CA-Clipper®

For DOS

Version 5.3

Programming and Utilities Guide

June 1995

COMPUTER ASSOCIATES®
Software superior by design.

PRINTED ON RECYCLED PAPER

© 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

Chapter 1 : Introduction

In This Guide 1–1
Part 1: Basic Concepts 1–2
Part 2: Programming 1–2
Part 3: Utilities 1–3

Chapter 2 : Basic Concepts

In This Chapter 2–1
The Structure of a CA-Clipper Program 2–2
A Typical CA-Clipper Program 2–3
The Main Procedure Definition 2–4
Function and Procedure Calls 2–5
Variable Declaration 2–6
Function and Procedure Definitions 2–7
Preprocessor Directives 2–8
Comments 2–8
Continuation 2–9
Reserved Words 2–10
Functions and Procedures 2–10
Defining Functions and Procedures 2–11
Calling Functions and Procedures 2–13
Passing Parameters 2–14
Passing by Value 2–16
Passing by Reference 2–17
Passing Arrays and Objects as Arguments 2–18
Argument Checking 2–19

Passing Arguments from the DOS Command Line 2–20
Returning Values from Functions 2–20
Recursion 2–21
Control Structures 2–22
Looping Structures 2–22
FOR...NEXT 2–23
DO WHILE...ENDDO 2–23
EXIT and LOOP 2–24
Decision-Making Structures 2–25
IF...ENDIF 2–25
DO CASE...ENDCASE 2–26
Error Handling Structures 2–26
Variables 2–27
Lexically Scoped Versus Dynamically Scoped Variables 2–27
Lifetime and Visibility of Variables 2–28
Declaring Variables 2–29
The Scope of a Declaration 2–30
Referring to Variables 2–31
Ambiguous Variable References 2–32
Avoiding Ambiguous Variable References 2–32
Qualifying Variable References 2–33
Creation and Initialization 2–33
Local Variables 2–34
Static Variables 2–35
Private Variables 2–36
Public Variables 2–37
Field Variables 2–38
Expressions 2–39
Data Types 2–40
Character 2–40
Memo 2–42
Date 2–43
Numeric 2–45
Logical 2–46
NIL 2–47

Operators 2–48
Terminology 2–48
Error Handling 2–50
String Operators 2–50
Date Operators 2–51
Mathematical Operators 2–51
Relational Operators 2–52
Logical Operators 2–53
Assignment Operators 2–54
Simple Assignment 2–55
Inline Assignment 2–56
Compound Assignments 2–57
Increment and Decrement Operators 2–58
Special Operators 2–60
Operator Precedence 2–61
Precedence of Categories 2–62
Precedence within a Category 2–63
Preincrement and Predecrement 2–63
Mathematical 2–64
Relational 2–64
Logical 2–64
Assignment 2–65
Postincrement and Postdecrement 2–65
Parentheses 2–65
The Macro Operator 2–66
Text Substitution 2–67
Compile and Run 2–68
Relationship to Commands 2–69
Using with Command Keywords 2–69
Using with Command Arguments 2–69
Using with Lists 2–71
Macros and Arrays 2–71
Macros and Code Blocks 2–72
Using with Database Command Conditions 2–73
Invoking Procedures and Functions 2–73
External References 2–74
Nested Macro Definitions 2–74

Arrays 2–75
Creating Arrays 2–75
Addressing Array Elements 2–76
Assigning Values to Array Elements 2–77
Multidimensional Arrays 2–79
Literal Arrays 2–80
Arrays as Function Arguments and Return Values 2–81
Traversing an Array 2–82
FOR...NEXT 2–82
AEVAL() 2–82
Empty Arrays 2–83
Determining the Size of an Array 2–84
Comparing Arrays 2–84
Changing the Size of an Array 2–85
Inserting and Deleting Array Elements 2–86
Copying Elements and Duplicating Arrays 2–86
Sorting an Array 2–87
Searching an Array 2–88
Code Blocks 2–89
Defining a Code Block 2–89
Executing a Code Block 2–90
Scope of Variables Within a Code Block 2–90
Using Code Blocks 2–92
Evaluation of Macros in Code Blocks 2–93
Storing and Compiling Code Blocks at Runtime 2–95
Objects and Messages 2–96
Classes 2–96
Instances 2–97
Instance Variables 2–98
Sending Messages 2–98
Accessing Exported Instance Variables 2–99
The Database System 2–100
Work Areas 2–101
Accessing Work Areas 2–101
Work Area Attributes 2–104

