLUSIVE
RECALL (multiple records)	FLOCK() or USE...EXCLUSIVE
REPLACE (multiple records)	FLOCK() or USE...EXCLUSIVE
UPDATE ON	FLOCK() or USE...EXCLUSIVE

For each invocation of FLOCK(), there is one attempt to lock the database file, and the result is returned as a logical value. A file lock fails if another user currently has a file or record lock for the same database file or EXCLUSIVE USE of the database file. If FLOCK() is successful, the file lock remains in place until you UNLOCK, CLOSE the DATABASE, or RLOCK().

By default, FLOCK() operates on the currently selected work area as shown in the example below.

Notes

- ***SET RELATION:*** CA-Clipper does not automatically lock all work areas in the relation chain when you lock the current work area, and an UNLOCK has no effect on related work areas.

Examples

- This example uses FLOCK() for a batch update of prices in Inventory.dbf:

```
USE Inventory NEW
IF FLOCK()
   REPLACE ALL Inventory->Price WITH ;
         Inventory->Price * 1.1
ELSE
   ? "File not available"
ENDIF
```

- This example uses an aliased expression to attempt a file lock in an unselected work area:

```
USE Sales NEW
USE Customer NEW
//
IF !Sales->(FLOCK())
   ? "Sales is in use by another"
ENDIF
```

Files

Library is CLIPPER.LIB.

See Also

RLOCK(), SET EXCLUSIVE*, UNLOCK, USE

FOPEN() function

Open a binary file

Syntax

```
FOPEN(<cFile>, [<nMode>]) → nHandle
```

Arguments

<*cFile*> is the name of the file to open, including the path if there is one.

<*nMode*> is the requested DOS open mode indicating how the opened file is to be accessed. The open mode is composed of elements from the two types of modes described in the tables below. If just the Access Mode is used, the file is opened non-sharable. The default open mode is zero, which indicates non-sharable and read-only.

FOPEN() Access Modes

Mode	Fileio.ch	Operation
0	FO_READ	Open for reading (default)
1	FO_WRITE	Open for writing
2	FO_READWRITE	Open for reading or writing

The Sharing Modes determine how other processes may access the file.

FOPEN() Sharing Modes

Mode	Fileio.ch	Operation
0	FO_COMPAT	Compatibility mode (default)
16	FO_EXCLUSIVE	Exclusive use
32	FO_DENYWRITE	Prevent others from writing
48	FO_DENYREAD	Prevent others from reading
64	FO_DENYNONE	Allow others to read or write
64	FO_SHARED	Same as FO_DENYNONE

The Access Modes in combination (+) with the Sharing modes determine the accessibility of the file in a network environment.

Returns

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

Description

FOPEN() is a low-level file function that opens an existing binary file for reading and writing, depending on the <*nMode*> argument. Whenever there is an open error, use FERROR() to return the DOS error number. For example, if the file does not exist, FOPEN() returns -1 and FERROR() returns 2 to indicate that the file was not found. See FERROR() for a complete list of error numbers.

If the specified file is opened successfully, the value returned is the DOS handle for the file. This value is similar to an alias in the database system and is required to identify the open file to other file functions. It is, therefore, important to assign the return value to a variable for later use as in the example below.

Warning! *This function allows low-level access to DOS files and devices. It should be used with extreme care and requires a thorough knowledge of the operating system.*

Notes

- ***Accessing files in other directories:*** FOPEN() does not obey either SET DEFAULT or SET PATH. Instead, it searches the current DOS directory and path setting unless a path is explicitly stated as part of the <*cFile*> argument.

Examples

- This example uses FOPEN() to open a file with sharable read/write status and displays an error message if the open fails:

```
#include "Fileio.ch"
//
nHandle := FOPEN("Temp.txt", FO_READWRITE + FO_SHARED)
IF FERROR() != 0
   ? "Cannot open file, DOS error ", FERROR()
   BREAK
ENDIF
```

Files Library is CLIPPER.LIB, header file is Fileio.ch.

See Also FCLOSE(), FCREATE(), FERROR()

FOR statement

Execute a block of statements a specified number of times

Syntax

```
FOR <idCounter> := <nStart> TO <nEnd>
    [STEP <nIncrement>]
    <statements>...
    [EXIT]
    <statements>...
    [LOOP]
NEXT
```

Arguments

<idCounter> is the name of the loop control or counter variable. If the specified *<idCounter>* is not visible or does not exist, a private variable is created.

<nStart> is the initial value assigned to *<idCounter>*. If *<nIncrement>* is negative, *<nStart>* must be less than *<nEnd>*.

TO <nEnd> defines the final value of *<idCounter>*. If *<nIncrement>* is negative, *<nStart>* must be greater than *<nEnd>*; otherwise, *<nStart>* must be less than *<nEnd>*.

STEP <nIncrement> defines the amount *<idCounter>* is changed for each iteration of the loop. *<nIncrement>* can be either positive or negative. If the STEP clause is not specified, *<idCounter>* is incremented by one for each iteration of the loop.

EXIT unconditionally branches control from within a FOR...NEXT construct to the statement immediately following the nearest NEXT statement.

LOOP branches control to the most recently executed FOR or DO WHILE statement.

Description

FOR...NEXT is a control structure that executes a block of statements a specified number of times. The control structure loops from the initial value of *<idCounter>* to the boundary specified by *<nEnd>*, moving through the range of values of the control variable for an increment specified by *<nIncrement>*. All expressions in the FOR statement are reevaluated for each iteration of the loop. The *<nStart>* and *<nEnd>* expressions, therefore, can be changed as the control structure operates.

A FOR loop operates until *<idCounter>* is greater than *<nEnd>* or an EXIT statement is encountered. Control then branches to the statement following the corresponding NEXT statement. If a LOOP statement is encountered, control branches back to the current FOR statement.

If *<nIncrement>* is a negative value, *<idCounter>* is decremented rather than incremented. The FOR loop, however, continues until *<idCounter>* is less than *<nEnd>*. This means that *<nEnd>* must be less than *<nStart>* when the FOR loop begins.

FOR loops are useful for traversing arrays where *<idCounter>* is used as the array subscript. See the example below.

FOR...NEXT constructs may be nested within any other control structures to any depth. The only requirement is that each control structure is properly nested.

Examples

- This example traverses an array in ascending order:

```
nLenArray := LEN(aArray)
FOR i := 1 TO nLenArray
   <statements>...
NEXT
```

- To traverse an array in descending order:

```
nLenArray := LEN(aArray)
FOR i := nLenArray TO 1 STEP -1
   <statements>...
NEXT
```

See Also

AEVAL(), BEGIN SEQUENCE, DO CASE, DO WHILE, IF

FOUND() function

Determine if the previous search operation succeeded

Syntax

```
FOUND() → lSuccess
```

Returns

FOUND() returns true (.T.) if the last search command was successful; otherwise, it returns false (.F.).

Description

FOUND() is a database function that determines whether a search operation (i.e., FIND, LOCATE, CONTINUE, SEEK, or SET RELATION) succeeded. When any of these commands are executed, FOUND() is set to true (.T.) if there is a match; otherwise, it is set to false (.F.).

If the search command is LOCATE or CONTINUE, a match is the next record meeting the scope and condition. If the search command is FIND, SEEK or SET RELATION, a match is the first key in the controlling index that equals the search argument. If the key value equals the search argument, FOUND() is true (.T.); otherwise, it is false (.F.).

The value of FOUND() is retained until another record movement command is executed. Unless the command is another search command, FOUND() is automatically set to false (.F.).

Each work area has a FOUND() value. This means that if one work area has a RELATION set to a child work area, querying FOUND() in the child returns true (.T.) if there is a match.

By default, FOUND() operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression (see example below).

FOUND() will return false (.F.) if there is no database open in the current work area.

Examples

- This example illustrates the behavior of FOUND() after a record movement command:

```
USE Sales INDEX Sales
? INDEXKEY(0)              // Result: SALESMAN
SEEK "1000"
? FOUND()                  // Result: .F.
SEEK "100"
? FOUND()                  // Result: .T.
SKIP
? FOUND()                  // Result: .F.
```

- This example tests a FOUND() value in an unselected work area using an aliased expression:

```
USE Sales INDEX Sales NEW
USE Customer INDEX Customer NEW
SET RELATION TO CustNum INTO Sales
//
SEEK "Smith"
? FOUND(), Sales->(FOUND())
```

- This code fragment processes all Customer records with the key value "Smith" using FOUND() to determine when the key value changes:

```
USE Customer INDEX Customer NEW
SEEK "Smith"
DO WHILE FOUND()
   .
   . <statements>
   .
   SKIP
   LOCATE REST WHILE Name == "Smith"
ENDDO
```

Files Library is CLIPPER.LIB.

See Also CONTINUE, EOF(), LOCATE, SEEK, SET RELATION, SET SOFTSEEK

FREAD() function

Read characters from a binary file into a buffer variable

Syntax

```
FREAD(<nHandle>, @<cBufferVar>, <nBytes>) → nBytes
```

Arguments

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

<cBufferVar> is the name of an existing and initialized character variable used to store data read from the specified file. The length of this variable must be greater than or equal to *<nBytes>*. *<cBufferVar>* must be passed by reference and, therefore, must be prefaced by the pass-by-reference operator (@).

<nBytes> is the number of bytes to read into the buffer.

Returns

FREAD() returns the number of bytes successfully read as an integer numeric value. A return value less than *<nBytes>* or zero indicates end of file or some other read error.

Description

FREAD() is a low-level file function that reads characters from a binary file into an existing character variable. It reads from the file starting at the current DOS file pointer position, advancing the file pointer by the number of bytes read. All characters are read including control, null, and high-order (above CHR(127)) characters.

FREAD() is similar in some respects to both FREADSTR() and FSEEK(). FREADSTR() reads a specified number of bytes from a file up to the next null (CHR(0)) character. FSEEK() moves the file pointer without reading.

If there is an error during the file read, FERROR() returns the DOS error number. See FERROR() for the list of error numbers.

Warning! *This function allows low-level access to DOS files and devices. It should be used with extreme care and requires a thorough knowledge of the operating system.*

Examples

- This example uses FREAD() after successfully opening a file to read 128 bytes into a buffer area:

```
#define F_BLOCK   128
//
cBuffer := SPACE(F_BLOCK)
nHandle := FOPEN("Temp.txt")
//
IF FERROR() != 0
   ? "File open error:", FERROR()
ELSE
   IF FREAD(nHandle, @cBuffer, F_BLOCK) <> F_BLOCK
      ? "Error reading Temp.txt"
   ENDIF
   FCLOSE(nHandle)
ENDIF
```

Files

Library is CLIPPER.LIB.

See Also

BIN2I(), BIN2L(), BIN2W(), FCLOSE(), FCREATE(), FERROR(), FOPEN(), FREADSTR(), FSEEK(), FWRITE()

FREADSTR() function

Read characters from a binary file

Syntax

```
FREADSTR(<nHandle>, <nBytes>) → cString
```

Arguments

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

<nBytes> is the number of bytes to read, beginning at the current DOS file pointer position.

Returns

FREADSTR() returns a character string up to 65,535 (64K) bytes. A null return value ("") indicates an error or end of file.

Description

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

Warning! *This function allows low-level access to DOS files and devices. It should be used with extreme care and requires a thorough knowledge of the operating system.*

Examples

- This example displays the ASCII values of the first 16 bytes of a text file:

```
#include "Fileio.ch"
//
nHandle := FOPEN("New.txt", FC_NORMAL)
IF FERROR() != 0
   ? "File open error:", FERROR()
ELSE
   cString := FREADSTR(nHandle, 16)
   ? cString
   FCLOSE(nHandle)
ENDIF
```

Files Library is CLIPPER.LIB.

See Also BIN2I(), BIN2L(), BIN2W(), FERROR(), FOPEN(), FREAD(), FSEEK(), FWRITE()

FRENAME() function

Change the name of a file

Syntax

```
FRENAME(<cOldFile>, <cNewFile>) → nSuccess
```

Arguments

<cOldFile> is the name of the file to be renamed, including the file extension. A drive letter and/or path name may also be included as part of the file name.

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

Returns

FRENAME() returns -1 if the operation fails and zero if it succeeds. In the case of a failure, FERROR() can be used to determine the nature of the error.

Description

FRENAME() is a file function that changes the name of a specified file to a new name and is identical to the RENAME command.

When FRENAME() is called, *<cOldFile>* is renamed only if it is located in the current DOS directory or in the specified path. FRENAME() does not use SET DEFAULT or SET PATH to locate *<cOldFile>*.

If the source directory is different from the target directory, the file moves to the target directory. In the instance that either *<cNewFile>* exists or is currently open, FRENAME() fails and returns -1, indicating that it did not perform its designated action. The nature of the error can be determined with FERROR().

Warning! *Files must be CLOSEd before renaming. Attempting to rename an open file will produce unpredictable results. When a database file is renamed, the associated memo (.dbt) file must also be renamed. Failure to do so may compromise the integrity of your databases.*

Examples

- This example demonstrates a file rename:

```
IF FRENAME("OldFile.txt", "NewFile.txt") == -1
   ? "File error:", FERROR()
ENDIF
```

Files Library is CLIPPER.LIB.

See Also CLOSE, ERASE, FERASE(), FERROR(), FILE(), RENAME

FSEEK() function

Set a binary file pointer to a new position

Syntax

```
FSEEK(<nHandle>, <nOffset>, [<nOrigin>]) → nPosition
```

Arguments

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

<nOffset> is the number of bytes to move the file pointer from the position defined by *<nOrigin>*. It can be a positive or negative number. A positive number moves the pointer forward, and a negative number moves the pointer backward in the file.

<nOrigin> defines the starting location of the file pointer before FSEEK() is executed. The default value is zero, representing the beginning of file. If *<nOrigin>* is the end of file, *<nOffset>* must be zero or negative.

Methods of Moving the File Pointer

Origin	Fileio.ch	Description
0	FS_SET	Seek from beginning of file
1	FS_RELATIVE	Seek from the current pointer position
2	FS_END	Seek from end of file

Returns

FSEEK() returns the new position of the file pointer relative to the beginning of file (position 0) as an integer numeric value. This value is without regard to the original position of the file pointer.

Description

FSEEK() is a low-level file function that moves the file pointer forward or backward in an open binary file without actually reading the contents of the specified file. The beginning position and offset are specified as function arguments, and the new file position is returned. Regardless of the function arguments specified, the file pointer cannot be moved beyond the beginning or end of file boundaries.

Warning! *This function allows low-level access to DOS files and devices. It should be used with extreme care and requires a thorough knowledge of the operating system.*

Examples

- This example uses FSEEK() to determine the length of a file by seeking from the end of file. Then, the file pointer is reset to the beginning of file:

```
#include "Fileio.ch"
//
// Open the file read-only
IF (nHandle := FOPEN("Temp.txt")) >= 0
   //
   // Get length of the file
   nLength := FSEEK(nHandle, 0, FS_END)
   //
   // Reset file position to beginning of file
   FSEEK(nHandle, 0)
   FCLOSE(nHandle)
ELSE
   ? "File open error:", FERROR()
ENDIF
```

- This pseudofunction positions the file pointer at the last byte in a binary file:

```
#define FileBottom(nHandle);
      (FSEEK(nHandle, 0, FS_END))
```

- This pseudofunction positions the file pointer at the first byte in a binary file:

```
#define FileTop(nHandle);
      (FSEEK(nHandle, 0))
```

- This pseudofunction reports the current position of the file pointer in a specified binary file:

```
#define FilePos(nHandle);
      (FSEEK(nHandle, 0, FS_RELATIVE))
```

Files Library is CLIPPER.LIB, header file is Fileio.ch.

See Also FCLOSE(), FCREATE(), FERROR(), FOPEN(), FREAD(), FREADSTR(), FWRITE()

FUNCTION statement

Declare a user-defined function name and formal parameters

Syntax

```
[STATIC] FUNCTION <idFunction>[(<idParam list>)]
   [LOCAL <identifier> [[:= <initializer>], ... ]]
   [STATIC <identifier> [[:= <initializer>], ... ]]
   [FIELD <identifier list> [IN <idAlias>]]
   [MEMVAR <identifier list>]
   .
   . <executable statements>
   .
   RETURN <exp>
```

Arguments

<idFunction> is the name of the user-defined function to be declared. User-defined function names can be any length, but only the first 10 characters are significant. Names can contain any combination of characters, numbers, or underscores, but must begin with a character or an underscore. Leading underscores are not recommended since they are reserved for internal functions.

<idParam list> is the declaration of one or more parameter variables. Variables specified in this list are declared local.

STATIC FUNCTION declares a user-defined function that can be invoked only by procedures and user-defined functions declared in the same program (.prg) file.

LOCAL declares and optionally initializes a list of variables or arrays whose visibility and lifetime is the current function.

STATIC declares and optionally initializes a list of variables or arrays whose visibility is the current user-defined function and lifetime is the duration of the program.

FIELD declares a list of identifiers to use as field names whenever encountered. If the IN clause is specified, referring to the declared name includes an implicit reference to the specified alias.

MEMVAR declares a list of identifiers to use as private or public memory variables or arrays whenever encountered.

<identifier> and ***<identifier list>*** are labels to be used as variable or array names.

<initializer> is a value to which an array or variable is originally set in an inline expression.

RETURN <exp> passes control back to the calling procedure or user-defined function, returning the result of *<exp>* as the value of the function. Each function must have at least one RETURN statement that returns a value. RETURN statements can occur anywhere in the body of a function.

Description

The FUNCTION statement declares a user-defined function and an optional list of local variables to receive parameters often referred to as formal parameters. A user-defined function is a subprogram comprised of a set of declarations and statements executed whenever you refer to *<idFunction>* followed by an open and closed parentheses pair. A function definition begins with a FUNCTION statement which is the FUNCTION declaration and ends with the next FUNCTION statement, PROCEDURE statement, or end of file.

Functions encapsulate a computational block of code and then later create expressions using the value returned. Functions and procedures increase both readability and modularity, isolate change, and help manage complexity.

A function in CA-Clipper is the same as a procedure, except that it must return a value. The returned value can be any data type including an array, a code block, or NIL. Each function must begin with a FUNCTION statement and contain at least one RETURN statement with an argument. Function declarations cannot be nested within other function definitions. A user-defined function can be used wherever standard functions are supported, including expressions.

The visibility of function names falls into two classes. Functions that are visible anywhere in a program are referred to as public functions and declared with a FUNCTION statement. Functions that are visible only within the current program (.prg) file are referred to as static functions and declared with a STATIC FUNCTION statement. Static functions have filewide scope.

Static functions limit visibility of a function name, thereby restricting access to the function. Because of this, subsystems defined within a single program (.prg) file can provide an access protocol with a series of public functions and conceal the implementation details of the subsystem within static functions and procedures. Since the static function references are resolved at compile time, they preempt references to public functions which are resolved at link time. This ensures that within a program file, a reference to a static function executes that function if there is a name conflict with a public function.

For more information on user-defined functions, variable declarations, and parameter passing, refer to the "Basic Concepts" chapter in the *Programming and Utilities Guide.*

Notes

- ***Calling a user-defined function:*** Use the same notation to call a user-defined function as when calling a standard CA-Clipper function:

  ```
  <idFunction>([<argument list>])
  ```

  You can call a user-defined function within an expression or on a line by itself. If called on a line by itself, the return value is ignored.

  You can also call a user-defined function as an aliased expression by prefacing it with an alias and enclosing it in parentheses:

  ```
  <idAlias>->(<idFunction>(<argument list>))
  ```

  When you call a user-defined function as an aliased expression, the work area associated with *<idAlias>* is selected, the expression is executed, and the original work area is then reselected. You can specify an aliased expression on a line by itself, as you would any other expression.

  A user-defined function in CA-Clipper may call itself recursively. This means you can make a reference to a user-defined function in statements or expressions of the same user-defined function definition.

- ***Parameters:*** User-defined functions, like procedures, can receive parameters passed from a calling procedure, user-defined function, or DOS command line. A parameter is a place holder for a value or reference. In CA-Clipper, there are two ways to express parameters: you can declare a list of local variable names as a part of the FUNCTION declaration (referred to as *formal* parameters), or you can specify a list of private variables in a separate PARAMETERS statement. Note that you cannot mix a declaration of formal parameters with a PARAMETERS statement. Attempting this will result in a fatal compiler error.

  Functions receive parameters in the order passed. In CA-Clipper, the number of parameters does not have to match the number of arguments passed. You can skip arguments or omit them from the end of the argument list. A parameter not receiving a value or reference is initialized to NIL. You can skip a parameter by passing NIL. If arguments are specified, PCOUNT() returns the position of the last argument passed. (If more arguments are passed than are parameters, they are ignored.)

  Parameters specified in a user-defined function can receive arguments passed by value or reference. The default method for expressions and variables is by value. This includes variables that contain references to arrays and objects. All variables except field variables, when prefaced with the pass-by-reference operator (@), are passed by reference. Field variables cannot be passed by reference and are always passed by value.

Examples

- This example demonstrates a user-defined function that formats numeric values as currency:

```
? Currency( 1000 )              // Result: $1,000.00

FUNCTION Currency( nNumber )
   LOCAL cNumber
   IF nNumber < 0
      cNumber := TRANSFORM(-1 * nNumber, ;
            "999,999,999,999.99")
      cNumber := PADL("($" + LTRIM(cNumber) + ")", ;
            LEN(cNumber))
   ELSE
      cNumber := TRANSFORM(nNumber, ;
            "999,999,999,999.99")
      cNumber := PADL("$" + LTRIM(cNumber), ;
            LEN(cNumber))
   ENDIF
   RETURN cNumber
```

- This example demonstrates a user-defined function that takes a character string formatted as a comma-separated list and returns an array with one element per item:

```
aList := ListAsArray("One, Two")
// Result: {"One", "Two"}

FUNCTION ListAsArray( cList )
   LOCAL nPos
   // Define an empty array
   LOCAL aList := {}
   //
   DO WHILE (nPos := AT(",", cList)) != 0
      // Add a new element
      AADD(aList, SUBSTR(cList, 1, nPos - 1))
      cList := SUBSTR(cList, nPos + 1)
   ENDDO
   AADD(aList, cList)
   //
   // Return the array
   RETURN aList
```

- This example checks for a skipped argument by comparing the parameter to NIL:

```
FUNCTION MyFunc( param1, param2, param3 )
    IF param2 == NIL
       param2 := "default value"
    ENDIF
    .
    . <statements>
    .
    RETURN NIL
```

- This example uses the user-defined function, Currency() (defined above), as an aliased expression:

```
USE Invoices NEW
USE Customer NEW
? Invoices->(Currency(Amount))
```

See Also LOCAL, PARAMETERS, PCOUNT(), PRIVATE, PROCEDURE, PUBLIC, RETURN, STATIC

FWRITE() function

Write to an open binary file

Syntax

```
FWRITE(<nHandle>, <cBuffer>, [<nBytes>])
   → nBytesWritten
```

Arguments

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

<cBuffer> is the character string to write to the specified file.

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

Returns

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

Description

FWRITE() is a low-level file function that writes data to an open binary file from a character string buffer. You can either write all or a portion of the buffer contents. Writing begins at the current file position, and the function returns the actual number of bytes written.

If FWRITE() results in an error condition, FERROR() can be used to determine the specific error.

Warning! *This function allows low-level access to DOS files and devices. It should be used with extreme care and requires a thorough knowledge of the operating system*

Examples

- This example copies the contents of one file to another:

```
#include "Fileio.ch"
#define F_BLOCK   512
//
cBuffer := SPACE(F_BLOCK)
nInfile := FOPEN("Temp.txt", FO_READ)
nOutfile := FCREATE("Newfile.txt", FC_NORMAL)
lDone := .F.
//
DO WHILE !lDone
    nBytesRead := FREAD(nInfile, @cBuffer, F_BLOCK)
    IF FWRITE(nOutfile, cBuffer, nBytesRead) < ;
                nBytesRead
       ? "Write fault: ", FERROR()
       lDone := .T.
    ELSE
       lDone := (nBytesRead == 0)
    ENDIF
ENDDO
//
FCLOSE(nInfile)
FCLOSE(nOutfile)
```

Files Library is CLIPPER.LIB.

See Also FCLOSE(), FCREATE(), FERROR(), FOPEN(), I2BIN(), L2BIN()

GBMPDISP() function

Display a bitmap (.BMP) file on screen

Syntax

```
GBMPDISP(<aBmpArray> | <cFile>,<nX>,<nY>,[<nTransColor>]
    → NIL
```

Arguments

<aBmpArray> defines the pointer created with GBMPLOAD().

-OR-

<***cFile***> defines the .BMP file to be displayed.

<***nX***> is the position of the starting point on the X-axis (column).

<***nY***> is the position of the starting point on the Y-axis (line).

<***nTransColor***> is the color to be excluded.

If <*nColor*> is set, all colors in the image are displayed except <*nColor*>.

Returns

GBMPDISP() always returns NIL.

Description

GBMPDISP() displays a bitmap (.BMP) or icon (.ICO) file previously loaded into memory or display a BMP directly from disk. This function can be used only if you have set the screen to a graphic mode using SET VIDEOMODE. GBMPDISP() respects the constraints defined by GSETCLIP().

Warning! *The current color palette is used when you display a .BMP. Be aware that if you allow users to modify the color palette, the colors used in creating your .BMP may be changed because of the current palette components.*

Notes

It is possible to create visual effects on top of a .BMP, such as a button that appears to be depressed when the user clicks on it. This can be done by superimposing a rectangle in XOR mode:

```
#include "Llibg.ch"
LOCAL aMyButton
SET VIDEOMODE TO LLG_VIDEO_VGA_640_480_16
// Load the BMP
aMyButton := gBmpLoad ("MyButton.BMP")

   // Display the BMP, always in SET mode
gBmpDisp (aMyButton,100,200)

.
.  // Your code
.

IF mState()[LLM_STATE_LEFT] == LLM_BUTTON_DOWN

   // When the mouse button is pressed, we superimpose a BMP

   gRect( 100,;
            200,;
            100 + aMyButton[LLG_BMP_X],;
            200 + aMyButton[LLG_BMP_Y],;
            LLG_FILL,;
            8,;
            LLG_MODE_XOR)

ENDIF

   // As long as the button is down...
DO WHILE mState()[LLM_STATE_LEFT] == LLM_BUTTON_DOWN
.
.  // Your code
.
ENDDO

   // When the mouse button is released we reload
   // the BMP to restore the original look

   gRect(100,;
          200,;
          100 + aMyButton[LLG_BMP_X],;
          200 + aMyButton[LLG_BMP_Y],;
          LLG_FILL,;
          8,;
         LLG_MODE_XOR)
```

Examples

- This example displays buttons previously stored in aIcons using GBMPLOAD():

```
gBmpDisp(aIcons,100,200)
```

- This example loads and displays a background without using a variable. In this case, the area occupied in VMM when the array is loaded, is freed by CA-Clipper once the display ends.

```
gBmpDisp(gBmpLoad("MyDesk.BMP") , 0 , 0 )
```

See Also: GBMPLOAD(), GSETPAL(), GWRITEAT(), SET VIDEOMODE

GBMPLOAD() function

Load a bitmap (.BMP) or icon (.ICO) file into memory

Syntax

```
GBMPLOAD(<cFileName>) → aBmpArray
```

Arguments

<cFileName> is a character value that specifies the name of .BMP or .ICO file to load into VMM memory.

Returns

GBMPLOAD() returns a reference to *<aBmpArray>* which is the pointer to the VMM region containing the .BMP or .ICO (not a black and white icon). The first two elements of *<aBmpArray>* contain the height and width of the .BMP or .ICO in pixels. Do not modify this array before passing it to GBMPDISP(). The following table shows the array structure:

aBmpArray Structure

Array Position	Return Value
LLG_BMP_X	Returns size on the X axis
LLG_BMP_Y	Returns size on the Y axis

The size and elements of this array should never be changed.

Description

GBMPLOAD() allows you to load one or more .BMP or .ICO files into memory without having to display them. This function is useful when you want to load a group of small .BMP files which are used often in an application (e.g., buttons). It avoids having to load the .BMP file each time you display it.

Warning! *It is not a good idea to keep backgrounds or images in memory unless you have a great deal of memory. This also includes any variable pointing to the variables which contain the .BMP.*

Notes

When a .BMP or .ICO is stored in a LOCAL variable, the memory occupied in the VMM is automatically freed when the function returns. You can free the memory at any time by assigning NIL to the variable. It is not necessary to be in graphic mode to use GBMPLOAD(), but it is necessary to be in graphic mode before calling GBMPDISP().

Examples

- This example stores all the buttons used by an application in an array:

```
STATIC aIcons:= {}
AADD(aIcons, GBMPLOAD('ARROW_U.BMP'))
AADD(aIcons, GBMPLOAD('ARROW_D.BMP'))
AADD(aIcons, GBMPLOAD('ARROW_L.BMP'))
AADD(aIcons, GBMPLOAD('ARROW_R.BMP'))
// Set video to graphic mode
SET VIDEOMODE TO LLG_VIDEO_VGA_640_480_16
// Display the ARROW_U.BMP
GBMPDISP(aIcons[1], 100, 200)
```

Files

Library is LLIBG.LIB, header file is Llibg.ch.

See Also

GBMPDISP(), GMODE(), GSETPAL(), SET VIDEOMODE, GWRITEAT()

GELLIPSE() function

Draw an ellipse or circle

Syntax

```
GELLIPSE(<nXc>, <nYc>, <nRadiusX>, <nRadiusY>,
   [<nDegStart>], [<nDegEnd>], [<nStyle>], [<nColor>],
   [<nMode>], [<nOutLineColor>], [<nHeight3D>]) → NIL
```

Arguments

*<**nXc**>*, *<**nYc**>*, *<**nRadiusX**>*, and *<**nRadiusY**>* define the ellipse center point coordinates in pixels.

Note: If *<nRadiusX>* and *<nRadiusY>* have the same value, the result will be a circle.

*<**nDegStart**>* is the value of the starting angle in degrees. The default value is 0.

*<**nDegEnd**>* is the value of the final angle in degrees. The default value is 360.

Note: Changing the values of *<nDegStart>* and *<nDegEnd>* allows you to draw arcs of a circle or ellipse and sections of a "pie chart."

*<**nStyle**>* defines the style of the ellipse using one of the constants listed in the table below:

Ellipse Style Constants

Constant	Description
LLG_FILL	The ellipse is displayed first and then filled with the color and mode specified below
LLG_FRAME	Display of the ellipse is restricted to its contour in the color and mode specified below

*<**nColor**>* is a numeric value representing the display color. If this parameter is missing, the last color specified in a call to a CA-Clipper function is used. The value range is limited to the number of colors available in the selected video mode. In 16-color modes, valid values are between 0 and 15. In 256-color modes, valid values are between 0 and 255.

<*nMode*> is a numeric value that represents the display mode. The following are valid <*nMode*> values:

Display Mode Constants

Constant	Description
LLG_MODE_SET	Display in SET mode (ignores any pixels present under the line displayed). This is the most common display mode.
LLG_MODE_AND	Display in AND mode (executes an AND on pixels present under the line at display time and on the display color).
LLG_MODE_OR	Display in OR mode (executes an OR on pixels present under the line at display time and on the display color).
LLG_MODE_XOR	Display in XOR mode (executes an XOR on pixels present under the line at display time and on the display color). See note.

Note: This method allows you to move objects around on the screen without damaging the background. To retrieve the initial background, just repeat the call for display in XOR mode. If the display mode parameter is missing, the last mode specified in a call to a CA-Clipper function is used.

<***nOutLineColor***> is a numeric value representing the outline color. If this parameter is missing, the last color specified in a call to a CA-Clipper function is used. The value range is limited to the number of colors available in the selected video mode. In 16-color modes, valid values are between 0 and 15. In 256-color modes, valid values are between 0 and 255.

<***nHeight3D***> is a numeric value representing the height of the 3-D effect in pixels.

Returns

GELLIPSE() always returns NIL.

Description

GELLIPSE() draws an ellipse or circle. This function can be used only if you have set the screen to a graphic mode using SET VIDEOMODE. This function respects the constraints defined by GSETCLIP().

Examples

- This example displays an ellipse in a region limited by clipping:

```
// Switch to graphic mode
SET VIDEOMODE TO LLG_VIDEO_VGA_640_480_16
// Restrict the display region to one portion of the screen
GSETCLIP(100,100,300,300)
// Draw a quarter section of an ellipse
GELLIPSE(200, 200, 160, 230, 045,;

    135, LLG_FILL, 12, LLG_MODE_SET)
QUIT                          // End of application
```

Files

Library is LLIBG.LLB, header file is Llibg.ch.

See Also

GLINE(), GPOLYGON(), GRECT(), GWRITEAT(), GBMPDISP(), SET VIDEOMODE

Get class

Provides objects for interactive editing of database fields and variables

Class Function

```
GETNEW([<nRow>], [<nCol>], [<bBlock>], [<cVarName>],
   [<cPicture>], [<cColorSpec>]) → oGet
```

Description

A Get object is a general purpose mechanism for editing data. It is used in CA-Clipper to implement the @...GET and READ commands. Get objects provide a sophisticated architecture for formatting and editing data, including cursor navigation and data validation. Data validation is performed via user-supplied code blocks, and display formatting can be controlled using standard picture strings.

Normally, a Get object is associated with a particular variable (referred to as the GET variable). The Get object does not directly access this variable; instead, the variable is manipulated by evaluating a supplied code block. When a Get object is created using the standard @...GET command, a code block is automatically created which provides access to the variable named in the command. For a Get object created with the GETNEW() function, you must specify an appropriate code block.

A Get object evaluates the supplied code block to retrieve the value of the GET variable. The value is converted to a displayable form and copied into a buffer referred to as the editing buffer. You can display the editing buffer on the screen for editing. Programmable behaviors allow the navigation of the editing buffer and subsequent copying of its contents back to the GET variable.

Returns

GetNew() returns a new Get object with the row, col, block, picture, and colorSpec instance variables set from the supplied arguments.

Exported Instance Variables

`badDate`

Contains a logical value indicating that the editing buffer does not represent a valid date. Get:badDate contains true (.T.) when the Get object is a date type and the date represented by the contents of the editing buffer is invalid. Get:badDate contains false (.F.) when the date is valid or the GET is not editing a date value.

`block` (Assignable)

Contains a code block that associates the Get object with a variable. The code block takes an optional argument that should assign the value of the argument to the variable. If the argument is omitted, the code block should return the current value of the variable.

If the GET variable is an array element, Get:block always returns the base of the array. The subscript(s) in the expression are stored internally. Thus, in the case of GETs on array elements, you cannot assign or retrieve the value of the array element by executing Get:block. Setting and getting may be done on array element GET variables (and simple variables as well) by use of the varGet() and varPut() methods, defined below.

Note: Use of the varGet and varPut messages—instead of directly evaluating the variable block in the GET—is the preferred method of accessing the GET variable.

`buffer` (Assignable)

Contains a character value which is the editing buffer used by the Get object. Get:buffer is meaningful only when the Get object has input focus. At other times, it contains NIL and any attempts to assign a new value are ignored.

`cargo` (Assignable)

Contains a value of any data type unused by the Get system. Get:cargo is provided as a user-definable slot, allowing arbitrary information to be attached to a Get object and retrieved later.

`changed`

Contains a logical value indicating whether the Get:buffer has changed since the GET received input focus. Get:changed contains true (.T.) if the buffer has changed; otherwise, it contains false (.F.).

`clear` (Assignable)

Contains a logical value indicating whether the editing buffer should be cleared before any more values are entered. Get:clear is set true (.T.) by Get:setFocus() and Get:undo() when the G variable is a numeric type or when Get:picture contains the "@K" picture function. At all other times, it contains false (.F.).

`col` (Assignable)

Contains a numeric value defining the screen column where the Get is displayed.

`colorSpec` (Assignable)

Contains a character string defining the display attributes for the Get object. The string must contain two color specifiers. The first, called the unselected color, determines the color of the Get object when it does not have input focus. The second, called the selected color, determines the color of the GET when it has input focus.

If no colors are specified, Get:colorSpec is initialized using the current SETCOLOR() colors. The SETCOLOR() unselected and enhanced colors are used as the Get object's unselected and selected colors, respectively. See the SETCOLOR() entry in this chapter for more information on color specifiers.

`decPos`

Contains a numeric value indicating the decimal point position within the editing buffer. Get:decPos is meaningful only when the value being edited is numeric and the Get object has input focus. Otherwise, it contains NIL.

`exitState` (Assignable)

Contains a numeric value used in the CA-Clipper version of Getsys.prg to record the means by which a Get object was exited.

GET Exit States

Number	Getexit.ch	Meaning
0	GE_NOEXIT	No exit attempted, prepare for editing
1	GE_UP	Go to previous
2	GE_DOWN	Go to next
3	GE_TOP	Go to first
4	GE_BOTTOM	Go to last
5	GE_ENTER	edit normal end
6	GE_WRITE	Terminate READ state with save
7	GE_ESCAPE	Terminate READ state without save
8	GE_WHEN	WHEN clause unsatisfied

Getexit.ch contains manifest constants for the Get:exitState values.

`hasFocus`

Contains a logical value that indicates if the Get object has input focus. Get:hasFocus contains true (.T.) if the Get object has input focus; otherwise, it contains false (.F.).

`message` (Assignable)

Contains a character string that is displayed on the Get system's status bar line when the GET has input focus. Typically, it describes the anticipated contents of, or user response to the GET. Refer to the READ command for details pertaining to the Get system's status bar.

`minus` (Assignable)

Contains a logical value indicating that a minus sign (-) has been added to the editing buffer. Get:minus is set to true (.T.) only when the Get object is a numeric type, the current value of the editing buffer is zero and the last change to the editing buffer was the addition of the minus sign. It is cleared when any change is made to the buffer.

`name` (Assignable)

Contains a character string representing the name of the GET variable. This value is optional and can be assigned at your discretion. With Get objects created using the standard @...GET command, Get:name always contains the GET variable name.

The Get object itself ignores this variable. It is used by the standard READ command and READMODAL() function to implement the READVAR() function (see the READVAR() entry in this chapter).

`original`

Contains a value of any data type that is a copy of the value in the GET variable at the time the Get object acquired input focus. This value implements the Get:undo message. Get:original is meaningful only while the GET has input focus. At all other times, it contains NIL.

`picture` (Assignable)

Contains a character value defining the PICTURE string that controls formatting and editing for the Get object. See the @...GET entry in this chapter for more information on PICTURE strings.

`pos`

Contains a numeric value indicating the position of the cursor within the editing buffer. Get:pos is meaningful only when the Get object has input focus. At all other times, it contains NIL.

`postBlock` (Assignable)

Contains an optional code block that validates a newly entered value. If present, the Get:postBlock should contain an expression that evaluates to true (.T.) for a legal value and false (.F.) for an illegal value. For Get objects created with the standard @...GET...VALID command, Get:postBlock is assigned the expression specified in the VALID clause.

The Get object itself ignores this variable. It is used by the standard READ command to implement the VALID clause.

During postvalidation the Get:postBlock is passed a reference to the current Get object as an argument. Note that this differs from the CA-Clipper 5.0 behavior where the Get:postBlock was passed the updated value of the GET variable, and a logical value representing whether the Get object had been edited.

`preBlock` (Assignable)

Contains an optional code block that decides whether editing should be permitted. If present, the Get:preBlock should evaluate to true (.T.) if the cursor enters the editing buffer; otherwise, it should evaluate to false (.F.). For Get objects created with the standard @...GET...WHEN command, Get:preBlock is initialized with the expression specified in the WHEN clause.

The Get object itself ignores this variable. It is used by the standard READ command to implement the WHEN clause.

During prevalidation, the Get:preBlock is passed a reference to the current Get object as an argument. Note that this behavior differs from CA-Clipper 5.0 where no arguments were passed to Get:preBlock when it was run during prevalidation.

`reader`

Contains a code block to implement special READ behaviors for any GET. If Get:reader contains a code block, READMODAL() evaluates that block to READ the GET (the Get object is passed as an argument to the block). The block may in turn call any desired function to provide custom editing of the Get object. If Get:reader does not contain a code block, READMODAL() uses a default read procedure (GetReader()) for the Get object.

Note that Get:reader allows particular Get objects to have specialized READ behaviors without changing the standard READMODAL() function. This preserves compatibility for GETs which are to be handled in the customary way and also eliminates potential conflicts between different extensions to the GET/READ system.

`rejected`

Contains a logical value indicating whether the last character specified by a Get:insert or Get:overStrike message was placed in the editing buffer. Get:rejected contains true (.T.) if the last character was rejected; otherwise, it contains false (.F.). Note that any subsequent text entry message resets this variable.

`row` (Assignable)

Contains a numeric value defining the screen row where the GET displays.

`subscript` (Assignable)

Contains an array of numeric values representing the subscripts of a GET array element. Each element of Get:subscript represents a dimension of the GET array. For example:

```
@ 1,1 GET aTestA[4]        // Get:subscript contains {4}
@ 1,1 GET aTestB[3,6]      // Get:subscript contains {3,6}
```

If the GET does not involve an array, Get:subscript contains NIL.