Database Files 2–105
Memo Files 2–105
Database File Attributes 2–106
Database Operations 2–106
Record Scoping 2–107
The Record Processing Primitive—DBEVAL() 2–109
Index Files 2–109
Creating 2–110
Opening 2–110
Ordering 2–111
Searching 2–111
Updating 2–112
Closing 2–112
The Input/Output System 2–113
Text Mode 2–113
Graphic Mode 2–113
Console Operations 2–115
Full-Screen Operations 2–116
Controlling Screen Color 2–117
Controlling Output Destination 2–117
Directing Output to the Printer 2–117
Directing Output to a File 2–119
The Keyboard System 2–120
Changing the Keyboard Buffer Size 2–120
Putting Characters in the Keyboard Buffer 2–121
Reading Characters from the Keyboard Buffer 2–121
Controlling Predefined Keys 2–122
Reassigning Key Definitions 2–123
Clearing the Keyboard Buffer 2–123
The Low-Level File System 2–124
Opening a File 2–124
Reading from a File 2–125
Writing to a File 2–125
Manipulating the File Pointer 2–126
Closing a File 2–126
Error Detection 2–127

Chapter 3: The Runtime Environment

In This Chapter 3–1
Setting the Workstation Environment 3–2
Files and Buffers 3–3
Changing the DOS Environment Size 3–4
Specifying the Location of Executable Files 3–4
Temporary Files 3–5
Specifying the Location of COMMAND.COM 3–5
Configuring the Serial Port 3–6
Setting the Application Environment 3–7
Application Command Line 3–9
The CLIPPER Environment Variable 3–9
Saving/Restoring EMM Page Frame—BADCACHE 3–10
Preventing Extended Cursor Use—CGACURS 3–10
Specifying Number of Dynamic Overlay File Handles—DYNF 3–10
Configuring Expanded Memory—E 3–11
Specifying the Number of Files—F 3–12
Displaying Memory Configuration Details at Startup—INFO 3–12
Preventing Detection of Idle Time—NOIDLE 3–13
Specifying Maximum Swap File Size—SWAPK 3–14
Specifying Swap File Location—SWAPPATH 3–14
Specifying Temporary File Location—TEMPPATH 3–14
Excluding Available Memory—X 3–15
The BLINKER Environment Variable 3–15
Controlling Amount of EMS Memory—/CEnnn,nn 3–16
Controlling Amount of XMS Memory—/CXnnn,nn 3–16
Controlling Operating Size of Blinker Overlay Pool—/OOnnn 3–16
Controlling Allocation of Overlay Pool—/OPc 3–17
Controlling Allocation of Overlay Pool—/OUc 3–17
Setting the Network Environment 3–17
Network Hardware Requirements 3–17
Assigning Rights 3–18
Setting Up Network Devices 3–19
The Application Batch File 3–20

Chapter 4 : Network Programming

In This Chapter 4–1
LAN Requirements for CA-Clipper 4–3
Using Shared Mode 4–3
When to Obtain Exclusive Use 4–4
Other File Open Operations 4–4
Retrying After an Open Failure 4–6
Locking 4–7
File Locking 4–9
Record Locking 4–9
Automatic Locking 4–10
Unlocking 4–11
Resolving a Failure 4–12
Overlays on a Network 4–13
Update Visibility 4–14
The Initiator 4–14
The Operating System and Other Processes 4–14
The Physical Disk 4–15
Abnormal Termination 4–16
Network Printing 4–16
Setting Up the Network Printer 4–17
Program Design Considerations 4–17
Printing to a File 4–18

Chapter 5 : Programming in Graphic Mode

In This Chapter 5–1
Overview of Graphic Mode 5–2
    Programming Similarities 5–2
    Programming Differences 5–3
        The Text Mode Technique 5–3
        The Graphic Mode Technique 5–4
Invoking Graphic Mode 5–5
Using Graphic Mode 5–6
    Shadowing Restrictions 5–7
    Enhancing an Application 5–8
    Tips and Techniques 5–9
        Push Buttons 5–9
        Bitmaps and Icons 5–9
        X/Y and Row/Column Coordinates 5–9
Compiling and Linking 5–10
Managing Bitmap and Font Resources 5–11
    RES2BML.EXE 5–11
    BML2RES.COM 5–12
    BMLDIR.COM 5–12
    Using Bitmap and Font Library Files 5–13
Troubleshooting 5–14
    Common Problems 5–14
    Software Compatibility 5–16
    Hardware Compatibility 5–17
    Memory Shortage 5–18

Chapter 6 : Introduction to TBrowse

In This Chapter 6–1
TBrowse Overview 6–2
Basic Browse Operations 6–3
    Creating TBrowse Objects 6–3
    Main Loop 6–4
        Stabilization 6–5
        Handling Keystrokes 6–6
Optimization of the Browse 6–7
    Calculated Fields, Picture Clauses, and Custom Headers 6–7
    Quicker Response Time 6–8
    Multi-User Issues 6–9
    Repositioning the Record Pointer 6–10
    Using TBrowse:cargo and TBColumn:cargo 6–12
Browsing with Get 6–13
    Determining Whether the Record Has Moved 6–16
Adding Color 6–18
    TBrowse:colorSpec 6–18
    TBColumn:defColor 6–19
    TBColumn:colorBlock 6–20
    Controlling the Highlight 6–22
    TBrowse:colorRect() 6–23
Controlling the Scope 6–24
    Viewing a Specific Key Value 6–26
    Browsing Search Results 6–30

Chapter 7 : Introduction to the Get System

In This Chapter 7–1
The Get Class 7–2
    Creating Get Objects 7–2
    Get:block 7–3
    Setting and Retrieving Values 7–4
    Get:cargo 7–5
    Get Class Protocol 7–6
The Get System 7–7
The Read Layer 7–8
Extending the Read Layer 7–9
    Nested Reads 7–9
    Using GETACTIVE() 7–10
        Changing Gets Through SET KEYs 7–10
        Improving Help Routines 7–12
    Using Code Blocks for WHEN/VALID 7–14
        Dynamic Pictures 7–14
    Processing the Entire GetList 7–16
Creating a New Read Layer 7–17
    Basic Guidelines 7–17
    Important Implementation Rules 7–18
    Implementation Steps 7–19
    READ VALID 7–19
    Gets With Messages 7–22
The Reader Layer 7–25
Creating New Reader Layers 7–26
    Basic Guidelines 7–26
    Important Implementation Rules 7–27
    Incremental Date Get 7–28
The Get Function Layer 7–31
Creating New Get Function Layers 7–32

Chapter 8 : Introduction to the Menu System

In This Chapter 8–1
Menu Terminology 8–2
Overview of Class Components 8–3
    Class Functions 8–3
    Instance Variables 8–4
    Methods 8–4
Relationships Between Menu Classes 8–5
    TopBarMenu 8–5
    MenuItem 8–5
    PopUpMenu 8–6
    More on Menu Items 8–7
        Accelerator Keys 8–7
        Action vs. Menu 8–8
        Shortcut Keys 8–8
        Status Messages 8–9
Menu Properties 8–9
    Specifying Colors 8–12
    Menu Separators 8–13
    Toggles 8–13
    Disabling Menu Items 8–15
    Cascading Menus 8–15
    Menu Item Indicators 8–16
Activating the Menu 8–17

Chapter 9 : Error Handling Strategies

In This Chapter 9–1
Overview of Exception Handling Concepts 9–2
Error Scoping 9–4
Raising Errors 9–6
Handling Errors 9–8
CA-Clipper Exception Mechanisms 9–10
The SEQUENCE Construct 9–11
The Posted Error Block 9–16
SEQUENCE vs. Error Block 9–19
Error Objects 9–21
A Model Example 9–23
The "System Error" Line-Up: Error.ch 9–24
Carrying Error-Processing Protocols 9–25
A Baseline Strategy 9–28
A Comprehensive Example 9–30
What Belongs in RECOVER? 9–32
Network Processing 9–37