`type`

Contains a single letter representing the data type of the GET variable. For more information on the values that represent data types, refer to the VALTYPE() entry in this chapter.

`typeOut`

Contains a logical value indicating whether the most recent message attempted to move the cursor out of the editing buffer, or if there are no editable positions in the buffer. Get:typeOut contains true (.T.) if the last message satisfied this condition. Note, Get:typeOut is reset by any message that moves the cursor.

Exported Methods

State Change Methods

`assign()` → *self*

Assigns the value in the editing buffer to the GET variable by evaluating Get:block with the buffer value supplied as its argument. This message is meaningful only when the Get object has input focus.

`colorDisp([`<*cColorString*>`])` → *self*

Get:colorDisp() is a method that changes a Get object's colors and redisplays it. It is exactly equivalent to assigning Get:colorSpec and issuing Get:display().

`display()` → *self*

Displays the GET on the screen. If the Get object has input focus, the Get:buffer displays in its selected color and the cursor is placed at the screen location corresponding to the current editing position within the buffer. If the GET does not have input focus, the Get:block is evaluated and the result displays in the GET's unselected color with no cursor.

`hitTest(`<*nRow*>`,` <*nColumn*>`)` → *self*

Determines the screen position specified by <*nRow*> and <*nColumn*> is on the Get object.

Applicable Hit Test Return Values

Value	Constant	Description
0	HTNOWHERE	The mouse cursor is not within the region of the screen that the GET occupies
-1025	HTCAPTION	The mouse cursor is on the GET's caption
-2049	HTCLIENT	The mouse cursor is on the GET

Button.ch contains manifest constants for the Get:hitTest() return values.

`killFocus() → self`

Takes input focus away from the Get object. Upon receiving this message, the Get object redisplays its editing buffer and discards its internal state information.

`reset() → self`

Resets the Get object's internal state information. This includes resetting the editing buffer to reflect the GET variable's current value and setting the cursor position to the first editable position within the buffer. This message has meaning only when the Get object has input focus.

`setFocus() → self`

Gives input focus to the Get object. Upon receiving this message, the Get object creates and initializes its internal state information, including the exported instance variables: Get:buffer, Get:pos, Get:decPos, and Get:original. The contents of the editing buffer are then displayed in the GET's selected color.

`undo() → self`

Sets the GET variable back to the value it had when the GET acquired input focus. This message has meaning only while the GET has input focus.

The effect of the Get:undo() message is equivalent to assigning the GETvariable from the saved value in Get:original and then sending the Get:reset() message.

`unTransform() → xValue`

Converts the character value in the editing buffer back to the data type of the original variable. Get:assign() is equivalent to Get:varPut (Get:unTransform()).

`updateBuffer() → self`

Sets the editing buffer to reflect the current value of the Get variable, and redisplays the GET. This message has meaning only while the GET has input focus.

`varGet() → GetVarValue`

Returns the current value of the GET variable. For simple GET variables this is equivalent to executing Get:block:

```
aGet:varGet() == EVAL(aGet:block)
```

However, if the GET variable is an array element, EVAL(aGet:block) will not return the value of the GET variable; in this case, you must use varGet(). An example of varGet() may be found in the READMODAL() function, defined in Getsys.prg.

`varPut() → Value`

Sets the GET variable to the passed value. For simple GET variables this is equivalent to executing Get:block with an argument:

```
aGet:varPut(aValue) == EVAL(aGet:block, aValue)
```

However, if the GET variable is an array element, EVAL(aGet:block, aValue) will not set the value of the GET variable; in this case use of varPut() is required.

Cursor Movement Methods

`end() → self`

Moves the cursor to the rightmost editable position within the editing buffer.

`home() → self`

Moves the cursor to the leftmost editable position within the editing buffer.

`left() → self`

Moves the cursor left to the nearest editable position within the editing buffer. If there is no editable position to the left, the cursor position is left unchanged.

`right() → self`

Moves the cursor right to the nearest editable position within the editing buffer. If there is no editable position to the right, the cursor position is left unchanged.

`toDecPos() → self`

Moves the cursor to the immediate right of the decimal point position in the editing buffer. This message is only meaningful when editing numeric values.

`wordLeft() → self`

Moves the cursor one word to the left within the editing buffer. If the cursor is already at the leftmost editable position, it is left unchanged.

`wordRight() → self`

Moves the cursor one word to the right within the editing buffer. If the cursor is already at the rightmost editable position, it is left unchanged.

Editing Methods

`backspace() → self`

Deletes the character to the left of the cursor moving the cursor one position to the left. If the cursor is already at the leftmost editable position in the editing buffer, this message has no effect.

`delete() → self`

Deletes the character under the cursor.

`delEnd() → self`

Deletes from the current character position to the end of the GET, inclusive.

`delLeft() → self`

Deletes the character to the left of the cursor.

`delRight() → self`

Deletes the character to the right of the cursor.

`delWordLeft() → self`

Deletes the word to the left of the cursor.

`delWordRight() → self`

Text Entry Methods

`insert(<cChar>) → self`

Inserts <*cChar*> into the editing buffer at the current cursor position, shifting the existing contents of the buffer to the right. The cursor is then placed one position to the right of the inserted string.

`overStrike(<cChar>) → self`

Puts <*cChar*> into the editing buffer at the current cursor position, overwriting the existing contents of the buffer. The cursor is placed one position to the right of the inserted string.

Examples

- This example creates a new Get object, assigns some new attributes to its instance variables, and then edits the GET with the READMODAL() function:

```
LOCAL cVar := SPACE(10)
//

// Create a new et object
objGet := GetNew()
//

// Assign some instance variables
objGet:row := 10
objGet:col := 10
//

// Assign the name of the associated
// variable and the block
objGet:name := "cVar"
objGet:block := { |cValue| IF(PCOUNT() > 0,;
                    cVar := cValue, cVar) }
//
objGet:picture := "@!"
objGet:colorSpec := "BG+/B, W+/BG"
objGet:postBlock := { |oGet| !EMPTY(oGet:varGet()) }
//

// Edit the single Get object
READMODAL({objGet})
```

Files

Source file is Getsys.prg.

See Also

@...GET, READ, READMODAL(), READVAR(), SETCOLOR()

GETACTIVE() function

Return the currently active Get object

Syntax

```
GETACTIVE (<[oGet>]) → objGet
```

Arguments

<***oGet***> is a reference to a Get object.

Returns

GETACTIVE() returns the Get object referenced by <*oGet*>. If <*oGet*> is not specified, then the current active Get object within the current READ is used. If there is no READ active when GETACTIVE() is called, it returns NIL.

Description

GETACTIVE() is an environment function that provides access to the active Get object during a READ. The current active Get object is the one with input focus at the time GETACTIVE() is called.

Examples

- This code uses a WHEN clause to force control to branch to a special reader function. Within this function, GETACTIVE() retrieves the active Get object:

```
@ 10, 10 GET x
@ 11, 10 GET y WHEN MyReader()
@ 12, 10 GET z
READ

// Called just before second get (above)
// becomes current
FUNCTION MyReader
    LOCAL objGet                // Active Get holder
    objGet := GETACTIVE()       // Retrieve current
                                // active Get
    BarCodeRead( objGet )
    RETURN (.F.)                // Causes Get to be
                                // skipped in READ
```

Files Library is CLIPPER.LIB, source file is Getsys.prg.

See Also @...GET, READ, READMODAL()

GETAPPLYKEY() function

Apply a key to a Get object from within a reader

Syntax

```
GETAPPLYKEY(<oGet>, <nKey>, <GetList>, <oMenu>,
   <nMsgRow>, <nMsgLeft>, <nMsgRight>, <cMsgColor>)
    → NIL
```

Arguments

<***oGet***> is a reference to a Get object.

<***nKey***> is the INKEY() value to apply to <*oGet*>.

<***GetList***> is a reference to the current list of Get objects.

<***oMenu***> is a reference to any top bar menu.

<***nMsgRow***> is a numeric value representing the row of the message bar.

<***nMsgLeft***> is a numeric value representing the left column of the message bar.

<***nMsgRight***> is a numeric value representing the right column of the message bar.

<***cMsgColor***> is a character string representing the colors to be used for the message bar text.

Returns

GETAPPLYKEY() always returns NIL.

Description

GETAPPLYKEY() is a Get system function that applies an INKEY() value to a Get object. Keys are applied in the default way. That is, cursor movement keys change the cursor position within the GET, data keys are entered into the GET, etc.

If the key supplied to GETAPPLYKEY() is a SET KEY, GETAPPLYKEY() will execute the set key and return; the key is not applied to the Get object.

Notes

- ***Focus:*** The Get object must be in focus before keys are applied. Refer to Get:setFocus and Get:killFocus for more information.
- ***CLEAR GETS:*** The Get object must be in focus before keys are applied. Refer to Get:setFocus and Get:killFocus for more information.

Examples

This example will apply keystrokes until Exit:

```
WHILE (oGet:exitState == GE_NOEXIT)
    GETAPPLYKEY (oGet, INKEY(0), GetList, oMenu, nMsgRow,;
    nMsgLeft, nMsgRight, nMsgColor)
ENDDO
```

Files

Library is CLIPPER.LIB, source file is Getsys.prg.

See Also

GETDOSETKEY(), GETPOSTVALIDATE(), GETPREVALIDATE(), GETREADER(), READMODAL()

GETDOSETKEY() function

Process SET KEY during GET editing

Syntax

```
GETDOSETKEY(<bKeyBlock>, <oGet>) → NIL
```

Arguments

<oGet> is a reference to the current Get object.

<bKeyBlock> is the code block to execute.

Returns

GETDOSETKEY() always returns NIL.

Description

GETDOSETKEY() is a function that executes a SET KEY code block, preserving the context of the passed Get object.

Note that the procedure name and line number passed to the SET KEY block are based on the most recent call to READMODAL().

Notes

- If a CLEAR GETS occurs in the SET KEY code, Get:exitState is set to GE_ESCAPE. In the standard system this cancels the current Get object processing and terminates READMODAL().

Examples

- The following example determines if the last key pressed, nKey, has a SET KEY associated with it. If it does, then GETDOSETKEY is called to execute that block on the current GET.

```
IF ((bKeyBlock := SETKEY (nKey)) == NIL)
   GETDOSETKEY (bKeyBlock, oGet)
ENDIF
```

Files

Library is CLIPPER.LIB, source file is Getsys.prg.

See Also

GETAPPLYKEY(), GETPOSTVALIDATE(), GETPREVALIDATE(), GETREADER(), READMODAL()

GETENV() function

Retrieve the contents of a DOS environment variable

Syntax

```
GETENV(<cEnvironmentVariable>) → cString
```

Arguments

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

Returns

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

Description

GETENV() is an environment function that lets you retrieve information from the DOS environment into an application program. Typically, this is configuration information, including path names, that gives the location of files (database, index, label, or reports). This function is particularly useful for network environments.

Notes

- ***Empty return value:*** If you are certain that an environment variable exists and yet GETENV() always returns a null string (""), be sure there are no spaces between the environment variable name and the first character of the string assigned to it in the DOS SET command.
- ***Compatibility:*** In previous releases of CA-Clipper, the function was called GETE(). This abbreviated form of GETENV() is still operational.

Examples

- This example retrieves the current DOS PATH setting, making it the current CA-Clipper PATH:

```
cPath := GETENV("PATH")
SET PATH TO (cPath)
```

- This example uses environment variables to configure the specific locations of files. When you set up a system, define environment variables that contain the location of various file types as well as the CLIPPER environment variable (see "The Runtime Environment" chapter in the *Programming and Utilities Guide*), like this:

```
C>SET LOC_DBF=<database file path>
C>SET LOC_NTX=<index file path>
C>SET LOC_RPT=<report file path>
```

  In the configuration section of your application program, assign the contents of the environment variables to variables. Then when you access a file, preface the reference with the path variable as follows:

```
cDdfDirectory := GETENV("LOC_DBF")
USE (cDdfDirectory + "Invoices")
```

Files

Library is CLIPPER.LIB.

GETPOSTVALIDATE() function

Postvalidate the current Get object

Syntax

```
GETPOSTVALIDATE(<oGet>) → lSuccess
```

Arguments

<oGet> is a reference to the current Get object.

Returns

GETPOSTVALIDATE() returns a logical value indicating whether the Get object has been postvalidated successfully.

Description

GETPOSTVALIDATE() is a Get system function that validates a Get object after editing, including evaluating Get:postBlock (the VALID clause) if present.

The return value indicates whether the GET has been postvalidated successfully. If a CLEAR GETS is issued during postvalidation, Get:exitState is set to GE_ESCAPE and GETPOSTVALIDATE() returns true (.T.).

Notes

- In the default system, a Get:exitState of GE_ESCAPE cancels the current GET and terminates READMODAL().

Examples

- This example calls GETPOSTVALIDATE to determine whether or not the VALID clause of oGet is satisfied. If not, then the user is not allowed to exit from the Get object.

```
IF (! GETPOSVALIDATE (oGet))
    oGet : exitState := GE_NOEXIT
ENDIF
```

Files

Library is CLIPPER.LIB, source file is Getsys.prg.

See Also

GETAPPLYKEY(), GETDOSETKEY(), GETPREVALIDATE(), GETREADER(), READMODAL()

GETPREVALIDATE() function

Prevalidate a Get object

Syntax

```
GETPREVALIDATE(<oGet>) → lSuccess
```

Arguments

<*oGet*> is a reference to the current Get object.

Returns

GETPREVALIDATE() returns a logical value indicating whether the Get object has been prevalidated successfully.

Description

GETPREVALIDATE() is a function that validates the Get object for editing, including evaluating Get:preBlock (the WHEN clause) if it is present. The logical return value indicates whether the GET has been prevalidated successfully.

Get:exitState is also set to reflect the outcome of the prevalidation:

Get:exitState Values

Getexit.ch	Meaning
GE_NOEXIT	Indicates prevalidation success, okay to edit
GE_WHEN	Indicates prevalidation failure
GE_ESCAPE	Indicates that a CLEAR GETS was issued

Note that in the default system, a Get:exitState of GE_ESCAPE cancels the current GET and terminates READMODAL().

Examples

- This example demonstrates the GETPREVALIDATE() function.

```
IF GETPREVALIDATE (oGet)
   // process the get
ELSE
   // WHEN clause not satisfied
   // give a warning to the user
ENDIF
```

Files

Library is CLIPPER.LIB, source file is Getsys.prg.

See Also

GETAPPLYKEY(), GETDOSETKEY(), GETPOSTVALIDATE(), GETREADER(), READMODAL()

GETREADER() function

Execute standard READ behavior for a Get object

Syntax

```
GETREADER(<oGet>, <GetList>, <oMenu>, <nMsgRow>,
   <nMsgLeft>, <nMsgRight>, <cMsgColor>) → NIL
```

Arguments

<*oGet*> is a reference to a Get object.

<***GetList***> is an array of all the Get objects in the current Get list.

<***oMenu***> is a reference to a TopBarMenu object.

<***nMsgRow***> is a numeric value representing the row number on the screen where the message bar is located.

<***nMsgLeft***> is a numeric value representing the left border of the row bar.

<***nMsgRight***> is a numeric value representing the right border of the row bar.

<***cMsgColor***> is a character string representing the color string to be used for the message bar.

Returns

GETREADER() always returns NIL.

Description

GETREADER() is a GET function that implements the standard READ behavior for GETs. By default, READMODAL() uses the GETREADER() function to read Get objects. GETREADER() in turn uses other functions in Getsys.prg to do the work of reading the Get object.

Notes

- If a Get object's Get:reader instance variable contains a code block, READMODAL() will evaluate that block in lieu of the call to GETREADER(). For more information refer to the Get:reader reference.

Examples

- This example sets the current Get object to the first GET in the Get list. Then, a READ is performed on this GET which has no menu object, but includes a message bar at row 25 from column 0 to column 80. The color of the text on the message bar is white with a red background.

```
oGet := GetList [1]
GETREADER (oGet, Getlist, NIL,25,;
            0,80,"W+/R")
```

Files

Library is CLIPPER.LIB, source file is Getsys.prg.

See Also

GETAPPLYKEY(), GETDOSETKEY(), GETPOSTVALIDATE(), GETPREVALIDATE(), READMODAL()

GFNTERASE() function

Erase a font from memory

Syntax

```
GFNTERASE(<aFont>) → NIL
```

Arguments

<***aFont***> is a pointer to the VMM region where the .FND font is loaded. <*aFont*> would have been created with an earlier call to GFNTLOAD().

Returns

GFNTERASE() always returns NIL.

Description

GFNTERASE() erases a specified font from memory. Since CA-Clipper is unable to reallocate the memory occupied by a loaded font (i.e., no automatic garbage collection), it is your responsibility to release this memory. This can be done with the following:

```
aFont2Erase := GFNTERASE(aFont2Erase)
```

Examples

- This example loads a Font file called MyFont .FND:

```
FUNCTION ShowOneFont (cString)
    LOCAL aFont
             // Load a specific font file into memory
    aFont := GFNTLOAD("MyFont.FND")
             // Display cString using the loaded font
    GWRITEAT(X , Y , cString, nColor, LLG_MODE_SET, aFont)
    // *Important*  You must erase the font from memory if it is
    // no longer used.  This is because CA-Clipper's VMM is
    // unable to automatically free the memory occupied by aFont.
    GFNTERASE(aFont)
    RETURN NIL
```

Files Library is LLIBG.LIB, header file is Llibg.ch.

See Also GFNTLOAD(), GFNTSET()

GFNTLOAD() function

Load a font file into memory

Syntax

```
GFNTLOAD(<cFontFile>) → aFont
```

Arguments

<***cFontFile***> is a character value that represents the name of the font file to load.

Returns

GFNTLOAD() returns a pointer to the VMM region where the font is loaded. This value can later be used as input to GFNTERASE().

Description

CA-Clipper supports two types of fonts:

1. Video ROM fonts—.FND

Video ROM .FND fonts are used in all the standard CA-Clipper terminal calls such as DEVOUT(), TBROWSE(), and GWRITEAT(). The size of .FND fonts is always 16x8. If you want to use a different size font, such as 8x8, you will need to supply the font file.

2. Windows Bitmap fonts—.FNT

The Windows Bitmap .FNT fonts can be used with the GWRITEAT() function.

- ***Using .FND fonts:*** The .FND fonts are 16 pixels high. Therefore, the font is displayed from pixel 1 to pixel 16 of each screen row. CA-Clipper allows you to control the number of pixel rows to use for the font:

  ```
  GFNTSET(aFont, nClipTop, nClipBottom)
  ```

  Pass <*aFont*> as NIL if you only want to change the <*nClipTop*> and <*nClipBottom*>.

  This feature provides the ability to display a one pixel frame around a GET or MENU PROMPT, etc. without having to use GWRITEAT(), because DEVOUT() (@..SAY..) is faster than GWRITEAT().

- ***Using .FNT fonts:*** Be aware that .FNT fonts are proportional. For example, this means that the characters "m" and "i" will not be the same number of pixels in width.

  To use .FNT fonts, load an .FNT font into memory :

  ```
  aFntFont := GFNTLOAD("MyFnt.FNT")
  ```

  ```
  GWRITEAT( X, Y, cString, nColor, LLG_MODE_SET, aFont )
  ```

  By passing LLG_MODE_NIL to GWRITEAT(), you can retrieve the width of the string to be displayed on screen (without displaying it).

  <*nWidth*> will contain the # of pixels of the string:

  ```
  nWidth := GWRITEAT( X , Y, cString, nColor, LLG_MODE_NIL,
  aFntFont)
  ```

  <*nColumns*> will contain the # of columns:

  ```
  nColumns := nWidth / 8.
  ```

Examples

- This example loads a Font file called MyFont.FND:`

  ```
  // Function ShowOneFont (cString)
  LOCAL aFont
  // Load a specific font file (MyFont.FND) into memory
  aFont := GFNTLOAD("MyFont.FND")
  // Display cString using the loaded font
  GWRITEAT(X, Y, cString, nColor, LLG_MODE_SET, aFont)
  //  *Important*
  // You must erase the font from memory if it is no longer
  // needed.
  // This is because CA-Clipper's VMM is unable to automatically
  // free the memory occupied by aFont.
  GFNTERASE(aFont)
  RETURN
  ```

Files

Library is LLIBG.LIB, header file is Llibg.ch.

See Also

GFNTSET(), GFNTERASE().

GFNTSET() function

Set an already loaded font as active

Syntax

```
GFNTSET(<aFont>, [<nClipTop>, <nClipBottom>])
    → aClipInfo
```

Arguments

<***aFont***> represents the pointer to the VMM region where the .FND font is loaded. <*aFont*> would have been created with GFNTLOAD().

<***nClipTop***> and <***nClipBottom***> specify how many pixel rows to use when displaying standard text output. This provides the ability to display a one pixel frame around a GET.

Returns

GFNTSET() returns a two-element array containing font clipping information.

Description

GFNTSET() sets an already loaded font as active. When GFNTSET() is called, all subsequent DEVOUT() calls will use the new font. This means that you can use multiple .FND fonts at the same time. To go back to the standard ROM font, use GFNTSET(LLG_FNT_ROM). You can also use an .FND font in a GWRITEAT() as shown in the following code:

```
aFndFont := GFNTLOAD("TestFnd.FND")
nColor   := 3
GWRITEAT(1, 1, "This is some text", nColor, LLG_MODE_SET, ;
         aFndFont)
```

GFNTSET() returns a two-element array containing font clipping information. The structure of this array is { *nTopPixelRow*, *nBottomPixelRow* }. This feature allows text to fit at a reduced height. A common requirement is to draw a frame around a GET without having to use a full text line over and under the GET. When you display a character with a DEVOUT() or a GET display method, the background always gets overwritten. So, if a frame is precisely drawn around the GET, part of the frame will be overwritten as soon as a single character is displayed.

In general, all regular fonts do not use the top and bottom row of pixels. This allows a displayed character to be clipped or restricted so that DEVPOS()/DISPLAY() will only erase the lines 1-14 (instead of 0-15) and will preserve the frame previously drawn.

For example, this is a line and frame character display. Limited display (1-14) will erase the frame. Text of all fonts is displayed in lines 1-14 so as not to disturb the frame.

```
0  ______________________________
1  XXXXXXXXXXXX
2  XXXXXXXXXXXX
3       XX
4       XX
5       XX
6       XX
7       XX
8       XX
9       XX
10      XX
11      XX
12      XX
13      XX
14      XX
15 ______________________________
```

Examples

- This example loads a font file called MyFont.FND:

```
FUNCTION ShowOneFont (cString)
   LOCAL aFontBig, aFontSmall
           // Load a couple of fonts into memory
   aFontBig    := GFNTLOAD("BigFnt.FND")
   aFontSmall  := GFNTLOAD("SmallFnt.FND")
           // Set the BigFnt.FND to be active.  As soon as
           // GFNTSET() is called, @...SAY,
           // DEVOUT() etc. will use the new font.
   GFNTSET(aFontBig)
   @1,1 SAY "This text should appear in the font 'BigFnt.FND'"
           // Display something in a Big font
   GWRITEAT(X , Y , "Using Big Font...",   nColor, LLG_MODE_SET)
   GWRITEAT(X , Y , "Using Small Font...", nColor, ;
                    LLG_MODE_SET,;    aFontSmall)
           // Set the SmallFnt.FND to be active
   GFNTSET(aFontSmall)
           // Display something in a Small font
   GWRITEAT(X , Y , "Using Small Font...", nColor, G_MODE_SET)
           // Use the standard ROM font
   GFNTSET(G_FNT_ROM)
           // Display something
   GWRITEAT(X , Y , "Using ROM Font...", nColor, G_MODE_SET)
           // *Important*  You must erase the fonts from memory
           // if it is no longer used.
           // This is because CA-Clipper's VMM is unable to
           // free the memory occupied by aFont
   GFNTERASE(aFontBig)
   GFNTERASE(aFontSmall)
   RETURN  NIL
```

Files Library is LLIBG.LIB, header file is Llibg.ch.

See Also GFNTERASE(), GFNTLOAD()

GFRAME() function

Draw a frame with a 3-D look

Syntax

```
GFRAME(<nXStart>, <nYStart>, <nXEnd>, <nYEnd>,
   <nColorBack>, <nColorLight>, <nColorDark>,
   <nWidthUp>, <nWidthRight>, <nWidthDown>,
   <nWidthLeft>, [<nMode>], [<nStyle>]) → NIL
```

Arguments

<nXStart>, ***<nYStart>***, ***<nXEnd>***, and ***<nYEnd>*** define frame coordinates in pixels.

<nColorBack> is a numeric value representing the display color of the border background.

<nColorLight> is a numeric value representing the light display color for borders.

<nColorDark> is a numeric value representing the dark display color for borders.

The range of color values is limited to the number of colors available in the selected video mode. In 16-color modes, valid values are 0-15. In 256-color modes, valid values are 0-255.

<nWidthUp> is a numeric value representing the thickness of the frame borders. The values *<nWidthRight>*, *<nXStart>*, *<nYStart>*, *<nXEnd>*, *<nYEnd>* are the external dimensions *<nWidthDown>* of the frame, and the width of the borders is measured *<nWidthLeft>* within the rectangle.

<nMode> represents the frame mode. The following are valid *<nMode>* values:

Frame Mode Constants

Constant	Description
LLG_MODE_SET	Display in SET mode (ignores any pixels present under the line displayed). This is the most common display mode.
LLG_MODE_AND	Display in AND mode (executes an AND on pixels present under the line at display time and on the display color).

Frame Mode Constants (cont.)

Constant	Description
LLG_MODE_OR	Display in OR mode (executes an OR on pixels present under the line at display time and on the display color).
LLG_MODE_XOR	Display in XOR mode (executes an XOR on pixels present under the line at display time and on the display color). See note.

<***nStyle***> defines the style of the ellipse using one of the constants listed in the table below:

Style Mode Constants

Constant	Description
LLG_FILL	Fills the frame with nColorBack. This is the default.
LLG_FRAME	Displays the frame in 3-D without filling in the frame with nColorBack.

Note: This method allows you to move objects around on the screen without damaging the background. To retrieve the initial background, just repeat the call for display in XOR mode. If the display mode parameter is missing, the last mode specified in a call to a CA-Clipper function is used.

Returns

GFRAME() always returns NIL.

Description

GFRAME() draws box frames with a 3-D look using three appropriately selected colors. For example, gray for the background, black for the dark color and white for the light color. This function can be used only if you have set the screen to a graphic mode using GMODE(). This function respects the constraints defined by GSETCLIP().

Examples

- This example shows how to display a series of frames with thin borders (3 pixels):

```
Switch to graphic mode
SET VIDEOMODE TO LLG_VIDEO_VGA_640_480_16
// Buttons
GFRAME(0, 0, 53, 399, 7, 15, 8, 3, 3, 3, 3, LLG_MODE_SET)
// Coordinates
GFRAME(0, 400, 53, 456, 7, 15, 8, 3, 3, 3, 3, LLG_MODE_SET)
// Painting
GFRAME(54, 0, 639, 399, 7, 15, 8, 3, 3, 3, 3, LLG_MODE_SET)
// Palette
GFRAME(54, 400, 639, 456, 7, 15, 8, 3, 3, 3, 3, LLG_MODE_SET)
// Messages
GFRAME(0, 457, 639, 479, 7, 15, 8, 3, 3, 3, 3, LLG_MODE_SET)
QUIT                          // End of application
```

Files Library is LLIBG.LIB, header file is Llibg.ch.

See Also DISPBOX(), GBMPDISP(), GLINE(), GRECT(), GWRITEAT()

GGETPIXEL() function

Get color information for a pixel

Syntax

```
GGETPIXEL(<nX>, <nY>) → nColor
```

Arguments

<***nX***> is a numeric value representing the position on the X-axis (row).

<***nY***> is a numeric value representing the position on the Y-axis (column).

Returns

GGETPIXEL() returns the value of a specific pixel.

Description

GGETPIXEL() can be used to get the color of a specific pixel.

Examples

- The following example retrieves the color of pixel at X-coordinate 100 and Y-coordinate 100:

```
nColor := GGETPIXEL(100, 100)
```

Files

Library is LLIBG.LIB, header file is Llibg.ch.

GLINE() function

Draw a line in graphic mode

Syntax

```
GLINE(<nXStart>, <nYStart>, <nXEnd>, <nYEnd>,
   [<nColor>], [<nMode>]) → NIL
```

Arguments

<nX1>, *<nY1>*, *<nX2>*, and *<nY2>* define the line coordinates in pixels.

<nColor> is a numeric value representing the display color. If the parameter is missing, the last color specified in a call to a CA-Clipper function is used. The range of values is limited to the number of colors available in the selected video mode. In 16-color modes, valid values are between 0 and 15. In 256-color modes, valid values are between 0 and 255.

Display Mode Constants

Constant	Description
LLG_MODE_SET	Display in SET mode (ignores any pixels present under the line displayed). This is the most common display mode.
LLG_MODE_AND	Display in AND mode (executes an AND on pixels present under the line at display time and on the display color).
LLG_MODE_OR	Display in OR mode (executes an OR on pixels present under the line at display time and on the display color).
LLG_MODE_XOR	Display in XOR mode (executes an XOR on pixels present under the line at display time and on the display color). See note.

Note: This method allows you to move objects around on the screen without damaging the background. To retrieve the initial background, just repeat the call for display in XOR mode. If the display mode parameter is missing, the last mode specified in a call to a CA-Clipper graphic function is used.

Returns

GLINE() always returns NIL.

Description

GLINE() draws lines on the screen and respects the constraints defined by GSETCLIP(). This function can be used only if you have set the screen to a graphic mode using SET VIDEOMODE.

Examples

- This example displays a line in a region restricted by clipping:

```
// Switch to graphic mode
SET VIDEOMODE TO LLG_VIDEO_VGA_640_480_16
// Restrict the display region to a portion of the screen
GSETCLIP(100, 100, 300, 300)
// Draw a line using color 6
GLINE(0, 0, 400, 400, 6, LLG_MODE_SET)
QUIT                          // End of application
```

Files

Library is LLIBG.LIB, header file is Llibg.ch.

See Also

GBMPDISP(), GLINE(), GRECT(), GWRITEAT()

GMODE() function

Switch video mode

Syntax

```
GMODE([<nMode>]) → aOldState
```

Arguments

<nMode> is a numeric value that represents the video mode.

The following are valid *<nMode>* values. They are all #defines in Llibg.ch.

Video Mode Constants

Constant	Description
LLG_VIDEO_TXT	Switch to text mode
LLG_VIDEO_VGA_640_480_16	Switch to VGA 640x480x16 mode
LLG_VIDEO_VESA_800_592_16	Switch to VESA 800x592x16
LLG_VIDEO_VESA_1024_768_16	Switch to VESA 1024x768x16
LLG_VIDEO_VESA_1280_1024_16	Switch to VESA 1280x1024x16
LLG_VIDEO_VESA_640_480_256	Switch to VESA 640x480x256
LLG_VIDEO_VESA_800_592_256	Switch to VESA 800x592x256
LLG_VIDEO_VESA_1024_768_256	Switch to VESA 1024x768x256
LLG_VIDEO_VESA_1280_1024_256	Switch to VESA 1280x1024x256

Note: LLG_VIDEO_TEXT and LLG_VIDEO_VGA_640_480_16 are the only two video modes applicable to CA-Clipper.

You can precede *<nMode>* by a minus sign. Supplying a negative argument in this manner causes GMODE() to check if the specified mode is available without switching to that mode.

Returns

If the video mode is available, GMODE() will return an array containing the settings for the requested video mode. The following table shows the array structure:

aOldState Structure

Array Position	Return Value
LLG_MODE_TEXT_ROW	Numeric value representing the number of text rows on screen.
LLG_MODE_TEXT_COL	Numeric value representing the number of text columns on screen.
LLG_MODE_GRAPH_ROW	Numeric value representing the number of graphic rows on screen.
LLG_MODE_GRAPH_COL	Numeric value representing the number of graphic columns on screen.
LLG_MODE_FONT_ROW	Numeric value representing the number of graphic rows per character.
LLG_MODE_FONT_COL	Numeric value representing the number of graphic columns per character.
LLG_MODE_COLOR_MAX	Numeric value representing the number of available colors.
LLG_MODE_IN_USE	Numeric value representing the video mode that was in use before calling GMODE(). See table in the Arguments section.
LLG_MODE_LIB_VERSION	Numeric value representing the version of the LLIBG library.
LLG_MODE_LST_COLOR	Numeric value representing the last color used in a GXXX()function. This value will be used in the next call to a GXXX() function if the color parameter is not defined. The initial value is 7.
LLG_MODE_LST_MODE	Numeric value representing the last mode (SET | AND | OR | XOR).

If *<nMode>* is omitted, GMODE() returns the current video mode settings.

Description

The GMODE() function is used to change the video mode or to retrieve information about the current video mode. CA-Clipper graphic mode supports the video modes described above. LLG_VIDEO_TEXT and LLG_VIDEO_VGA_640_480_16 are the only two video modes applicable to CA-Clipper. You can switch between modes at any time in your application. To switch to regular text mode use GMODE(LLG_VIDEO_TXT). The video will not change if GMODE() is called without parameters. GMODE() will return the current video mode information only.

When switching from LLG_VIDEO_TXT to LLG_VIDEO_VGA_640_480_16 mode, all displayed text lines are converted to the equivalent graphic mode display. This conversion does not happen when switching back to text mode. If you need an application to start in graphic mode with an empty screen, call CLS before calling GMODE (LLG_VIDEO_VGA_640_480_16).

Important! *To switch video modes, the SET VIDEOMODE command must be used. Without this command, the video mode may not be switched properly.*

Notes

During video mode switching, the library calls the CA-Clipper SETMODE() function with 25,80 as parameters. If you use other values, the call to GMODE() should be followed with a call to SETMODE (nYourRow,nYourCol).

In graphic mode, you have access to all available screen lines and columns. Any call to SETMODE() with a number of lines less than or equal to 30 causes CA-Clipper graphic mode to use a 16-pixel font that yields 30 lines.

For example, if you call SETMODE (25,80) with a font that is 8 pixels wide by 16 pixels high, and switch video mode to LLG_VIDEO_VGA_640_480_16, the same font is used. You get an additional 5 rows of text to the specified 25 i.e.,(480/16) = 30 rows of text!

CA-Clipper uses the following chart for calculating font sizes based on video settings:

SETMODE(nRow, nColumn) Calculation Chart

nRow	Fonts	Number of Rows Available
31- 34	14-pixel font	34
43	See following note	
60+	8-pixel font	60

Note: The 43-line mode being an EGA emulation of a VGA card, is not available in 640 x 480 x 16 mode.

The following are the available settings:

Available Settings

Wide / High Pixels	Column / Row
640 x 480	80 x 30 or 80 x 60
800 x 592	100 x 37 or 100 x 74
1024 x 768	128 x 48 or 100 x 96
1280 x 1024	160 x 64 or 160 x 128

During the execution of a QUIT or on the last RETURN of your application, the application is automatically set back to the starting video mode, which is usually text mode.

Note: CA-Clipper uses the ROM fonts of your VGA video card. This allows you to speed up text display. It guarantees zero use of main memory since no font loading is required, even in graphic mode. You can also avoid having to provide one or more font files with the .EXE.

Examples

- This example executes an entire application in graphic mode:

```
// Switch to graphic mode
SET VIDEOMODE TO LLG_MODE_VGA_640_480_16

// This is equivalent to GMODE (LLG_VIDEO_VGA_640_480_16)
// Your CA-Clipper instructions will now operate
// in graphic mode!
.
.
.
QUIT                            // End of application
```

- This example stores the graphic mode parameters in variables which can be used later on in an application:

```
// Switch to graphic mode and save the current graphic mode
// settings
SET VIDEOMODE TO LLG_MODE_VGA_640_480_16
aMode := GMODE()
// Store the different values
nTxtRow     := aMode[LLG_MODE_TEXT_ROW]
nTxtCol     := aMode[LLG_MODE_TEXT_COL]
nGraRow     := aMode[LLG_MODE_GRAPH_ROW]
nGraCol     := aMode[LLG_MODE_GRAPH_COL]
nFonRow     := aMode[LLG_MODE_FONT_ROW]
nFonCol     := aMode[LLG_MODE_FONT_COL]
nColorMax   := aMode[LLG_MODE_COLOR_MAX]
nVideoMode  := aMode[LLG_MODE_IN_USE]
nLibVer     := aMode[LLG_MODE_LIB_VERSION]
nLstColor   := aMode[LLG_MODE_LST_COLOR]
nLstMode    := aMode[LLG_MODE_LST_MODE]
```

- This example creates two pseudo-functions which return the number of lines and columns in pixels, similar to the way MAXROW() and MAXCOL() return line and column values:

```
#translate GMAXROW()    => GMODE()[LLG_MODE_GRAPH_ROW]
#translate GMAXCOL()    => GMODE()[LLG_MODE_GRAPH_COL]
```

- This is a similar function that returns the number of colors available:

```
#translate GMAXCOLOR() => GMODE()[LLG_MODE_COLOR_MAX]
```

Files Library is LLIBG.LIB, header file is Llibg.ch.

See Also SET VIDEOMODE

GO command

Move the pointer to the specified identity

Syntax

```
GO[TO] <xIdentity> | BOTTOM | TOP
```

Arguments

<xIdentity> is a unique value guaranteed by the structure of the data file to reference a specific item in a data source (database). In a .dbf, identity is the record number. In other data formats, identity is the unique primary key value.

BOTTOM specifies the last logical record in the current work area.

TOP specifies the first logical record in the current work area.

Description

GO[TO] is a database command that positions the record pointer in the current work area at the specified *identity*. In an Xbase data structure, this identity is the record number because every record, even an empty record, has a record number. In data structures of different design, identity may be defined as something other than record number.

Examples

- This example saves the current record number, searches for a key, and then restores the record pointer to the saved position:

```
FUNCTION KeyExists( xKeyExpr )

   LOCAL nSavRecord := RECNO()      // Save the current record
                                    // pointer position
   LOCAL lFound

   SEEK xKeyExpr
   IF ( lFound := FOUND() )
      .
      .  < statements >
      .
   ENDIF

   GOTO nSavRecord     // Restore the record pointer position

   RETURN ( lFound )
```

See Also

DBGOTO(), LASTREC(), RECNO(), SET DELETED, SET FILTER, SET RELATION, SKIP

GPOLYGON() function

Draw a polygon on screen

Syntax

```
GPOLYGON(<aVertex>, <lFilled>, <nMode>, <nOutLine>,
   <nFillColor>) → NIL
```

Arguments

<***aVertex***> is an array value representing an array of polygon vertices coordinates.

<***lFilled***> is a logical value that creates a filled or non-filled polygon. LLG_FILL or LLG_FRAME are valid constants.

<***nMode***> is a numeric value representing the video Mode. The following are valid <*nMode*> values. They are all # defines in Llibg.ch.

Constant	Description
LLG_MODE_SET	Display in SET mode (ignores any pixels present under the line displayed). This is the most common display mode.
LLG_MODE_AND	Display in AND mode (executes an AND on pixels present under the line at display time and on the display color).
LLG_MODE_OR	Display in OR mode (executes an OR on pixels present under the line at display time and on the display color).
LLG_MODE_XOR	Display in XOR mode (executes an XOR on pixels present under the line at display time and on the display color). See note.

Note: This method allows you to move objects around on the screen without damaging the background. To retrieve the initial background, just repeat the call for display in XOR mode. If the display mode parameter is missing, the last mode specified in a call to a CA-Clipper graphic function is used.

<***nOutLine***> is a numeric value representing the outline color.

<***nFillColor***> is a numeric value representing the fill color.

Returns

GPOLYGON() always returns NIL.

Description

GPOLYGON() can be used to create a polygon. Simply pass an array of coordinates which make up the polygon.

Examples

- The following array of vertices would draw a triangle:

```
aVertex := {{1,5},;       // First point
{20,45},;                 // Second point
{65,145} ;                // Third point}
```

Files

Library is LLIBG.LIB, the header file is Llibg.ch.

See Also

GELLIPSE(), GLINE(), GRECT()

GPUTPIXEL() function

Draw a pixel on the screen

Syntax

```
GPUTPIXEL(<nX>, <nY>, <nColor>, <nMode>) → NIL
```

Arguments

<nX> and *<nY>* define pixel coordinates for row and column on the screen.

<nColor> is a numeric value representing the display color of the pixel.

The range of values is limited to the number of colors available in the selected video mode. In 16-color modes, valid values are between 0 and 15. In 256-color modes, valid values are between 0 and 255.

<nMode> is the numeric value representing the video mode. The following are valid *<nMode>* values. They are all # defines in Llibg.ch.

Video Mode Constants

Constant	Description
LLG_MODE_SET	Display in SET mode (ignores any pixels present under the line displayed). This is the most common display mode.
LLG_MODE_AND	Display in AND mode (executes an AND on pixels present under the line at display time and on the display color).
LLG_MODE_OR	Display in OR mode (executes an OR on pixels present under the line at display time and on the display color).
LLG_MODE_XOR	Display in XOR mode (executes an XOR on pixels present under the line at display time and on the display color). See note.