Chapter 10 : CA-Clipper Compiler—CLIPPER.EXE

In This Chapter 10–1
Invoking the CA-Clipper Compiler 10–2
Using CLIPPERCMD 10–3
Compiler Options 10–4
The Compiler Script File 10–9
The Compiler Return Code 10–9
How the CA-Clipper Preprocessor Works 10–10
How CA-Clipper Compiles 10–12
The Compile and Link Batch File 10–13
Header Files 10–14
Output Files 10–15
Object Files 10–15
Temporary Files 10–15
Preprocessed Output Listing 10–16
Changing the Size of the Environment 10–16

Chapter 11 : CA-Clipper Protected Mode Linker—EXOSPACE.EXE

In This Chapter 11–1
Overview of CA-Clipper/Exospace 11–2
Invoking CA-Clipper/Exospace 11–4
The CA-Clipper/Exospace Return Code 11–7
The Compile and Link Batch File 11–7
Output Files 11–8
    Executable Files (.EXE) 11–8
    Map Files (.MAP) 11–9
    Temporary Files 11–9
How CA-Clipper/Exospace Searches for Files 11–10
    Library Files (.LIB) 11–10
    Object Files (.OBJ) 11–11
    Script Files (.LNK) 11–11
Linker Command Reference 11–12
    Compatibility with Other Linkers 11–13
        .RTLink Compatibility 11–13
        Blinker Compatibility 11–14
    EXOSPACE CLIPPER 501 11–15
    EXOSPACE ENVIRONMENT CLIPPER 11–16
    EXOSPACE ENVIRONMENT OVERRIDE 11–17
    EXOSPACE EXECUTABLE CLIPPER 11–18
    EXOSPACE EXECUTABLE NODELETE 11–20
    EXOSPACE PACKAGE 11–21
    EXOSPACE PROCEDURE DEPTH 11–23
    FILE 11–24
    LIBRARY 11–25
    MAP 11–26
    MODULE...FROM 11–27
    NODEFLIB 11–28
    OUTPUT 11–29
    STACK 11–30

Troubleshooting 11–31
If the Application Does Not Run 11–31
How Does It Fail? 11–33
Software Compatibility 11–35
Memory Shortage 11–38
Extended Memory Not Available 11–38
Using the High Memory Area 11–38
Bad Memory 11–39
Problems When Running Under DPMI 11–40
Problems in Third-Party Libraries 11–41
Environment Compatibility Issues 11–42
PC/AT Compatibility 11–42
Extended Memory 11–43
DOS Versions 11–47

Chapter 12 : CA-Clipper Real Mode Linker— BLINKER.EXE

In This Chapter 12–1
Overview of Blinker 12–2
Upgrade Offer 12–3
Blinker Command Reference 12–4
(comment) 12–4
// (comment) 12–5
@ (nested script) 12–5
BEGINAREA 12–6
BLINKER CACHE EMS 12–7
BLINKER CACHE XMS 12–8
BLINKER CLIPPER PAGE 12–9
BLINKER ENVIRONMENT CLIPPER 12–9
BLINKER ENVIRONMENT NAME 12–10
BLINKER ENVIRONMENT OVERRIDE 12–11
BLINKER EXECUTABLE CLIPPER 12–12
BLINKER EXECUTABLE NODELETE 12–14
BLINKER LINK EMS 12–14

BLINKER LINK PAGEFRAME 12–15
BLINKER LINK XMS 12–15
BLINKER MESSAGE DUPLICATES 12–16
BLINKER MESSAGE NOBLINK 12–17
BLINKER MESSAGE NOWARNING 12–18
BLINKER MESSAGE WINK 12–18
BLINKER OVERLAY OPSIZE 12–19
BLINKER OVERLAY PAGEFRAME 12–20
BLINKER OVERLAY THRESHOLD 12–21
BLINKER OVERLAY UMB 12–22
BLINKER PROCEDURE DEPTH 12–23
DEFINE 12–24
DEFLIB 12–25
ECHO 12–25
ENDAREA 12–26
EXTDICTIONARY 12–26
FILE 12–27
LIBRARY 12–28
MAP 12–29
MIXCASE 12–29
MODULE 12–30
MURPHY 12–32
NOBELL 12–32
NODEFLIB 12–33
NOEXTDICTIONARY 12–33
NOTABLEOFCONTENTS 12–34
OUTPUT 12–34
READONLY 12–35
SEARCH 12–35
SECTION INTO 12–36
STACK 12–37
UPPERCASE 12–38
VERBOSE 12–38
WORKFILE 12–39
Blinker Function Reference 12–40
BLIVERNUM 12–40

Chapter 13 : CA-Clipper Debugger—CLD.LIB

In This Chapter 13–1
Starting the Debugger 13–4
Preparing Your Programs for Debugging 13–4
Programming Considerations 13–4
Compiling Your Source Code 13–6
Invoking the Debugger 13–7
From DOS 13–7
Linking the Debugger 13–9
Controlling Execution of Application—/D or /d 13–9
Permitting Concurrent Viewing of Application and Debug Screens—/S or /s 13–10
Setting screen lines—/4[3] 13–10
Setting screen lines—/5[0] 13–10
From Windows 13–11
Invoking with Alt+D 13–11
Using ALTD() 13–12
How the Debugger Searches for Files 13–13
Using a Script File 13–14
Getting Help 13–16
Leaving the Debugger 13–16
The Debugger Display 13–17
The Debugger Menus 13–17
The Menu Bar 13–17
Menu Operations 13–19
Menu Commands 13–21
The Function Keys 13–22
The Debugger Windows 13–23
Window Operations 13–24
The Code Window 13–29
The Command Window 13–31
The Watch Window 13–34
The Monitor Window 13–37
The Callstack Window 13–39
The Help Window 13–41
The View Sets Window 13–42

The View Workareas Window 13–43
The Set Colors Window 13–44
Dialog Boxes 13–45
Debugging a Program 13–46
Executing Program Code 13–47
Modes of Execution 13–47
Finding Program Errors 13–50
Viewing Program Output 13–52
Inspecting Data and Expressions 13–53
Using Watchpoints and Tracepoints 13–53
Creating New Variables 13–56
Inspecting Program Code 13–57
Using Breakpoints 13–57
Navigating the Call Stack 13–60
Viewing Files 13–61
Accessing DOS 13–62
Menu Command Reference 13–63
?|?? 13–64
BP 13–66
Callstack 13–68
Delete 13–69
File DOS 13–71
File Exit 13–71
File Open 13–72
File Resume 13–73
Help 13–74
List 13–76
Locate Case 13–77
Locate Find 13–78
Locate Goto 13–79
Locate Next 13–80
Locate Previous 13–81
Monitor All 13–82
Monitor Local 13–83
Monitor Private 13–84
Monitor Public 13–85

Monitor Sort 13–86
Monitor Static 13–87
Num 13–88
Options Codeblock 13–89
Options Color 13–90
Options Exchange 13–91
Options Line 13–92
Options Menu 13–93
Options Mono 13–94
Options Path 13–95
Options Preprocessed 13–96
Options Restore 13–97
Options Save 13–98
Options Swap 13–99
Options Tab 13–100
Point Breakpoint 13–101
Point Delete 13–102
Point Tracepoint 13–103
Point Watchpoint 13–104
Run Animate 13–105
Run Go 13–106
Run Next 13–107
Run Restart 13–108
Run Speed 13–109
Run Step 13–110
Run To 13–111
Run Trace 13–112
View 13–112
View App 13–114
View Callstack 13–115
View Sets 13–116
View Workareas 13–117
Window Iconize 13–118
Window Move 13–119
Window Next 13–119
Window Prev 13–120