Note: This method allows you to move objects around on the screen without damaging the background. To retrieve the initial background, just repeat the call for display in XOR mode. If the display mode parameter is missing, the last mode specified in a call to a CA-Clipper graphic function is used.

Returns

GPUTPIXEL() always returns NIL.

Description

This function can be used to change the color of a specific pixel on the screen.

Examples

- The following example will put a white pixel at X-coordinate 200 and Y-coordinate 100:

```
GPUTPIXEL (200, 100, 15, LLG_MODE_SET)
```

Files

Library is LLIBG.LIB, header file is Llibg.ch.

See Also

GWRITEAT()

GRECT() function

Draw a rectangle in graphic mode

Syntax

```
GRECT(<nXStart>, <nYStart>, <nXEnd>, <nYEnd>,
   [<nStyle>], [<nColor>], [<nMode>])    → NIL
```

Arguments

<***nXStart***>, <***nYStart***>, <***nXEnd***> and <***nYEnd***> define the rectangle coordinates in pixels.

<***nStyle***> defines the style of the rectangle using one of the constants listed in the table below:

Style Mode Constants

Constant	Description
LLG_FILL	The rectangle is displayed first and then it is filled with the color and mode specified below
LLG_FRAME	Display of the rectangle is restricted to its contour in the color and mode specified below

<***nColor***> is a numeric value that represents the display color. If this parameter is missing, the last color specified in a call to a CA-Clipper graphic function is used. The value range of values is limited to the number of colors available in the selected video mode. In 16-color modes, valid values are between 0 and 15. In 256-color modes, valid values are between 0 and 255.

<***nMode***> represents the display mode. The following are valid <*nMode*> values:

Display Mode Constants

Constant	Description
LLG_MODE_SET	Display in SET mode (ignores any pixels present under the line displayed). This is the most common display mode.
LLG_MODE_AND	Display in AND mode (executes an AND on pixels present under the line at display time and on the display color).

Display Mode Constants (cont.)

Constant	Description
LLG_MODE_OR	Display in OR mode (executes an OR on pixels present under the line at display time and on the display color).
LLG_MODE_XOR	Display in XOR mode (executes an XOR on pixels present under the line at display time and on the display color). See note.

Note: This method allows you to move objects around on the screen without damaging the background. To retrieve the initial background, just repeat the call for display in XOR mode. If the display mode parameter is missing, the last mode specified in a call to a CA-Clipper graphic function is used.

Returns

GRECT() always returns NIL.

Description

GRECT() draws filled or empty rectangles on the screen. This function can be used only if you have set the screen to a graphic mode using GMODE(). It respects the constraints defined by GSETCLIP().

Examples

- This example shows how to display a rectangle in a region restricted by clipping:

```
// Switch to graphic mode
SET VIDEOMODE LLG_VIDEO_VGA_640_480_16)
// Restrict the display region to one portion of the screen
GSETCLIP(100,100,300,300)
// Draw a filled rectangle that is partially clipped
GRECT(120,120,350,350,LG_FILL,2,LLG_MODE_SET)
QUIT                            // End of application
```

Files

Library is LLIBG.LIB, header file is Llibg.ch.

See Also

GELLIPSE(), GLINE(), GPOLYGON(), GWRITEAT(), GBMPDISP()

GSETCLIP() function

Define the allowed display area

Syntax

```
GSETCLIP([<nX1>, <nY1>, <nX2>, <nY2>]) → aOldRegion
```

Arguments

<***nX1***> is a numeric value representing the left X-axis (column) position of the region in pixels.

<***nY1***> is a numeric value representing the top Y-axis (row) position of the region in pixels.

<***nX2***> is a numeric value representing the right X-axis (column) position of the region in pixels.

<***nY2***> is a numeric value representing the bottom Y-axis (row) position of the region in pixels.

Either all parameters should be specified or none.

Returns

GSETCLIP() returns an array of four numbers representing the new clipping region if you specify all arguments, and the current region if you specify none.

Description

GSETCLIP() limits the active display to a portion of the screen. This function applies only to CA-Clipper graphic mode functions beginning with "G", such as GRECT(), GLINE(), GBMPDISP(). Text emulation functions such as DEVOUT() and QOUT() are not affected by setting clipping regions. To enhance redraw speed, clipping of the function for rewriting text in GWRITEAT() graphic mode is simplified. CA-Clipper graphic mode checks whether each character displayed fits entirely in the clipping region. If it does, the character is displayed; otherwise, it is not.

Notes

The clipping region is initialized to be the entire screen. To specify maximum clipping (which amounts to eliminating clipping altogether), you need to pass parameters to GSETCLIP() corresponding to the entire screen. There are two ways to do this:

- Save the clipping state immediately after the change. This is equivalent to the whole screen at that instant:

```
SET VIDEOMODE TO LLG_VIDEO_VGA_640_480_16
// Retrieve clipping value
aMaxClip := GSETCLIP()
// Execute clipping
GSETCLIP(<nCoord list>)
.
.
.
// Your code
.
.
.
// Retrieve clipping value
GSETCLIP(aMaxClip[1] , aMaxClip[2] , aMaxClip[3] , aMaxClip[4])
```

- Use GMODE() which returns the screen size in pixels:

```
// Switch to graphic mode
SET VIDEOMODE TO LLG_VIDEO_VGA_640_480_16

// Perform clipping
GSETCLIP(<nCoord list>)
.
.
.
// Your code
.
.
.
// Retrieve clipping value
GSETCLIP(0, 0, ;
GMODE()[LLG_MODE_GRAPH_ROW], ;
GMODE()[LLG_MODE_GRAPH_COL])
```

Examples

- This example restricts display to a portion of the screen:

```
// Switch to graphic mode
SET VIDEOMODE TO LLG_VIDEO_VGA_640_480_16

// Restrict the display region
GSETCLIP(100, 100, 300, 300)

DO WHILE ! (INKEY() == K_ESC)
   AreaDraw()                    // Call a drawing function
ENDDO
QUIT                             // End of application
```

- This example stores clipping region parameters for later use:

```
// Switch to graphic mode
SET VIDEOMODE TO LLG_VIDEO_VGA_640_480_16

// Retrieve current values
aOldRegion := GSETCLIP()

// Restrict display region to a portion of the screen
GSETCLIP(100,100,300,300)

DO WHILE !(INKEY() == K_ESC)
   AreaDraw()
   // Call a function that draws something
ENDDO

// Reset original clipping regions
GSETCLIP(aOldRegion[1],;
   aOldRegion[2],;
   aOldRegion[3],;
   aOldRegion[4])
QUIT                          // End of application
```

Files Library is LLIBG.LIB, header file is Llibg.ch.

See Also GLINE(), GRECT(), GWRITEAT(), GBMPDISP()

GSETEXCL() function

Define a screen region to be excluded from display

Syntax

```
GSETEXCL([<nExclArea>]|[<nTop>, <nLeft>, <nBottom>,
   <nRight>, <nType>]) → NIL
```

Arguments

<nExclArea> is a numeric value representing the exclusion area number.

<nTop>, <nLeft>, <nBottom>, <nRight>, and ***<nType>*** define the exclusion area coordinates.

Constant	Description
LLM_COOR_TEXT	*<nTop>* which is the default, is the row line number.
LLM_COOR_GRAPH	*<nTop>* is the row in pixels.

Note: Either zero, *<nExclArea>*, or all other parameters should be specified.

Returns

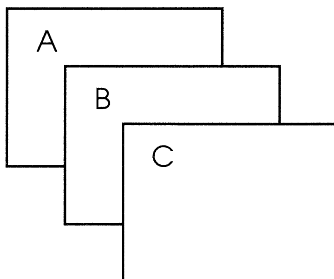
If *<nExclArea>* is specified, an array of exclusion area coordinates for *<nExclArea>* is returned. GSETEXCL(LLG_EXCL_KILL_ALL) deletes all previously defined exclusion areas. If no arguments are specified, GSETEXCL() returns the number of active exclusion areas. If *<nTop>*, *<nLeft>*, *<nBottom>*, *<nRight>*,and *<nType>* are specified, NIL is returned.

Description

This function is used to prevent output from being displayed in a defined region of the screen. The GSETEXCL() function is probably the most complex function in CA-Clipper graphic mode. If you have never used the DISPBEGIN() and DISPEND() functions, please read CA-Clipper's documentation for further explanation of these two functions. Essentially, these functions are used to trap or "buffer" all screen output. DISPBEGIN() initiates the buffering and DISPEND() stops the buffering and updates the screen.

GSETEXCL() is the opposite of GSETCLIP(). GSETEXCL() and GSETCLIP() must never be used at the same time. A call to GSETCLIP() will destroy all previously defined exclusion areas. CA-Clipper internally maintains exclusion areas as if they were multiple inclusion areas.

To understand GSETEXCL(), imagine a cascading window system with three windows, A, B and C:



Our goal is to display information in window A while it is covered by windows B and C.

- ***Old text mode technique:*** Using CA-Clipper in text mode and assuming you saved each window region prior to painting the window boxes, you would probably do the following:

  1. Issue a DISPBEGIN() to activate video buffering.

  2. Pop off each screen by individually saving the current screens and immediately restoring that window's previous background, starting with window C, and then doing window B. For example,

     ```
     cNewScrC := SAVESCREEN(nTop, nLeft, nBottom, nRight)
     RESTSCREEN(nTop, nLeft, nBottom, nRight,cOldScrC)
     cNewScrB := SAVESCREEN(nTop, nLeft, nBottom, nRight)
     RESTSCREEN(nTop, nLeft, nBottom, nRight,cOldScrB)
     ```

  3. At this point, window A is the only window visible and it is easy to manipulate or paint information in window A.

  4. You would then save and redisplay window B and C, for example,

     ```
     cOldScrC := SAVESCREEN(nTop, nLeft, nBottom, nRight)
     RESTSCREEN(nTop, nLeft, nBottom, nRight,cNewScrC)
     cOldScrB := SAVESCREEN(nTop, nLeft, nBottom, nRight)
     RESTSCREEN(nTop, nLeft, nBottom, nRight,cNewScrB)
     ```

5. Issue a DISPEND() because up to now the user has not seen any screen activity since every screen output function has been buffered!

   By issuing a DISPEND(), ALL of the buffered data is instantly flushed to the screen. Essentially the user sees only the final screen and not the intermediate screen activity along the way. The user is not aware of the displaying and redisplaying of windows B and C. This provides a smooth visual effect. This buffering method is available in text mode because display orders are fast, simple and the memory needed to save a full screen is small (4KB).

- ***New graphic mode technique:*** The situation is totally different in graphic mode. DISPBEGIN() and DISPEND() are not enabled in CA-Clipper graphic mode. CA-Clipper uses exclusion areas instead of removing windows until we have the desired window current. This exclusion area concept is used in almost every graphical system, including Microsoft Windows. An exclusion area is simply a defined screen region which is prevented from having data written to it.

  In the example above, you would declare windows B and C as exclusion areas. This prevents data from being written to these windows. Therefore, when you modify information in window A, windows B and C are not affected. The above example using exclusion areas would be as follows:

```
GSETEXCL(nTop, nLeft, nBottom, nRight) // Window B coordinates
GSETEXCL(nTop, nLeft, nBottom, nRight) // Window C coordinates
// Manipulate or paint information in window A.
GSETEXCL (LLG_EXCL_KILL_ALL)
```

Notes

You must reset the exclusion areas once the screen work is completed; otherwise, the exclusion areas remain active. This essentially prevents any data from ever being written to those screen regions. As you can see, DISPBEGIN() and DISPEND() require a lot of work to manage a simple cascade of windows, and the overhead in graphic mode renders the technique unusable. Exclusion zones do not consume memory, are faster, and are simpler to use.

Files

Library is LLIBG.LIB, header file is Llibg.ch.

See Also

DISPBEGIN(), DISPEND(), and GSETCLIP()

GSETPAL() function

Change components of a color

Syntax

```
GSETPAL(<nColor>, <nRedValue>, <nGreenValue>,
   <nBlueValue>) → aOldPalette

GSETPAL() → aOldPalette

GSETPAL(<aPalette>) → aOldPalette

GSETPAL(<nColor>, <aRGB>) → aOldPalette
```

Arguments

<***nColor***> is a numeric value that represents a display color.

The range of values for the parameter is limited to the number of colors available in the selected video mode. In 16-color modes, valid values are between 0 and 15. In 256-color modes, valid values are between 0 and 255.

<***nRedValue***> is a numeric value that represents the red component assigned to <*nColor*>. Its possible values are from 0 to 63.

<***nGreenValue***> is a numeric value that represents the green component assigned to <*nColor*>. Its possible values are from 0 to 63.

<***nBlueValue***> is a numeric value that represents the blue component assigned to <*nColor*>. Its possible values are from 0 to 63.

<***aFullOldPalette***> is an array of components in the palette. The structure of the array is { *nRedValue, nGreenValue, nBlueValue* }.

<***aRGB***> is an array of components in the palette. The structure of the array is { *nRedValue, nGreenValue, nBlueValue* }.

Returns

<***aOldPalette***> represents an array of components in palette:

```
aOldPalette := { nRedValue, nGreenValue, nBlueValue }
```

Description

The GSETPAL() function changes a color's basic component values. This function must be called for each color if you want to change the palette. It can be used in text or graphic mode.

Notes

CA-Clipper saves the VGA | VESA environment including the palette. It also saves the first call to GMODE() or SET VIDEOMODE. When you do a QUIT or reach the last RETURN of your application, your initial video mode is automatically restored. You do not need to save the original color palette.

Examples

- This example stores a video mode's color components in a multidimensional array. It is identical to calling GSETPAL() with no parameters:

```
FUNCTION aPalSave()
   LOCAL aPalStore := {}
   LOCAL nI        := 0

// For all colors in the current video mode
   FOR nI := 1 TO GMODE()[LLG_MODE_COLOR_MAX]
// Save color reference and color components
      AADD(aPalStore, {nI-1,;
         GSETPAL(nI-1)[1],;
         GSETPAL(nI-1)[2],;
         GSETPAL(nI-1)[3]})
   NEXT nI
   // Return an array of arrays
   RETURN aPalStore
```

- This example loads all color components of a video mode using a multidimensional array which contains the individual components:

```
FUNCTION aPalRest(aPalette)
   LOCAL nI := 0
   // For all the arrays in the major array
   FOR nI := 1 TO LEN(aPalette)
   // Reset the color components
      GSETPAL(aPalette[nI,1],;
         aPalette[nI,2],;
         aPalette[nI,3],;
         aPalette[nI,4];)
   NEXT nI
   RETURN aPalette
```

Files

Library is LLIBG.LIB, header file is Llibg.ch.

GWRITEAT() function

Draw graphic text without background

Syntax

```
GWRITEAT(<nColStart>, <nLnStart>, <cString>, [<nColor>],
   [<nMode>], [<aFont>]) → nWidth
```

Arguments

<nColStart> and *<nLnStart>* define graphic text coordinates in pixels.

*<cString>*is a character value representing the character string to be displayed.

*<nColor>*is a numeric value representing the display color. If the parameter is missing, the last color specified in a call to a CA-Clipper graphic function is used.

The range of values is limited to the number of colors available in the selected video mode. In 16-color modes, valid values are between 0 and 15. In 256-color modes, valid values are between 0 and 255.

<nMode> is a numeric value representing the display mode. The following are valid *<nMode>* values. They are all #defines in Llibg.ch.

Display Mode Constants

Constant	Description
LLG_MODE_SET	Display in SET mode (ignores any pixels present under the line displayed). This is the most common display mode.
LLG_MODE_AND	Display in AND mode (executes an AND on pixels present under the line at display time and on the display color).
LLG_MODE_OR	Display in OR mode (executes an OR on pixels present under the line at display time and on the display color).
LLG_MODE_XOR	Display in XOR mode (executes an XOR on pixels present under the line at display time and on the display color). See note.
LLG_MODE_NIL	Allows you to compute the width of the string without displaying anything on the screen. Be aware that .FNT fonts are proportional, which means that an "m" and an "i" do not use the same number of pixels.

Note: This method allows you to move objects around on the screen without damaging the background. To retrieve the initial background, just repeat the call for display in XOR mode. If the display mode parameter is missing, the last mode specified in a call to a CA-Clipper graphic function is used.

<***aFont***> is an optional font array pointer which was created with the GFNTLOAD("MyFont.FND") function:

```
FUNCTION ShowOneFont (cString)
    LOCAL aFont
    // Load a specific font file into memory
    aFont := GFNTLOAD("MyFont.FND")
    // Display cString using the loaded font
    GWRITEAT(X , Y , cString, nColor, LLG_MODE_SET, aFont)
    // *Important*
    // You must erase the font if it is no longer used.
    GFNTERASE(aFont)
    RETURN   NIL
```

Returns

GWRITEAT() returns the width of the written text in pixels.

Description

GWRITEAT() displays text in graphic mode without affecting the background. It is important not to confuse the functions DEVOUT() and GWRITEAT(). Even when you are in graphic mode, you should continue to use DEVPOS(), DEVOUT(), and QOUT() for all "normal" displays. The functions related to GETs, TBROWSE, etc. use both foreground and background colors, and text can only be displayed in areas whose size is a multiple of the selected font size.

Notes

GWRITEAT() should be used for graphic enhancements, such as a window title within its frame (as you do not want to overwrite the window frame's borders), or to display information in graphic areas where you want to explicitly handle whether or not the background is erased before a new display, as in adding a title to a bitmap.

Unlike DEVOUT(), this function receives graphic coordinates only. This allows you to display text pixel by pixel. Since GWRITEAT() works with pixel coordinates instead of row and column coordinates, you may need to properly calculate nX and nY. This function can be used only if you have set the screen to a graphic mode using SET VIDEOMODE().

This function respects the constraints defined by GSETCLIP().

Examples

- This example writes a see-through title on the frame of a 3-D box:

```
// Switch to graphic mode
SET VIDEOMODE TO LLG_VIDEO_VGA_640_480_16
// Display a 3D box of constant width 16x16x16x16
DISPBOX(nTop, nLeft, nBottom, nRight, LLG_BOX_GRAY_SQUARE)
// Write the alias name transparently in the 3D frame
GWRITEAT( nLeft * GMODE()[LLG_MODE_FONT_COL],;
nTop   * GMODE()[LLG_MODE_FONT_ROW],;
ALIAS(SELECT()),;
4,;
LLG_MODE_SET)
QUIT                          // End of application
```

Files Library is LLIBG.LIB, header file is Llibg.ch.

See Also GBMPDISP(), GFNTLOAD(), GLINE(), GRECT()

HARDCR() function

Replace all soft carriage returns in a character string with hard carriage returns

Syntax

```
HARDCR(<cString>) → cConvertedString
```

Arguments

<cString> is the character string or memo field to be converted.

Returns

HARDCR() returns a character string up to 65,535 (64K) characters in length.

Description

HARDCR() is a memo function that replaces all soft carriage returns (CHR(141)) with hard carriage returns (CHR(13)). It is used to display long character strings and memo fields containing soft carriage returns with console commands. In CA-Clipper, console commands (including REPORT and LABEL FORM) do not automatically convert soft carriage returns to hard carriage returns, making it necessary for you to explicitly make the conversion. Soft carriage returns are added by MEMOEDIT() when lines wrap.

Notes

- ***HARDCR() in REPORT and LABEL FORMs:*** If HARDCR() is used in a REPORT FORM or LABEL FORM contents expression and nowhere else, you must declare it EXTERNAL to ensure that it is linked.
- HARDCR() does not remove a soft carriage return if it is specified alone (i.e. HARDCR (CHR(141)). It requires the combination of a soft carriage return and a line feed, i.e., CHR(141) + CHR(10).

Examples

- To display a memo field formatted with the automatic word wrapping of MEMOEDIT():

```
USE Sales NEW
? HARDCR(Sales->Notes)
```

Files

Library is EXTEND.LIB.

See Also

?|??, @...SAY, EXTERNAL*, LABEL FORM, MEMOTRAN(), REPORT FORM, STRTRAN()

HEADER() function

Return the current database file header length

Syntax

```
HEADER() → nBytes
```

Returns

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

Description

HEADER() is a database function that is used with LASTREC(), RECSIZE(), and DISKSPACE() to create procedures for backing up files.

By default, HEADER() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression (see example below).

Examples

- This example determines the header size of Sales.dbf:

```
USE Sales NEW
? HEADER()            // Result: 258
```

- This example defines a pseudofunction, DbfSize(), that uses HEADER() with RECSIZE() and LASTREC() to calculate the size of the current database file in bytes:

```
#define DbfSize()     ((RECSIZE() * LASTREC()) + ;
       HEADER() + 1)
```

Later you can use DbfSize() as you would any function:

```
USE Sales NEW
USE Customer NEW
? DbfSize()
? Sales->(DbfSize())
```

Files Library is EXTEND.LIB.

See Also DISKSPACE(), LASTREC(), RECSIZE()

I2BIN() function

Convert a CA-Clipper numeric to a 16-bit binary integer

Syntax

```
I2BIN(<nInteger>) → cBinaryInteger
```

Arguments

<*nInteger*> is an integer numeric value to be converted. Decimal digits are truncated.

Returns

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

Description

I2BIN() is a low-level file function that converts an integer numeric value to a character string formatted as a binary integer—least significant byte first. I2BIN() is used with FWRITE() to convert a CA-Clipper numeric to a standard binary form. The inverse of I2BIN() is BIN2I().

Examples

- This example opens a database file using low-level file functions and writes a new date of the last update to bytes 1-3:

```
#include "Fileio.ch"
//
nHandle = FOPEN("Sales.dbf", FO_READWRITE)
//
// Convert date of last update to int
nYear = I2BIN(90)
nMonth = I2BIN(12)
nDay = I2BIN(15)
//
// Point to the date of last update
FSEEK(nHandle, 1, FS_SET)
//
// Write the new update date using only the first byte
FWRITE(nHandle, nYear, 1)
FWRITE(nHandle, nMonth, 1)
FWRITE(nHandle, nDay, 1)
FCLOSE(nHandle)
```

Files

Library is EXTEND.LIB, source file is SOURCE\SAMPLE\EXAMPLEA.ASM.

See Also

BIN2I(), BIN2L(), BIN2W(), CHR(), FWRITE(), L2BIN()

IF statement

Execute one of several alternative blocks of statements

Syntax

```
IF <lCondition1>
   <statements>...
[ELSEIF <lCondition2>]
   <statements>...
[ELSE]
   <statements>...
END[IF]
```

Arguments

<lCondition> is a logical control expression. If it evaluates to true (.T.), all following statements are executed until an ELSEIF, ELSE, or ENDIF is encountered.

ELSEIF <lCondition> identifies statements to execute if the associated condition evaluates to true (.T.) and all preceding IF or ELSEIF conditions evaluate to false (.F.). Any number of ELSEIF statements can be specified within the same IF...ENDIF control structure.

ELSE identifies statements to execute if the IF and all preceding ELSEIF conditions evaluate to false (.F.).

Description

The IF control structure works by branching execution to statements following the first true (.T.) evaluation of the IF or any ELSEIF condition. Execution then continues until the next ELSEIF, ELSE, or ENDIF is encountered whereupon execution branches to the first statement following the ENDIF.

If no condition evaluates to true (.T.), control passes to the first statement following the ELSE statement. If an ELSE statement is not specified, control branches to the first statement following the ENDIF statement.

IF...ENDIF structures may be nested within IF...ENDIF structures and other control structure commands. These structures, however, must be nested properly.

The IF...ELSEIF...ENDIF form of the IF construct is identical to DO CASE...ENDCASE. There is no specific advantage of one syntax over the other. The IF construct is also similar to the IF() function which can be used within expressions.

Examples

- This example evaluates a number of conditions using an IF...ELSEIF...ENDIF construct:

```
LOCAL nNumber := 0
//
IF nNumber < 50
   ? "Less than 50"
ELSEIF nNumber = 50
   ? "Is equal to 50"
ELSE
   ? "Greater than 50"
ENDIF
```

See Also

BEGIN SEQUENCE, DO CASE, DO WHILE, FOR, IF()

IF() function

Return the result of an expression based on a condition

Syntax

```
IF(<lCondition>, <expTrue>, <expFalse>) → Value
```

Arguments

<lCondition> is a logical expression to be evaluated.

<expTrue> is the value, a condition-expression, of any data type, returned if *<lCondition>* is true (.T.).

<expFalse> is the value, of any date type, returned if *<lCondition>* is false (.F.). This argument need not be the same data type as *<expTrue>*.

Returns

IF() returns the evaluation of *<expTrue>* if *<lCondition>* evaluates to true (.T.) and *<expFalse>* if it evaluates to false (.F.). The value returned is the data type of the valid condition-expression.

Description

IF() is a logical conversion function. It is one of the most powerful and versatile functions in CA-Clipper. It provides a mechanism to evaluate a condition within an expression. With this ability you can convert a logical expression to another data type.

Examples

- This example converts a logical data value to a numeric data value:

```
lPaid = .T.
? IF(lPaid, 1, 0)              // Result: 1
```

- In this example a logical field is formatted depending on whether the Customer is past due or not:

```
@ ROW() + 1, 25 SAY IF(lPaid, SPACE(10), "Go get 'em")
```

- If you are printing forms, you can print an indicating symbol in different columns depending on the value of a logical field:

```
@ ROW(), IF(InHospital, 10, 12) SAY "X"
```

- You can also use IF() to force the LABEL FORM to print blank lines. Enter the following expression when you create the label with RL.EXE:

```
IF(EMPTY(Company), CHR(255), Company)
```

Files Library is CLIPPER.LIB.

See Also DO CASE, IIF()

IIF() function

Return the result of an expression based on a condition

Syntax

```
[I]IF(<lCondition>, <expTrue>, <expFalse>) → Value
```

Arguments

<lCondition> is a logical expression to be evaluated.

<expTrue> is the value, a condition-expression, of any data type, returned if *<lCondition>* is true (.T.).

<expFalse> is the value, of any date type, returned if *<lCondition>* is false (.F.). This argument need not be the same data type as *<expTrue>*.

Returns

IIF() returns the evaluation of *<expTrue>* if *<lCondition>* evaluates to true (.T.) and *<expFalse>* if it evaluates to false (.F.). The value returned is the data type of the valid condition-expression.

Description

IIF() is a logical conversion function. It is one of the most powerful and versatile functions in CA-Clipper. It provides a mechanism to evaluate a condition within an expression. With this ability you can convert a logical expression to another data type.

Examples

- This example converts a logical data value to a numeric data value:

```
lPaid = .T.
? IIF(lPaid, 1, 0)                // Result: 1
```

- In this example a logical field is formatted depending on whether the Customer is past due or not:

```
@ ROW() + 1, 25 SAY IIF(lPaid, SPACE(10), "Go get 'em")
```

- If you are printing forms, you can print an indicating symbol in different columns depending on the value of a logical field:

```
@ ROW(), IIF(InHospital, 10, 12) SAY "X"
```

- You can also use IIF() to force the LABEL FORM to print blank lines. Enter the following expression when you create the label with RL.EXE:

```
IIF(EMPTY(Company), CHR(255), Company)
```

Files Library is CLIPPER.LIB.

See Also DO CASE, IF()

INDEX command

Create an index file

Syntax

```
INDEX ON <expKey> [TAG <cOrderName>] [TO
   <cOrderBagName>] [FOR <lCondition>] [ALL]
   [WHILE <lCondition>] [NEXT <nNumber>]
   [RECORD <nRecord>] [REST]
   [EVAL <bBlock>] [EVERY <nInterval>]
   [UNIQUE] [ASCENDING|DESCENDING]
   [USECURRENT] [ADDITIVE]
   [CUSTOM] [NOOPTIMIZE]
```

Note: Although both the TAG and the TO clauses are optional, you must specify at least one of them.

Arguments

<expKey> is an expression that returns the key value to place in the index for each record in the current work area. *<expKey>* can be character, date, logical, or numeric type. The maximum length of the index key expression is determined by the driver.

TAG <cOrderName> is the name of the order to be created. *<cOrderName>* can be any CA-Clipper expression that evaluates to a string constant.

TO <cOrderBagName> is the name of a disk file containing one or more orders. The active RDD determines the order capacity of an order bag. The default DBFNTX driver only supports single-order bags, while other RDDs may support multiple-order bags (e.g., the DBFCDX and DBFMDX drivers). You may specify *<cOrderBagName>* as the file name with or without a path name or extension. If an extension is not provided as part of *<cOrderBagName>*, CA-Clipper will use the default extension of the current RDD.

Both the TAG and TO clauses are optional, but you must use at least one of them.

FOR <lCondition> specifies the conditional set of records on which to create the order. Only those records that meet the condition are included in the resulting order. *<lCondition>* is an expression that may be no longer than 250 characters under the DBFNTX and DBFNDX drivers. The maximum value for these expressions is determined by the RDD. The FOR condition is stored as part of the order bag and used when updating or recreating the index using the REINDEX command. Duplicate key values are not added to the order bag.

Drivers that do not support the FOR condition will produce an "unsupported" error.

The FOR clause provides the only scoping that is maintained for all database changes. All other scope conditions create orders that do not reflect database updates.

ALL specifies all orders in the current or specified work area. ALL is the default scope of INDEX .

WHILE <lCondition> specifies another condition that must be met by each record as it is processed. As soon as a record is encountered that causes the condition to fail, the INDEX command terminates. If a WHILE clause is specified, the data is processed in the controlling order. The WHILE condition is transient (i.e., it is not stored in the file and not used for index updates and REINDEXing purposes). The WHILE clause creates *temporary* orders, but these orders are not updated.

Drivers that do not support the WHILE condition will produce an "unsupported" error.

Using the WHILE clause is more efficient and faster than using the FOR clause. The WHILE clause only processes data for which *<lCondition>* is true (.T.) from the current position. The FOR clause, however, processes all data in the data source.

NEXT <nNumber> specifies the portion of the database to process. If you specify NEXT, the database is processed in the controlling order for the *<nNumber>* number of identities. The scope is transient (i.e., it is not stored in the order and not used for REINDEXing purposes).

RECORD <nRecord> specifies the processing of the specified record.

REST specifies the processing of all records from the current position of the record pointer to the end of file (EOF).

EVAL <bBlock> evaluates a code block every *<nInterval>*, where *<nInterval>* is a value specified by the EVERY clause. The default value is 1. This is useful in producing a status bar or odometer that monitors the indexing progress. The return value of *<bBlock>* must be a logical data type. If *<bBlock>* returns false (.F.), indexing halts.

EVERY <nInterval> is a clause containing a numeric expression that modifies the number of times *<bBlock>* is EVALuated. The EVERY option of the EVAL clause offers a performance enhancement by evaluating the condition for every *n*th record instead of evaluating *every* record ordered. The EVERY keyword is ignored if you specify no EVAL condition.

UNIQUE specifies that the key value of each record inserted into the order be unique. Duplicate key values are not added to the order.

ASCENDING specifies that the keyed pairs be sorted in increasing order of value. If neither ASCENDING nor DESCENDING is specified, ASCENDING is assumed. Although not stored as an explicit part of the file, ASCENDING is an implicit file attribute that is understood by the REINDEX command.

Drivers that do not support the ASCENDING condition will produce an "unsupported" error. The following keywords are new to CA-Clipper 5.3.

DESCENDING specifies that the keyed pairs be sorted in decreasing order of value. Using this keyword is the same as specifying the DESCEND() function within *<expKey>*, but without the performance penalty during order updates. If you create a DESCENDING index, you will not need to use the DESCEND() function during a SEEK. DESCENDING is an attribute of the file, where it is stored and used for REINDEXing purposes.

Drivers that do not support the DESCENDING condition will produce an "unsupported" error.

USECURRENT specifies that only records in the controlling order—and within the current range as specified by ORDSETSCOPE()—will be included in this order. This is useful when you have already created a conditional order and want to reorder the records which meet that condition, and/or to further restrict the records meeting a condition. If not specified, all records in the database file are included in the order.

ADDITIVE specifies that any open orders should remain open. If not specified, all open orders are closed before creating the new one. Note, however, that the production index file is never closed.

CUSTOM specifies that a custom built order will be created for RDDs that support them. A custom built order is initially empty, giving you complete control over order maintenance. The system does not automatically add and delete keys from a custom built order. Instead, you explicitly add and delete keys using ORDKEYADD() and ORDKEYDEL(). This capability is excellent for generating pick lists of specific records and other custom applications.

NOOPTIMIZE specifies that the FOR condition will not be optimized. If NOOPTIMIZE is not specified, the FOR condition will be optimized if the RDD supports optimization.

Description

The INDEX command adds a set of keyed pairs, ordered by <*expKey*> to a file specified by <*cOrderBagName*> using the database open in the current work area.

In RDDs that support production or structural indexes (e.g., DBFCDX, DBFMDX), if you specify a tag but do not specify an order bag, the tag is created and added to the order bag. If no production or structural index exists, it will be created and the tag will be added to it.

When using RDDs that support multiple order bags, you must explicitly SET ORDER (or ORDSETFOCUS()) to the desired controlling order. If you do not specify a controlling order, the data file will be viewed in natural order.

If <*cOrderBagName*> does not exist, it is created in accordance with the RDD in the current or specified work area.

If <*cOrderBagName*> exists and the RDD specifies that order bags can only contain a single order, <*cOrderBagName*> is erased and the new order is added to the order bag and to the order list in the current or specified work area.

If <*cOrderBagName*> exists and the RDD specifies that order bags can contain multiple tags, <*cOrderName*> is created if it does not already exist; otherwise, <*cOrderName*> is replaced in <*cOrderBagName*> and the order is added to the order list in the current or specified work area.

ASCENDING or DESCENDING specifies the sequence of keyed pairs in the order. If neither clause is specified, the default is ASCENDING.

If you specify the UNIQUE clause, the resulting order will contain only unique records. Some RDDs may do this by only including record references to a key value once. Others may produce a runtime recoverable error as a non-unique key insertion is attempted.

The EVAL clause lets you specify a code block to be evaluated as each record is placed in the order. The EVERY clause lets you modify how often *<bBlock>* is called. Instead of evaluation as each record is placed in the order, evaluation only occurs as every *<nInterval>* records are placed in the order.

The INDEX command accepts certain clauses that let the user create conditional and partial orders. Some orders are intended to be maintained across the application, others are considered "temporary" orders.

The FOR clause provides the only order scoping that is permanent and can be maintained across the life of the application. The string passed as the FOR condition is stored within the order for later use in maintaining the order. Though only accessing part of a database, orders created using this clause exist as long as the database is active. The FOR clause lets you create *maintainable scoped* orders.

The WHILE, NEXT, REST and RECORD clauses process data from the current position of the database cursor in the default or specified work area. If you specify these clauses, the order list remains open and the active order is used to organize the database while it is being created. These clauses let you create temporary (*non-maintainable*) orders. Orders created using these clauses contain records in which *<lCondition>* is true (.T.) at the location of the record pointer.

Notes

RDD support: Not all RDDs support all aspects of the INDEX command. See the "Replaceable Database Driver Architecture" chapter in the *Drivers Guide* for details on a particular RDD.

Examples

- The following example creates a simple order (index) based on one field (Acct):

```
USE Customer NEW
INDEX ON Customer->Acct TO CuAcct
```

- This example creates a conditional order (index) based on a FOR clause. This index will contain only records whose field TransDate contains a date greater than or equal to January 1, 1995:

```
USE Invoice NEW
INDEX ON Invoice->TransDate      ;
   TO InDate      ;
   FOR ( Invoice->TransDate >= CTOD( "01/01/95" ) )
```

- This example creates an order in a multiple-order bag (i.e., a tag in an index that can support multiple tags in an index file):

```
USE Customer NEW
INDEX ON Customer->Acct TAG CuAcct TO Customer
```

- The following example creates an order that calls a routine, MyMeter, during its creation:

```
#define MTR_INCREMENT   10

USE Customer NEW
INDEX ON Customer->Acct TO CuAcct EVAL ;
      {|| MYMETER() } EVERY MTR_INCREMENT

FUNCTION MYMETER()

   STATIC nRecsDone := 0

   nRecsDone := += MTR_INCREMENT
   ? ( nRecsDone/LASTREC() ) * 100

   RETURN (.T.)
```

See Also

CLOSE, DBCREATEINDEX(), DBORDERINFO(), DBREINDEX(), DBSEEK(), DBSETINDEX(), DBSETORDER(),DESCEND(), DTOS(), INDEXKEY(), INDEXORD(), ORDCONDSET(), ORDCREATE(), ORDKEYADD(), ORDKEYDEL(), ORDSCOPE(), REINDEX, SEEK, SET INDEX, SET ORDER, SORT, SOUNDEX(), USE

INDEXEXT() function

Return the default index extension based on the database driver currently linked

Syntax

```
INDEXEXT() → cExtension
```

Returns

Unless you have linked another database driver, INDEXEXT() returns ".ntx" to indicate that the default CA-Clipper driver is in effect. If the dBASE III PLUS compatible database driver is linked, the function returns ".ndx".

Description

INDEXEXT() returns the default index file extension by determining which database driver is currently linked. Note that it is preferable to use ORDBAGEXT() than INDEXEXT().

Notes

- INDEXEXT() returns the default index extension of the driver loaded, not the actual index file extension. If no driver is loaded, ".ntx" is returned.

Examples

- In this example, INDEXEXT() creates an existence test for the Customer index file independent of the database driver linked into the current program:

```
USE Customer NEW
//
IF .NOT. FILE("Customer" + INDEXEXT())
   INDEX ON CustName TO Customer
ENDIF
```

Files Library is CLIPPER.LIB.

See Also INDEXKEY(), INDEXORD(), ORDBAGEXT()

INDEXKEY() function

Return the key expression of a specified index

Syntax

```
INDEXKEY(<nOrder>) → cKeyExp
```

Arguments

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

Returns

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

Description

INDEXKEY() is a database function that determines the key expression of a specified index in the current work area and returns it as a character string. To evaluate the key expression, specify INDEXKEY() as a macro expression like this: &(INDEXKEY(<*nOrder*>)).

INDEXKEY() has a number of applications, but two specific instances are important. Using INDEXKEY(), you can TOTAL on the key expression of the controlling index without having to specify the key expression in the source code. The other instance occurs within a DBEDIT() user function. Here, you may want to determine whether or not to update the screen after the user has edited a record. Generally, it is only necessary to update the screen if the key expression of the controlling index has changed for the current record. Both of these examples are illustrated below.

By default, INDEXKEY() operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression (see example below).

Examples

- This example accesses the key expression of open indexes in the current work area:

```
#define ORD_NATURAL     0
#define ORD_NAME        1
#define ORD_SERIAL      2
//
USE Customer INDEX Name, Serial NEW
SET ORDER TO ORD_SERIAL
? INDEXKEY(ORD_NAME)        // Result: Name index exp
? INDEXKEY(ORD_SERIAL)      // Result: Serial index exp
? INDEXKEY(ORD_NATURAL)     // Result: Serial index exp
```

- This example accesses the key expression of the controlling index in an unselected work area:

```
USE Customer INDEX Name, Serial NEW
USE Sales INDEX Salesman NEW
? INDEXKEY(0); Customer->(INDEXKEY(0))
```

- This example uses INDEXKEY() as part of a TOTAL ON key expression. Notice that INDEXKEY() is specified using a macro expression to force evaluation of the expression:

```
USE Sales INDEX Salesman NEW
TOTAL ON &(INDEXKEY(0)) FIELDS SaleAmount TO ;
      SalesSummary
```

- This example uses INDEXKEY() to determine whether the DBEDIT() screen should be updated after the user has edited the current field value. Generally, you must update the DBEDIT() screen if the user changes a field that is part of the controlling index key. FieldEdit() is a user-defined function called from a DBEDIT() user function to edit the current field if the user has pressed an edit key.

```
#include "Dbedit.ch"
#define ORD_NATURAL  0
FUNCTION FieldEdit()
   LOCAL indexVal
   // Save current key expression and value
   indexVal = &(INDEXKEY(ORD_NATURAL))
   .
   . <code to GET current field value>
   .
   // Refresh screen if key value has changed
   IF indexVal != &(INDEXKEY(ORD_NATURAL))
      nRequest = DE_REFRESH
   ELSE
      nRequest = DE_CONT
   ENDIF
   RETURN nRequest
```

Files Library is CLIPPER.LIB.

See Also INDEX, INDEXEXT(), INDEXORD(), SET INDEX, SET ORDER, USE

INDEXORD() function

Return the order position of the controlling index

Syntax

```
INDEXORD() → nOrder
```

Returns

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

Description

INDEXORD() is a database function that determines the position of the controlling index in the list of index files opened by the last USE...INDEX or SET INDEX TO in the current work area. It is often useful to save the last controlling index so it can be restored later.

By default, INDEXORD() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression (see example below).

Examples

- This example uses INDEXORD() to save the current order. After changing to a new order, it uses the saved value to restore the original order:

```
USE Customer INDEX Name, Serial NEW
nOrder := INDEXORD()                 // Result: 1
SET ORDER TO 2
? INDEXORD()                         // Result: 2
SET ORDER TO nOrder
? INDEXORD()                         // Result: 1
```

- This example uses an aliased expression to determine the order number of the controlling index in an unselected work area:

```
USE Sales INDEX Salesman, CustNum NEW
USE Customer INDEX Name, Serial NEW
? Sales->(INDEXORD())                // Result: 1
```

Files

Library is CLIPPER.LIB.

See Also

INDEX, INDEXKEY(), SET INDEX, SET ORDER, USE

INIT PROCEDURE statement

Declare an initialization procedure

Syntax

```
INIT PROCEDURE <idProcedure> [(<idParam list>)]
   [FIELD <idField list> [IN <idAlias>]]
   [LOCAL <identifier> [[:= <initializer>]]]
   [MEMVAR <identifer list>]
   .
   . <executable statements>
   .
   [RETURN]
```

Arguments

INIT PROCEDURE declares a procedure that will be executed at program startup.

<idProcedure> is the name of the initialization procedure to declare. Initialization procedure names can be any length, but only the first 10 characters are significant. Names may not begin with an underscore but can contain any combination of characters, numbers, or underscores.

<idParam list> is the declaration of one or more parameter variables. Variables specified in this list are declared local.

FIELD declares a list of identifiers to use as field names whenever encountered. If the IN clause is specified, referring to the declared name includes an implicit reference to the specified alias.

LOCAL declares and, optionally, initializes a list of variables or arrays whose visibility and lifetime is the current procedure.

MEMVAR declares a list of identifiers to use as private or public memory variables or arrays whenever encountered.

RETURN passes control to the next initialization procedure or the first executable routine in the program, if no other initialization procedures are pending.

Description

The INIT PROCEDURE statement declares a procedure that will be executed at program startup. INIT procedures are called prior to the first executable statement in a CA-Clipper application, and are useful for performing common initialization tasks such as reading configuration settings, or opening a communications port.

INIT PROCEDUREs are executed implicitly by CA-Clipper at program startup. The visibility of initialization procedures is restricted to the system; therefore, it is not possible to call an INIT PROCEDURE from a procedure or user-defined function. Each INIT PROCEDURE receives a copy of the DOS command line arguments used to invoke the application.

Control passes from one INIT PROCEDURE to the next until all procedures in the initialization list have been called. Control then passes to the first executable statement in the program.

The ANNOUNCE statement declares a module identifier for a source (.prg) file. Once declared, INIT PROCEDUREs are referenced by this module identifier. An application may use any number of initialization procedures by explicitly REQUESTing their module identifiers.

The INIT PROCEDUREs requested for an application are collectively referred to as the initialization list. There is no default execution order of procedures in the initialization list; however, the following rules apply:

- The CA-Clipper initialization procedure, CLIPINIT, is always called first
- If an INIT PROCEDURE is declared in the same source (.prg) file as the application's primary (root) routine, it will be the last initialization procedure called

CLIPINIT is called first to establish system integrity by installing the default error recovery system (ErrorSys). Once CLIPINIT has finished executing, control passes to the next INIT PROCEDURE in the initialization list.

If an error is raised during system initialization, the system returns to DOS, and pending initialization procedures are not called.

Examples

- The following example uses both INIT and EXIT PROCEDUREs to save and restore the context of the operating system. You can have your program, "Myfile.prg", REQUEST SaveDos:

```
ANNOUNCE SaveDos

#define DOS_SCREEN    1
#define DOS_ROW       2
#define DOS_COL       3
#define DOS_CURSOR    4
#define DOS_COUNT     4

STATIC saSaveDos[ SD_COUNT ]

INIT PROCEDURE dosSave()
   SAVE SCREEN TO saSaveDos[ DOS_SCREEN ]
   saSaveDos[ DOS_ROW ]    := ROW()
   saSaveDos[ DOS_COL ]    := COL()
   saSaveDos[ DOS_CURSOR ] := SETCURSOR()
   RETURN

EXIT PROCEDURE dosRestore()
   RESTORE SCREEN FROM saSaveDos[ DOS_SCREEN ]
   SETPOS   ( saSaveDos[ DOS_ROW ], saSaveDos[ DOS_COL ] )
   SETCURSOR( saSaveDos[ DOS_CURSOR ] )
   RETURN
```

See Also

ANNOUNCE, REQUEST, EXIT PROCEDURE

INKEY() function

Extract a character from the keyboard buffer or a mouse event

Syntax

```
INKEY([<nSeconds>] [,<nEventMask>]) → nInkeyCode
```

Arguments

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

<nEventMask> specifies which events should be returned. If ***<nEventMask>*** is omitted, the value represented by the SET EVENTMASK command will be used. If there is no SET EVENTMASK command issued, the default value that will be used is 128 (keyboard events only).

This parameter can be any combination of the following values. The constant values listed below are defined in Inkey.ch.

Inkey Constants

Constant	Value	Descripiton
INKEY_MOVE	1	Mouse Events
INKEY_LDOWN	2	Mouse Left Click Down
INKEY_LUP	4	Mouse Left Click Up
INKEY_RDOWN	8	Mouse Right Click Down
INKEY_RUP	16	Mouse Right Click Up
INKEY_KEYBOARD	128	Keyboard Events
INKEY_ALL	159	All Mouse and Keyboard Events

Returns

INKEY() returns an integer value from -39 to 386 for keyboard events and integer values from 1001 to 1007 for mouse events. This value identifies either the key extracted from the keyboard buffer or the mouse event that last occurred. If the keyboard buffer is empty, and no mouse events are taking place, INKEY() returns 0. INKEY() returns values for all ASCII characters, function, Alt+Function, Ctrl+Function, Alt+Letter, and Ctrl+Letter key combinations.

Description

INKEY() is a keyboard function that extracts the next key pending in the keyboard buffer or the next mouse event and returns a value representing the appropriate event. The value is also saved internally and can be accessed using LASTKEY(). If the <*nSeconds*> argument is specified and there are no pending keys in the buffer, program execution pauses until a key appears in the keyboard buffer, or an appropriate mouse event occurs, or <*nSeconds*> has elapsed. The time INKEY() waits is based on the operating system clock and is not related to the microprocessor speed. If <*nSeconds*> is zero, program execution pauses until a key is pressed or an unmasked mouse event occurs. Note that INKEY() is not a wait state and, therefore, SET KEYs are not active.

INKEY() is similar to the NEXTKEY() function. Unlike INKEY(), however, NEXTKEY() reads, but does not extract the key or mouse event. This is useful when you need to test for a key or mouse event without processing it.

INKEY() is the basic primitive of the CA-Clipper system for fetching keys and mouse events. It is used for polling the keyboard, polling the mouse, or pausing program execution. As an example, you can use INKEY() to terminate commands with a record scope such as LIST, LABEL FORM, and REPORT FORM by including it in a WHILE condition.

Examples

- The following example will inform INKEY() to terminate if the left mouse button has been clicked or the right mouse button has been clicked. If no events of this type occur within 10 seconds, INKEY() will terminate.

```
? INKEY( 10 , INKEY_LDOWN + INKEY_RDOWN )
```

*INPUT command

Enter the result of an expression into a variable

Syntax

```
INPUT [<expPrompt>] TO <idVar>
```

Arguments

<expPrompt> is an optional prompt displayed before the input area. The prompt can be an expression of any data type.

TO <idVar> specifies the name of the variable to be assigned the evaluation result. If *<idVar>* is not visible or does not exist, a private variable is created and assigned the result.

Description

INPUT is a console command and wait state that takes entry from the keyboard in the form of a valid expression of up to 255 characters and any data type. When invoked, INPUT sends a carriage return/linefeed to the screen, displays the prompt, and begins accepting keyboard input at the first character position following the prompt. Input greater than MAXCOL() wraps to the next line.

Pressing Return terminates entry of the expression. The expression is then compiled and evaluated using the macro operator (&), and the result assigned to *<idVar>*. If the variable is not visible or does not exist, it is created as a private. If no expression is entered, no action is taken.

INPUT supports only two special keys: Backspace and Return. Esc is not supported. Backspace deletes the last character typed. Return confirms entry and is the only key that can terminate an INPUT.

Examples

- In this example INPUT assigns a keyboard entry to an existing local variable:

```
LOCAL exp
INPUT "Expression: " TO exp
IF exp != NIL
    ? exp
ELSE
    ? "No expression entered"
ENDIF
```

Files

Library is CLIPPER.LIB.

See Also

ACCEPT

INT() function

Convert a numeric value to an integer

Syntax

```
INT(<nExp>) → nInteger
```

Arguments

<nExp> is a numeric expression to be converted to an integer.

Returns

INT() returns an integer numeric value.

Description

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

Examples

- These examples demonstrate the results of various invocations of the INT() function:

```
? INT(100.00)                // Result: 100
? INT(.5)                    // Result: 0
? INT(-100.75)               // Result: -100
```

Files

Library is CLIPPER.LIB.

See Also

ROUND()

ISALPHA() function

Determine if the leftmost character in a string is alphabetic

Syntax

```
ISALPHA(<cString>) → lBoolean
```

Arguments

<cString> is the character string to be examined.

Returns

ISALPHA() returns true (.T.) if the first character in *<cString>* is alphabetic; otherwise, it returns false (.F.).

Description

ISALPHA() is a character function that determines if the specified string begins with an alphabetic character. An alphabetic character consists of any uppercase or lowercase letter from A to Z. ISALPHA() returns false (.F.) if the string begins with a digit or any other character.

Examples

- These examples demonstrate various results of ISALPHA():

```
? ISALPHA("AbcDe")                // Result: .T.
? ISALPHA("aBcDE")                // Result: .T.
? ISALPHA("1BCde")                // Result: .F.
? ISALPHA(".FRED")                // Result: .F.
```

Files Library is EXTEND.LIB.

See Also ISDIGIT(), ISLOWER(), ISUPPER(), LOWER(), UPPER()

ISCOLOR() function

Determine if the current computer has color capability

Syntax

```
ISCOLOR() | ISCOLOUR() → lBoolean
```

Returns

ISCOLOR() returns true (.T.) if there is a color graphics card installed; otherwise, it returns false (.F.).

Description

ISCOLOR() is a screen function that allows you to make decisions about the type of screen attributes to assign (color or monochrome). Note that some monochrome adapters with graphics capability return true (.T.).

Examples

- This example installs color attribute variables at runtime:

```
IF ISCOLOR()
   cBox  = "BG+/B, W/N"
   cSays = "BG/B, W/N"
   cGets = "W/N, N/W"
ELSE
   cBox  = "W+"
   cSays = "W/N, N+/W"
   cGets = "W/N, N/W"
ENDIF
.
. <statements>
.
SETCOLOR(cSays)
```

Files

Library is CLIPPER.LIB.

See Also

SETCOLOR()

ISDIGIT() function

Determine if the leftmost character in a character string is a digit

Syntax

```
ISDIGIT(<cString>) → lBoolean
```

Arguments

<cString> is the character string to be examined.

Returns

ISDIGIT() returns true (.T.) if the first character of the character string is a digit between zero and nine; otherwise, it returns false (.F.).

Description

ISDIGIT() is a character function that determines whether the first character in a string is a numeric digit between zero and nine. If any other character is the first character of the *<cString>*, ISDIGIT() returns false (.F.).

ISDIGIT() is useful where you need to know if the current character string is a number before converting it to a numeric value with the VAL() function.

Examples

- These examples demonstrate various results of ISDIGIT():

```
? ISDIGIT("AbcDe")          // Result: .F.
? ISDIGIT("1abcd")          // Result: .T.
? ISDIGIT(".12345")         // Result: .F.
```

Files Library is EXTEND.LIB.

See Also ISALPHA(), ISLOWER(), ISUPPER(), LOWER(), UPPER()

ISDISK() function

Check if a disk drive is available

Syntax

```
ISDISK(<cDrive>) → lAvailable
```

Arguments

<cDrive> is the letter of the disk drive to be checked.

Returns

ISDISK() returns true (.T.) if a disk drive is available; otherwise, it returns false (.F.).

Examples

- This example builds a string that contains all currently available drives on your system:

```
FUNCTION AllDrives()
   LOCAL wI, cDrives := ""
   FOR wI := 1 TO 26
      IF ISDISK( Chr( wI + 64 ) )
         cDrives := cDrives + Chr( wI+64 )
      ENDIF
   NEXT
RETURN cDrives
```

Files

Library is EXTEND.LIB.

See Also

DIRCHANGE(), DISKNAME(), CURDIR()

ISLOWER() function

Determine if the leftmost character is a lowercase letter

Syntax

```
ISLOWER(<cString>) → lBoolean
```

Arguments

<cString> is the character string to be examined.

Returns

ISLOWER() returns true (.T.) if the first character of the character string is a lowercase letter; otherwise, it returns false (.F.).

Description

ISLOWER() is a character function that determines whether the first character of a character string is lowercase. It is the inverse of ISUPPER() which determines whether a character begins with an uppercase character.

Both ISLOWER() and ISUPPER() relate to the LOWER() and UPPER() functions which actually convert lowercase characters to uppercase, and vice versa.

Examples

- These examples demonstrate various results of ISLOWER():

```
? ISLOWER("aBcDe")          // Result: .T.
? ISLOWER("AbcDe")          // Result: .F.
? ISLOWER("1abcd")          // Result: .F.
? ISLOWER("abcd")           // Result: .T.
```

Files

Library is CLIPPER.LIB.

See Also

ISALPHA(), ISDIGIT(), ISUPPER(), LOWER(), UPPER()

ISPRINTER() function

Determine whether LPT1 is ready

Syntax

```
ISPRINTER() → lReady
```

Returns

ISPRINTER() returns true (.T.) if LPT1 is ready; otherwise, it returns false (.F.).

Description

ISPRINTER() is a printer function that determines whether the parallel port (LPT1) is online and ready to print. ISPRINTER() is hardware-dependent and, therefore, only works on IBM BIOS compatible systems.

You can check ISPRINTER() to make sure the printer is ready before you begin a print operation; however, if an error occurs during the print operation, a runtime error is generated.

Examples

- This example tests the parallel port for readiness with up to 25 retries. If the parallel port is ready, the printer operation begins:

```
LOCAL nCount := 0, nTimes := 25, lReady
//
DO WHILE nCount++ <= nTimes .AND. !(lReady := ;
      ISPRINTER())
ENDDO
//
IF lReady
   REPORT FORM Sales TO PRINTER
ELSE
   ? "Printer not ready..."
   BREAK
ENDIF
```

Files

Library is EXTEND.LIB, source file is SOURCE\SAMPLE\EXAMPLEA.ASM.

See Also

SET DEVICE, SET PRINTER

ISUPPER() function

Determine if the leftmost character in a string is uppercase

Syntax

```
ISUPPER(<cString>) → lBoolean
```

Arguments

<cString> is the character string to be examined.

Returns

ISUPPER() returns true (.T.) if the first character is an uppercase letter; otherwise, it returns false (.F.).

Description

ISUPPER() is a character function that determines whether the first character of a string is uppercase. It is the inverse of ISLOWER(). Both ISUPPER() and ISLOWER() relate to the UPPER() and LOWER() functions which actually convert uppercase characters to lowercase, and vice versa.

Examples

- These examples illustrate ISUPPER() applied to various values:

```
? ISUPPER("AbCdE")          // Result: .T.
? ISUPPER("aBCdE")          // Result: .F.
? ISUPPER("$abcd")          // Result: .F.
? ISUPPER("8ABCD")          // Result: .F.
```

Files

Library is CLIPPER.LIB.

See Also

ISALPHA(), ISDIGIT(), ISLOWER(), LOWER(), UPPER()

Glossary

Glossary

Abbreviation

(preprocessor) A source token whose leftmost characters exactly match the leftmost characters of a keyword in a translation directive. Abbreviations must be at least four characters in length.

Activation

(procedure and function) The invocation and execution process for procedures. Each call to a procedure is referred to as an *activation* of that procedure. If the procedure in turn calls another procedure (or calls itself recursively), a *new activation* is said to have occurred. The earlier activation is then referred to as a *pending activation.*

Pending activations are often referred to as *higher level activations* or *higher level procedures*. When one procedure calls another (i.e., creates a new activation), the latter is often referred to as a *lower level activation* or a *lower level procedure.* (*See also:* Activation Record, Activation Stack, Function, Procedure)

Activation Record

(procedure and function) An internal data structure that contains information pertaining to an activation. (*See also:* Activation, Activation Stack)

Activation Stack

(procedure and function) An internal data structure that contains an activation record for the current activation and all pending activations. (*See also:* Activation, Activation Record, Stack)

Active Window

(debugger) The window to which all keystrokes (except those valid in the Command Window) apply. An active window is indicated by a highlighted border. The Tab and Shift+Tab keys are used to select the next and previous window, respectively.

Algorithm

(algorithm) A set of rules and/or a finite series of steps that will accomplish a particular task. (*See also:* Sequence, Selection, Iteration)

Alias

(database) The name of a work area; an alternate name given to a database file. Aliases are often used to give database files descriptive names and are assigned when the database file is opened. If no alias is specified when the database file is USEd, the name of the database file becomes the alias. (*See also:* Work Area)

An alias can be used to reference both fields and expressions (including user-defined functions). In order to reference an expression using an alias, the expression must be enclosed in parentheses.

Animate Mode
(debugger) The mode of execution in which an application runs one line at a time until a breakpoint or tracepoint is reached, with the execution bar moving to each line as it is executed.

Application
(configuration) A program designed to execute a set of interrelated tasks. Typically referring to a system designed to address a particular business purpose (e.g., Order Entry/Inventory/Invoicing, a document tracking database, or an insurance claims calculator).

Application Buttons
(Workbench) The buttons in the Application Browser pictorially represent each application in the repository. They visually convey the following information: name of the application and compilation status. (*See also:* Compilation Status)

Application Programming Interface (API)
(general) A set of functions that allow direct communication between extend functions and the CA-Clipper Virtual Memory Manager (VMM). It permits safe access to large amounts of memory.

The API enables programmers to interact with and control the features built into CA-Clipper 5.3 and to add specific functionality to C and Assembly language modules that call CA-Clipper routines.

Argument
(variable) Generally, a value or variable supplied in a function or procedure call, or an operand supplied to an operator. In function and procedure calls, arguments are often referred to as *actual parameters*. (*See also:* Parameter)

Array
(array) A data structure that contains an ordered series of values called *elements*. The elements of an array are referred to by ordinal number: the first element is number 1, the second is number 2, etc. A numeric expression used to specify an element of an array is referred to as a *subscript* or *index*. In CA-Clipper, the elements of an array may be values of any type, including references to other arrays. (*See also:* Array Reference, Nested Array, Subarray, Subscript)

Array Functions
(array) Those functions that specifically perform their tasks on arrays. (*See also:* Array, Function, Element, Subscript)

Array Iterator
(array) A function that traverses an array, performing an operation on each element. (*See also:* Array)

Array Reference
(array) A special data value that allows access to an array. In CA-Clipper, program variables and array elements cannot directly contain arrays; they may, however, contain array references. A variable that contains a reference to a particular array is said to *refer to* that array, and the array's elements may be accessed by applying a subscript to the variable.

If the value of a variable containing an array reference is assigned to a second variable, the second variable will contain a copy of the array reference; both variables then refer to the same array, and the array's elements may be accessed by applying a subscript to either variable. (*See also:* Array, Multi-dimensional Array, Nested Array, Subarray, Subscript)

ASCII

(general) An acronym for the American Standard Code for Information Interchange, the agreed upon standard for representing characters (alphabetic, symbolic, etc.) in the memory of the computer.

Assignment

(expression) The act of copying a new value into a variable. In CA-Clipper this is done with the simple assignment operators (=) and (:=), or the compound operators (+=, -=, *=, **=).

Attribute

(database) As a formal DBMS term, refers to a column or field in a table or database file. (*See also:* Column, Field)

Auto Layout

(Workbench) CA-Clipper's Auto Layout feature allows you to (*See also:* Form Editor, Menu Editor):

- Add predefined controls to a data window, based upon the fields in the associated data server using the Auto Layout toolbar button from within the Window Editor
- Add one or more standard, predefined menus to the current menu tree using the Auto Layout toolbar button from within the Menu Editor

Background Color

(user-interface) The color that appears behind displayed text of another color (the foreground color). Though background color and foreground color are usually different, you may render text invisible by choosing the same color for both. (*See also:* Foreground color)

Beginning of File

(database) The top of the database file. In CA-Clipper there is no beginning of file area or record. Instead, it is indicated by BOF() returning true (.T.) if an attempt is made to move the record pointer above the first record in the database file or the database file is empty.

Binary File

(file) A file that contains an unformatted sequence of bytes. Carriage-return, line-feed, or end of file characters have no special meaning in a binary file. Binary files include executable files, graphics files, or data files. (*See also:* Text File)

Binary Notation

(general) A coding system that represents values with the digits 1 or 0. The numerical base upon which computers are modeled. Also referred to as base 2, it is a mathematical representation of the state of a logic element. A 0 represents an *off* state, and a 1 indicates an *on* state. A sequence of eight binary digits, eight on's and off's, express one byte of program or data. (*See also:* Bit, Byte)

Binary Operator

(expression) An operator that operates on two operands. For example, the addition operator. (*See also:* Operator)

Bit

(general) One binary digit, the smallest element of code. Eight consecutive bits form one byte.

Blockify

(preprocessor) To change an expression in the source text into a code block definition. Blockifying is accomplished by surrounding the source text with braces and placing an empty block parameter list (a pair of vertical bars) just inside the braces. When the resulting code block is evaluated at runtime, the original expression will be evaluated.

Blockify Result Marker

(preprocessor) A result marker of the form <*{id}*>. *id* must correspond to the name of a match marker. A blockify result marker specifies that the corresponding source text is to be blockified. If the matched source text is a list of expressions, each expression in the list is individually blockified. If no source text matched the corresponding match marker, an empty result is produced.

Branching

(language) Changing the sequence of execution in a program. Execution normally proceeds in sequence from the top of a function or procedure to the bottom. When control is transferred to a statement that is not in sequence, execution is said to have *branched.*

Breakpoint

(debugger) A point at which an application pauses execution and returns control to the debugger.

Browser

(Workbench) A *browser* is a tool in the CA-Clipper Workbench that provides a convenient and organized way to view the data that is currently stored in your repository.

In CA-Clipper, you can browse:

- Applications
- Modules
- Entities
- Errors

Browse/Form View

(Workbench) Use the Browse/Form View toolbar button to toggle between browse view and form view. *Browse view* displays multiple records for each field in a data window, always maintaining one record as the "current" row, reflecting the data server's current position. *Form view,* on the other hand, displays only one record—the current one—at any given time.

Buffer

(general) A temporary data storage location in memory. As an example, a *disk input-output buffer* is an area of memory that stores data read from the disk to temporary locations while processing it.

Byte

(general) Eight bits of data, the smallest unit of information stored in the computer's memory. As an example, one byte is required to represent one ASCII character.

Calling Convention

(general) The Microsoft Mixed-Language Programming Guide defines calling convention as "the way a language implements a call."

Calling Program

(procedure and function) The procedure or user-defined function that transferred control to the currently executing procedure or function. When the current procedure or user-defined function terminates with a RETURN statement, the program regains control.

Call Stack

(debugger) A list containing the names of all pending activations at the current point in an application.

Callstack Window

(debugger) The window in which the call stack is displayed.

Cell

(database) In a table, a cell is the intersection of a Row and a Column. (*See also:* Column, Row, Table)

Character

(general) A letter, digit, punctuation mark, or special symbol stored and processed by the computer. (*See also:* ASCII)

(data type) A special data type consisting of one or more values in the IBM extended character set. Characters can be grouped together to form strings. The maximum size of a character string in CA-Clipper is 65,534 bytes. (*See also:* String)

Character Functions

(expression) Those functions that act upon individual characters or strings of ASCII characters in the performance of their tasks. (*See also:* ASCII, Function, String)

Child

(Workbench) A *child* is a submenu item within a menu structure. Use the Demote Item toolbar button to demote a menu item to the child level in a menu's hierarchy. (*See also:* Sibling)

Class

(object-oriented) A class defines the variables contained in an object and the operations applied when the object receives a message. Every object is an instance of a class and responds to messages of that class. Each object, however, has its own copy of the variables specified in the class definition. New objects are created in CA-Clipper by calling a special function that begins with the class name followed by the *New* suffix. (*See also:* Object, Instance Variable, Message, Method)

Clause

(command) An optional or required section of a CA-Clipper command beginning with a keyword that modifies or enhances the command.

Code Block

(data type) A special data type that refers to a piece of compiled program code. In a program, the source code that specifies the creation of a code block.

Code Window

(debugger) The window in which source code is displayed.

Collapse/Expand Data
(Workbench) To *collapse* an item when viewing information (for example, in the Source Code Editor) means to compress it, "hiding" any additional information that is subordinate to that item. At any time you can *expand* a collapsed item (thereby redisplaying its subordinate data).

In CA-Clipper, the following types of items can be collapsed and expanded:

- Branches in tree browsers (like Entity Browsers)
- Entities in the Source Code Editor
- Menus in the Menu Editor

Collision
(network) An attempt by more than one user (typically on a networked, multi-user system) to update a database simultaneously, usually resulting in data corruption. Generally due to poor implementation of the code to support multi-user operations. (*See also:* LAN, Local Area Network, Deadly Embrace)

Column
(database) A database term used to describe a field in a table or database file. (*See also:* Field)

(user-interface) A numeric value that represents a position on the display screen or on the printed page.

Command
(command) A statement to be translated by the CA-Clipper preprocessor into source code that will perform a particular operation. All CA-Clipper commands are defined in the standard header file, Std.ch, located in \CLIP53\INCLUDE. Also, the preprocessor directives that define a command. (*See also:* Statement, Header files, Std.ch)

Command Window
(debugger) The window in which commands are displayed and entered.

Comment
(language) Text in a source program that is ignored by the compiler. Usually used to make descriptive comments about the surrounding source code.

Compilation Status
(Workbench) The *compilation status* of items in the repository is denoted in the various browsers with either LED or symbol indicators:

- *Green* (or a check mark) indicates that the application (or module) has been compiled successfully, contains an executable file, and is ready to run provided that the .EXE has not been deleted from DOS
- *Red* (or X) indicates compilation errors or warnings
- *Blue* (or !) indicates that compilation has been successful, but there are compiler warnings
- *Yellow* (or ?) indicates that the application (or module) is in an indeterminate state: either it is new and needs to be compiled, or has been modified and needs to be recompiled

Compiler
(program) A program that translates source code output from the preprocessor into object code. The resulting object file can then be linked to produce an executable program using a linker. (*See also:* Linker, Object File, Program File)

Concatenate
(expression) To combine two groups of character data together by placing them in a sequence to form a new string of characters. (*See also:* Data Type)

Concurrency
(database) The degree to which data can be accessed by more than one user at the same time.

Condition
(command, database, expression) A logical expression that determines whether an operation will take place. With database commands, a logical expression that determines what records are included in an operation. Conditions are specified as arguments of the FOR or WHILE clause. (*See also:* Scope)

Conditional Compilation
(preprocessor) Selective exclusion by the preprocessor of certain source code. The affected source code is bracketed by the #ifdef and #endif directives. It is excluded if the argument in the #ifdef directive is not a #defined identifier.

Console Input/Output
(user-interface) Operation of the keyboard and display that emulates a simple typewriter-like interface. Console input echoes each key typed and provides processing for the Backspace and Return keys. Console output wraps to the next line when the output reaches the right edge of the visible display, and scrolls the display when the output reaches the bottom of the visible display. (*See also:* Full-screen Input/Output)

Constant
(variable) The representation of an actual value. For example, .T. is a logical constant, *string* is a character constant, 21 is a numeric constant. There are no date and memo constants.

Constant Array
(array) *See* Literal Array.

Control Structure
(language) Any program structure that alters the flow of program control. In CA-Clipper, these include:

- BEGIN SEQUENCE...END
- DO WHILE...ENDDO
- DO CASE...ENDCASE
- FOR...NEXT
- IF...ENDIF

Controlling Order
(database) The active order (index) for a particular work area. Only one order may control a work area at any time, and it controls the order in which the database is accessed during paging and searching.

Controlling/Master Index
(database) The index currently being used to refer to records by key value or sequential record movement commands. (*See also:* Index, Natural Order)

Conversion Functions
(expression) Generally referring to a category of functions whose purpose is to change one data type to another (e.g., to change a number or a date to a character string).

Cursor
(user-interface) An on-screen indicator used to show the current keyboard input focus, displayed as a block or underline character. The cursor moves in response to characters or control keys typed by the user. (*See also:* Highlight, Input Focus)

(debugger) The cursor indicates the current line and/or column position in the active window or dialog box. Note that some windows, such as the Monitor Window, do not utilize the cursor. When a window that does not utilize the cursor is active, the cursor appears in the Code Window.

Data Form
(user-interface) A data form is a form or data entry screen associated with a data server. It is *data-aware*—it "knows" about the data with which it is intended to operate via properties you specify for each control in the form. (*See also:* Auto Layout, Form Editor)

Data Independence
(modular programming) The technique of writing a program, function, or procedure in such a way that it will be able to perform its operation on data supplied to it without a *built-in* description of the data format.

Data Server
(Workbench) A data server is a high-level, abstract entity designed as a consistent GUI interface for Xbase database files. The data server acts as a database describer, defining its file name, its fields (or columns), the order in which it is accessed, etc. It provides a mechanism for extending field definitions beyond the basic name, type, and length information stored in the database file structure—this last part is accomplished using a *field spec*. (*See also:* Field Specification (Field Spec))

Data Store
(file, general) An area of memory or disk in which data is *store*d and from which it is retrieved while being processed.

Data Type
(data type) The category of a data value. A data type is distinguished by the set of allowable values for that type, the set of operators that can be applied, and the storage format used to represent these values. In CA-Clipper, the following data types are defined: character, numeric, date, logical, array, object, code block, and NIL. Program variables may contain values of any type. Database field variables are limited to character, numeric, date, logical, and a special type called memo which is treated the same as character.

Database
(database) An aggregation of related operational data used by an application system. A database can contain one or more data files or tables. (*See also:* Field, Record, Tuple, View)

Date Functions
(expression) Functions that operate on date values (as opposed to character, numeric or other values). (*See also:* Function)

Date Type
(data type) A special data type consisting of digits to store year, month, and day values. Operations on date values are based on chronological values.

DB Server Editor
(Workbench) Use the Source Code Editor to create a data server, import the structure of an existing database file, and specify its properties and fields. (*See also:* Data Server, FieldSpec Editor, Visual Editor)

DBMS
(database) An acronym for the term *database management system*. A DBMS is a software system that mediates access to a database through a data manipulation language.

DDL
(general) Data Description Language. A set of symbols and words, and the rules governing their use as meaningful descriptions of the data components and their relationships within a database. Besides describing the elements and their relationships, the DDL also specifies the sorting and search fields.

Deadly Embrace
(network) In a multi-user database management system, the result of an unforeseen flaw that has allowed a situation wherein two or more simultaneous operations cannot be completed because each has instigated locks on files or records that the other needs to complete its task.

Debugger
(debugger) A tool used to track down errors in a program.

Debugging
(error handling) A phase of software development where errors are identified and fixed.

Declaration
(variable) A statement used by the compiler to define a variable, procedure, or function identifier. The scope of the declaration is determined by the position of the declaration statement in the source file. (*See also:* Identifier, Scope)

Decrement
(expression) To decrease a value by a fixed amount, usually one. In CA-Clipper, the decrement operator (--) can be used to decrement a numeric value in a variable. (*See also:* Assignment, Increment)

Define
(preprocessor) To define an identifier to the preprocessor, using the #define directive, and optionally specify text to be substituted for occurrences of the identifier.

Delimited File

(file, database) A text file that contains variable-length database records with each record separated by a carriage-return/line feed pair (CHR(13) + CHR(10)) and terminated with an end of file mark (CHR(26)). Each field within a delimited file is variable length, not padded with either leading or trailing spaces, and separated by a comma. Character strings are optionally delimited to allow for embedded commas. (*See also:* Database File, Text File, SDF File)

Delimiter

(general) A character or other symbol that marks a boundary.

Desktop

(user-interface) The background area of the screen on which all windows, icons, and dialog boxes appear.

Destination

(expression) The variable or array element to receive data in an assignment. (*See also:* Assignment)

(general) The work area, file, or device to which data is sent.

Device

(general) Either an actual physical component of the computer system such as a printer or a DOS *handle* that refers to it (e.g., PRN:), or a *logical device* that behaves and is addressed the same way as a *physical device* (e.g., a print spooler).

Dialog box

(debugger) A box displayed from within the debugger whenever further input is required.

Dimension

(array) The maximum number of subscripts required to specify an array element. For example, a two-dimensional array must have two subscripts, a three-dimensional array must have three subscripts, and so on. (*See also:* Subscript)

Directive

(preprocessor) An instruction to the preprocessor. Preprocessor directives must appear on a separate line, and must begin with a hash mark (#). Their scope or effect extends from the point where they are encountered to the end of the source file in which they appear.

Directory

(file) The major operating system facility for cataloging files. A directory contains a list of files and references to child directories (subdirectories), and is identified by name. Directories can be nested forming a hierarchical tree structure. The operating system provides a number of facilities that allow users to create and delete directories. (*See also:* Disk, File, Path, Volume)

Disk

A magnetic storage medium designed for long-term storage. Disks come in two varieties: hard disks (fast but fixed) or floppy disks (slow but removable). A disk can be partitioned into multiple volumes, each containing a tree-structured directory system that holds files accessible by programs. (*See also:* Volume, Directory, File)

DLL

(general) Abbreviation for a Microsoft Windows dynamic link library.

DML
(database) Data Manipulation Language. The set of commands and functions that control change and movement operations like input/output, reporting and sorting on data elements in a database.

Drive
(file) A disk drive or a letter (normally followed by a colon) that designates a disk drive. On most computers, the letters A and B refer to floppy disk drives; other letters refer to fixed disk drives or *logical* drives (e.g., fixed disk partitions or network drives).

Dumb Stringify Result Marker
(preprocessor) A result marker of the form #<{*id*}>. *id* must correspond to the name of a match marker. A dumb stringify result marker specifies that the corresponding source text is to be enclosed in quotes. If the matched source text constitutes a list of expressions, each expression in the list is individually stringified. If no source text was matched, an empty pair of quotes is produced.

Dynamic
(general) Used generically to refer to data or algorithms that change with time. Often used specifically to describe algorithms that automatically adjust to prevailing conditions.

Dynamic Overlay
(linker) A portion (memory page) of program code that can be moved into and out of memory on a least recently used basis. Dynamic overlays are allocated in 1K increments and the movement of code is managed by the overlay manager at runtime.

Dynamic Scoping
(variable) A method of determining an item's existence or visibility based on the state of a program during execution. Example: A CA-Clipper public variable may or may not be visible within a particular function, depending on whether the variable has been created and whether a previously called function has obscured it by creating a private variable with the same name. (*See also:* Lexical Scoping, Scope of a Variable)

Element
(array) A component unit of an array, usually referred to by a numeric subscript or index. (*See also:* Array, Subscript)

EMM
(memory) Expanded Memory Manager. A driver that provides a hardware-independent interface between application software and the expanded memory hardware.

Empty Result
(preprocessor) An absence of result text; the effect of certain result markers when the corresponding match marker did not match any source text (but when the translation directive as a whole was matched). An empty result simply implies that no result text is written to output.

EMS Memory
(memory) Expanded Memory Specification, or formally, the Lotus/Intel/Microsoft Expanded Memory Specification. EMS Memory is a functional definition of a bank-switched memory-expansion subsystem. This system consists of hardware expansion modules and a resident driver program.

Encapsulation
(modular programming) Generically, the design of a function or program that obeys the principle of information hiding. A function or program is said to be encapsulated if other programs or functions have no knowledge of its inner workings. A data structure is said to be encapsulated if knowledge of its internal organization is limited to a single function or module. (*See also:* Information Hiding, Lexical Scoping, Modularity, Side Effect)

End of File
(database) The bottom of a database file. In CA-Clipper, this is LASTREC() + 1 and is indicated by EOF() returning true (.T.).

Enhanced Color
(user-interface) The color used to display GETs or PROMPTs (if INTENSITY is ON). (*See also:* Standard Color)

Entity
(Workbench) In CA-Clipper, *applications* consist of *modules*, which consist of *entities*.

An *entity* is a component that has a distinct name and can be edited. Some examples are: functions, procedures, globals, constants, forms, and menus. (*See also:* Module)

Environment Variables
(configuration) Operating system variables that can be used to communicate configuration information to executable programs. Environment variables are manipulated using the DOS SET command. The CA-Clipper compiler and linker respond to certain environment variables. CA-Clipper programs can inspect the settings of environment variables using the GETENV() function.

Error
(error handling) The presence of some element of an operation that does not satisfy the requirements of the operation. An error causes failure when encountered, which in turn raises an exception. (*See also:* Exception, Failure, Runtime Error)

Error Handling
(error handling) The concept of including code in a program so that exceptions to normal operational states that occur during the program execution can be anticipated and dispatched with the least possible detrimental consequences to the use of the program and the data being worked on. (*See also:* Exceptions, Runtime Errors)

Evaluate
(expression) To execute part of a program in order to produce a value. For an expression, to execute the program code associated with the expression and return the resulting value. For the macro operator, to compile the macro string, execute the resulting program code, and return the resulting value. (*See also:* Expression)

Exception
(error handling) An occurrence of an abnormal condition during the execution of an operation. An exception is said to be raised when an operation fails. (*See also:* Error, Failure, Runtime Error)

Exclusive
(network) In a network, to assure that no other user will write data to a file, it may be opened in an Exclusive mode. Only the user opening the file exclusively may then access it until exclusive use of the file (by closing it, or opening it SHARED) is relinquished.

Executable File
(configuration) A file output from the linker directly executable from the operating system command line. Executable files have an .EXE extension. (*See also:* Linker)

Execution Bar
(debugger) The highlight bar which is positioned on the line of code to be executed next.

Exported Instance Variables
(object-oriented) *See* Instance Variables.

Expression
(expression) A combination of constants, identifiers, operators, and functions that yield a single value when evaluated.

Extend Routine
(extend) A routine written in a language other than CA-Clipper that can be called from CA-Clipper with or without return values.

Extend System API
(extend) A set of C and Assembly-callable routines that provide interface and services to extend routines.

Extended Memory
(general) Extended Memory is the IBM designation for physical memory above one megabyte that can be accessed by an 80286 or greater CPU in protected mode. (*See also:* Protected Mode)

Extension
(file) A portion of a file name normally used for identifying the type or originating program of a file. (*See also:* Drive, File Name, Path)

Failure
(error handling) The inability of an operation to satisfy its purpose. When a failure occurs an exception is raised. Failures are due in large part to errors. (*See also:* Exception, Error, Retry, Runtime Error)

Far Pointer
(memory addressing) A PC memory address consisting of both a 64K segment selector and an offset value which expands to a 20-bit address.

Field
(database) The basic column unit of a database file. A field has four attributes: name, type, length, and decimals if the type is numeric. (*See also:* Database, Record, Tuple, Vector, View)

Field Specification (Field Spec)
(Workbench) A *field specification* (or *field spec*) is essentially a template, that is, a set of properties (such as picture clauses, validation and formatting rules, etc.) that are related to a field but are *independent* of any particular data server. One of its advantages is that any changes made to a field spec are automatically propagated to all data servers that use that field spec. (*See also:* Data Server)

Field Variable

(variable, database) A variable that refers to data in a database field, as opposed to data in memory. (*See also:* Local Variable, Memory Variable, Private Variable, Public Variable, Static Variable, Variable)

FieldSpec Editor

(Workbench) Use the FieldSpec Editor to create and modify field specs *independent of* any field or data server with which they may be associated. (See also: Data Server, Field Specification, Visual Editor)

File

(file) A file is an organized collection of bytes stored on disk, maintained by the operating system, and referenced by name. Its internal structure is solely determined by its creator. (See also: Binary File, Database File, Text File)

File Handle

(file) An integer numeric value returned from FOPEN() or FCREATE() when a file is opened or created. This value is used to identify the file for other operations until it is closed.

File Locking

(network) The process by which a user is guaranteed exclusive access to a database file. The file is only available to the user that applied the lock. (*See also:* Record Locking)

File Name

(file) The name of a disk file that may optionally include a drive designator, path, and extension. (*See also:* Drive, Extension, Path)

File Server

(network) A computer on a network dedicated to providing data storage to other computers (i.e., workstations) for the purpose of sharing information among multiple users. File servers tend to provide other services such as E-Mail service and shared printer support as well.