Window Size 13–120
Window Tile 13–121
Window Zoom 13–122

Chapter 14 : Program Maintenance—RMAKE.EXE

In This Chapter 14–1
Invoking RMAKE 14–2
The RMAKE Environment Variable 14–3
RMAKE Options 14–4
How RMAKE Works 14–6
How RMAKE Searches for Files 14–8
Make Files 14–8
Target and Dependency Files 14–8
The Make File 14–10
Using Quotation Marks 14–11
Line Continuation 14–11
Comments 14–12
Dependency Rules 14–12
Makepath Examples 14–13
Inference Rules 14–15
Setting Environment Variables 14–17
Directives 14–19
Macros 14–22
User-Defined Macros 14–23
Predefined Macros 14–25
A Complete Make File 14–27

Chapter 15 : Program Editor—PE.EXE

In This Chapter 15–1
Invoking the Program Editor 15–2
Navigation and Editing 15–3
Leaving the Program Editor 15–5
The PE System Architecture 15–5

Chapter 16 : Database Utility—DBU.EXE

In This Chapter 16–1
Invoking the Database Utility 16–2
The Main DBU Screen 16–3
The Menu Bar 16–3
The Message and Prompt Area 16–5
Dialog Boxes 16–5
Buttons 16–6
Fill-in Fields 16–7
Scrolling Lists 16–7
Closing a Dialog Box 16–8
Windows 16–8
Work Areas 16–9
Files 16–10
Indexes 16–10
Fields 16–11
Navigation on the Main Screen 16–12
Leaving DBU 16–13
The DBU Menus 16–13
F1 Help 16–15
F2 Open 16–16
Database 16–16
Index 16–17
View 16–17
F3 Create 16–18
Database 16–18
Index 16–23
F4 Save 16–24
View 16–24
Struct 16–24
F5 Browse 16–25
Database 16–25
View 16–26

F6 Utility 16–28
Copy 16–28
Append 16–30
Replace 16–30
Pack 16–31
Zap 16–32
Run 16–32
F7 Move 16–33
Seek 16–33
Goto 16–34
Locate 16–34
Skip 16–35
F8 Set 16–35
Relation 16–36
Filter 16–39
Fields 16–40
The DBU System Architecture 16–41

Chapter 17 : Report and Label Utility—RL.EXE

In This Chapter 17–1
Loading the Report and Label Utility 17–2
Creating and Modifying Reports 17–2
Creating or Modifying a Report 17–2
Defining Report Columns 17–3
Deleting a Column 17–6
Inserting a New Column 17–6
Locating a Column 17–6
Defining Report Options 17–7
Defining Groups 17–9
Saving the Report Definition 17–11
Printing a Report 17–11
Reporting from Related Work Areas 17–12

Creating and Modifying Labels 17–13
Creating or Modifying a Label 17–13
The Label Editor Screen 17–14
Defining the Label Dimensions and Formatting 17–15
Standard Label Formats 17–16
Defining the Label Contents 17–16
Blank Lines 17–17
Saving the Label Design 17–18
Printing the Labels 17–19
Leaving RL 17–19
The RL System Architecture 17–20

Chapter 18 : Online Documentation—NG.EXE

In This Chapter 18–1
Loading the Instant Access Engine 18–2
Using Memory-Resident Mode 18–2
Using Pass Through Mode 18–3
Accessing the Instant Access Engine 18–3
How the Instant Access Engine Searches for Files 18–4
Using the Access Window 18–5
The Menu Bar 18–5
Selecting Menus and Menu Items 18–6
Sizing and Moving the Access Window 18–6
Getting Help 18–7
Viewing a Documentation Database 18–8
The Short Entry List 18–9
Searching a Short Entry List 18–10
Expanding an Entry—Moving Down a Level 18–10
See Also References 18–11
Moving Up a Level 18–12
Selecting a New Documentation Database 18–13
Instant Access Engine Navigation Keys 18–14

Configuring the Instant Access Engine 18–15
Toggling Color 18–16
Toggling Auto Lookup 18–16
Changing the Hot Key 18–17
Saving the New Configuration 18–17
Leaving the Instant Access Engine 18–18
Exiting the Instant Access Engine 18–18
Uninstalling the Instant Access Engine 18–18

Appendix A : CLIPPER.BML

Index

Chapter 1 Introduction

In This Guide

This is the *Programming and Utilities Guide* for CA-Clipper. It contains conceptual information on programming and creating applications using the CA-Clipper development system, as well as usage information for each utility in the package.