File-wide Declaration

(variable) A variable declaration statement that has the scope of the entire source file. File-wide declarations are specified before the first procedure or function declaration in a program file, and the program file must be compiled with the /N option. (*See also:* Scope, Storage Class)

Foreground Color

(user-interface) The color of text appearing on the screen, usually on a different colored background. Though foreground color and background color are usually different, you may render text invisible by choosing the same color for both. (*See also:* Background Color)

Form Editor

(Workbench) Use the Form Editor to create forms, such as dialog boxes and data forms; define their properties and controls; and modify existing forms using standard editing techniques. (See also: Data Form, Visual Editor)

Form Feed

(general) A special character (CHR(12)) that by convention causes most printers to move the printhead to the top of the next page. (*See also:* Hard Carriage Return, Line Feed)

Full-Screen Input/Output
(user-interface) A style of operation of the keyboard and display used for complex data entry and display tasks. Full-screen input and output are generally performed using the @..SAY, @..GET and READ commands. Full-screen output is distinguished from console-style output by the fact that control characters (e.g., backspace, carriage-return) are not processed, and wrapping and scrolling do not occur at the boundaries of the visible display area. (*See also:* Console Input/Output)

Function
(procedure and function) An executable block of code with an assigned name. Alternately, the collection of source code statements that define a function. Certain functions are supplied as part of CA-Clipper; others are defined by the programmer using the FUNCTION or PROCEDURE declaration statements. The latter are referred to as *user-defined* functions.

The terms *procedure* and *function* are generally interchangeable. By convention, a function returns a value, while a procedure does not. (*See also:* Activation, Parameter, Procedure)

Group
(linker) An Intel 8086 addressing classification defining a collection of segments to be addressed using the same segment register. Note that a group is not a section but rather a logical concept used only for addressing.

GUI
(user-interface) Abbreviation for Graphical User Interface. Used generically by CA-Clipper to refer to graphical user interfaces like Windows.

Hard Carriage Return
(general) An explicit carriage-return character at the end of a line in a text file, as opposed to a soft carriage return that might be inserted into text by a program designed to handle word-wrapping. A hard carriage-return character is generated by the expression (CHR(13)) where a soft carriage-return character is generated with the expression (CHR(141)).

Header File
(configuration, preprocessor) A source file containing manifest constant definitions; command or pseudofunctions; and/or program statements merged into another source file using the #include preprocessor directive. (*See also:* Program File, Source File, Std.ch)

Help Window
(debugger) The window in which on-line help is displayed.

Hexadecimal
(general) A representation of a value in base-16 rather than decimal, which is base-10. Hexadecimal values are easily converted to and from binary (base-2), which is the form of data the computer actually uses. Hexadecimal values are represented by digits zero through nine and A through F for values between 10 and 15.

High-Level Wrapper Functions
(database) A CA-Clipper-callable function that consists of calls to lower-level internals. Specifically, in the RDD subsystem, a high-level wrapper function contains calls to a work area's RDD Virtual Jump Table.

Hidden
(modular programming) The resulting state of a program module written to conform to the principals of information hiding. (*See also:* Information Hiding)

Highlight
(user-interface) Indicates input focus for menus, browsers, or GETs. With menus and browsers, the currently selected item or cell has input focus and is displayed in the current enhanced color or inverse video. With GETs, the current GET is highlighted in the current enhanced color or inverse video while the other GETs are displayed in the current standard color if an unselected color setting is active. (*See also:* Cell, Standard Color, Enhanced Color, Input focus, Unselected Color)

IBM Extended Character Set
(general) The character set built into the ROM of the IBM-PC. This character set is a superset of ASCII, containing additional special characters (such as a line drawing character set) that may be used to enhance your program screens.

IDE
(user-interface) Abbreviation for Integrated Development Environment. Used generically by CA-Clipper to refer to Windows-based application development environments such as the CA-Clipper Workbench.

Identifier
(preprocessor, procedure and function, variable) A name that identifies a function, procedure, variable, constant or other named entity in a source program. In CA-Clipper, identifiers must begin with an alphabetic character and may contain alphabetic characters, numeric characters, and the underscore character.

Identity
(database) A unique value guaranteed by the structure of the data file to reference a specific record in a database even if the record is empty. In the Xbase file (.dbf), the identity is the record number; but it could be the value of a unique primary key or even the offset of an array in memory.

Include File
(preprocessor) *See* Header File.

Increment
(expression) To increase a value by a fixed amount, usually one. In CA-Clipper the increment operator (++) can be used to increment a numeric value in a variable.

Incremental Linking
(linker) The ability to link only the modules of an application that have been changed, greatly increasing the speed in which the link occurs. (*See also:* Linking, Module)

Index
(database) An ordered set of key values that provides a logical ordering of the records in an associated database file. Each key in an index is associated with a particular record in the database file. The records can be processed sequentially in key order, and any record can be located by performing a SEEK operation with the associated key value. (*See also:* Controlling/Master Index, Key Value, Natural Order)

Information Hiding

(procedure and function) A fundamental programming principle that states that functions and programs should conceal their inner workings from other functions and programs. Stated simply: a function should possess only the knowledge necessary for it to accomplish its task. When one function calls another, the calling function should possess only the knowledge explicitly required to call the other function. (*See also:* Encapsulation, Lexical Scoping, Modularity, Side Effect)

Initialize

(variable) To assign a starting value to a variable. If initialization is specified as part of a declaration or variable creation statement, the value to be assigned is called an *initializer*. (*See also:* Assignment)

Input Focus

(user-interface) The GET, browse cell, or menu item where user interaction can take place is said to have input focus. The item with input focus usually is displayed in enhanced color or inverse video.

Insert Mode

(user-interface) A data entry mode entered when the user presses the insert key. When this mode is active, characters are inserted at the cursor position. Text to the right of the cursor is shifted right. (*See also:* Overstrike Mode)

Inspecting

(debugger) The process of examining work areas, variables, expressions and activations inside the debugger.

Instance Variable

(object-oriented) Instance variables are the attributes or the data portion of an object as defined by the object's class. Each object, when created, is given its own unique set of instance variables initialized to their default values. Instance variables that are accessible are called *exported* instance variables. Exported instance variables can be inspected and—in some cases—assigned using the send operator. The instance variables of an object persist as long as the object it belongs to. (*See also:* Class, Object)

Instantiation

(object-oriented) Creation of an *instance* of a class, i.e., an object.

Integer

(data type) A number with no decimal digits. Note that CA-Clipper does not provide a separate data type for integer values.

Iteration

(algorithm) One of the three basic building blocks of algorithm development (the others are sequence and selection). Iteration refers to operations that are performed repeatedly, usually until some condition is satisfied. (*See also:* Selection, Sequence)

Join

(database) An operation that takes two tables as operands and produces one table as a result. It is, in fact, a combination of other operations including selection and projection. (*See also:* Selection, Projection)

Key Expression
(database) An expression, typically based on one or more database fields, that when evaluated, yields a key value for a database record. Key expressions are most often used to create indexes or for summarization operations. (*See also:* Index, Key Value)

Key Value
(database) The value produced by evaluating a key expression. When placed in an index, a key value identifies the logical position of the associated record in its database file. (*See also:* Index, Key Expression)

Keyboard Buffer
(user-interface) An area of memory dedicated to storing input from the keyboard while a program is unable to process the input. When the program is able to accept the input, the keyboard buffer is emptied.

Keyboard Polling
(user interface) The periodic system access of the keyboard buffer as one of the system services.

(system) That part of the system services that seeks and reports activity at the keyboard. It is one Input element of the DOS I/O channels.

Keyed Pair
(database) A pair consisting of a key value and an identity.

Keyword
(command, language) A word that has a special meaning to a compiler or other utility program. Commands, directives, or options are often recognized by examining supplied text to see if it contains keywords.

LAN
(network) An acronym for Local Area Network. Generally used to describe a system by which microcomputers are connected together to perform such functions as file and peripheral sharing, electronic mail, and centralized backup of data. (*See also:* Local Area Network)

Lexical Scoping
(variable) A method of determining an item's existence, visibility, or applicability (i.e., the item's *scope*) by it's position within the text of a program. (*See also:* Local, Variable, Scope of a Variable)

Lexically Scoped Variable
(variable) A variable that is only accessible in a particular section of a program, where that section is defined using simple textual rules. For example, a local variable is only accessible within the procedure that declares it. (*See also:* Dynamic Scoping, Private Variable, Public Variable, Static Variable)

Library
(linker) A file containing one or more object modules. Modules are extracted by linker and combined with object files to form an executable (.EXE) file.

Library File
(configuration) A file containing one or more object modules. The linker searches specified libraries to resolve references to functions or procedures that were not defined in the object files being linked. (*See also:* Linker, Module, Object File)

Lifetime of a Variable
(variable) The period of time during which a variable retains its assigned value. The lifetime of a variable depends on its storage class. (*See also:* Scope of a Variable, Visibility)

Line Feed
(general) A special character (CHR(10)) that by convention causes the cursor or printhead to move to the next line or to terminate a line in a text file. It is usually used in combination with a hard carriage return. (*See also:* Form Feed, Hard Carriage Return)

Linker
(program) A program that combines object files created by a compiler to produce an executable program. The linker examines the supplied object files to resolve symbol references between modules. If a module refers to a symbol that is not defined by any of the modules, the linker searches one or more libraries to resolve the reference. (*See also:* Library File, Object File)

Linking
(linker) The process in which object files and libraries are combined and references are resolved to produce a relocatable memory image (generally, an executable).

List
(expression, command) A list of expressions, field names, or file names, separated by commas specified generally as command, procedure, or function arguments. Code blocks can also execute a list of expressions.

List Match Marker
(preprocessor) A match marker indicating a position that will successfully match a list of one or more arbitrarily complex expressions. A list match marker marks a part of a command that is expected to consist of a list of programmer-supplied expressions. A list match marker has the form *<id,...>*. *id* associates a name with the match marker. The name can be used in a result marker to specify how the matching source text is to be handled.

Literal
(data type) A source code element interpreted literally (as encountered) and assumed to have no abstract meaning. Generally a constant. (*See also:* Constant).

Literal Array
(array) In CA-Clipper, an array specified by enclosing a series of expressions in curly ({}) braces. A literal array is an expression that evaluates to an array reference. (*See also:* Array, Array Reference)

Local Area Network
(network) A system by which microcomputers are connected (via coaxial cable, optical fiber, twisted pair phone wire, or other media), relying on sophisticated operating software to perform such functions as file and peripheral sharing, electronic mail, and centralized backup of data. (*See also:* LAN).

Local Variable
(variable) A variable that exists and retains its value only as long as the procedure in which it is declared is active (i.e., until the procedure returns control to a higher level procedure). Local variables are lexically scoped; they are accessible by name only within the procedure where they are declared. (*See also:* Dynamic Scoping, Lexical Scoping, Private Variable, Public Variable, Static Variable)

Locklist
(database) A list of the records that are currently locked in the work area.

Logical Type
(data type) A special data type consisting of true (.T.) or false (.F.) values. (*See also:* Condition)

Logify
(preprocessor) To change an expression in the source text into a logical value. Logifying is accomplished by surrounding the expression with periods.

Logify Result Marker
(preprocessor) A result marker of the form #<.*id*.>. *id* must correspond to the name of a match marker. This result marker writes true (.T.) to the result text if any input text is matched; otherwise, it writes false (.F.) to the result text. The input text itself is not written to the result text.

Macro
(expression) In CA-Clipper, an operation that allows source code to be compiled and executed at runtime. In CA-Clipper, the macro symbol (&) does not perform *text substitution* unless embedded within a character string. Instead, it is generally treated as a unary operator that operates on a character string. The text in the character string is compiled *on the fly* using a special runtime compiler. The resulting code is then executed, and the value obtained is returned as the result of the macro operation. (*See also:* Code Blocks, Unary Operator)

Maintainable Scoped Orders
(database) Scoped (filtered) orders created using the FOR clause. The FOR condition is stored in the index header. Orders of this type are correctly updated using the expression to reflect record updates, deletions and additions.

Make
(program) A program used to maintain multi-file program systems. A make program takes as its input a file (make file) specifying the relationships between files. When executed, the make program compares the date and time stamps of specified target files to the specified dependent files. If any of the dependent files have a more recent date and time stamp than the associated target files, a series of actions is performed. (*See also:* Make File)

Make File
(configuration) A text file used as input to a make utility containing the specifications and actions required to build a program or a system of programs. This file is often referred to as a description file. (*See also:* Make)

Manifest Constant
(preprocessor) An identifier specified in a #define directive. The preprocessor substitutes the specified result text whenever it encounters the identifier in the source text.

Map File (.MAP)
(linker) The map file (.MAP) contains information about symbol and segment addresses within the memory image created by a linker. It is generated when requested through the use of the appropriate command line switch.

Mask
(ASCII) A set of characters that represent the valid data set for some matching or selection process. A character pattern used as a filter in evaluation of a character string. An ASCII or binary filter.

(binary) Data in a specific storage location that replaces characters in the accumulator. Also, a modifier in a logical operation.

Master Index
(database) *See* Controlling/Master Index.

Match
(preprocessor) A successful comparison of source text with a match pattern (or part of a match pattern).

Match Marker
(preprocessor) A construct used in a match pattern to indicate a position that will successfully match a particular type of source text. There are several types of match markers, each of which will successfully match a particular type of source text.

Match Pattern
(preprocessor) The part of a translation directive that specifies the format of source text to be affected by the directive. A match pattern generally consists of words and match markers.

Memo Type
(data type, database) A special database field type consisting of one or more characters in the IBM extended character set. The maximum size of a memo field in CA-Clipper is 65,534 bytes. A memo field differs only from a character string by the fact it is stored in a separate memo (.dbt) file and the field length is variable length. (*See also:* Character, String)

Memory Variable
(variable) In general, a variable that resides in memory, as opposed to a database field variable. Sometimes used specifically to refer to variables of the MEMVAR storage class (private and public variables), as opposed to static or local variables. (*See also:* Field Variable, Local Variable, Private Variable, Public Variable, Static Variable, Variable)

Menu
(user-interface) An on-screen list of choices from which the user selects. Menus range from simple to elaborate forms. Two examples are menus that *pull-down* from the top of the screen (an elaborate type requiring more programming), or a simple list of numbered items from which the user selects by entering the appropriate number. (*See also:* Menu Editor)

Menu Bar
(debugger) The bar at the top of the debugger screen, on which the available menu choices are displayed.

Menu Editor

(Workbench) Use the Menu Editor to create menu structures, define their properties and menu items, and modify existing menus using standard editing techniques. (*See also:* Auto Layout, Menu, Visual Editor)

Messages

(object-oriented) A message is the way an object is requested to perform some action. Messages are sent to an object and composed of the object name, the send operator, and the selector name followed by arguments enclosed in parentheses. The selector has the same name as the method it is calling. Sending a message produces a return value, much like a function call, with the return value varying depending on the operation performed. (*See also:* Object, Method, Instance Variable)

Metasymbol

(language) Descriptive symbols used in syntax to represent information that must be supplied as part of a source code statement. A metasymbol is constructed using two information components: a data type prefix and a logical descriptor.

Metasymbol table

Prefix	Type
a	Array
b	Code block
c	Character
d	Date
exp	Expression
id	Literal identifier
l	Logical
m	Memo
n	Numeric
obj, o	Object
u	Undetermined
x	Extended expression

Within a syntax statement, metasymbols are generally delimited by square or pointed brackets ([], <>); the delimiters are not part of the syntax.

Method

(object-oriented) A method is the operation performed in response to a message sent to an object. (*See also:* Class, Message, Object)

Modularity

(modular programming) Roughly, a measure of a system's adherence to the principles of modular programming. The principles of modular programming are not precisely defined, but may be said to comprise these basic ideas: programs should be organized as well-defined *modules*; modules should correspond with syntactic units of the programming language (such as functions or source files); a module should accomplish a well-defined task; a module should interact with as few other modules as possible; interactions between modules should be explicitly specified in the source code for the modules; modules should obey the principle of information hiding. (*See also:* Encapsulation, Information Hiding, Lexical Scoping, Side Effect)

Module

(modular programming) Generically, a procedure or function (or a set of related procedures and functions) that can be treated as a unit. Sometimes used to refer specifically to the code in a single object file, normally the result of compiling a single source file. (*See also:* Object File, Source File)

Module

(linker) A portion of the object code that is a discrete unit. If any part of a module is linked, the entire module must be linked.

(Workbench) In CA-Clipper, *applications* consist of *modules*, which consist of *entities*.

Modules are "containers" for entities and can be one of three types in CA-Clipper:

- The Binary Objects module which contains the binary definitions of all form, menu, data server, and field spec entities created using the Workbench's four editors
- Program modules (usually .prg files)
- Header modules (usually .ch files)

Module Buttons

(Workbench) The buttons in a Module Browser pictorially represent each module in the current application. They are arranged in alphabetical order and visually convey the following information:

- The name of the module
- The type
- The compilation status
- The debug mode status
- The number of entities it contains

(*See also:* Compilation Status, Entity, Module)

Modulus

(mathematics) The correct term for the % operator, as used in CA-Clipper, is *modulo*. It produces the remainder of a division operation (e.g., 9 modulo 47 ==> 2). The CA-Clipper function, MOD(), performs a modulo operation on two numbers.

Traditionally, *modulus* is actually the term for an entirely different mathematical value (i.e., the absolute value of a complex number, computed by adding the squares of each and taking the positive square root of the sum). The same word also has a specific and different meaning in Physics.

Monitor Window

(debugger) The window in which monitored variables are displayed.

Monitored variable

(debugger) A variable which is selected by the options on the Monitor Menu and displayed in the Monitor Window.

Multi-tasking

(general) Computer operations in which one CPU handles several tasks. System operations in which a single CPU handles programs in a manner that seems simultaneous to one or more users while it interleaves the programs, executing them in segments.

Multidimensional Array

(array) In CA-Clipper, an array whose elements consist entirely of references to other arrays (called *subarrays*). The elements of the subarrays may, in turn, contain references to other arrays. Arrays organized in this fashion are said to be *nested*. Each level of nesting may be viewed as a *dimension* of the main array, and the elements of the subarrays may be accessed by applying multiple subscripts to the main array. (*See also:* Array, Array Reference, Nested Array, Subscript)

Multiple-Order Bag

(database) An order bag that can contain any number of orders; a multiple-tag index. The .cdx and .mdx files are examples of multiple-order bags.

Name

(general) A routine's public symbol. Often the words "symbol," "public symbol," and "name" will be used interchangeably.

Naming Convention

(general) The way a compiler alters the name of a routine before placing it in an object file. According to the Microsoft Mixed-Language Programming Guide, this term "refers to the way that a compiler alters the name of a routine before placing it in an object file."

Natural Order

(database) For a database file, the order determined by the sequence in which records were originally entered into the file. Also called unindexed order. (*See also:* Index)

Nested Array

(array) In CA-Clipper, two arrays are said to be *nested* if one of them contains a reference to the other. When an array contains a reference to a second array, the second array is sometimes called a *subarray* of the first array. (*See also:* Array, Array Reference, Multidimensional Array, Subscript)

NIL

(data type) A special data type that has only one allowable value. The special value (NIL) is automatically assigned to all uninitialized variables except publics and is also passed as a substitute when arguments are omitted in a procedure or function call.

Non-Maintainable/Temporary Orders

(database) Orders created using the WHILE or NEXT clauses. These orders are useful because they can be created quickly. However, the conditions in these clauses are *not* stored in the index header. Therefore, orders of this type are *not* correctly updated to reflect record updates, deletions and additions. They are only for temporary use.

Nondedicated Server

(equipment) A computer that serves as both a workstation and a network server.

Normal Stringify Result Marker

(preprocessor) A result marker of the form <"*id*">. *id* must correspond to the name of a match marker. A normal stringify result marker specifies that the corresponding source text is to be enclosed in quotes. If the matched source text is a list, each element of the list is individually stringified. If no source text was matched, an empty result is produced.

Normalization
(database) The process of elimination and consolidation of redundant data elements in a database system.

Numeric Type
(data type) A special data type consisting of values that indicate magnitude. Numeric values consist of digits between zero and nine, a sign, and a decimal point.

Object
(object-oriented, data type) An object is an instance of a class. Each object has one or more attributes (called instance variables) and a series of operations (methods) that execute when a message is sent to the object. The object's instance variables can only be accessed or assigned by sending messages to the object. Objects are created by calling a special function associated with a class. (*See also:* Class, Instance Variable, Message, Method)

Object File
(configuration) A file that contains the output of a compiler or other language translator, generally the result of compiling a single source file. Object files are linked to create an executable program. (*See also:* Linking, Source File)

Operand
(expression) A value that is operated on by an operator, or the term in an expression that specifies such a value. For example, in the expression $x + 5$, x and 5 are operands. (*See also:* Operator)

Operating System
The basic software program that organizes and services the computer and its peripheral devices. The operating system supported by CA-Clipper, MS/PC-DOS, is organized into several layers as follows:

- *Loader* is the layer which brings the operating system software into memory.
- *BIOS* is the basic hardware interface layer that provides services to the kernel and consists of initialization code and device drivers.
- *Kernel* is the application interface layer and provides services for process control, memory management, peripheral support, and a file system.
- *User interface shell (COMMAND.COM)* provides basic services to the user including an interactive mode, directory management, and a service for loading and executing application programs.
- *Support programs* provide extended operating services not resident in the user interface shell.

Operator
(expression) A symbol that identifies a basic operation. For example, the multiplication operator (*) denotes that two values are to be multiplied. Operators are categorized as either unary or binary, depending on whether they require one or two operands, respectively. (*See also:* Binary Operator, Operand, Unary Operator)

Optional Clause

(command, preprocessor) A portion of a match pattern that is enclosed in square ([ ]) brackets. An optional clause specifies part of a match pattern that need not be present for source text to match the pattern. An optional clause may contain any of the components legal within a match pattern, including other optional clauses. When a match pattern contains a series of optional clauses that are immediately adjacent to each other, the matching portions of the source text are not required to appear in the same order as the clauses in the match pattern. If an optional clause is matched by more than one part of the source text, the multiple matches may be handled using a repeating clause in the result pattern.

Order

(database) A named mechanism (index) that provides logical access to a database according to the keyed-pairs. This term encompasses both single indices and the tags in multiple-tag indices.

Orders are not, themselves, data files. They provide access to data that gives the appearance of an ordering of the data in a specific way. This ordering is defined by the relationships between keyed-pairs. An order does not change the physical (the natural or entry) order of data in a database.

Order Bag

(database) A container that holds zero or more orders. Normally a disk or memory file. A traditional index like .ntx is an order bag that holds only one order. A multiple-tag index (.mdx or .cdx) is an order bag that holds zero or more orders. Though order bags may be a memory or disk file, CA-Clipper 5.3 only supports order bags as disk files.

Order List

(database) A list of all the orders available to the database in the specified work area.

Ordinal Position

(general) The numeric position within a sequence.

(programming) The position of an array or list element within its array or list. The number(s) that represent that position.

Overlay

(linker) A section of an executable program that shares memory with other sections of the same program. An overlay is read into memory when the code residing in it is requested by the root (non-overlaid) section or another overlay. (*See also:* Dynamic overlay)

Overstrike Mode

(user-interface) A data entry mode entered when the user presses the insert key. When this mode is active, characters are entered at the cursor position and text to the right of the cursor remains stationary. (*See also:* Insert Mode)

Page Frame

(memory) A 64 KB block of memory that is made available to applications as four contiguous 16 KB pages.

Pan

(user interface) Apparent horizontal movement of a cursor or view across an area. Sometimes mistakenly applied to apparent vertical movement which is, correctly, scrolling. (*See also:* Scrolling)

Parameter
(variable) A identifier that receives a value or reference passed to a procedure or user-defined function. A parameter is sometimes referred to as a *formal parameter*. (*See also:* Activation, Argument, Function, Procedure, Reference)

Parameter-passing Convention
(general) The specific format in which particular parameters are meant to be transferred.

Path
(file) A literal string that specifies the location of a disk directory in the tree structured directory system. A path specification consists of the following elements: an optional disk drive letter followed by a colon, an optional backslash indicating that the path starts at the root directory of the specified drive, the names of all the directories from the root directory to the target directory, separated by backslash (\) characters. Example: C:\CLIP53\INCLUDE. A path list is a series of path specifications separated by semicolons.

Picture
(user-interface) A string that defines the format for data entry or display in a GET, SAY, or the return value of TRANSFORM(). Picture strings are comprised of functions which affect the formatting as a whole and a series of template characters that affect formatting on a character by character basis. (*See also:* Template)

Pointer
(memory) A variable that *points* to another variable. Usually a pointer is a memory address. As one of C's strongest features, pointers provide the means by which functions can modify their calling arguments. They are also useful for optimizing routines.

Port
(general) A designation for the hardware that allows the processor to communicate with peripheral devices.

Postfix Notation
(general) Inline placement of decrement (--) or increment (++) operators after the variable so the operators are applied to the value after an assignment.

Example:

```
LOCAL x, y
x := 99
Y = x++    // postfix increment
? Y        // Y = 99
```

Precedence
(expression) The stature of an operator in the hierarchy that determines the order in which expressions are evaluated. For example, the expression *5 + 2 * 3* is interpreted as *5* + (2 * *3*) because the multiply operator (*) has a higher precedence than the addition operator (+). (*See also:* Expression)

Prefix Notation
(general, operators) Inline placement of decrement (--) or increment (++) operators before the variable so the operators are applied to the value before an assignment.

Example:

```
LOCAL x, y
x := 99
Y = ++x    // prefix increment
? Y        // Y = 100
```

Glossary

Preprocessor

(program, preprocessor) A translation program that prepares source code for compilation by applying selective text replacements. The replacements to be made are specified by directives in the source file. In CA-Clipper, the preprocessor operates transparently as a part of the compiler program (CLIPPER.EXE). (*See also:* Compiler)

Preview Menu Bar

(Workbench) As you define a menu structure in the Menu Editor, each new entry is added to the *preview menu bar,* a prototypical menu bar at the top of the Menu Editor window. The preview menu bar is partially operational—showing description messages in the status bar and allowing submenus to be pulled down—but nothing actually happens when you make a selection. Its purpose is to give you visual feedback while you are designing a menu structure. (*See also:* Menu Editor)

Primitive

(procedure and function) A simple, low-level function used by other high-level functions or programs to perform a more complex task.

Print Spooler

A program running either on a local workstation or on the file server that captures print jobs to a file and then queues them for later printing. Print spoolers generally operate as background tasks in order to facilitate printing while other tasks are operating in the foreground.

Private Variable

(variable) A variable of the MEMVAR storage class. Private variables are created dynamically at runtime using the PRIVATE statement. They are accessible within the creating procedure and any lower level procedures unless obscured by another private variable with the same name. (*See also:* Activation, Dynamic Scoping, Function, Lexical Scoping, Local Variable, Procedure, Public Variable, Static Variable)

Procedure

(procedure and function) An executable block of code with an assigned name. Alternately, the collection of source code statements that define a procedure. In CA-Clipper, this can be a source (.prg) file, a format (.fmt) file, an explicitly declared procedure (PROCEDURE), or an explicitly declared function (FUNCTION).

The terms *procedure* and *function* are generally interchangeable. By convention, a function returns a value, while a procedure does not. (*See also:* Activation, Function, Parameter)

Procedure File

(configuration) An ASCII text file containing CA-Clipper procedure and function definitions, usually ending with a .prg extension; a program file. (*See* Program File)

Program Editor

(program) An executable program that allows the creation and editing of text files or programs from within DOS. (*See also:* File Server, Source Code Editor)

Program File

(configuration) An ASCII text file containing CA-Clipper source code. Program files usually end with a .prg extension. The compiler reads the program file, translates the source code, and produces an object file that is then linked to produce an executable program. (*See also:* Linking, Object File, Source Code)

Projection

(database) A DBMS term specifying a subset of fields. In CA-Clipper, the analogy is the FIELDS clause. (*See also:* Join, Selection)

Prompt

(user-interface) A series of characters displayed on the screen indicating that input from the keyboard is expected.

Protected Mode

(CPU) In this mode, an 80286 or higher processor retains instruction compatibility with the 8086 but can directly address 16 MB of memory.

Pseudofunction

(preprocessor) A function-like construct that is replaced with another expression via the #define directive, rather than compiled into a conventional function call. Pseudofunctions may contain parenthesized arguments that may be included in the substituted text.

Public Variable

(variable) A variable of the MEMVAR storage class. Public variables are created dynamically at runtime using the PUBLIC statement and are accessible from any procedure at any level unless obscured by a private variable with the same name. (*See also:* Activation, Dynamic Scoping, Function, Lexical Scoping, Local Variable, Private Variable, Procedure, Static Variable)

Query

(database) A request for information to be retrieved from a database. Alternately, a data structure in which such a request is encoded.

(general) A query is also a general term used when you want to interrogate a setting or an exported instance variable for its current value.

Queue

(general) A data structure of variable length where elements are added to one end and retrieved from the other. A queue is often described as *first in, first out*. (*See also:* Stack, Print Spooler)

Real Memory

(memory) Memory available to an application at time of execution.

Real Mode

(CPU) 8086 emulation mode. In this mode, a computer cannot directly address *extended* memory, and an 80286 or higher processor performs as if it were a fast 8088.

Record

(database) A record in the traditional database paradigm is a row of one or more related columns (fields) of data. In the expanded architecture of CA-Clipper, a record could be data that does not exactly fit this definition.

A record is, in this expanded context, data associated with a single *identity*. In an Xbase data structure, this corresponds to a row (fields associated with a record number); in other data structures, this may not be the case.

In this document we use "record" in the traditional sense, but you should be aware that CA-Clipper permits expansion of the meaning of record.

Record

(database) The basic row unit of a database file consisting of one or more field elements. (*See also:* Database, Field, Table, Tuple)

Record Locking

(network) The process by which one user obtains exclusive access to a record in a database to prevent another user from attempting to write data to it concurrently. A record lock must be applied prior to writing to a database in use by more than one user.

Recovery

(error handling) The process of attempting to handle an exception or runtime error. Generally, recovery consists of three possible actions: terminate processing, retry the failed operation, or resume processing with the next operation. In all cases, the environment of the program must be restored to a stable state. (*See also:* Exception, Error, Failure, Retry, Runtime Error)

Recursion

(expression) The calling of a procedure by a statement in that same procedure. When a procedure calls itself it is said to *recurse*. A recursive call causes a new activation of the procedure. If the source code for the procedure includes a declaration of local variables, a new set of local variables is created for each activation. A private variable created by the procedure is associated with the activation in which it is created and is visible in that activation and any lower level activations, unless obscured by a private variable created in a lower level activation. (*See also:* Activation, Function, Private Variable, Procedure)

Reference

(array, variable) A special value that refers indirectly to a variable or array. If one variable contains a reference to a second variable (achieved by passing the second variable by reference in a function or procedure call), operations on the first variable (including assignment) are *passed through* to the second variable. If a variable contains a reference to an array, the elements of the array can be accessed by applying a subscript to the variable. (*See also:* Array Reference, Parameter)

Regular Match Marker

(preprocessor) A match marker indicating a position that will successfully match an arbitrarily complex expression in the source text. A regular match marker generally marks a part of a command that is expected to consist of arbitrary programmer-supplied text, as opposed to a keyword or other restrictive component. In order for the source text to match, it must constitute a properly formed expression. A regular match marker has the form <*id*>. *id* associates a name with the match marker. The name can be used in a result marker to specify how the matching source text is to be handled.

Regular Result Marker

(preprocessor) A result marker of the form <*id*>. *id* must correspond to the name of a match marker. This result marker writes the matched input text to the result text or writes nothing if no input text is matched.

Relation

(database) A link between database files that allows the record pointer to move in more than one database file based on the value of a common field or expression. This allows information to be accessed from more than one database file at a time.

Relational Database System

(database) A system that stores data in rows and columns, without system dependencies within the data. In other words, relationships between different databases are not stored in the actual database itself, as is the case in a system that uses record pointers.

Relative Addressing

(user-interface) To refer to a memory address, array element, screen location, or printer location with respect to another value, rather than referring to a specific address or element.

Repeating Clause

(preprocessor) A portion of a result pattern surrounded by square ([ ]) brackets. The text specified by the repeating clause is written to output once for each successfully matched match marker in the corresponding match pattern.

Repository

(Workbench) The *repository* is where the CA-Clipper Workbench stores all application components, and it automatically manages the relationships between the various components of an application.

Restricted Match Marker

(preprocessor) A match marker indicating a position that will successfully match one or more specified keywords. A restricted match marker marks a part of a command that is expected to be a keyword. A restricted match marker has the form <*id: wordList*> where *wordList* is a list of one or more keywords. Source text is successfully matched only if it matches one of the keywords (or is an acceptable abbreviation). *id* associates a name with the match marker. The name can be used in a result marker to specify how the matching source text is to be handled.

Result Pattern
(preprocessor) The part of a translation directive that specifies the text to be substituted for source text that matches the match pattern. A result pattern generally consists of operators and result markers.

Result Text
(preprocessor) The text that results from formatting matched source text using a result pattern. If the result text matches a match pattern in another preprocessor directive, then it becomes the source text for that directive. Otherwise, the result text is passed as input to the compiler.

Retry
(network) Upon failing to lock a record in a database that is opened in shared mode, retry refers to attempting the write again based upon certain programmatic parameters. (*See also:* Shared, Exclusive)

(error handling) After an exception has been raised and the conditions of a failure corrected, an attempt is made to re-execute the failed operation. (*See also:* Exception, Error, Failure, Runtime Error)

Return Value
(procedure and function) The value or reference returned by a function or method from a function call or message send.

Robust
(general) Strength, durability. The quality of an application or language, or element that permits its effective use, without failure, in many environments and situations.

Routine
(general) Any procedure, function, or other complete lexical unit.

Row
(database) A group of related column or field values that are treated as a single entity. It is the same as a CA-Clipper record. (*See also:* Column, Field, Record)

(user-interface) A numeric expression that evaluates to an integer identifying a screen or printer row position.

Run Mode
(debugger) The mode of execution in which an application executes without pausing, until a breakpoint or tracepoint is reached.

Runtime Error
(error handling) An error that halts a program while it is executing.

Scope
(command, database) In a database command, a clause that specifies a range of database records to be addressed by the command. The scope clause uses the qualifiers ALL, NEXT, RECORD, and REST to define the record scope. (*See also:* Condition)

Scoreboard
(user-interface) An area of the display on line 0 beginning at column 60 that displays status information during certain data entry operations.

Screen Coordinates
(user-interface) The area of the screen beginning at the origin of the screen (0,0) and incrementing as X- and Y-coordinates. Note that the relation of coordinates to pixels depends on the mapping mode that Windows uses.

Script File

(configuration) A text file that contains command input to a compiler, linker, or other utility program. A script file is often used in lieu of equivalent keyboard input. For the CA-Clipper compiler, script files contain a list of source files to be compiled into a single object file.

(debugger) A file in which frequently used debugger commands are stored and from which those commands can be executed.

Scrolling

(user-interface) The action that takes place when the user attempts to move the cursor or highlight beyond the window boundary to access information not currently displayed. (*See also:* Window)

SDF File

(file, database) A text file that contains fixed-length database records with each record separated by a carriage return/line feed pair (CHR(13) + CHR(10)) and terminated with an end of file mark (CHR(26)). Each field within an SDF file is fixed-length with character strings padded with trailing spaces and numeric values padded with leading spaces. There are no field separators. (*See also:* Database, Text File, Delimited File)

Search Condition

(database) *See* Condition, Scope

Section

(linker) Load module portion of an .EXE or .OVL file loaded into memory as a single unit. In a program with overlays, the root section containing the main program module loads when the program is executed. Other sections are loaded as overlays when modules within them are invoked. (*See also:* Dynamic Overlay, Linking)

Segment

(linker) Code or data handled by the linker as a indivisible unit.

Segment Handle (VM API)

(linker) The identifier of a particular Virtual Memory segment.

Selection

- (algorithm) One of the three basic building blocks of algorithm development (the others are sequence and iteration). Selection allows control to flow along a number of possible paths, depending on the circumstance encountered.
- (database) A DBMS term that specifies a subset of records meeting a condition. The selection itself is obtained with a selection operator. In CA-Clipper, the analogy is the FOR clause.

(*See also:* Sequence, Iteration, Join, Projection)

Self

(object-oriented) An object-oriented term describing a reference to the object that received the current message. In CA-Clipper, this reference is often the return value of a message send.

Send Operator
(object-oriented) A new operator (:) in CA-Clipper used to send messages to user interface objects.

Separator
(file, database) The character or set of characters that differentiate fields or records from one another. In CA-Clipper, the DELIMITED and SDF file types have separators. The DELIMITED file uses a comma as the field separator and a carriage return/line feed pair as the record separator. The SDF file type has no field separator, but also uses a carriage return/line feed pair as the record separator. (*See also:* Delimiter)

Sequence
- (algorithm) One of the three basic building blocks of algorithm development (the others are selection and iteration). A sequence is a series of discrete steps that must be performed in a particular order.
- (language) In CA-Clipper, a series of statements enclosed in a BEGIN SEQUENCE control structure. (*See also:* Algorithm, Iteration, Selection)

Set Colors Window
(debugger) The window in which the Debugger color settings can be displayed and modified.

Shared
(network) A mode in which a file is opened that allows it to be accessed by more than one user at the same time. The inverse of Exclusive.

Shortcutting
(expression) A compiler optimization that causes expressions to be evaluated only to the extent required to determine their outcome. For example, in the expression *f*() .OR. *g*() function *g* need not be executed if function *f* returns true (.T.). CA-Clipper performs shortcutting on all logical operators (.OR. .AND. .NOT.).

Sibling
(Workbench) A *sibling* is a menu item at the same level as another within a menu structure. Use the Promote Item toolbar button to promote a menu item to the sibling level in a menu's hierarchy. (*See also:* Child)

Side Effect
(modular programming) An *unexpected effect* of executing a function or program. When a function changes the state of a system in a way that is not explicitly specified by the function's name or calling protocol, the change is called a *side effect*. Reliance on side effects is contrary to the principles of modular programming. (*See also:* Encapsulation, Information Hiding, Lexical Scoping, Modularity)

Single Step Mode
(debugger) The mode of execution in which only the line of code highlighted by the execution bar is executed, and its output displayed.

Single-dimensional Array
(array) In CA-Clipper, an array whose elements do not contain references to other arrays. (*See also:* Array, Array Reference, Multi-dimensional Array, Nested Array, Subarray, Subscript)

Single-Order Bag
(database) An order bag that can contain only one order. The .ntx and .ndx files are examples of single-order bags.

Skeleton
(command) A wildcard mask used to specify a group of file names or memory variables. The * is used to specify one or more characters and the ? to specify a single character.

Smart Stringify Result Marker
(preprocessor) A result marker of the form <(*id*)>. *id* must correspond to the name of a match marker. A smart stringify result marker specifies that the corresponding source text is to be enclosed in quotes unless the source text was enclosed in parentheses. If the matched source text is a list, each element of the list is individually processed. If no source text was matched, an empty result is produced. The smart stringify result marker is used to implement commands that allow extended expressions (a part of a command that may be either an unquoted literal or a character expression).

Soft Carriage Return
(general) A carriage return that is introduced into text usually in order to implement some sort of wrap operation, as opposed to a Hard Carriage Return that was specifically entered into the text when it was created.

Sort Order
(array, database) Describes the various ways database files and arrays are ordered.

- Ascending

  Causes the order of data in a sort to be from lowest value to highest value.

- Descending

  Causes the order of data in a sort to be from highest value to lowest value.

- Chronological

  Causes data in a sort to be ordered based on a date value, from earliest to most recent.

- ASCII

  Causes data in a sort to be ordered according to the ASCII Code values of the data to be sorted.

- Dictionary

  The data in a sort is ordered in the way it would appear if the items sorted were entries in a dictionary of the English language.

- Collating Sequence

  Data in a sort will be placed in sequence following the order of characters in the IBM Extended Character Set.

- Natural

  The order in which data was entered into the database.

Source Code
(procedure and function) The textual representation of a program or procedure. (*See also:* Source File, Object File)

Glossary

Source Code Editor
(Workbench) Use the Source Code Editor to create new entities; edit existing entities using standard editing techniques—such as cut, copy, paste, delete, search for and replace text, and insert and delete lines of code. (*See also:* Program Editor)

Source File
(configuration) *See* Program File, Header File.

Source Text
(preprocessor) Text from a source file, processed by the preprocessor. Source text is examined to see if it matches a previously specified match pattern. If so, the corresponding result pattern is substituted for the matching source text.

Specialization
(Error class) The conditional processing performed by an error handler, usually in a series of case statements.

Spooler
(network) *See* Print Spooler.

Stabilization
(class) Stabilization is a TBrowse class process accomplished through the stabilize() method. The process includes a call to a browsing method and a testing of the instance variable.

After all the browse objects in a browsing area are established, the area is drawn on the screen, the internal browsing cursor is initialized, and the data is displayed in the proper locations. The process is incremental and may be interrupted at any time. It continues until all objects in a browse area are *stabilized*.

Stack
(general) A data structure of variable length whose elements are added and retrieved from the same end. A stack is often described as *first in, last out*. (*See also:* Queue)

Standard Color
(user-interface) The color pair definition that is used by all output options (such as SAY and ?), with the exception of GETs and PROMPTs, that use the enhanced color pair. (*See also:* Enhanced Color).

Statement
(language) In CA-Clipper, the basic unit of source code. A statement is normally a single line of text. Multiple statements can be placed on the same line by separating them with semicolons. A statement may be continued to another line by placing a semicolon at the end of the line to be continued. If the text of a statement matches a command definition (defined with a preprocessor directive), it is translated into the form specified by the command definition. (*See also:* Command)

Static Overlay
(linker) A section of the program that is not always resident in RAM and shares memory with other sections. The section that is currently in use is loaded into memory, allowing a larger program to execute in less available RAM.

Static Variable
(variable) A variable that exists and retains its value for the duration of execution. Static variables are lexically scoped; they are only accessible within the procedure that declares them, unless they are declared as *file-wide*, in which case they are accessible to any procedure in the source file that contains the declaration. (*See also:* Dynamic Scoping, Lexical Scoping, Local Variable, Private Variable, Public Variable)

Status Bar
(Workbench) Almost every window, browser, and editor in the CA-Clipper Workbench contains a status bar that displays helpful, informative text about the current window, a toolbar button, or selected menu command.

Std.ch
(preprocessor) The *standard header file* containing definitions for all CA-Clipper commands.

Storage Class
(variable) Defines the two characteristics of variables: lifetime and visibility. (*See also:* Lifetime, Scope, Visibility)

String
(data type) Generically, a value of type character. In source code, a series of characters enclosed in single or double quotes. (*See also:* Character)

Stringify
(preprocessor) To change source text into a literal character string by surrounding the text with quotes.

Stub
(error handling) A procedure used for debugging purposes that only simulates the intended actions of the real procedure. It may display an indicating message, return a constant value, or do nothing.

Subarray
(array) In CA-Clipper, an array that is referred to by an element of another array. (*See also:* Array, Array Reference, Multi-dimensional Array, Nested Array, Subscript)

Subclass (noun)
(class, API programming) A class that inherits from another class.

Subclass (verb)
(class, API programming) The act of extending or modifying the behavior of a class through inheritance.

Subdirectory
(file) *See* Directory.

Subscript
(array) A numeric value used to designate a particular element of an array. Applying a subscript to an array is called *subscripting* the array. In CA-Clipper programs, subscripting is specified by enclosing a numeric expression in square ([ ]) brackets after the name of a program variable. The variable is then said to be *subscripted.* (*See also:* Array, Array Reference, Multi-dimensional Array, Nested Array, Subarray)

Substring
(data type) A string within a string, usually to be specified as an argument of a function or command.

Swap Space (VM API)
(linker) Region of real memory in which the Virtual Memory Manager loads virtual memory segments.

Swapfile
(linker) Also known as the *workfile*, used by a linker to swap data and code in and out of memory during the linking process.

Symbol
(linker) An assigned name for a value representing a constant or the address of code or data. There are four types of symbols used by the linker defined as follows:

- *Absolute symbol:* a constant
- *Relative symbol:* address of code or data
- *Public symbol:* accessed by modules other than the module in which they are defined. Public symbols are used to share procedures and variables between modules. As such, the relative address of a public symbol is assigned by the compiler during compilation.
- *External symbol:* a public symbol not defined in the current module. Generally, these are references into CLIPPER.LIB or EXTEND.LIB, but the compiler generates them whenever there is a procedure or user-defined function referenced but not compiled into the current module.

Syntax
(language) The rules that dictate the form of statements or commands as defined by the implementors of the language. Also, a complete description of the forms that a statement or command can take.

Table
(database) A DBMS term defining a collection of column definitions and row values. In CA-Clipper, it is represented and referred to as a database file.

Tag
(database) A set of keyed-pairs that provides ordered access to the table based on a key value. Usually, an order in a multiple-order index (order).

Template
(user-interface) A mask that specifies the format in which data should be displayed. For example, you might want to store phone numbers as "9999999999" to save space, but use a template to display the number to the user as "(999) 999-9999."

Text File
(file) A file consisting entirely of ASCII characters. Each line is separated by a carriage return/line feed pair (CHR(13) + CHR(10)) and the file is terminated with a end of file mark (CHR(26)). (*See also:* Delimited File, Program File, SDF File)

Text Replacement
(preprocessor) The process of removing portions of input text and substituting different text in its place.

Toggle
(command) As a verb, to choose between an *on* or *off* state. As a noun, a value or setting that can be either on or off. A toggle is often represented using a logical value, with true (.T.) representing on, and false (.F.) representing off.

Token
(preprocessor) An elemental sequence of characters having a collective meaning. The preprocessor groups characters into tokens as it reads the input text stream. Tokens include identifiers, keywords, constants, and operators. White space, and certain special characters, serve to mark the end of a token to the preprocessor.

Tool Palette
(Workbench) The tool palette is a feature of the Form Editor that contains a set of icons that provide for drag-and-drop placement of GUI controls on a form.

Toolbar
(Workbench) Almost every window, browser, and editor in the CA-Clipper Workbench contains a customized toolbar that provides buttons as shortcuts to commonly used menu commands. Most toolbars have the same set of common buttons on the left, and buttons specific to the particular editor or browser on the right.

Trace Mode
(debugger) A mode of execution similar to Single Step Mode, the difference being that Trace Mode traces over function and procedure calls.

Tracepoint
(debugger) A variable or expression whose value is displayed in the Watch Window, and which causes an application to pause whenever that value changes.

Translation Directive
(preprocessor) A preprocessor instruction containing a translation rule. The two translation directives are #command and #translate.

Translation Rule
(preprocessor) The portion of a translation directive containing a match pattern followed by the special symbol (=>) followed by a result pattern.

Truncate
(expression) To remove insignificant information from the end of an item of data. With numerics, to ignore any part of the number that falls outside of the specified precision.

Tuple
(database) A formal DBMS term that refers to a row in a table or a record in a database file. In DIF files, tuple also refers to the equivalent of a CA-Clipper record. (*See also:* Database, Field, Record)

Two-dimensional Array
(array) An array that has two dimensions. In CA-Clipper, an array whose elements contain references to other arrays, all of which have the same length and do not refer to other arrays. (*See also:* Array, Array Reference, Nested Array, Subscript)

Typeahead Buffer
(user-interface) *See* Keyboard Buffer.

Unary Operator
(expression) An operator that operates on a single operand. For example, the .NOT. operator. (*See also:* Binary Operator, Operator)

Undefine
(preprocessor) To remove an identifier from the preprocessor's list of defined identifiers via the #undefine directive.

Undefined Symbol
(linker) An *unresolved symbol* that was never declared public by a module, but which is referenced by another module. After the public symbol definition is encountered, the symbol becomes defined (resolved). When a symbol is referenced, but not defined, it is said to be undefined.

Unselected Color
(user-interface) The color pair definition used to display all but the current GET, or the GET that has input focus. If this color setting is specified, the current GET is displayed using the current enhanced color. (*See also:* Enhanced Color).

Update
(database) The process of changing the value of fields in one or more records. Database fields are updated by various commands and the assignment operator.

User Function
(user-interface) A user-defined function called by ACHOICE(), DBEDIT(), or MEMOEDIT() to handle key exceptions. A user function is supplied to one of these functions by passing a parameter consisting of a string containing the function's name.

User Interface
(user-interface) The way a program interacts with its user (i.e., menu operation and selection, data input methods, etc.)

User-defined Function
(procedure and function) *See* Function.

Variable
(variable) An area of memory that contains a stored value. Also, the source code identifier that names a variable. (*See also:* Local Variable, Private Variable, Public Variable, Static Variable)

Vector
(database) In a DIF file, vector refers to the equivalent of a CA-Clipper field. (*See also:* Database, Field, Record, Tuple)

Verb
(command) The first word of a command that describes the action to perform. (*See also:* Command)

View
(database) A DBMS term that defines a virtual table. A virtual table does not actually exist but is derived from existing tables and maintained as a definition. The definition in turn is maintained in a separate file or as an entry in a system dictionary file. In CA-Clipper, views are supported only by DBU.EXE and are maintained in .vew files. (*See also:* Database, Field, Record)

View Sets Window
(debugger) The window in which CA-Clipper status settings can be inspected.

View Workareas Window
(debugger) The window in which work area information is displayed.

Virtual Memory

(general) Disk space, RAM drive, or expanded memory used as random access memory.

(VM API) The Virtual Memory Application Programmer Interface is a group of functions that permits a limited amount of real memory to emulate a much larger *virtual* memory space. Extend routines call these functions, directly communicating with the Virtual Memory Manager.

Visibility

(variable) The set of conditions under which a variable is accessible by name. A variable's visibility depends on its storage class. (*See also:* Dynamic Scoping, Lexical Scoping)

Visual Editor

(Workbench) A *visual editor* is a WYSIWYG environment that provides for visual development of applications using standard Windows techniques such as point-and-click and drag-and-drop placement of controls on a window.

The CA-Clipper Workbench's Menu Editor, Form Editor, DB Server Editor, and FieldSpec Editor are visual editors. (*See also:* DB Server Editor, FieldSpec Editor, Form Editor, Menu Editor)

Volume

(file) A unit of disk storage uniquely identified by a *label* and of fixed size. A hard disk can be partitioned into one or more volumes by an operation system utility. Volumes are subdivided into one or more directories organized in tree structure. (*See also:* Directory, Disk)

Wait State

(user-interface) A wait state is any mode that extracts keys from the keyboard except for INKEY(). These modes include ACHOICE(), DBEDIT(), MEMOEDIT(), ACCEPT, INPUT, READ and WAIT.

Watch Window

(debugger) The window in which watchpoints and tracepoints are displayed.

Watchpoint

(debugger) A variable or expression whose value is displayed in the Watch Window and updated as an application executes.

Wild Match Marker

(preprocessor) A match marker indicating a position that will successfully match any source text. A wild match marker matches all source text from the current position to the end of the source line. A wild match marker has the form <**id**>. *id* associates a name with the match marker. The name can be used in a result marker to specify how the matching source text is to be handled.

Window

(user-interface) A rectangular screen region used for display. A window may be the same size or smaller than the physical screen. Attempting to display information that extends beyond the specified boundaries of the window clips the output at the window edge.

Word
(preprocessor) A series of characters in a match pattern or result pattern. Source text matches a word in a match pattern if the text is identical to the word or is an acceptable abbreviation of it. A word that appears in a result pattern is copied unmodified into the result text.

Word Wrapping
(user-interface) The process of continuing the current text on the next line of a display when a boundary is reached and breaking the text on a word boundary.

Work Area
(database) The basic containment area of a database file and its associated indexes. Work areas can be referred to by alias name, number, or a letter designator. (*See also:* Alias)

Workfile
(linker) *See* Swapfile.

Workstation
(network) A personal computer connected to a network used to run applications and front end processes. (*See also:* File Server)

Index

!

! (negate), 2-54

! (RUN), See RUN

!= (not equal), 2-64

#

(not equal), 2-64

define, 2-498

#command | #translate, 2-1

#define, 2-12, 2-26

#else, 2-18, 2-20

#endif, 2-18, 2-20

#error, 2-17

#ifdef, 2-18

#ifndef, 2-20

#include, 2-22, 2-83, 2-554, 2-891, 2-903

#stdout, 2-25

#undef, 2-26

#xcommand, 2-28

#xtranslate, 2-28

$

$ (substring), 2-29, 2-170, 2-777

%

% (modulus), 2-30

%= (modulus and assign), 2-68

&

& (compile and run), 2-31

(

() (group), 2-39

*

* (compatibility), 1-3

* (multiplication), 2-40

** (exponentiation), 2-41, 2-402, 2-585

*= (multiplication and assign), 2-68

Index

+

+ (addition), 2-42

+ (concatenation), 2-42

++ (increment), 2-44

+= (addition and assign), 2-68

+= (concatenation and assign), 2-68

-

- (concatenation), 2-46

- (subtraction), 2-46

-- (decrement), 2-48

-= (concatenation and assign), 2-68

-= (subtraction and assign), 2-68

-> (alias), 2-50

.

.AND., 2-53

.dbt files, 2-1006

.ndx files, 2-532, 2-539, 2-1006

.NOT., 2-54

.ntx files, 2-532, 2-539, 2-1006

.OBJ files, 2-903

.OR., 2-55

/

/ (division), 2-56

/= (division and assign), 2-68

:

:, 2-57, 2-59, 2-444, 2-547, 2-580, 2-734, 2-736, 2-747, 2-940, 2-943

<

< (less than), 2-9, 2-61

<= (less than or equal), 2-63

<> (not equal), 2-64

=

= (assign), 2-66, 2-734, 2-943

= (equal), 2-71, 2-165, 2-878

== (exactly equal), 2-73, 2-878

>

> (greater than), 2-75

>= (greater than or equal), 2-77

?

?, 2-79, 2-757, 2-900

??, See ?

@

@ (pass-by-reference), 2-81

@...BOX, 2-83

@...CLEAR, 2-85

@...GET, 2-86, 2-117, 2-218, 2-220, 2-782, 2-792, 2-861, 2-871, 2-884, 2-889
    implementation, 2-459

@...GET CHECKBOX, 2-97

@...GET LISTBOX, 2-101

@...GET PUSHBUTTON, 2-106

@...GET RADIOGROUP, 2-110

@...PROMPT, 2-115, 2-625, 2-894

@...SAY, 2-86, 2-117, 2-874, 2-884, 2-892

@...TO, 2-122

@.TBROWSE, 2-113

(

[] (array element, 2-124

^

^ (exponentiation), 2-41

^= (exponentiation and assign), 2-68

{

{}, 2-126

|

| | (code block argument delimiter), 2-126

A

AADD(), 2-127

ABS(), 2-129

Absolute value, 2-129

ACCEPT, 2-130

ACHOICE(), 2-131

Achoice.ch, 2-134

ACLONE(), 2-138,

ACOPY(), 2-139,

addItem(), 2-572, 2-727, 2-772, 2-981

ADEL(), 2-141

ADIR(), 2-142

AEVAL(), 2-144, 2-757

AFIELDS(), 2-146

AFILL(), 2-148

AINS(), 2-150

ALERT(), 2-151

Index

Alias
database file, 2-1008
default, 2-1008
field, 2-50, 2-412
memory variable, 2-615
obtaining name, 2-153
operator, 2-50
selecting work area with, 2-852
work area, 2-1008

ALIAS(), 2-153

ALLTRIM(), 2-154, *See also* LTRIM(), RTRIM()

ALTD(), 2-155

Alternate file, 2-853

ANNOUNCE, 2-156

aOldState Structure, 2-492

APPEND BLANK, 2-157

APPEND FROM, 2-158

Appending new record, 2-251

Arithmetic mean, 2-172

Array functions
AADD(), 2-127
ACLONE(), 2-138
ACOPY(), 2-139
ADEL(), 2-141
ADIR(), 2-142
AEVAL(), 2-144
AFIELDS(), 2-146
AFILL(), 2-148
AINS(), 2-150
ARRAY(), 2-162
ASCAN(), 2-165
ASIZE(), 2-167
ASORT(), 2-168
ATAIL(), 2-171
EMPTY(), 2-382
LEN(), 2-563

ARRAY(), 2-162

Arrays
adding new elements, 2-127
and disk directories, 2-143
and field attributes, 2-146
and memory files, 2-814, 2-833
and the macro operator, 2-35
as return values, 2-445, 2-820
as used with ACHOICE(), 2-135
as used with DBEDIT(), 2-271
assigning values to, 2-944
changing the size of, 2-127, 2-167
comparison, 2-73
constant, 2-126
copying elements, 2-139
creating, 2-126, 2-163, 2-580, 2-734, 2-747, 2-940
creating database files from, 2-263, 2-329
deleting elements, 2-141
determining data type of, 2-995, 2-1012
determining number of elements in, 2-563
duplicating subarrays, 2-138
empty, 2-127, 2-382
evaluating code blocks on, 2-144
filling, 2-148
GetList, 2-792
growing, 2-127, 2-167
initializing, 2-580, 2-734, 2-747, 2-940
inserting elements, 2-150
literal, 2-126, 2-141
local, 2-580
maximum number of elements, 2-580, 2-735, 2-747, 2-940
multidimensional, 2-138, 2-139, 2-564, 2-734, 2-747
operators, 2-59, 2-66, 2-73, 2-124, 2-126
passing as parameters, 2-370, 2-718
polygon coordinates, 2-499
private, 2-734
public, 2-747
releasing, 2-803
scanning for a value, 2-165
shrinking, 2-167
sorting, 2-168

static, 2-940
syntax for creating, 2-126
traversing, 2-144, 2-373, 2-432
vertices example, 2-499

ASC(), 2-164

ASCAN(), 2-165
example, 2-145

ASCII code, 2-541

ASCII files
creating from a character string, 2-614
creating from a memo field, 2-614
displaying contents, 2-993
reading, 2-160, 2-607
replacing memo field with, 2-607
writing, 2-237, 2-853

ASIZE(), 2-167

ASORT(), 2-168

Assembly language interface, 2-205

Assignment operators, 2-59, 2-66, 2-68

AT(), 2-170, *See also* RAT()

ATAIL(), 2-171

AVERAGE, 2-172

B

Backup
creating a, 2-517
example, 2-357

BEGIN SEQUENCE, 2-173
and error recovery, 2-387

Beginning of file, 2-200, 2-441

Bell, 2-215, 2-855, 2-977

BIN2I(), 2-176

BIN2L(), 2-177

BIN2W(), 2-178

Binary file, *See also* Low-level file functions
closing a, 2-405
creating a, 2-407
opening a, 2-430
reading a, 2-435, 2-437, 2-441
writing to a, 2-449

Bitmap
adding title to, 2-513
array structure, 2-454
fonts, 2-481
loading, 2-454

Blank values,testing for, 2-382

BLOB
definition, 2-179
drivers supporting, 2-179

BLOB functions
BLOBDIRECTEXPORT(), 2-179
BLOBDIRECTGET(), 2-181
BLOBDIRECTIMPORT(), 2-183
BLOBDIRECTPUT(), 2-186
BLOBEXPORT(), 2-188
BLOBGET(), 2-190
BLOBIMPORT(), 2-192
BLOBROOTGET(), 2-194
BLOBROOTLOCK(), 2-196
BLOBROOTPUT(), 2-197
BLOBROOTUNLOCK(), 2-199

BLOBDIRECTEXPORT(), 2-179

BLOBDIRECTGET(), 2-181

BLOBDIRECTIMPORT(), 2-183

BLOBDIRECTPUT(), 2-186

BLOBEXPORT(), 2-188

BLOBGET(), 2-190

BLOBIMPORT(), 2-192

BLOBROOTGET(), 2-194

BLOBROOTLOCK(), 2-196

BLOBROOTPUT(), 2-197

BLOBROOTUNLOCK(), 2-199

BOF(), 2-200, *See also* EOF()

Boolean algebra, 2-53, 2-54, 2-55

Boundary conditions
  indicating, 2-977
  testing for, 2-201, 2-385

Box drawing, 2-83, 2-122

Box.ch, 2-83

Branching
  conditional, 2-371, 2-520
  unconditional, 2-369

BREAK, 2-173, 2-783

BREAK(), 2-202

Browse
  built-in, 2-204
  user-defined, 2-953
  user-defined, 2-270, 2-958

BROWSE(), 2-203, *See also* DBEDIT()
  implementation, 2-953, 2-958

Button storing example, 2-453, 2-455

C

C interface, 2-205

CA-Clipper return code, setting, 2-395

Calculations
  AVERAGE, 2-172
  COUNT, 2-240
  SUM, 2-952

CALL, 2-205, 2-1017

Calling conventions
  command, 2-718
  function, 2-444, 2-718

CANCEL, See QUIT

CAPTION, 2-87, 2-97, 2-102, 2-106, 2-110

CASE, 2-371

CDOW(), 2-208

Change directory, 2-868, 2-899

Character (escape character), 2-9

Character functions
  ALLTRIM(), 2-154
  ASC(), 2-164
  AT(), 2-170
  CTOD(), 2-246
  EMPTY(), 2-382
  ISALPHA(), 2-544
  ISDIGIT(), 2-546
  ISLOWER(), 2-548
  ISUPPER(), 2-550
  LEFT(), 2-562
  LEN(), 2-563
  LOWER(), 2-587
  LTRIM(), 2-588
  PADC(), 2-716
  PADL(), 2-716
  PADR(), 2-716
  RAT(), 2-777
  REPLICATE(), 2-808
  RIGHT(), 2-822
  RTRIM(), 2-829
  SOUNDEX(), 2-937
  SPACE(), 2-938
  STRTRAN(), 2-947
  STUFF(), 2-948
  SUBSTR(), 2-950
  TRANSFORM(), 2-988
  UPPER(), 2-1003
  VAL(), 2-1011

Character string
  comparison, 2-61, 2-63, 2-64, 2-71, 2-73, 2-75, 2-77, 2-878, 2-1003
  delete and insert characters in, 2-948
  determining length of, 2-563
  extracting a substring, 2-562, 2-777, 2-822, 2-950
  finding a substring in, 2-170, 2-777, 2-947
  maximum length of, 2-515, 2-562, 2-808, 2-822, 2-938, 2-950

operators, 2-29, 2-42, 2-46, 2-59, 2-61, 2-63, 2-64, 2-66, 2-68, 2-71, 2-73, 2-75, 2-77
replicating a, 2-808, 2-938
search and replace a substring in, 2-947
test for alphabetic, 2-544
test for digit, 2-546
test for lowercase, 2-548
test for null, 2-382
test for uppercase, 2-550
writing contents to an ASCII file, 2-614

Character string formatting
converting to lowercase, 2-587
converting to uppercase, 2-1003
extracting a single line, 2-605, 2-629, 2-634
extracting a substring, 2-950
padding with blank spaces, 2-716, 2-938
stripping carriage returns, 2-515, 2-612
trimming blank spaces, 2-154, 2-588, 2-829

CheckBox class
bitmaps, 2-209
buffer, 2-210
capCol, 2-210
capRow, 2-210
caption, 2-210
cargo, 2-211
CheckBox() function, 2-209
col, 2-211
colorSpec, 2-211
display method, 2-213
examples, 2-214
fBlock, 2-211
HasFocus, 2-212
hitTest method, 2-213
killFocus method, 2-213
message, 2-212
MouseCol, 2-213
MouseRow, 2-213
NewState, 2-214
row, 2-212
sBlock, 2-212
select method, 2-214
setFocus method, 2-214
style, 2-212
typeOut, 2-212

CheckBox(), 2-209

CHR(), 2-215

Classes
CheckBox, 2-209
Error, 2-387, 2-393
Get, 2-459, 2-792
ListBox, 2-567
MenuItem, 2-619
PopUpMenu, 2-724
RadioButto, 2-761
RadioGroup, 2-768
Scrollbar, 2-841
TBColumn, 2-953
TBrowse, 2-958
TopBarMenu, 2-979

CLEAR ALL, 2-217

CLEAR GETS, 2-218

CLEAR MEMORY, 2-219

CLEAR SCREEN, 2-220

CLEAR TYPEAHEAD, 2-221

Clipper variable, 2-748

CLIPPER.LIB, 2-995, 2-1012

Clipping region, 2-504

Clock, twenty-four hour, 2-847, 2-976

CLOSE, 2-222, 2-256
CLOSE ALL, 2-222, 2-255
CLOSE ALTERNATE, 2-222
CLOSE DATABASES, 2-222
CLOSE FORMAT, 2-222
CLOSE INDEXES, 2-222

close(), 2-727

Index

Closing files
all, 2-217
before deletion, 2-337, 2-386, 2-409
before renaming, 2-439, 2-804
by type, 2-222
database, 2-590
low-level, 2-405
on program termination, 2-759
with USE, 2-1006

CLS, 2-220

CMONTH(), 2-223

Code block functions
AEVAL(), 2-144
DBEVAL(), 2-276
EMPTY(), 2-382
EVAL(), 2-397

Code blocks
and ERRORBLOCK(), 2-393
and the macro operator, 2-36
assigning to a key, 2-927
compared to macro expressions, 2-36
compiling with macro operator, 2-36
creating, 2-126
displaying data within, 2-757
evaluating, 2-397
evaluating for arrays, 2-144
evaluating for database records, 2-277
operators, 2-59, 2-66, 2-126
printing within, 2-757
set-get for memory variables, 2-617
set-get for fields, 2-415
used to sort arrays, 2-169

COL(), 2-224

COLOR, 2-87

Color
disabling enhanced, 2-889
enhanced, 2-87, 2-98, 2-103, 2-111, 2-133, 2-625, 2-871
obtaining current settings, 2-922
setting codes, 2-922
standard, 2-85, 2-115, 2-117, 2-133
testing for, 2-545
unselected, 2-87, 2-98, 2-103, 2-111, 2-133

Color components, 2-510

Color functions,COLORSELECT(), 2-225

COLORSELECT(), 2-225

Command directives, #command | #translate, 2-1

COMMAND.COM, 2-831

Commands
? | ??, 2-79
@...BOX, 2-83
@...CLEAR, 2-85
@...GET, 2-86
@...GET CHECKBOX, 2-97
@...GET LISTBOX, 2-101
@...GET PUSHBUTTON, 2-106
@...GET RADIOGROUP, 2-110
@...GET TBROWSE, 2-113
@...PROMPT, 2-115
@...SAY, 2-117
@...TO, 2-122
ACCEPT, 2-130
APPEND BLANK, 2-157
APPEND FROM, 2-158
AVERAGE, 2-172
CALL, 2-205
CANCEL, 2-207
CLEAR ALL, 2-217
CLEAR GETS, 2-218
CLEAR MEMORY, 2-219
CLEAR SCREEN, 2-220
CLEAR TYPEAHEAD, 2-221
CLOSE, 2-222
COMMIT, 2-227
CONTINUE, 2-229
COPY FILE, 2-231
COPY STRUCTURE, 2-232
COPY STRUCTURE EXTENDED, 2-234
COPY TO, 2-236
COUNT, 2-240
CREATE, 2-241
CREATE FROM, 2-243

DELETE, 2-335
DELETE FILE, 2-337
DIR, 2-348
DISPLAY, 2-365
EJECT, 2-381
ERASE, 2-386
FIND, 2-424
GO, 2-497
INPUT, 2-542
JOIN, 2-551
KEYBOARD, 2-553
LABEL FORM, 2-556
LIST, 2-565
LOCATE, 2-583
MENU TO, 2-625
NOTE, 2-661
PACK, 2-715
QUIT, 2-759
READ, 2-782
RECALL, 2-797
REINDEX, 2-801
RELEASE, 2-803
RENAME, 2-804
REPLACE, 2-806
REPORT FORM, 2-809
RESTORE, 2-814
RESTORE SCREEN, 2-816
RUN, 2-831
SAVE, 2-833
SAVE SCREEN, 2-835
SEEK, 2-848
SELECT, 2-850
SET ALTERNATE, 2-853
SET BELL, 2-855
SET CENTURY, 2-856
SET COLOR, 2-857
SET CONFIRM, 2-861
SET CONSOLE, 2-862
SET CURSOR, 2-864
SET DATE, 2-865
SET DECIMALS, 2-867
SET DEFAULT, 2-868
SET DELETED, 2-870
SET DELIMITERS, 2-871
SET DESCENDING, 2-873
SET DEVICE, 2-874
SET EPOCH, 2-875
SET ESCAPE, 2-876
SET EVENTMASK, 2-877
SET EXACT, 2-878
SET EXCLUSIVE, 2-880
SET FILTER, 2-882
SET FIXED, 2-883
SET FORMAT, 2-884
SET FUNCTION, 2-886
SET INDEX, 2-887
SET INTENSITY, 2-889
SET KEY, 2-890
SET MARGIN, 2-892
SET MEMOBLOCK, 2-893
SET MESSAGE, 2-894
SET OPTIMIZE, 2-895
SET PATH, 2-898
SET PRINTER, 2-900
SET PROCEDURE, 2-903
SET RELATION, 2-904
SET SCOPE, 2-906
SET SCOPEBOTTOM, 2-907
SET SCOPETOP, 2-908
SET SCOREBOARD, 2-909
SET SOFTSEEK, 2-910
SET TYPEAHEAD, 2-912
SET UNIQUE, 2-913
SET VIDEOMODE, 2-914
SET WRAP, 2-915
SKIP, 2-933
SORT, 2-935
STORE, 2-943
SUM, 2-952
TEXT, 2-974
TOTAL, 2-986
TYPE, 2-993
UNLOCK, 2-997
UPDATE, 2-999
USE, 2-1005
user-defined, 2-277
WAIT, 2-1015
ZAP, 2-1020

COMMIT, 2-227, 2-257, 2-259

Compatibility
- indicator, 1-3
- previous releases, 2-474, 2-792

Compatibility commands
- !, 2-831
- CALL, 2-206
- CANCEL, 2-207
- CLEAR ALL, 2-217
- DECLARE, 2-334
- DIR, 2-348
- DO, 2-369
- FIND, 2-424
- NOTE, 2-661
- RESTORE SCREEN, 2-816
- SAVE SCREEN, 2-835
- SET COLOR, 2-857
- SET EXACT, 2-878
- SET FORMAT, 2-884
- SET PROCEDURE, 2-903
- SET UNIQUE, 2-913
- STORE, 2-943
- WAIT, 2-1015

Compatibility functions
- ADIR(), 2-142
- AFIELDS(), 2-147
- DBEDIT(), 2-270
- DBF(), 2-279
- FKLABEL(), 2-425
- FKMAX(), 2-426
- MOD(), 2-635
- READKEY(), 2-789
- RECCOUNT(), 2-798
- WORD(), 2-1017

Compilation,conditional, 2-18, 2-20

Compiler switches
- /B, 2-581, 2-941
- /D, 2-18
- /I, 2-22
- /L, 2-741
- /M, 2-370
- /N, 2-412, 2-615, 2-941
- /U, 2-26
- /W, 2-412, 2-615

Compiling
- a .prg file, 2-370
- an .fmt file, 2-884

Concurrency control
- and BLOBDIRECTEXPORT(), 2-179
- and BLOBDIRECTIMPORT(), 2-183
- and BLOBEXPORT(), 2-188
- and BLOBIMPORT(), 2-192

Condition evaluation, 2-522, 2-524

Console commands
- ? | ??, 2-79, 2-757
- ACCEPT, 2-130
- and COL(), 2-224
- and HARDCR(), 2-515
- and ROW(), 2-827
- DISPLAY, 2-365
- INPUT, 2-542
- LABEL FORM, 2-556
- LIST, 2-565
- REPORT FORM, 2-809
- SET ALTERNATE, 2-853
- SET CONSOLE, 2-862
- SET MARGIN, 2-892
- SET PRINTER, 2-900
- TEXT...ENDTEXT, 2-974
- TYPE, 2-993
- WAIT, 2-1015

Console functions,QOUT() | QQOUT(), 2-757

Constants
- display mode, 2-457, 2-502
- ellipse style, 2-456
- style mode, 2-502
- video mode, 2-500

CONTINUE, 2-229, 2-583

Control structures
- BEGIN SEQUENCE, 2-173
- decision-making, 2-520
- decision-making, 2-18, 2-20, 2-371
- DO CASE, 2-371
- DO WHILE, 2-373
- error handling, 2-173
- FOR, 2-432

IF, 2-520
looping, 2-373, 2-432
nesting, 2-173, 2-371, 2-373, 2-432, 2-520
preprocessor, 2-18, 2-20

Controls, GUI
check box, 2-97, 2-209
list box, 2-101, 2-567
menu item, 2-619
pop-up menu, 2-724
push button, 2-106
radio button, 2-761
radio button group, 2-110, 2-768
scroll bar, 2-572, 2-841
top bar menu, 2-979

Conventions
language, 1-4, 1-6
manual, 1-4, 1-6

Conversion
ASCII code to character, 2-215
character to ASCII code, 2-164
character to date, 2-246
character to numeric, 2-1011
character to soundex, 2-937
date to ANSI string, 2-379
date to character, 2-208, 2-223, 2-378
date to numeric, 2-250, 2-377, 2-637, 2-1019
double to integer, 2-1017
integer to binary, 2-519
integer to numeric, 2-176, 2-177
numeric to binary, 2-555
numeric to character, 2-945
numeric to integer, 2-543
real to integer, 2-543
to invert, 2-341
to lowercase, 2-587
to uppercase, 2-1003
unsigned integer to numeric, 2-178

Conversion functions
ASC(), 2-164
BIN2I(), 2-176
BIN2L(), 2-177
BIN2W(), 2-178
CDOW(), 2-208
CHR(), 2-215
CMONTH(), 2-223
CTOD(), 2-246
DAY(), 2-250
DESCEND(), 2-341
DOW(), 2-377
DTOC(), 2-378
DTOS(), 2-379
HARDCR(), 2-515
I2BIN(), 2-519
IF(), 2-522
IIF(), 2-524
L2BIN(), 2-555
MEMOTRAN(), 2-612
MONTH(), 2-637
SOUNDEX(), 2-937
STR(), 2-945
TRANSFORM(), 2-988
VAL(), 2-1011
WORD(), 2-1017
YEAR(), 2-1019

COPY, 2-357
COPY FILE, 2-231
COPY STRUCTURE, 2-232
COPY STRUCTURE EXTENDED, 2-234
COPY TO, 2-236

COPY FILE, 2-231

COPY STRUCTURE, 2-232

COPY STRUCTURE EXTENDED, 2-234

COPY TO, 2-236

COUNT, 2-240

CREATE, 2-241

CREATE FROM, 2-243

Index

Creating
polygon, 2-498
specified directories, 2-353

CTOD(), 2-246

CURDIR(), 2-248

D

Data dictionary, 2-244

Data display
?|??, 2-79
@...SAY, 2-117
BROWSE(), 2-204
DBEDIT(), 2-270
DISPLAY, 2-365
LIST, 2-565
SET INTENSITY, 2-889
TRANSFORM(), 2-990

Data entry
@...GET, 2-86, 2-783
ACCEPT, 2-130
APPEND BLANK, 2-157
BROWSE(), 2-204
controlling Esc during, 2-876
controlling exit keys during, 2-786
controlling insert mode during, 2-788
DBEDIT(), 2-270
example, 2-382
INPUT, 2-542
MEMOEDIT(), 2-599
navigation keys, 2-783
obtaining help during, 2-795
SET BELL, 2-855
SET CONFIRM, 2-861
SET CURSOR, 2-864
SET DELIMITERS, 2-871
SET FORMAT, 2-884
WAIT, 2-1015

Data fields,information about, 2-280

Data type
array, 2-59, 2-66, 2-73, 2-124, 2-126
character, 2-29, 2-42, 2-46, 2-59, 2-61, 2-63, 2-64, 2-66, 2-71, 2-73, 2-75, 2-77
checking, 2-994, 2-1012
code block, 2-59, 2-66, 2-126
date, 2-42, 2-44, 2-46, 2-48, 2-59, 2-61, 2-63, 2-64, 2-66, 2-71, 2-73, 2-75, 2-77
logical, 2-53, 2-54, 2-55, 2-59, 2-61, 2-63, 2-64, 2-66, 2-71, 2-73, 2-75, 2-77
memo, 2-29, 2-42, 2-46, 2-59, 2-61, 2-63, 2-64, 2-66, 2-71, 2-73, 2-75, 2-77
NIL, 2-59, 2-64, 2-66, 2-71, 2-73
numeric, 2-30, 2-40, 2-41, 2-42, 2-44, 2-46, 2-48, 2-56, 2-59, 2-61, 2-63, 2-64, 2-66, 2-71, 2-73, 2-75, 2-77
object, 2-57, 2-73, 2-387, 2-459, 2-953, 2-958

Data validation
example, 2-382, 2-447
RANGE, 2-90, 2-247, 2-783, 2-876
TYPE(), 2-994
VALID, 2-89, 2-558, 2-783, 2-861, 2-1001
VALTYPE(), 2-1012

Database commands
APPEND BLANK, 2-157
APPEND FROM, 2-159
AVERAGE, 2-172
COMMIT, 2-227
CONTINUE, 2-229
COPY STRUCTURE, 2-232
COPY STRUCTURE EXTENDED, 2-234
COPY TO, 2-237
COUNT, 2-240
CREATE, 2-241
CREATE FROM, 2-243
DELETE, 2-335
DISPLAY, 2-365
GO, 2-497
JOIN, 2-551
LABEL FORM, 2-556
LIST, 2-565
LOCATE, 2-583
PACK, 2-715
RECALL, 2-797

REPLACE, 2-806
REPORT FORM, 2-809
SEEK, 2-848
SET DELETED, 2-870
SET DESCENDING, 2-873
SET EXCLUSIVE, 2-880
SET FILTER, 2-882
SET MEMOBLOCK, 2-893
SET OPTIMIZE, 2-895
SET RELATION, 2-904
SET SCOPE, 2-906
SET SCOPEBOTTOM, 2-907
SET SCOPETOP, 2-908
SET SOFTSEEK, 2-910
SET UNIQUE, 2-913
SKIP, 2-933
SORT, 2-935
SUM, 2-952
TOTAL, 2-986
UPDATE, 2-999
USE, 2-1005
ZAP, 2-1020

Database drivers, 2-610
changing, 2-319
DBFCDX, 2-610
DBFMEMO, 2-610
DBFNTX, 2-610
determining current, 2-319
identifying available RDDs, 2-778, 2-780
RDDs, 2-159, 2-237, 2-244, 2-338, 2-526
setting default RDD, 2-781

Database files
calculating size of, 2-517, 2-560, 2-800
closing, 2-590, 2-1006
creating, 2-611, 2-935, 2-986
creating from an array, 2-262, 2-328
date of last update, 2-590
determining if open, 2-1010
field names, 2-417
header length, 2-517
information about, 2-282, 2-284, 2-293
mass updating, 2-428, 2-999
maximum number in a relation, 2-905
number of fields in, 2-406
number of records in, 2-560
obtaining name, 2-279
opening, 2-332, 2-1006
processing sequentially, 2-385, 2-406
processing with DBEVAL(), 2-276
record pointer identity, 2-799
relating, 2-308, 2-313, 2-324, 2-904
removing all records, 2-1020
searching, 2-669, 2-848
size of record, 2-800
sorting, 2-935
summarizing records, 2-986
totalling, 2-952, 2-986
traversing, 2-373
updating, 2-204, 2-270, 2-428, 2-599, 2-823, 2-999

Database functions
ALIAS(), 2-153
BLOBDIRECTEXPORT(), 2-179
BLOBDIRECTGET(), 2-181
BLOBDIRECTIMPORT(), 2-183
BLOBDIRECTPUT(), 2-186
BLOBEXPORT(), 2-188
BLOBGET(), 2-190
BLOBIMPORT(), 2-192
BLOBROOTGET(), 2-194
BLOBROOTLOCK(), 2-196
BLOBROOTPUT(), 2-197
BLOBROOTUNLOCK(), 2-199
BOF(), 2-200
DBAPPEND(), 2-251
DBCLEARFILTER(), 2-252
DBCLEARINDEX(), 2-253
DBCLEARRELATION(), 2-254
DBCLOSEALL(), 2-255
DBCLOSEAREA(), 2-256
DBCOMMIT(), 2-257
DBCOMMITALL(), 2-259
DBCREATE(), 2-261
DBCREATEINDEX(), 2-264
DBDELETE(), 2-266
DBEDIT(), 2-268
DBEVAL(), 2-276
DBF(), 2-279
DBFIELDINFO(), 2-280
DBFILEGET(), 2-282
DBFILEPUT(), 2-284

DBFILTER(), 2-286
DBGOBOTTOM(), 2-288
DBGOTO(), 2-290
DBGOTOP(), 2-292
DBINFO(), 2-293
DBORDERINFO(), 2-297
DBRECALL(), 2-302
DBRECORDINFO(), 2-304
DBREINDEX(), 2-306
DBRELATION(), 2-307
DBRSELECT(), 2-312
DBSEEK(), 2-315
DBSELECTAREA(), 2-317
DBSETDRIVER(), 2-319
DBSETFILTER(), 2-320
DBSETORDER(), 2-323
DBSETRELATION(), 2-324
DBSKIP(), 2-326
DBSTRUCT(), 2-328
DBUNLOCK(), 2-330
DBUNLOCKALL(), 2-331
DBUSEAREA(), 2-332
DELETED(), 2-340
EOF(), 2-384
FCOUNT(), 2-406
FIELDBLOCK(), 2-414
FIELDGET(), 2-416
FIELDNAME(), 2-417
FIELDPOS(), 2-419
FIELDPUT(), 2-420
FIELDWBLOCK(), 2-421
FLOCK(), 2-427
FOUND(), 2-433
HEADER(), 2-517
INDEXKEY(), 2-533
INDEXORD(), 2-536
LASTREC(), 2-560
LUPDATE(), 2-590
MEMOEDIT(), 2-599
ORDCONDSET(), 2-669
RECNO(), 2-799
RECSIZE(), 2-800
RLOCK(), 2-823
SELECT(), 2-852
USED(), 2-1010

Database structure
  copying, 2-232
  loading into an array, 2-147, 2-262, 2-328

Database, inheriting characteristics, 2-610

Date commands
  SET CENTURY, 2-856
  SET DATE, 2-865
  SET EPOCH, 2-875

Date formatting
  control century display, 2-856
  control default century, 2-875
  convert to day name, 2-208
  convert to day of month number, 2-250
  convert to day of week number, 2-377
  convert to month name, 2-223
  convert to month number, 2-637
  convert to year number, 2-1019
  define display format, 2-865
  with character strings, 2-378