Further information on many of the topics covered in this guide can be found in the *Reference Guide*. In addition, you will find much of the utilities portion of the *Programming and Utilities Guide* packaged in other forms to make access more convenient.

For online information accessible while operating your program editor or any other development utility, use *The Guide to CA-Clipper Utilities* database (see "Online Documentation—NG.EXE" chapter of this guide for documentation). For a brief summary of the utilities, use the *Quick Reference Guide*.

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.

Part 1: Basic Concepts

This section builds on the "Language Reference" chapter of the *Reference Guide* by focusing on various aspects of the programming language and explaining how some of the language components fit together. Although not as exhaustive as the "Language Reference," this chapter gives you a different view of the language, based on systems and other logical groupings, that may help you better understand how to use CA-Clipper to accomplish your programming tasks.

Part 2: Programming

This section is divided into several chapters, each of which gives you specific details on a particular aspect of the CA-Clipper system that may require further programming. It contains conceptual information on programming and creating applications using the following subsystems: TBrowse, The GET System, and Error Handling, as well as chapters on "The Runtime Environment," "Network Programming," and "Programming in Graphic Mode." Further reference information can also be found in the "Language Reference" chapter of the *Reference Guide*.

Part 3: Utilities

This section documents each utility supplied with the CA-Clipper package in a separate chapter. The complete command line syntax as well as all options are completely specified for each utility. In addition, usage information, including navigation and selection, is provided for the menu-driven utilities.

The following utilities are covered:

- CA-Clipper Compiler—CLIPPER.EXE
- CA-Clipper Linker—BLINKER.EXE
- CA-Clipper Debugger—CLD.LIB
- Program Maintenance—RMAKE.EXE
- Program Editor—PE.EXE
- Database Utility—DBU.EXE
- Report and Label Utility—RL.EXE
- The Guide to CA-Clipper—NG.EXE

The symbols and conventions used to document the command line syntax and options for these utilities are the same as those used in the *Reference Guide*. For instance, square brackets ([ ]) delimit optional items, angle brackets (< >) delimit user-supplied items, and required keywords are shown in uppercase letters. If you are unfamiliar with the conventions, refer to the introduction of the *Reference Guide* for further details.

Chapter 2
Basic Concepts

In This Chapter

The "Language Reference" chapter of the *Reference Guide* introduced you to programming in CA-Clipper by defining all of the components (for example, commands, functions, and operators) that go together to make up the programming language. This chapter builds on the "Language Reference" material by focusing on various aspects of the programming language and explaining how some of the language components fit together. Although not as exhaustive as the "Language Reference," this chapter gives you a different view of the language, based on systems and other logical groupings, that may help you better understand how to use CA-Clipper to accomplish your programming tasks.

This chapter provides an overview of the CA-Clipper language constructs and its major systems. The following topics are covered:

- Structure of a CA-Clipper Program
- Functions and Procedures
- Control Structures
- Variables
- Expressions
- Data Types
- Operators
- The Macro Operator

- Arrays
- Code Blocks
- Objects and Messages
- Database, Input/Output, Keyboard, and Low-level File Systems

The Structure of a CA-Clipper Program

A CA-Clipper program is a collection of statements that adhere to the rules defined by the CA-Clipper language. A program statement can take many different forms including:

- Command invocation
- Function call (for example, library, pseudo, or user-defined)
- Procedure call
- Preprocessor directive
- Control structure or declaration statement
- Assignment statement
- Comment

A program is stored in a text file with a (.prg) extension. In general, white space (for example, blanks, tabs) is ignored by the compiler, allowing you to format your programs for readability. However, the compiler interprets the carriage return/linefeed pair as the end of a statement (see the Continuation section for exceptions).