Date functions
  CDOW(), 2-208
  CMONTH(), 2-223
  DATE(), 2-249
  DAY(), 2-250
  DOW(), 2-377
  DTOC(), 2-378
  DTOS(), 2-379
  EMPTY(), 2-382
  MAX(), 2-591
  MIN(), 2-628
  MONTH(), 2-637
  TRANSFORM(), 2-988
  YEAR(), 2-1019

DATE(), 2-249,

Dates
  blank, 2-246, 2-377, 2-378, 2-379, 2-382, 2-590, 2-637, 2-1019
  calculations with, 2-250, 2-377, 2-637, 2-1019
  comparison, 2-61, 2-63, 2-64, 2-71, 2-73, 2-75, 2-77
  leap year, 2-250
  literal, 2-246

maximum of two, 2-591
minimum of two, 2-628
operators, 2-42, 2-44, 2-46, 2-48, 2-59, 2-61, 2-63, 2-64, 2-66, 2-68, 2-71, 2-73, 2-75, 2-77
range of, 2-856, 2-865, 2-875
used with RANGE, 2-247

Day of week
character, 2-208
numeric, 2-377

DAY(), 2-250

DBAPPEND(), 2-251

dBASE III PLUS compatibility
and PUBLIC Clipper, 2-748
DBF(), 2-279
DO, 2-370
FKLABEL(), 2-425
FKMAX(), 2-426
for BROWSE command, 2-203
indexing, 2-532, 2-539
LABEL FORM, 2-556
MOD(), 2-635
PRIVATE, 2-735
READKEY(), 2-789
REPORT FORM, 2-810
RETURN TO MASTER, 2-173, 2-820
RLOCK(), 2-824
SET FORMAT, 2-884
SET PRINTER, 2-901
SET PROCEDURE, 2-903
UNLOCK, 2-997

dBASE III PLUS, database driver, 2-532, 2-539

DBCLEARFILTER(), 2-252

DBCLEARINDEX(), 2-253

DBCLEARRELATION(), 2-254

DBCLOSEALL(), 2-255

DBCLOSEAREA(), 2-256

DBCOMMIT(), 2-257

DBCOMMITALL(), 2-259

DBCREATE(), 2-261

DBCREATEINDEX(), 2-264

DBDELETE(), 2-266

DBEDIT(), 2-268,
implementation, 2-953, 2-958

Dbedit.ch, 2-270

DBEVAL(), 2-276, 2-757

DBF(), 2-279,

DBFIELDINFO(), 2-280

DBFILEGET(), 2-282

DBFILEPUT(), 2-284

DBFILTER(), 2-286

DBFMEMO driver, 2-610

DBGOBOTTOM(), 2-288

DBGOTO(), 2-290

DBGOTOP(), 2-292

DBINFO(), 2-293

Dbinfo.ch, 2-282, 2-284, 2-293, 2-297, 2-304

DBORDERINFO(), 2-297

DBRECALL(), 2-302

DBRECORDINFO(), 2-304

DBREINDEX(), 2-306

DBRELATION(), 2-307

DBRLOCK(), 2-309

DBRLOCKLIST(), 2-311

DBRSELECT(), 2-312

DBSEEK(), 2-315

DBSELECTAREA(), 2-317

DBSETDRIVER(), 2-319

DBSETFILTER(), 2-320

DBSETINDEX(), 2-322

DBSETORDER(), 2-323

DBSETRELATION(), 2-324

DBSKIP(), 2-326

DBSTRUCT(), 2-328

Dbstruct.ch, 2-261, 2-280, 2-328

DBUNLOCK(), 2-314, 2-330

DBUNLOCKALL(), 2-331

DBUSEAREA(), 2-332

Debugger, 2-155, 2-581, 2-941

Debugging, 2-741, 2-743, 2-795

Decimal display, 2-867, 2-883

Declaration statements
- EXTERNAL, 2-403
- FIELD, 2-412
- FUNCTION, 2-443
- LOCAL, 2-580
- MEMVAR, 2-615
- PROCEDURE, 2-736
- STATIC, 2-940

Declarations
- compile-time, 2-615
- compile-time, 2-287, 2-307, 2-412, 2-581, 2-940

DECLARE, See PRIVATE

Default drive, 2-423, 2-868, 2-898

DELETE, 2-266, 2-335, 2-1020

DELETE FILE, 2-337,

DELETE TAG, 2-338

Deleted records
- deleting all records, 2-1020
- detecting status of, 2-340
- filtering, 2-340, 2-870
- ignoring, 2-340, 2-870
- marking, 2-266, 2-336
- processing, 2-340, 2-870
- reinstating, 2-302, 2-797
- removing, 2-715
- sorting, 2-936
- updating, 2-1000

DELETED(), 2-340,

Deleting,sub-directories, 2-354

Delimited files
- reading, 2-160
- writing, 2-238

delItem(), 2-572, 2-728, 2-772, 2-981

DESCEND(), 2-341

DEVOUT(), 2-343

DEVOUTPICT(), 2-344

DEVPOS(), 2-346

DIR, 2-348,

DIRCHANGE(), 2-350

Directives
- #command | #translate, 2-1
- #define, 2-12, 2-26
- #else, 2-18, 2-20
- #endif, 2-18, 2-20
- #error, 2-17
- #ifdef, 2-18
- #ifndef, 2-20
- #include, 2-22
- #stdout, 2-25
- #undef, 2-26
- #xcommand, 2-28
- #xtranslate, 2-28

Directories
- changing of, 2-350
- creating directory, 2-353
- removing of, 2-354

Directory functions
- DIRCHANGE(), 2-350
- DIRMAKE(), 2-353
- DIRREMOVE(), 2-354

DIRECTORY(), 2-351

Directry.ch, 2-145, 2-351

DIRMAKE(), 2-353

DIRREMOVE(), 2-354

Disk directory, 2-143, 2-348, 2-354, 2-355, 2-356, 2-547

Disk drive
- available space, 2-357
- changing current drive, 2-355

Disk functions
- DISKCHANGE(), 2-355
- DISKNAME(), 2-356
- ISDISK(), 2-547

DISKCHANGE(), 2-355

DISKNAME(), 2-356

DISKSPACE(), 2-357, 2-517

DISPBEGIN(), 2-358, 2-507

DISPBOX(), 2-360

DISPCOUNT(), 2-363
- determining display context, 2-363
- pending screen refresh requests, 2-363

DISPEND(), 2-364, 2-507
- buffering display updates, 2-364
- determining display context, 2-364

DISPLAY, 2-365

Display
- BMP file, 2-451
- limiting, 2-504

Display color, 2-456, 2-457, 2-510, 2-512

Display(), 2-845

display(), 2-213, 2-728, 2-754, 2-765, 2-772, 2-981

DISPOUT(), 2-367

DO, 2-369

DO CASE, 2-371,

DO WHILE, 2-373, , 2-384

DOS
- accessing from a CA-Clipper program, 2-831
- changing the prompt, 2-831
- controlling output device, 2-901
- current directory, 2-248
- current drive,returning, 2-356
- default directory, 2-352, 2-408, 2-430, 2-607
- default drive, 2-248, 2-357
- directory, 2-350
- environment variables, 2-22, 2-474, 2-866, 2-899, 2-1008
- error code, 2-390
- error number, 2-375
- ERRORLEVEL, 2-395
- file attributes, 2-143, 2-407, 2-423, 2-430
- file handles, 2-935
- file pointer, 2-435, 2-437, 2-441
- flushing buffers, 2-405, 2-933
- number of open files, 2-1006
- open mode, 2-430
- passing parameters from command line, 2-718
- path, 2-248, 2-607
- return code, 2-395, 2-405, 2-410, 2-429, 2-759
- returning control to, 2-820
- running commands, 2-831
- SET command, 2-474
- system date, 2-249
- system time, 2-847, 2-976
- testing print device, 2-549
- version name, 2-712

DOSERROR(), 2-375, 2-390

Double-click
- sensitivity, 2-595
- speed threshold, 2-595

DOW(), 2-377

DROPDOWN, 2-103

DTOC(), 2-378

DTOS(), 2-379

E

Edit fields, See @...GET

EJECT, 2-381

ELSE, 2-520

ELSEIF, 2-520

EMPTY(), 2-382

END, 2-173
- ENDCASE, 2-371
- ENDDO, 2-373
- ENDIF, 2-520
- ENDTEXT, 2-974

End of file, 2-384, 2-435, 2-441

ENDCASE, 2-371

ENDDO, 2-373

ENDIF, 2-520

ENDTEXT, 2-974

Environment commands
- SET BELL, 2-855
- SET DATE, 2-865
- SET DECIMALS, 2-867
- SET DEFAULT, 2-868
- SET DEVICE, 2-874
- SET EPOCH, 2-875
- SET EXACT, 2-878
- SET FIXED, 2-883
- SET PATH, 2-898

Environment functions,
- CURDIR(), 2-248
- DEVOUT(), 2-343
- DEVOUTPICT(), 2-344
- DEVPOS(), 2-346
- DIRECTORY(), 2-351
- DISKSPACE(), 2-357
- DISPOUT(), 2-367
- DOSERROR(), 2-375
- ERRORLEVEL(), 2-395
- GETACTIVE(), 2-470
- GETENV(), 2-474
- MEMORY(), 2-609
- NOSNOW(), 2-660
- OS(), 2-712
- PCOL(), 2-720
- PROW(), 2-745
- READEXIT(), 2-786
- READINSERT(), 2-788
- READVAR(), 2-795
- SETBLINK(), 2-919
- SETCURSOR(), 2-925
- SETMODE(), 2-929
- SETPOS(), 2-930

Environment variables
- CLIPPER, 2-1008
- INCLUDE, 2-22

EOF(), 2-384, , 2-910

ERASE, 2-386,

Error class, 2-387
- args, 2-387
- canDefault, 2-388
- canRetry, 2-388, 2-391
- canSubstitute, 2-388
- cargo, 2-388
- description, 2-388
- examples, 2-391
- filename, 2-389
- genCode, 2-388, 2-389
- operation, 2-389
- osCode, 2-390
- severity, 2-390
- subCode, 2-390
- subSystem, 2-390
- tries, 2-391

Error functions
- DOSERROR(), 2-375
- OUTERR(), 2-713

Error handling
- #error, 2-17
- ALERT(), 2-151
- BREAK(), 2-202
- error handling blocks, 2-393
- error objects, 2-393

ERRORBLOCK(), 2-393
file open, 2-881, 2-1008
general, 2-977
local error recovery, 2-174
low-level, 2-375, 2-405, 2-410, 2-430, 2-435, 2-449
network, 2-655
printer, 2-549
runtime, 2-387, 2-393, 2-741, 2-743

ERRORBLOCK(), 2-387, 2-393,

ERRORLEVEL(), 2-395, 2-759

ErrorNew(), 2-387

Exclusion areas, 2-507

Exclusive mode, 2-427, 2-823, 2-880, 2-1006, 2-1020

EXIT, 2-373, 2-431

EXIT PROCEDURE, 2-399

EXP(), 2-402, , 2-867

Exponent,maximum value of, 2-402

Exponentiation, 2-402

Export, 2-237, 2-406, 2-614

Expressions
determining data type of, 2-994, 2-1012
metasymbols used for, 1-5

EXTEND.LIB, 2-995, 2-1012

EXTERNAL, 2-403, 2-515, 2-612
and macro expressions, 2-37
used in header files, 2-22

F

FCLOSE(), 2-405

FCOUNT(), 2-406

FCREATE(), 2-407

FERASE(), 2-409,

FERROR(), 2-410

FIELD, 2-412

FIELD alias, 2-51

FIELDBLOCK(), 2-414

FIELDGET(), 2-416

FIELDNAME(), 2-417

FIELDPOS(), 2-419

FIELDPUT(), 2-420

Fields
attributes of, 2-417
changing value of, 2-806
count, 2-406
declaring, 2-412
editing, 2-86, 2-204, 2-270, 2-599
names of, 2-417
number of, 2-406
picklist, 2-406, 2-418
picklist example, 2-147, 2-891
total, 2-952, 2-986

FIELDWBLOCK(), 2-421

File
deleting from disk, 2-337, 2-386, 2-409
picklist example, 2-143
renaming, 2-439, 2-804
testing for the existence of, 2-423

File commands
DELETE FILE, 2-337
DIR, 2-348
ERASE, 2-386
RENAME, 2-804

Index

SET DEFAULT, 2-868
SET PATH, 2-898
TYPE, 2-993

File functions
ADIR(), 2-142
CURDIR(), 2-248
DIRECTORY(), 2-351
DISKSPACE(), 2-357
FERASE(), 2-409
FILE(), 2-423
FRENAME(), 2-439

File handle
accessing a, 2-435, 2-437, 2-441, 2-449
number available, 2-407, 2-430, 2-437
obtaining a, 2-407, 2-430
releasing a, 2-405

File locking
APPEND BLANK, 2-157
APPEND FROM, 2-159
DELETE, 2-335
FLOCK(), 2-427, 2-823
PACK, 2-715
RECALL, 2-797
REPLACE, 2-806
SET EXCLUSIVE, 2-880
SORT, 2-935
UNLOCK, 2-997
UPDATE, 2-999
USE, 2-1008

FILE(), 2-423

Fileio.ch, 2-407, 2-429, 2-441

Filename, extraction example, 2-950

Filters, optimizing, 2-895

FIND, See SEEK

findText(), 2-573

FKLABEL(), 2-425

FKMAX(), 2-426

FLOCK(), 2-427, *See also* RLOCK(), UNLOCK

Font
.FND, 2-480
erasing from memory, 2-480
loaded, 2-480
Video ROM, 2-481
Windows bitmap, 2-481

Font file loading example, 2-480

Fonts
.FNT, 2-481
clipping information, 2-483
displaying in lines, 2-484
file loading example, 2-482
file loading example(), 2-485
setting loaded font, 2-483
size calculation chart, 2-494
size of, 2-481
types, 2-481

FOPEN(), 2-429

FOR, 2-431, *See also* DO WHILE

Format files
multiple page, 2-884
using, 2-783, 2-884

Formatting
blocks of text, 2-974
data entry screens, 2-86, 2-97, 2-101, 2-106, 2-110, 2-113, 2-117, 2-224, 2-592, 2-593, 2-827, 2-884
logical data, 2-523, 2-525
numeric and character output, 2-945
printed output, 2-720, 2-745
with PICTURE clauses, 2-988

Formatting functions, TRANSFORM(), 2-988

Formfeed, 2-381

FOUND(), 2-229, 2-433, 2-584, 2-910

Frame,defining coordinates, 2-486

FREAD(), 2-176, 2-177, 2-178, 2-435, 2-519

FREADSTR(), 2-437

FRENAME(), 2-439, *See also* RENAME

FSEEK(), 2-441

Full-screen commands
@...BOX, 2-83
@...CLEAR, 2-85
@...GET, 2-86
@...PROMPT, 2-115, 2-625
@...SAY, 2-117
@...TO, 2-122
and COL(), 2-224
and MAXCOL(), 2-592
and MAXROW(), 2-593
and ROW(), 2-827
READ, 2-782
SET DEVICE, 2-874

Full-screen editing,navigation keys, 2-783

FUNCTION, 2-443, , 2-820

Function keys
and WAIT, 2-1015
names of, 2-425
number available, 2-425, 2-426, 2-886
programming, 2-886, 2-890

Functions
AADD(), 2-127
ABS(), 2-129
ACHOICE(), 2-131
ACLONE(), 2-138
ACOPY(), 2-139
ADEL(), 2-141
ADIR(), 2-142
AEVAL(), 2-144
AFIELDS(), 2-146
AFILL(), 2-148
AINS(), 2-150
ALERT(), 2-151
ALIAS(), 2-153
ALLTRIM(), 2-154
ALTD(), 2-155
ARRAY(), 2-162
ASC(), 2-164
ASCAN(), 2-165
ASIZE(), 2-167
ASORT(), 2-168
AT(), 2-170
ATAIL(), 2-171
BIN2I(), 2-176
BIN2L(), 2-177
BIN2W(), 2-178
BLOBDIRECTEXPORT(), 2-179
BLOBDIRECTGET(), 2-181
BLOBDIRECTIMPORT(), 2-183
BLOBDIRECTPUT(), 2-186
BLOBEXPORT(), 2-188
BOF(), 2-200
BREAK(), 2-202
BROWSE(), 2-203
CDOW(), 2-208
CHR(), 2-215
CMONTH(), 2-223
COL(), 2-224
COLORSELECT(), 2-225
CTOD(), 2-246
CURDIR(), 2-248
DATE(), 2-249
DAY(), 2-250
DBAPPEND(), 2-251
DBCLEARFILTER(), 2-252
DBCLEARINDEX(), 2-253
DBCLEARRELATION(), 2-254
DBCLOSEALL(), 2-255
DBCLOSEAREA(), 2-256
DBCOMMIT(), 2-257
DBCOMMITALL(), 2-259
DBCREATE(), 2-261
DBCREATEINDEX(), 2-264
DBDELETE(), 2-266
DBEDIT(), 2-268
DBEVAL(), 2-276
DBF(), 2-279
DBFIELDINFO(), 2-280
DBFILEGET(), 2-282
DBFILEPUT(), 2-284
DBFILTER(), 2-286
DBGOBOTTOM(), 2-288
DBGOTO(), 2-290
DBGOTOP(), 2-292
DBINFO(), 2-293
DBORDERINFO(), 2-297
DBRECALL(), 2-302
DBRECORDINFO(), 2-304
DBREINDEX(), 2-306
DBRELATION(), 2-307

DBRSELECT(), 2-312
DBSEEK(), 2-315
DBSELECTAREA(), 2-317
DBSETDRIVER(), 2-319
DBSETFILTER(), 2-320
DBSETORDER(), 2-323
DBSETRELATION(), 2-324
DBSKIP(), 2-326
DBSTRUCT(), 2-328
DBUNLOCK(), 2-330
DBUNLOCKALL(), 2-331
DBUSEAREA(), 2-332
DELETED(), 2-340
DESCEND(), 2-341
DEVOUT(), 2-343
DEVOUTPICT(), 2-344
DEVPOS(), 2-346
DirChange(), 2-350
DIRECTORY(), 2-351
DIRMAKE, 2-353
DIRREMOVE(), 2-354
DISKCHANGE(), 2-355
DISKNAME(), 2-356
DISKSPACE(), 2-357
DISPBEGIN(), 2-358
DISPBOX(), 2-360
DISPCOUNT(), 2-363
DISPEND(), 2-364
DISPOUT(), 2-367
DOSERROR(), 2-375
DOW(), 2-377
DTOC(), 2-378
DTOS(), 2-379
EMPTY(), 2-382
EOF(), 2-384
ERRORBLOCK(), 2-393
ERRORLEVEL(), 2-395
EVAL(), 2-397
EXP(), 2-402
FCLOSE(), 2-405
FCOUNT(), 2-406
FCREATE(), 2-407
FERASE(), 2-409
FERROR(), 2-410
FIELD(), 2-417
FIELDBLOCK(), 2-414
FIELDGET(), 2-416
FIELDNAME(), 2-417
FIELDPOS(), 2-419
FIELDPUT(), 2-420
FIELDWBLOCK(), 2-421
FILE(), 2-423
FKLABEL(), 2-425
FKMAX(), 2-426
FLOCK(), 2-427
FOPEN(), 2-429
FOUND(), 2-433
FREAD(), 2-435
FREADSTR(), 2-437
FRENAME(), 2-439
FSEEK(), 2-441
FWRITE(), 2-449
GBMPDISP(), 2-451
GBMPLOAD(), 2-454
GELLIPSE(), 2-456
GETACTIVE(), 2-792
GETAPPLYKEY(), 2-792
GETDOSETKEY(), 2-792
GETENV(), 2-474
GETPOSTVALIDATE(), 2-792
GETPREVALIDATE(), 2-792
GETREADER(), 2-792
GFNTERASE(), 2-480
GFNTLOAD(), 2-481
GFNTSET(), 2-483
GFRAME(), 2-486
GGETPIXEL(), 2-489
GLINE(), 2-490
GMODE(), 2-492
GPOLYGON(), 2-498
GPUTPIXEL(), 2-500
GRECT(), 2-502
GSETCLIP(), 2-504
GSETEXCL(), 2-507
GSETPAL(), 2-510
GWRITEAT(), 2-512
HARDCR(), 2-515
HEADER(), 2-517
I2BIN(), 2-519
IF(), 2-522
IIF(), 2-524
INDEXEXT(), 2-532

INDEXKEY(), 2-533
INDEXORD(), 2-536
INKEY(), 2-540
INT(), 2-543
ISALPHA(), 2-544
ISCOLOR(), 2-545
ISDIGIT(), 2-546
ISDISK(), 2-547
ISLOWER(), 2-548
ISPRINTER(), 2-549
ISUPPER(), 2-550
L2BIN(), 2-555
LASTKEY(), 2-558
LASTREC(), 2-560
LEFT(), 2-562
LEN(), 2-563
LOG(), 2-585
LOWER(), 2-587
LTRIM(), 2-588
LUPDATE(), 2-590
MAX(), 2-591
MAXCOL(), 2-592
MAXROW(), 2-593
MCOL(), 2-594
MDBLCLK(), 2-595
MEMOEDIT(), 2-596
MEMOLINE(), 2-605
MEMOREAD(), 2-607
MEMORY(), 2-609
MEMOSETSUPER(), 2-610
MEMOTRAN(), 2-612
MEMOWRIT(), 2-614
MEMVARBLOCK(), 2-617
MENUMODAL(), 2-623
MHIDE(), 2-627
MIN(), 2-628
MLCOUNT(), 2-629
MLCTOPOS(), 2-631
MLEFTDOWN(), 2-633
MLPOS(), 2-634
MOD(), 2-635
MONTH(), 2-637
MPOSTOLC(), 2-638
MPRESENT(), 2-640
MRESTSTATE(), 2-641
MRIGHTDOWN(), 2-642
MROW(), 2-643
MSAVESTATE(), 2-644
MSETBOUNDS(), 2-645
MSETCLIP(), 2-646
MSETCURSOR(), 2-648
MSETPOS(), 2-649
MSHOW(), 2-650
MSTATE(), 2-652
NETERR(), 2-655
NETNAME(), 2-657
NEXTKEY(), 2-658
NOSNOW(), 2-660
ORDBAGEXT(), 2-663
ORDBAGNAME(), 2-664
ORDCOND(), 2-666
ORDCONDSET(), 2-669
ORDCREATE(), 2-673
ORDDESCEND(), 2-675
ORDDESTROY(), 2-677
ORDFOR(), 2-678
ORDISUNIQUE(), 2-680
ORDKEY(), 2-682
ORDKEYADD(), 2-684
ORDKEYCOUNT(), 2-686
ORDKEYDEL(), 2-688
ORDKEYGOTO(), 2-691
ORDKEYNO(), 2-693
ORDKEYVAL(), 2-695
ORDLISTADD(), 2-697
ORDLISTCLEAR(), 2-699
ORDLISTREBUILD(), 2-700
ORDNAME(), 2-701
ORDNUMBER(), 2-703
ORDSCOPE(), 2-704
ORDSETFOCUS(), 2-706
ORDSETRELATION(), 2-708
ORDSKIPUNIQUE(), 2-710
OS(), 2-712
OUTERR(), 2-713
OUTSTD(), 2-714
PAD(), 2-716
PCOL(), 2-720
PCOUNT(), 2-722
PROCLINE(), 2-741
PROCNAME(), 2-743
PROW(), 2-745

QOUT(), 2-757
RAT(), 2-777
RDDLIST(), 2-778
RDDNAME(), 2-780
RDDSETDEFAULT(), 2-781
READEXIT(), 2-786
READINSERT(), 2-788
READKEY(), 2-789
READMODAL(), 2-792
READVAR(), 2-795
RECCOUNT(), 2-798
RECNO(), 2-799
RECSIZE(), 2-800
REPLICATE(), 2-808
RESTSCREEN(), 2-818
RIGHT(), 2-822
RLOCK(), 2-823
ROUND(), 2-825
ROW(), 2-827
RTRIM(), 2-829
SAVESCREEN(), 2-837
SCROLL(), 2-839
SECONDS(), 2-847
SELECT(), 2-852
SET(), 2-916
SETBLINK(), 2-919
SETCANCEL(), 2-920
SETCOLOR(), 2-922
SETCURSOR(), 2-925
SETKEY(), 2-927
SETMODE(), 2-929
SETPOS(), 2-930
SETPRC(), 2-931
SOUNDEX(), 2-937
SPACE(), 2-938
SQRT(), 2-939
STR(), 2-945
STRTRAN(), 2-947
STUFF(), 2-948
SUBSTR(), 2-950
TIME(), 2-976
TONE(), 2-977
TRANSFORM(), 2-988
TRIM(), 2-991
TYPE(), 2-994
UPDATED(), 2-1001
UPPER(), 2-1003
USED(), 2-1010
VAL(), 2-1011
VALTYPE(), 2-1012
VERSION(), 2-1014
WORD(), 2-1017
YEAR(), 2-1019

FWRITE(), 2-449, 2-555

G

GBMPDISP(), 2-451

GBMPLOAD(), 2-454

GELLIPSE(), 2-456

Get class, 2-459
assign method, 2-465, 2-466
backspace method, 2-468
badDate, 2-460
block, 2-459, 2-460, 2-465
buffer, 2-460, 2-465, 2-466
cargo, 2-460
changed, 2-461
clear, 2-461
col, 2-459, 2-461
colorDisp method, 2-465
colorSpec, 2-459, 2-461, 2-465
cursor movement methods, 2-467
decPos, 2-461, 2-466
delEnd method, 2-468
delete method, 2-468
delLeft method, 2-468
delRight method, 2-468
delWordLeft method, 2-468
display method, 2-465
editing methods, 2-468
end method, 2-467
examples, 2-469
exitState, 2-462
hasFocus, 2-462
hittest method, 2-466
home method, 2-467
insert method, 2-464, 2-469

killFocus method, 2-466
left method, 2-467
message, 2-462
methods, 2-465
minus, 2-462
name, 2-463
original, 2-463, 2-466
overStrike method, 2-464, 2-469
picture, 2-459, 2-463
picture method, 2-461
pos, 2-463, 2-466
postBlock, 2-463
preBlock, 2-464
reader, 2-464
rejected, 2-464
reset method, 2-466
right method, 2-467
row, 2-459, 2-464
setFocus method, 2-461, 2-466
subscript, 2-464
text entry methods, 2-469
toDecPos method, 2-468
type, 2-465
typeOut, 2-465
undo method, 2-461, 2-463, 2-466
unTransform method, 2-466
updateBuffer method, 2-467
varGet method, 2-467
varPut method, 2-467
varput method, 2-466
wordLeft method, 2-468
wordRight method, 2-468

Get system
Getsys.prg, 2-470, 2-471, 2-473, 2-476, 2-477, 2-478, 2-792

Get system commands
@...GET, 2-86, 2-459
@...GET CHECKBOX, 2-97
@...GET LISTBOX, 2-101
@...GET PUSHBUTTON, 2-106
@...GET RADIOGROUP, 2-110
@...GET TBROWSE, 2-113
CLEAR GETS, 2-218
READ, 2-459, 2-784

Get system functions
GETACTIVE(), 2-470
GETAPPLYKEY(), 2-471
GETDOSETKEY(), 2-473
GETPOSTVALIDATE(), 2-476
GETPREVALIDATE(), 2-477
GETREADER(), 2-478
READFORMAT(), 2-787
READKILL(), 2-791
READUPDATED(), 2-794

getAccel(), 2-728, 2-772, 2-983

GETACTIVE(), 2-470

GETAPPLYKEY(), 2-471

getData(), 2-574

GETDOSETKEY(), 2-473

GETENV(), 2-474, 2-866

getFirst(), 2-729, 2-981

getItem(), 2-574, 2-729, 2-772, 2-982

getLast(), 2-729, 2-982

GetNew(), 2-459

getNext(), 2-729, 2-982

GETPOSVALIDATE(), 2-476

getPrev(), 2-730, 2-983

GETPREVALIDATE(), 2-477

GETREADER(), 2-478

getShortct(), 2-730

getText(), 2-574

GFNTERASE(), 2-480

GFNTLOAD(), 2-481

GFNTSET(), 2-483

GFRAME(), 2-486

GGETPIXEL(), 2-489

GLINE(), 2-490

GMODE(), 2-492

GO, 2-288, 2-292, 2-497, 2-905

GOTO, See GO

GPOLYGON(), 2-498

GPUTPIXEL(), 2-500

Graphic mode, 2-492

Graphic mode execution example, 2-495

Graphic mode functions
    GBMPDISP(), 2-451
    GBMPLOAD(), 2-454
    GELLIPSE(), 2-456
    GFNTERASE(), 2-480
    GFNTLOAD(), 2-481
    GFRAME(), 2-486
    GPOLYGON(), 2-498
    GPUTPIXEL(), 2-500
    GRECT(), 2-502
    GSETCLIP(), 2-504

Graphics, 2-83, 2-122, 2-215, 2-545

GRECT(), 2-502

GSETCLIP(), 2-504

GSETEXCL(), 2-507

GSETPAL(), 2-510

GUISEND, 2-88, 2-99, 2-104, 2-108, 2-111, 2-114

GWRITEAT(), 2-512

H

Hard carriage return, 2-515, 2-612

HARDCR(), 2-515
    in REPORT and LABEL FORMs, 2-515

Header file
    Llibg.ch, 2-500

Header files
    Achoice.ch, 2-134
    Box.ch, 2-83
    Dbstruct.ch, 2-261, 2-328
    directory location, 1-2
    Directry.ch, 2-145, 2-351
    Error.ch, 2-389
    Examplep.ch, 2-425, 2-426
    Fileio.ch, 2-407, 2-429, 2-441
    general discussion, 2-22
    identifier scoping in, 2-22
    Inkey.ch, 2-135, 2-137, 2-270, 2-554, 2-558, 2-658, 2-891, 2-927
    Llibg.ch, 2-498, 2-627, 2-646, 2-650, 2-652
    Llibg.ch(), 2-360
    Memoedit.ch, 2-599
    nesting, 2-22
    path searching, 2-22
    Set.ch, 2-918
    Std.ch, 2-26, 2-277

HEADER(), 2-517

Help, 2-886, 2-890
    online, 1-1

HitTest(), 2-845

hitTest(), 2-213, 2-574, 2-730, 2-754, 2-765, 2-773, 2-984

I

I2BIN(), 2-519

IF, 2-520, *See also* DO CASE, IF()

IF(), 2-522, *See also* IF

IIF(), See IF(), 2-524, *See also* IF

Import, 2-159, 2-406

INCLUDE directory, 1-2

Include files, See Header files

INDEX, 2-264, 2-341, 2-526

Index files
- closing, 2-253, 2-1006
- controlling index, 2-433, 2-534, 2-536, 2-1006
- creating, 2-264, 2-913
- determining default extension of, 2-532
- existence test for, 2-532
- information about, 2-297
- opening, 2-1006
- order of, 2-323, 2-533, 2-536, 2-1006
- rebuilding, 2-306, 2-801, 2-913
- relative searching, 2-315, 2-905, 2-910
- searching, 2-315, 2-324, 2-434, 2-904, 2-910, 2-937
- updating, 2-715, 2-913

Index functions
- DBCLEARINDEX(), 2-253
- DBCREATEINDEX(), 2-264
- DBREINDEX(), 2-306
- DBSEEK(), 2-315
- DBSETORDER(), 2-323
- DESCEND(), 2-341
- FOUND(), 2-433
- INDEXEXT(), 2-532
- INDEXKEY(), 2-533
- INDEXORD(), 2-536
- ORDCONDSET(), 2-669
- SOUNDEX(), 2-937

Index key
- date, 2-379
- descending order, 2-200, 2-341
- determining expression, 2-534
- logical, 2-522, 2-524
- matching, 2-904, 2-999
- totalling on, 2-533, 2-986
- unique values, 2-913

Index/order,SEEK, 2-848

INDEXEXT(), 2-532

Indexing
- in descending order, 2-341
- on dates and character strings, 2-379
- on numbers and character strings, 2-946

INDEXKEY(), 2-533

INDEXORD(), 2-536

Inheritance chain, 2-610

INIT PROCEDURE, 2-537

INKEY, 2-877

INKEY(), 2-540
- converting return values, 2-215
- used to stuff keyboard, 2-553
- used with DISPLAY, 2-365
- used with LIST, 2-566
- used with SET KEY, 2-890

Inkey.ch, 2-135, 2-137, 2-270, 2-554, 2-558, 2-658, 2-891, 2-927

INPUT, 2-542

Insert mode, 2-788

InsItem(), 2-460, 2-732

insItem(), 2-576, 2-774, 2-984

Installation, default directory structure, 1-2

Index

Instance variables
assignable, 2-387, 2-953, 2-959
exported, 2-209, 2-387, 2-460, 2-620, 2-725, 2-749, 2-762, 2-768, 2-953, 2-959, 2-979
list of CheckBox, 2-209
list of Error, 2-387
list of Get class, 2-460
list of ListBox class, 2-568
list of MenuItem class, 2-620
list of PopUpMenu, 2-725
list of PushButton class, 2-749
list of RadioButto, 2-762
list of RadioGroup, 2-768
list of TBColumn class, 2-953
list of TBrowse, 2-959
list of TopBarMenu class, 2-979

INT(), 2-543,

Integer, 2-543

Interactive CREATE, 2-245

isAccel(), 2-766

ISALPHA(), 2-544

ISCOLOR(), 2-545

ISCOLOUR(), See ISCOLOR()

ISDIGIT(), 2-546

ISDISK(), 2-547

ISLOWER(), 2-548

isOpen(), 2-732

isPopUp(), 2-622

ISPRINTER(), 2-549

ISUPPER(), 2-550

Iterator functions
AEVAL(), 2-144
DBEVAL(), 2-276

J

JOIN, 2-551

K

KEYBOARD, 2-553
controlling the, 2-786, 2-788, 2-920, 2-927
number of programmable keys available, 2-890
polling the, 2-558, 2-658, 2-789
programming, 2-890
status, 2-909
toggling insert mode, 2-788

Keyboard buffer
and function keys, 2-886
clearing the, 2-221
extracting a key from, 2-541
handling with DBEDIT(), 2-271
maximum number of characters in, 2-912
reading pending key, 2-658
returning last key extracted, 2-558
stuffing the, 2-215, 2-553, 2-808

Keyboard commands
KEYBOARD, 2-553
SET ESCAPE, 2-876
SET EVENTMASK, 2-877
SET KEY, 2-890
SET TYPEAHEAD, 2-912

Keyboard functions
INKEY(), 2-540
READKEY(), 2-789
SETCANCEL(), 2-920
SETKEY(), 2-927

killFocus(), 2-213, 2-755, 2-766, 2-774

L

L2BIN(), 2-555

LABEL FORM, 2-403, 2-556
  blank lines in, 2-523, 2-525

Labels, 2-556
  example, 2-745
  printing with @...SAY, 2-931

LASTKEY(), 2-134, 2-558

LASTREC(), 2-517, 2-560

LEFT(), 2-562,

LEN(), 2-563

LIBRARY directory, 1-2

Library files, directory location, 1-2

Library functions, determining data type of, 2-1012

Linking
  declarations, 2-403
  external references, 2-369, 2-403, 2-884

LIST, 2-565

ListBox class
  addItem method, 2-572
  bitmap, 2-568
  bottom, 2-568
  buffer, 2-568
  capCol, 2-568
  capRow, 2-568
  caption, 2-568
  cargo, 2-569
  caseSensitive, 2-573
  close method, 2-572
  coldBox, 2-569
  colorSpec, 2-569
  delItem method, 2-572
  dropDown, 2-570
  exact, 2-573
  examples, 2-579
  fBlock, 2-570
  findText method, 2-573
  getData method, 2-574
  getItem method, 2-574
  getText method, 2-574
  hasFocus, 2-570
  hitTest method, 2-574
  hotBox, 2-570
  insItem method, 2-576
  isOpen, 2-571
  item, 2-578
  itemCount, 2-571
  killFocus method, 2-576
  left, 2-571
  ListBox() function, 2-567
  message, 2-571
  methods, 2-572
  mouseCol, 2-574
  mouseRow, 2-574
  nextItem method, 2-576
  open method, 2-576
  position, 2-572, 2-573, 2-574, 2-576, 2-577, 2-578
  prevItem method, 2-577
  right, 2-571
  sBlock, 2-571
  scroll method, 2-577
  select method, 2-577
  setData method, 2-578
  setFocus method, 2-578
  setItem method, 2-578
  setText method, 2-578
  text, 2-572, 2-573, 2-576, 2-578
  top, 2-571
  topItem, 2-571
  typeOut, 2-572
  vScroll, 2-572

ListBox(), 2-567

Llibg.ch, 2-498

llibg.ch, 2-500

LOCAL, 2-443, 2-580

LOCATE, 2-229, 2-583

LOCK(), *See also* RLOCK()

Locking, *See also* File locking, Record locking
  obtaining a file lock, 2-427
  obtaining a record lock, 2-823

Index

LOG(), 2-585, , 2-867

Logarithm
base 10 example, 2-585
natural, 2-402, 2-585

Logical
comparison, 2-61, 2-63, 2-64, 2-71, 2-73, 2-75, 2-77
operators, 2-53, 2-54, 2-55, 2-59, 2-61, 2-63, 2-64, 2-66, 2-71, 2-73, 2-75, 2-77

Logical functions
EMPTY(), 2-382
IF(), 2-522
IIF(), 2-524
TRANSFORM(), 2-988

LOOP, 2-373, 2-431

Looping structures
conditional, 2-373
counter, 2-432

Low-level file functions,I2BIN(), 2-519

Low-level file functions
BIN2I(), 2-176
BIN2L(), 2-177
BIN2W(), 2-178
FCLOSE(), 2-405
FCREATE(), 2-407
FERROR(), 2-375, 2-410
FOPEN(), 2-429
FREAD(), 2-435
FREADSTR(), 2-437
FSEEK(), 2-441
FWRITE(), 2-449
L2BIN(), 2-555

Low-level file I/O, See Low-level file functions

LOWER(), 2-587,

LPT1,testing readiness of, 2-549

LTRIM(), 2-588,

LUPDATE(), 2-590

M

Macro expressions
and ACHOICE(), 2-131
and external references, 2-37, 2-403
and index keys, 2-533
and local variables, 2-286, 2-307, 2-581
and MEMVAR declarations, 2-615
and static variables, 2-286, 2-307, 2-941
creating, 2-35
expanded in TEXT block, 2-974
TYPE(), 2-994, 2-1012

Macro operator
and arrays, 2-35
and code blocks, 2-36
compared to extended expressions, 2-33
limitations, 2-33
nesting, 2-32
used for compiling on the fly, 2-36
used for text substitution, 2-31

Macro variable
definition, 2-31
terminating, 2-31

Mailing labels, 2-556

Manifest constants
compared to variables, 2-13
defining, 2-13
removing, 2-26

Match markers, 2-3
Extended expression match marker, 2-5
List match marker, 2-4
Optional match clauses, 2-5
Restricted match marker, 2-4
Result match marker, 2-4
Wild match marker, 2-5

Match pattern
Literal values, 2-3
matching commands, 2-1
saving command, 2-1
Words, 2-3

Mathematical operators, 2-30, 2-40, 2-41, 2-42, 2-44, 2-46, 2-48, 2-56

MAX(), 2-591,

MAXCOL(), 2-592, 2-929

Maximum, 2-591

MAXROW(), 2-593, 2-929

MCOL(), 2-594, *See also* MROW()

MDBLCLK(), 2-595

Memo commands, SET MEMOBLOCK, 2-893

Memo fields
- assigning values to, 2-944
- browsing example, 2-603
- comparison, 2-61, 2-63, 2-64, 2-71, 2-73, 2-75, 2-77
- displaying of, 2-596
- editing, 2-599, 2-614, 2-831
- editing example, 2-603
- editing of, 2-596
- formatting, 2-515, 2-605, 2-612, 2-629, 2-634
- operators, 2-29, 2-42, 2-46, 2-59, 2-61, 2-63, 2-64, 2-66, 2-71, 2-73, 2-75, 2-77
- printing, 2-612
- printing example, 2-606
- processing line-by-line, 2-606, 2-630, 2-634
- replacing with ASCII file contents, 2-607
- user-defined edit window, 2-596, 2-597
- word wrapping in, 2-606, 2-629, 2-634
- writing contents to an ASCII file, 2-614

Memo functions
- BLOBDIRECTEXPORT(), 2-179
- BLOBDIRECTGET(), 2-181
- BLOBDIRECTIMPORT(), 2-183
- BLOBDIRECTPUT(), 2-186
- BLOBEXPORT(), 2-188
- BLOBGET(), 2-190
- BLOBIMPORT(), 2-192
- BLOBROOTGET(), 2-194
- BLOBROOTLOCK(), 2-196
- BLOBROOTPUT(), 2-197
- BLOBROOTUNLOCK(), 2-199
- HARDCR(), 2-515
- MEMOEDIT(), 2-597, 2-599
- MEMOLINE(), 2-605
- MEMOREAD(), 2-607
- MEMOTRAN(), 2-612
- MEMOWRIT(), 2-614
- MLCOUNT(), 2-629
- MLCTOPOS(), 2-631
- MLPOS(), 2-634
- MPOSTOLC(), 2-638

Memo-field operations, 2-610

MEMOEDIT
- insert, 2-600
- scroll, 2-600
- word wrap, 2-600

MEMOEDIT(), 2-596, 2-612, 2-909
- browse mode, 2-597
- controlling insert mode during, 2-788
- creating character string example, 2-603
- desribing current mode example, 2-604
- editing character string example, 2-603
- editing modes, 2-597
- editing text, 2-597
- initialization mode, 2-600
- writing contents of text file example, 2-604

Memoedit.ch, 2-599

MEMOLINE(), 2-605, *See also* MLCOUNT()
- memo function, 2-606

MEMOREAD(), 2-607

Memory
- available amount, 2-609
- dynamic, 2-609
- free pool, 2-609

Memory variable commands
- CLEAR MEMORY, 2-219
- RESTORE, 2-814
- SAVE, 2-833

MEMORY(), 2-609

Index

Memory-resident programs, 2-831

MEMOSETSUPER(), 2-610

MEMOTRAN(), 2-612
    in REPORT FORMs, 2-612

MEMOWRIT(), 2-614

MEMVAR, 2-615

MEMVAR alias, 2-51

MEMVARBLOCK(), 2-617

Menu commands
    @...PROMPT, 2-115
    MENU TO, 2-625
    SET MESSAGE, 2-894
    SET WRAP, 2-915

MENU TO, 2-115, 2-625

MenuItem class
    caption, 2-620
    cargo, 2-620
    checked, 2-620
    data, 2-620
    enabled, 2-620
    id, 2-621
    isPopUp method, 2-622
    MenuItem() function, 2-619
    message, 2-621
    shortcut, 2-621
    style, 2-621

MenuItem(), 2-619

MENUMODAL(), 2-623

Menus
    example, 2-372
    lightbar, 2-115, 2-134, 2-625, 2-889, 2-894, 2-915
    navigation, 2-625, 2-915
    pop-up, 2-132
    pop-up example, 2-818, 2-837
    top bar, 2-979
    user-defined, 2-134

MESSAGE, 2-87, 2-98, 2-102, 2-107, 2-111, 2-113

Messages, sending, 2-57

Metasymbol prefixes,table, 1-5

Methods
    list of CheckBox class, 2-213
    list of Get class, 2-465
    list of ListBox class, 2-572
    list of MenuItem class, 2-622
    list of PopUpMenu class, 2-727
    list of PushButton class, 2-754
    list of RadioButto class, 2-765
    list of TBColumn class, 2-956
    list of TBrowse class, 2-964
    list of TopBarMenu class, 2-981
    RadioGroup class, 2-772

MHIDE(), 2-627

MIN(), 2-628,

Minimum, 2-628

MLCOUNT(), 2-606, 2-629

MLCTOPOS(), 2-631

MLEFTDOWN(), 2-633

MLPOS(), 2-634

MOD(), See % (modulus), 2-635

Mode
    color, 2-500
    display, 2-502
    XOR, 2-452, 2-500, 2-502

Monitor, type installed, determining, 2-545

Month
    character, 2-223
    day of, 2-250
    numeric, 2-637

MONTH(), 2-637

Mouse functions
MCOL(), 2-594
MDBLCLK(), 2-595
MLEFTDOWN(), 2-633
MPRESENT(), 2-640
MRESTSTATE(), 2-641
MRIGHTDOWN(), 2-642
MROW(), 2-643
MSAVESTATE(), 2-644
MSETBOUNDS(), 2-645
MSETCURSOR(), 2-648
MSETPOS(), 2-649

MPOSOLC(), 2-638

MPRESENT(), 2-640

MRESTSTATE(), 2-641

MRIGHTDOWN(), 2-642

MROW(), 2-643

MSAVESTATE(), 2-644

MSETBOUNDS(), 2-645

MSETCLIP(), 2-646

MSETCURSOR(), 2-648

MSETPOS(), 2-649

MSHOW(), 2-650

MSTATE(), 2-652

Multiple record blocks, 2-251

N

Natural logarithm, 2-402, 2-585

Nested arrays, See Arrays

Nested GETs, See Nested READs

Nested READs, 2-784

Nested sub-directories, 2-353

NETERR(), 2-655

NETNAME(), 2-657

Network commands
SET EXCLUSIVE, 2-880
UNLOCK, 2-997
USE...EXCLUSIVE, 2-1006
USE...SHARED, 2-1006

Network functions
FLOCK(), 2-427
NETERR(), 2-655
NETNAME(), 2-657
RLOCK(), 2-823

Networking
APPEND BLANK, 2-157, 2-655
APPEND FROM, 2-159
controlling print device, 2-901
COPY TO, 2-237
DBUNLOCK(), 2-330
DBUNLOCKALL(), 2-331
DELETE, 2-335
obtaining a file lock, 2-427
obtaining a record lock, 2-824
PACK, 2-715
RECALL, 2-797
REINDEX, 2-801
releasing locks, 2-997
REPLACE, 2-806
SET DEVICE, 2-874
trapping errors, 2-655
USE, 2-656, 2-1008

NEXT, 2-431

nextItem(), 2-774

NEXTKEY(), 2-658

NIL
comparison, 2-64, 2-71, 2-73
operators, 2-59, 2-64, 2-66, 2-71, 2-73
testing for, 2-382

NOSNOW(), 2-660

NOTE, See * (comment)

Numeric
comparison, 2-61, 2-63, 2-64, 2-71, 2-73, 2-75, 2-77
display, 2-867, 2-883
maximum of two, 2-591
minimum of two, 2-628
operators, 2-30, 2-40, 2-41, 2-42, 2-44, 2-46, 2-48, 2-56, 2-59, 2-61, 2-63, 2-64, 2-66, 2-68, 2-71, 2-73, 2-75, 2-77
overflow, 2-402, 2-946
precision, 2-825
rounding, 2-825, 2-945, 2-1011
testing for zero, 2-382

Numeric functions
ABS(), 2-129
CHR(), 2-215
EMPTY(), 2-382
EXP(), 2-402
INT(), 2-543
LOG(), 2-585
MAX(), 2-591
MIN(), 2-628
ROUND(), 2-825
SQRT(), 2-939
STR(), 2-945
TRANSFORM(), 2-988
WORD(), 2-1017

O

Object create functions
CheckBox(),, 2-209
ErrorNew(), 2-387
GetNew(), 2-459
ListBox(), 2-567
MenuItem(), 2-619
PopUp(), 2-724
PushButton(), 2-749
RadioButto(), 2-761
RadioGroup(), 2-768
TBColumnNew(), 2-953
TBrowseDB(), 2-958
TBrowseNew(), 2-958
TopBar(),, 2-979

Objects,
comparison, 2-73
creating Error, 2-387
creating Get, 2-459
creating new objects, 2-89, 2-97, 2-101, 2-106, 2-110, 2-113, 2-387, 2-459, 2-953, 2-958
creating TBColumn, 2-953
creating TBrowse, 2-958
operators, 2-57, 2-59, 2-66, 2-73
sending messages, 2-57

open(), 2-732

Opening files
database, 2-1010
exclusive mode, 2-880
in a network environment, 2-655
path searching, 2-899
shared mode, 2-880
with USE, 2-1006

Operating system commands
COMMIT, 2-227
COPY FILE, 2-231
DIR, 2-348
ERASE, 2-386
RENAME, 2-804
RUN, 2-831
SET DEFAULT, 2-868
SET PATH, 2-898
TYPE, 2-993

Operating system functions, See Environment functions

Operators
$, 2-29
%, 2-30
&, 2-31
( ), 2-39
*, 2-40
**, 2-41
+, 2-42
++, 2-44
-, 2-46
->, 2-50
.AND., 2-53
.NOT., 2-54

.OR., 2-55
/, 2-56
:, 2-57
<, 2-61
<=, 2-63
<>, 2-64
=, 2-59, 2-66, 2-68, 2-71
==, 2-73
>, 2-75
>=, 2-77
@, 2-81
[ ], 2-124
{ }, 2-126
addition, 2-42
alias, 2-50
array, 2-59, 2-66, 2-73, 2-124, 2-126
array element, 2-124
assign, 2-66, 2-734, 2-943
assignment, 2-59, 2-66, 2-68
binary, 2-29, 2-30, 2-40, 2-41, 2-42, 2-46, 2-50, 2-53, 2-55, 2-56, 2-57, 2-59, 2-61, 2-63, 2-64, 2-66, 2-68, 2-71, 2-73, 2-75, 2-77
character, 2-29, 2-42, 2-46, 2-59, 2-61, 2-63, 2-64, 2-66, 2-68, 2-71, 2-73, 2-75, 2-77
code block, 2-59, 2-66, 2-126
compile and run, 2-31
compound assignment, 2-68
concatenation, 2-42, 2-46
constant array, 2-126
date, 2-42, 2-44, 2-46, 2-48, 2-59, 2-61, 2-63, 2-64, 2-66, 2-68, 2-71, 2-73, 2-75, 2-77
decrement, 2-48
division, 2-56
equal, 2-71, 2-165, 2-878
exactly equal, 2-73, 2-878
exponentiation, 2-41, 2-402, 2-585
greater than, 2-75
greater than or equal, 2-77
grouping, 2-39
increment, 2-44
inline addition and assign, 2-68
inline assign, 2-59, 2-68, 2-444, 2-580, 2-734, 2-736, 2-747, 2-940, 2-943
inline concatenation and assign, 2-68
inline division and assign, 2-68
inline exponentiation and assign, 2-68
inline modulus and assign, 2-68
inline multiplication and assign, 2-68
inline subtraction and assign, 2-68
less than, 2-9, 2-61
less than or equal, 2-63
logical, 2-53, 2-54, 2-55, 2-59, 2-61, 2-63, 2-64, 2-66, 2-71, 2-73, 2-75, 2-77
macro, 2-31
mathematical, 2-30, 2-40, 2-41, 2-42, 2-44, 2-46, 2-48, 2-56
memo, 2-29, 2-42, 2-46, 2-59, 2-61, 2-63, 2-64, 2-66, 2-71, 2-73, 2-75, 2-77
modulus, 2-30
multiple inline assign, 2-59
multiplication, 2-40
negate, 2-54
NIL, 2-59, 2-64, 2-66, 2-71, 2-73
not equal, 2-64
numeric, 2-30, 2-40, 2-41, 2-42, 2-44, 2-46, 2-48, 2-56, 2-59, 2-61, 2-63, 2-64, 2-66, 2-68, 2-71, 2-73, 2-75, 2-77
object, 2-57, 2-59, 2-66, 2-73
pass-by-reference, 2-81
postdecrement, 2-48
postfix, 2-44, 2-48
postincrement, 2-44
predecrement, 2-48
prefix, 2-44, 2-48
preincrement, 2-44
relational, 2-29, 2-53, 2-54, 2-55, 2-61, 2-63, 2-64, 2-71, 2-73, 2-75, 2-77
send, 2-57
special, 2-31, 2-39, 2-81, 2-124, 2-126
substring, 2-29, 2-170, 2-777
subtraction, 2-46
table of special, 1-4
unary, 2-31, 2-42, 2-44, 2-46, 2-48, 2-54, 2-81
using less than operator with <resultPattern>, 2-9

Optimizing,filters, 2-895

ORDBAGEXT(), 2-663

ORDBAGNAME(), 2-664

ORDCOND(), 2-666

ORDCONDSET(), 2-669

ORDCREATE(), 2-673

ORDDESCEND(), 2-675

ORDDESTROY(), 2-677

Order commands
- SET OPTIMIZE, 2-895
- SET SCOPE, 2-906
- SET SCOPEBOTTOM, 2-907
- SET SCOPETOP, 2-908

Order functions
- ORDBAGEXT(), 2-663
- ORDBNAME(), 2-664
- ORDCOND(), 2-666
- ORDCREATE(), 2-673
- ORDDESCEND(), 2-675
- ORDDESTROY(), 2-677
- ORDFORD(), 2-678
- ORDISUNIQUE(), 2-680
- ORDKEY(), 2-682
- ORDKEYADD(), 2-684
- ORDKEYCOUNT(), 2-686
- ORDKEYDEL(), 2-688
- ORDKEYGOTO(), 2-691
- ORDKEYNO(), 2-693
- ORDKEYVAL(), 2-695
- ORDLISTADD(), 2-697
- ORDLISTREBUILD(), 2-700
- ORDNAME(), 2-701
- ORDNUMBER(), 2-703
- ORDSCOPE(), 2-704
- ORDSETFOCUS(), 2-706
- ORDSETRELATION(), 2-708
- ORDSKIPUNIQUE(), 2-710

Orders
- adding key to custom order, 2-684
- adding orders to list, 2-697
- clearing current list, 2-699
- creating orders(), 2-673
- current record number, 2-693
- delete a key from custom order, 2-688
- deleting, 2-338
- descending flag, 2-675
- information about, 2-297
- key value of current record, 2-695
- moving record pointer, 2-710, 2-848
- moving to a specific record, 2-691
- number of keys, 2-686
- RDD support, 2-338
- rebuilding orders in list, 2-700
- relations
  - establishing, 2-708
  - work areas, relating, 2-708
- removing order, 2-677
- returning default extension, 2-663
- returning FOR expression of order, 2-678
- returning key expression of order, 2-682
- returning name of order, 2-701
- returning order name, 2-664
- returning order position, 2-703
- scoping key value boundaries, 2-704
- searching, 2-848
- setting conditions, 2-669
- setting focus, 2-706
- specifying ordering conditions, 2-666
- unique flag, 2-680

ORDFOR(), 2-678

ORDISUNIQUE(), 2-680

ORDKEY(), 2-682

ORDKEYADD(), 2-684

ORDKEYCOUNT(), 2-686

ORDKEYDEL(), 2-688

ORDKEYGOTO(), 2-691

ORDKEYNO(), 2-693

ORDKEYVAL(), 2-695

ORDLISTADD(), 2-697

ORDLISTCLEAR(), 2-699

ORDLISTREBUILD(), 2-700

ORDNAME(), 2-701

ORDNUMBER(), 2-703

ORDSCOPE(), 2-704

ORDSETFOCUS(), 2-706

ORDSETRELATION(), 2-708

ORDSKIPUNIQUE(), 2-710

OS(), 2-712

OTHERWISE, 2-371

OUTERR(), 2-713

Output functions, OUTSTD(), 2-714

Output to text file
- ? | ??, 2-79
- @...SAY, 2-117, 2-874
- DISPLAY, 2-365
- LABEL FORM, 2-556
- LIST, 2-565
- REPORT FORM, 2-810
- SET ALTERNATE, 2-853
- SET CONSOLE, 2-862
- SET PRINTER, 2-900
- TEXT, 2-974
- TYPE, 2-993

OUTSTD(), 2-714

P

PACK, 2-715, 2-1020

PADC(), 2-716

PADL(), 2-716

PADR(), 2-716

Page eject, 2-381, 2-931

Page offset, 2-892

Parameter
- column parameter, 2-600
- line parameter, 2-600
- maximum number for DO...WITH, 2-369
- number passed, 2-722
- omitting, 2-370, 2-445, 2-718, 2-722
- optional, 2-369, 2-445, 2-718, 2-722
- passing, 2-445, 2-722
- passing arrays as, 2-370, 2-718
- passing by reference, 2-205, 2-369, 2-445, 2-718
- passing by value, 2-205, 2-369, 2-445, 2-718
- passing from DOS command line, 2-718
- passing into local variables, 2-581
- passing private, 2-718
- passing with CALL, 2-1017

PARAMETERS, 2-443, 2-615, 2-718,

Parentheses, and precedence, 2-39

Pass by reference, 2-81, 2-435

Pass by value, 2-81

Path, 2-423, 2-868, 2-899
- extraction example, 2-777

PCOL(), 2-720, 2-892
- resetting with SETPRC(), 2-931

PCOUNT(), 2-722

Pending GETs, 2-218, 2-220, 2-782

Pending record blocks, 2-251

PICTURE, 2-87, 2-117, 2-988

PopUp(), 2-724

PopUpMenu class
addItem method, 2-727
border, 2-725
bottom, 2-725
cargo, 2-725
close method, 2-727
colorSpec, 2-726
current, 2-726
delItem method, 2-728
display method, 2-728
getAccel method, 2-728
getFirst method, 2-729
getItem method, 2-729
getLast method, 2-729
getNext method, 2-729
getPrev method, 2-730
getShortct method, 2-730
hitTest method, 2-730
InsItem method, 2-732
isOpen method, 2-732
itemCount, 2-726
left, 2-726
open method, 2-732
PopUp() function, 2-724
right, 2-726
select method, 2-732
setItem method, 2-733
top, 2-726
width, 2-727

Postfix notation, 2-44, 2-48

Prefix --, 2-48

Prefix notation, 2-44, 2-48

Preprocessor directives
#define, 2-12, 2-26
#else, 2-18, 2-20
#endif, 2-18, 2-20
#error, 2-17
#ifdef, 2-18
#ifndef, 2-20
#include, 2-22, 2-83, 2-553, 2-891, 2-903
#stdout, 2-25
#undef, 2-26
#xcommand, 2-28
#xtranslate, 2-28

Preprocessor identifiers
defining, 2-13, 2-26
testing existence of, 2-18
testing nonexistence of, 2-20

prevItem(), 2-774

Print margin
setting the left, 2-892

Printer functions
ISPRINTER(), 2-549
PCOL(), 2-720
PROW(), 2-745
SETPRC(), 2-931

Printing
? | ??, 2-79
@...GET, 2-94
@...SAY, 2-117, 2-874
a formfeed, 2-381
columnar reports, 2-810
DISPLAY, 2-365
EJECT, 2-381
example, 2-523, 2-525, 2-549, 2-934
form letter example, 2-974
handling errors, 2-175
labels with @...SAY, 2-931
LIST, 2-565
mailing labels, 2-556
moving the printhead, 2-347
obtaining column number, 2-720
obtaining row number, 2-745

relative addressing, 2-720, 2-745
resetting column number, 2-931
resetting row number, 2-931
sending control codes, 2-215, 2-720, 2-745, 2-931
SET CONSOLE, 2-862
SET PRINTER, 2-900
setting the margin, 2-892
spooling printed output, 2-902
testing for readiness of, 2-549
TEXT, 2-974
TYPE, 2-993

PRIVATE, 2-443, 2-615, 2-734,

PROCEDURE, 2-736, , 2-820

Procedures
compiling, 2-903
declaring, 2-736
executing, 2-369
line number, 2-741
name, 2-743
naming conventions, 2-736
passing parameters to, 2-370, 2-722
termination, 2-820

Process
child, 2-395
parent, 2-395

PROCLINE(), 2-741

PROCNAME(), 2-743

Program
termination, 2-820

Program control commands
QUIT, 2-759

Program control functions
IF(), 2-522
IIF(), 2-524
INKEY(), 2-541
LASTKEY(), 2-558
NEXTKEY(), 2-658
READKEY(), 2-789

Program control statements
DO, 2-369
RETURN, 2-820

Program termination, 2-759, 2-920

Projection, 2-551

PROW(), 2-745
resetting with SETPRC(), 2-931

Pseudofunctions
defining, 2-11, 2-13
removing, 2-26

PUBLIC, 2-443, 2-615, 2-747

PushButton class
bitmap, 2-749
bmpXOff, 2-749
bmpYOff, 2-750
buffer, 2-750
caption, 2-750, 2-751
capXOff, 2-750
capYOff, 2-751
cargo, 2-751
col, 2-751
colorSpec, 2-751
display method, 2-754
fBlock, 2-752
hasFocus, 2-752
hitTest method, 2-754
killFocus method, 2-755
message, 2-752
PushButton() function, 2-749
row, 2-753
sBlock, 2-753
select method, 2-756
setFocus method, 2-756
sizeX, 2-753
sizeY, 2-753
style, 2-753
typeOut, 2-754

PushButton(), 2-749

Q

QOUT(), 2-79, 2-757

QQOUT(), See QOUT()

Query, 2-287

QUIT, 2-759

R

RadioButto class
- bitmaps, 2-762
- buffer, 2-762
- capCol, 2-762
- capRow, 2-763
- caption, 2-762
- cargo, 2-763
- col, 2-763
- colorSpec, 2-763
- display method, 2-765
- fBlock, 2-764
- hasFocus, 2-764
- hitTest method, 2-765
- isAccel method, 2-766
- killFocus method, 2-766
- RadioButto() function, 2-761
- row, 2-764
- sBlock, 2-764
- select method, 2-766
- setFocus method, 2-767
- style, 2-765

RadioButto(), 2-761

RadioGroup class
- addItem method, 2-772
- block, 2-770
- bottom, 2-768
- buffer, 2-768
- capCol, 2-768
- capRow, 2-769
- caption, 2-769
- cargo, 2-769
- coldBox, 2-769
- colorSpec, 2-770
- delItem method, 2-772
- display method, 2-772
- getAccel method, 2-772
- getItem method, 2-772
- hasFocus, 2-770
- hitTest method, 2-773
- hotBox, 2-771
- insItem method, 2-774
- itemCount, 2-771
- killFocus method, 2-774
- left, 2-771
- message, 2-771
- nextItem method, 2-774
- prevItem method, 2-774
- RadioGroup() function, 2-768
- right, 2-771
- select method, 2-775
- setColor method, 2-775
- setFocus method, 2-776
- setStyle, 2-776
- top, 2-771
- typeOut, 2-771

RadioGroup(), 2-768

RANGE, 2-88, 2-909

RAT(), 2-777, *See also* AT()

RDD functions
- RDDLIST(), 2-778
- RDDNAME(), 2-780
- RDDSETDEFAULT(), 2-781

RDDLIST(), 2-778

RDDNAME(), 2-780

RDDSETDEFAULT(), 2-781

READ, 2-86, 2-218, 2-782, 2-792, 2-876, 2-877, 2-884, 2-891, 2-909
- controlling insert mode during, 2-788
- determining variable name, 2-795
- determining whether value changed, 2-1001
- exiting from a, 2-786
- implementation, 2-459, 2-464, 2-792

Read-only files, 2-1008

READEXIT(), 2-786

Reading foreign file formats, 2-159

READINSERT(), 2-788

READKEY(), 2-789

READMODAL(), 2-792

READVAR(), 2-581, 2-795, 2-941
    implementation, 2-463

RECALL, 2-302, 2-797

RECCOUNT(), See LASTREC()

RECNO(), 2-799

Record
    adding, 2-157, 2-159, 2-204, 2-884
    copy, 2-237
    count, 2-240, 2-560
    filtering, 2-286, 2-320, 2-882
    information about, 2-304
    pointer
        moving, 2-848
    position of pointer, 2-799
    processing with DBEVAL(), 2-277
    scoping, 2-277, 2-320, 2-882
    size, 2-800
    summarizing, 2-986
    total, 2-952, 2-986

Record locking
    APPEND BLANK, 2-157
    DBUNLOCK(), 2-330
    DBUNLOCKALL(), 2-331
    DELETE, 2-335
    RECALL, 2-797
    REPLACE, 2-806
    RLOCK(), 2-427, 2-823
    SET EXCLUSIVE, 2-880
    UNLOCK, 2-997

Record number
    matching, 2-905
    saving, 2-497

Record pointer
    and BOF(), 2-200
    and EOF(), 2-384
    and relative searching, 2-433, 2-910
    initial value, 2-1006
    moving, 2-315, 2-326, 2-933
    moving in several files simultaneously, 2-325, 2-904
    moving to first record, 2-292
    moving to last record, 2-288
    moving to specific identify, 2-290
    restoring to saved position, 2-497

RECSIZE(), 2-517, 2-800

Recursion, 2-445, 2-581

REINDEX, 2-306, 2-715, 2-801

Relational
    operators, 2-29, 2-53, 2-54, 2-55, 2-61, 2-63, 2-64, 2-71, 2-73, 2-75, 2-77

Relations
    and file locking, 2-428, 2-824
    child, 2-433, 2-905, 2-910
    clearing, 2-254
    creating, 2-325, 2-905
    cyclical, 2-905
    multiple child, 2-905
    order of, 2-307, 2-313
    parent, 2-905
    saving, 2-308, 2-313
    with UPDATE, 2-999

Relative printer addressing, 2-720, 2-745

Relative screen addressing, 2-224, 2-827

RELEASE, 2-803

Remove disk file, 2-337, 2-386, 2-409

RENAME, 2-804,

Renaming files, 2-439, 2-804

REPLACE, 2-806

Replaceable Database Drivers (RDDs), 2-159, 2-778, 2-780, 2-781

REPLICATE(), 2-808,

REPORT FORM, 2-403, 2-809

Reports, 2-810

REQUEST, 2-812

RESTORE, 2-287, 2-814

RESTORE SCREEN, See RESTSCREEN()

RESTSCREEN(), 2-818,

Result markers
- Blockify result marker, 2-7
- Dumb stringify result marker, 2-6
- Logify result marker, 2-8
- Normal stringify result marker, 2-7
- Regular result marker, 2-6
- Smart stringify result marker, 2-7
- table of marker forms, 2-6

Result pattern, 2-6
- Literal tokens, 2-6
- Repeating result clauses, 2-8
- specifying more than one statement, 2-9
- Words, 2-6

RETURN, 2-820

RETURN TO MASTER, emulating, 2-173, 2-820

Return values, user-defined function, 2-445, 2-820

RIGHT(), 2-822,

RL.EXE, 2-556, 2-810

RLOCK(), 2-823,

ROUND(), 2-825,

Rounding, 2-825, 2-945

ROW(), 2-827

RTRIM(), 2-829,

RUN, 2-395, 2-831, 2-868

S

SAVE, 2-287, 2-833

SAVE SCREEN, See SAVESCREEN()

SAVESCREEN(), 2-837,

Scoreboard, 2-909

Screen commands
- CLEAR GETS, 2-218
- CLEAR SCREEN, 2-220
- SET CONSOLE, 2-862
- SET CURSOR, 2-864
- SET DELIMITERS, 2-871
- SET INTENSITY, 2-889
- SET MESSAGE, 2-894
- SET SCOREBOARD, 2-909

Screen functions
- COL(), 2-224
- COLORSELECT(), 2-225
- DEVOUT(), 2-343
- DEVOUTPICT(), 2-344
- DISPBEGIN(), 2-358
- DISPBOX(), 2-360
- DISPCOUNT(), 2-363
- DISPEND(), 2-364
- DISPOUT(), 2-367
- ISCOLOR(), 2-545
- MAXCOL(), 2-592
- MAXROW(), 2-593
- RESTSCREEN(), 2-818
- ROW(), 2-827
- SAVESCREEN(), 2-837
- SCROLL(), 2-839
- SETBLINK(), 2-919
- SETCOLOR(), 2-922
- SETMODE(), 2-929

Screen handling
addressing, 2-592, 2-593
changing colors, 2-922
clearing the screen, 2-85, 2-220
control of cursor display, 2-864
defining a scrolling region, 2-839
defining mouse cursor's screen column position, 2-594
determining monitor type, 2-545
line zero messages, 2-909
maximum column number, 2-592
maximum row number, 2-593
obtaining column number, 2-224
obtaining row number, 2-827
pause output, 2-993
refreshing the screen, 2-533
relative addressing, 2-224, 2-827
saving and restoring screens, 2-818, 2-837
suppress display, 2-863, 2-902

SCROLL(), 2-839

scroll(), 2-577

Scrollbar class
BarLength, 2-842
Bitmaps, 2-842
Cargo, 2-842
ColorSpec, 2-843
Current, 2-843
Display method, 2-845
End, 2-843
HitTest method, 2-845
Offset, 2-843
orient, 2-844
Scrollbar() function, 2-841
Start, 2-844
Style, 2-844
ThumbPos, 2-845
Total, 2-845

Scrollbar(), 2-841

Search and replace, 2-947

Search commands
CONTINUE, 2-229
LOCATE, 2-584
testing after, 2-434

Search functions, *See also* Search commands
DBSEEK(), 2-315
FOUND(), 2-433
ORDCONDSET(), 2-669

Searching,phonetic, 2-937

SECONDS(), 2-847

SEEK, 2-315, 2-341, 2-905, 2-910

SEEK command, 2-848

SELECT, 2-312, 2-317, 2-850

SELECT(), 2-852

select(), 2-214, 2-577, 2-732, 2-756, 2-766, 2-775, 2-985

Selection, 2-551

SEND, 2-88, 2-99, 2-104, 2-108, 2-111, 2-114

SEQUENCE, See BEGIN SEQUENCE

SET ALTERNATE, 2-79, 2-853

SET BELL, 2-855,

SET CENTURY, 2-249, 2-856, 2-875

SET COLOR, See SETCOLOR()

SET CONFIRM, 2-861,

SET CONSOLE, 2-79, 2-862, 2-900

SET CURSOR, 2-864,

SET DATE, 2-246, 2-249, 2-856, 2-865, 2-875

SET DECIMALS, 2-402, 2-825, 2-867, , 2-883, 2-939, 2-1011

SET DEFAULT, 2-348, 2-356, 2-423, 2-868,

SET DELETED, 2-335, 2-560, 2-870, 2-936, 2-1000

SET DELIMITERS, 2-871

SET DESCENDING, 2-873

SET DEVICE, 2-94, 2-117, 2-874, 2-901

SET EPOCH, 2-856, 2-875

SET ESCAPE, 2-876, 2-891

SET EVENTMASK, 2-877

SET EXACT, 2-61, 2-63, 2-64, 2-71, 2-73, 2-75, 2-77, 2-165, 2-878

SET EXCLUSIVE, 2-880,

SET FILTER, 2-252, 2-286, 2-320, 2-560, 2-882

SET FIXED, 2-402, 2-825, 2-867, 2-883, , 2-939, 2-1011

SET FORMAT, 2-884,

SET FUNCTION, 2-886, 2-890, 2-1015

SET INDEX, 2-253

SET INDEX command, 2-887

SET INTENSITY, 2-889

SET KEY, 2-218, 2-220, 2-581, 2-783, 2-876, 2-886, 2-890, 2-920, 2-941, 2-1001, 2-1015

SET MARGIN, 2-892

SET MEMOBLOCK, 2-893

SET MESSAGE, 2-115, 2-894

SET OPTIMIZE, 2-895

SET ORDER, 2-323, 2-896

SET PATH, 2-423, 2-898,

SET PRINTER, 2-900

SET PROCEDURE, 2-903

SET RELATION, 2-254, 2-307, 2-324, 2-428, 2-904, 2-910, 2-997

SET SCOPE, 2-906

SET SCOPEBOTTOM, 2-907

SET SCOPETOP, 2-908

SET SCOREBOARD, 2-909

SET SOFTSEEK, 2-433, 2-905, 2-910

SET TYPEAHEAD, 2-912

SET UNIQUE, 2-913,

SET VIDEOMODE, 2-914

SET WRAP, 2-915

SET(), 2-916

Set.ch, 2-918

SETBLINK(), 2-919

SETCANCEL(), 2-891, 2-920

SETCOLOR, 2-919

SETCOLOR(), 2-133, 2-367, 2-922

setColor(), 2-775

SETCURSOR(), 2-925,

setData(), 2-578

setFocus(), 2-214, 2-756, 2-767, 2-776

setItem(), 2-578, 2-733, 2-985

SETKEY(), 2-927

SETMODE(), 2-929

SETPOS(), 2-930

SETPRC(), 2-931

setText(), 2-578

Shared mode, 2-427, 2-823, 2-880, 2-1006

SKIP, 2-200, 2-326, 2-384, 2-933

Soft carriage return, 2-515, 2-612

SORT, 2-357, 2-935

Sorting
- arrays, 2-168
- ascending order, 2-168, 2-935
- database files, 2-935
- descending order, 2-168, 2-935
- dictionary order, 2-168, 2-935

Sound functions, TONE(), 2-977

Soundex code, 2-937

SOUNDEX(), 2-937

SOURCE directory, 1-2

Source files
directory location, 1-2
line number, 2-741

SPACE(), 2-938,

Specifying
translation directive, 2-1
user-defined command, 2-1

SQRT(), 2-867, 2-939

Square root, 2-939

STATE, 2-98, 2-103, 2-107

Statements
ANNOUNCE, 2-156
BEGIN SEQUENCE:, 2-173
DECLARE, 2-334
DO, 2-369
DO CASE, 2-371
DO WHILE, 2-373
EXIT PROCEDURE, 2-399
EXTERNAL, 2-403
FIELD, 2-412
FOR, 2-431
FUNCTION, 2-443
IF, 2-520
INIT PROCEDURE, 2-537
LOCAL, 2-580
MEMVAR, 2-615
PARAMETERS, 2-718
PRIVATE, 2-734
PROCEDURE, 2-736
PUBLIC, 2-747
REQUEST, 2-812
RETURN, 2-820
STATIC, 2-940

STATIC, 2-443, 2-940

Statistics
AVERAGE, 2-172
COUNT, 2-240
SUM, 2-952

Std.ch, 2-2, 2-26, 2-277

STORE, 2-943,

STR(), 2-945

String, See Character string

STRTRAN(), 2-947

Structure extended file, 2-234, 2-241, 2-243, 2-262, 2-328

STUFF(), 2-948

STYLE, 2-99, 2-108

SUBSTR(), 2-950,

Substring
general, 2-950
left, 2-562
operator, 2-29
right, 2-822
search and replace, 2-947

SUM, 2-952,

Summarizing records, 2-986

Summation, 2-952

Super driver, 2-610

Symbols, table of special, 1-4

System functions
TYPE(), 2-995
UPDATED(), 2-1001
VALTYPE(), 2-1012

T

Table view, 2-204, 2-270

Tags, deleting, 2-338

TBColumn class, 2-953
- block, 2-953, 2-954, 2-956
- cargo, 2-953
- colorBlock, 2-954
- colSep, 2-954, 2-960
- defColor, 2-954
- examples, 2-957
- footing, 2-954
- footSep, 2-954, 2-960
- heading, 2-953, 2-954
- headSep, 2-955, 2-961
- lSetting, 2-956
- nStyle, 2-956
- postBlock, 2-955
- preBlock, 2-955
- setstyle method, 2-956
- TBColumnNew() function, 2-953
- width, 2-956

TBColumnNew(), 2-953

TBrowse class, 2-958
- addColumn method, 2-966
- applyKey method, 2-966
- autoLite, 2-959
- bBlock, 2-971
- border, 2-959
- cargo, 2-960
- colCount, 2-960, 2-966
- colorRect method, 2-967
- colorSpec, 2-960
- colPos, 2-960
- colSep, 2-954, 2-960
- colWidth method, 2-968
- configure method, 2-968
- delColumn method, 2-968
- down method, 2-964
- end method, 2-964
- examples, 2-966, 2-967, 2-971, 2-973
- footSep, 2-960
- forceStable method, 2-968
- freeze, 2-960
- getColumn method, 2-968
- goBottom method, 2-964
- goBottomBlock, 2-964
- goTop method, 2-961, 2-964
- goTopBlock, 2-961, 2-964
- headSep, 2-961
- hilite method, 2-968
- hitBottom, 2-961, 2-964, 2-965
- hitTop, 2-961, 2-965, 2-966
- home method, 2-964
- insColumn method, 2-970
- invalidate method, 2-970
- key handlers, 2-971
- left method, 2-964
- leftVisible, 2-961
- lSetting, 2-972
- mColPos, 2-962
- message, 2-962
- mRowPos, 2-962
- nBottom, 2-958, 2-962
- nKey, 2-971
- nLeft, 2-958, 2-962
- nRight, 2-958, 2-962
- nStyle, 2-972
- nTop, 2-958, 2-962
- pageDown method, 2-965
- pageUp method, 2-965
- panEnd method, 2-965
- panHome method, 2-965
- panLeft method, 2-965
- panRight method, 2-965
- picture, 2-955
- refreshAll method, 2-970
- refreshCurrent method, 2-970
- right method, 2-965
- rightVisible, 2-962
- rowCount, 2-963
- rowPos, 2-963
- setColumn method, 2-970
- SetKey, 2-966
- setKey method, 2-971
- setstyle method, 2-972
- skipBlock, 2-961, 2-963
- stabilization, 2-973

stabilize method, 2-964, 2-973
stable, 2-964
up method, 2-966

TBrowseDB(), 2-958

TBrowseNew(), 2-958

Temporary files, 2-614, 2-936

TEXT, 2-974

Text files, See ASCII files

Time
calculations, 2-847
elapsed seconds, 2-847
formatting, 2-976

Time functions
SECONDS(), 2-847
TIME(), 2-976

TIME(), 2-976,

TONE(), 2-977

TopBar(), 2-979

TopBarMenu class
addItem method, 2-981
cargo, 2-979
colorSpec, 2-979
current, 2-980
delItem method, 2-981
display method, 2-981
getAccelmethod, 2-983
getFirst method, 2-981
getItem method, 2-982
getLast method, 2-982
getNext method, 2-982
getPrev method, 2-983
hitTest method, 2-984
insItem method, 2-984
itemCount, 2-980
left, 2-980
right, 2-980
row, 2-980
select method, 2-985
setItem method, 2-985
TopBar() function, 2-979

TOTAL, 2-986,

TRANSFORM(), 2-988,

Translation directives, 2-28
#command | #translate, 2-1

Trim, 2-154, 2-588, 2-829

TRIM(), See RTRIM(), 2-991

TYPE, 2-993

TYPE(), 2-994,

Typeahead buffer, See Keyboard buffer

U

Uniqueness attribute, 2-913

UNLOCK, 2-330, 2-331, 2-997

Unlocking, 2-330, 2-331, 2-997

UPDATE, 2-999

UPDATED(), 2-1001

UPPER(), 2-1003,

USE, 2-256, 2-332, 2-1005, 2-1010

USED(), 2-1010

User function
MEMOEDIT(), 2-597

User interface functions
ACHOICE(), 2-132
BROWSE(), 2-203
DBEDIT(), 2-268
MEMOEDIT(), 2-596, 2-597
READMODAL(), 2-792

User-defined commands, 2-28, 2-277

User-defined functions
and data validation, 2-86, 2-218, 2-220, 2-783
calling conventions, 2-444
compiling, 2-903

declaring, 2-444
determining data type of, 2-1012
executing, 2-444
naming conventions, 2-444
nesting, 2-445
passing arguments to, 2-444
termination, 2-820

V

VAL(), 2-1011

VALID, 2-88, 2-98, 2-102, 2-107, 2-111, 2-113, 2-218, 2-220, 2-447
implementation, 2-463

VALTYPE(), 2-581, 2-941, 2-1012

Variable names
precedence, 2-581, 2-615, 2-734, 2-748, 2-941
resolving references, 2-412, 2-581, 2-615, 2-734, 2-748, 2-941, 2-943

Variables
compared to manifest constants, 2-13
creating date, 2-246
creation, 2-580, 2-718, 2-734, 2-747, 2-940, 2-943
hidden, 2-580, 2-734, 2-747, 2-833
initialization, 2-59, 2-66, 2-68, 2-580, 2-734, 2-747, 2-940
local, 2-286, 2-307, 2-580, 2-940
maximum number of local, 2-580
maximum number of private, 2-734
maximum number of public, 2-748
maximum number of static, 2-940
naming conventions, 2-943
private, 2-718, 2-734, 2-803
public, 2-747, 2-803
releasing, 2-217, 2-219, 2-803, 2-820, 2-943
restore from a file, 2-581, 2-814, 2-941
save to a file, 2-580, 2-833, 2-941
screen, 2-818, 2-837
static, 2-286, 2-307, 2-940

Version number,obtaining of, 2-1014

VERSION(), 2-1014

Video mode, 2-492

video mode settings, 2-504

Video ROM fonts, 2-481

View, 2-286, 2-307, 2-312

Virtual join, 2-307, 2-312, 2-904

VMM
memory, 2-454
region, 2-480, 2-481

W

WAIT, 2-1015,

Wait states
ACCEPT, 2-130
ACHOICE(), 2-134
and INKEY(), 2-540
and KEYBOARD, 2-553
and NEXTKEY(), 2-658
and READVAR(), 2-795
and SET KEY, 2-890
and SET TYPEAHEAD, 2-912
and SETCANCEL(), 2-920
DBEDIT(), 2-271
INPUT, 2-542
MENU TO, 2-625
READ, 2-784
READMODAL(), 2-792
WAIT, 2-1015

WHEN, 2-88, 2-98, 2-102, 2-107, 2-111, 2-113
implementation, 2-464

WHILE, See DO WHILE

Wildcard characters, 2-143, 2-348, 2-352, 2-423, 2-803, 2-833

Windowing, 2-204, 2-270, 2-839

WITH, 2-369

WORD(), 2-1017

Work area
changing, 2-317, 2-850
default aliases, 2-850
determining whether used, 2-1010
next available, 2-1008
number available, 2-850, 2-852, 2-1006
number of related, 2-312
obtaining number of, 2-852
positioning record pointer in, 2-497
selecting next available, 2-850, 2-852

Workstation, determining name of, 2-657

Writing foreign file formats, 2-237

Y

YEAR(), 2-1019

Year, numeric, 2-1019

Z

ZAP, 2-1020

Index