A Typical CA-Clipper Program

The following example program, ListBox.prg, is used throughout the discussion in this section to illustrate the various components of a CA-Clipper program and to give you an idea of what a typical program looks like:

```
/* ListBox.prg
   Display a menu and process a menu choice
   Compile with /M, /N, or /W
*/

// Manifest constant definitions
#define S_ADD       " Add"
#define S_EDIT      " Edit"
#define S_REPORT    " Report"
#define S_QUIT      " Quit"

// Filewide variable declarations go here

PROCEDURE Main()

   // Local variable declarations
   LOCAL cChoice

   // Continuous loop redisplays ListBox after each
   // function return
   DO WHILE .T.

      // Function call to create ListBox with
      // default option of Quit
      cChoice := CreateListBox(S_QUIT)

      // Case structure to make function call
      // based on return value of CreateListBox()
      DO CASE

      // Functions called in the CASE structure are
      // defined in a separate .prg file, which you
      // must create, compile, and link with this
      // application
      CASE cChoice = S_ADD
         AddRecs()
      CASE cChoice = S_EDIT
         EditRecs()
      CASE cChoice = S_REPORT
         Reports()
      OTHERWISE
         EXIT      // Break out of continuous loop
      ENDCASE
   ENDDO
   RETURN
```

```
FUNCTION CreateListBox(cChoice)
   CLEAR
   @ 1, 31, 6, 45 GET cChoice LISTBOX {S_ADD, ;
      S_EDIT, ;
      S_REPORT, ;
      S_QUIT}
   SET MESSAGE TO 23 CENTER
   READ
   CLEAR
   RETURN cChoice
```

The Main Procedure Definition

The main program file in any CA-Clipper application usually has a main procedure (or function) definition located at the top of the program (.prg) file. In our example, the main procedure definition begins with the following statement:

```
PROCEDURE Main()
```

This statement marks the beginning of the procedure and gives it a name. After the application is compiled and linked, it is this procedure that serves as a startup routine when the application is executed. In fact, if your application is designed with a main procedure, you must compile with the /N option or, from the Workbench, uncheck Create Default Procedure in the Compiler Options dialog box to prevent the compiler from generating a startup procedure of its own.

Like other procedure and function definitions, the main procedure consists of statements, commands, and function calls that perform a particular task. Typically, variable declarations are placed at the top of the procedure.

Following the variable declarations is the main procedure body, which usually consists of one or more function calls. Depending on the nature of the application, you can place the function calls in a control structure as in ListBox.prg:

```
DO WHILE .T.

    cChoice := CreateListBox(S_QUIT)

    DO CASE
    CASE cChoice = S_ADD
        AddRecs()
    CASE cChoice = S_EDIT
        EditRecs()
    CASE cChoice = S_REPORT
        Reports()
    OTHERWISE
        EXIT
    ENDCASE
ENDDO
```

Finally, the end of the main procedure can be marked with a RETURN statement. The RETURN statement is not a required part of any procedure definition but is included for readability. If no RETURN statement is present, the next PROCEDURE or FUNCTION statement (or the end of file) is used to indicate the end of the previous definition.

Function and Procedure Calls

Within a routine, calls to other routines are made using the routine name. For functions, the name must be followed by its arguments enclosed in parentheses. A function call is an expression whose data type is determined by its return value. Therefore, a function call can be part of another expression. Function calls are the same regardless of how the function is defined. For example, calls to the standard CA-Clipper library functions are identical to your own user-defined function calls.

The following line of code from ListBox.prg illustrates a function call as part of another statement:

```
cChoice := CreateListBox(S_QUIT)
```

Procedures, on the other hand, always return NIL. Thus, a call to a procedure is normally made as a separate program statement. A procedure can, however, accept parameters. Parameters are passed to a procedure using the same syntax as a function (for example, a parameter list enclosed in parentheses).

Thus, in ListBox.prg, AddRecs, EditRecs, and Reports could be defined as functions or procedures without affecting the calling convention.

Variable Declaration

In CA-Clipper, dynamic variables can be created and initialized without formal declaration statements. The default scope of an undeclared dynamic variable is private to the routine in which the variable is first initialized. In addition, CA-Clipper is a weak-typed language, which means that the data type of a variable is never declared—only the *scope*, or visibility.

Lexically scoped variables must be declared. In ListBox.prg, there is one declaration statement that sets up a local variable to receive the value of a menu choice:

```
LOCAL cChoice
```

This statement declares the variable *cChoice* as a local variable. Except in one case (that is, the PUBLIC statement), variable declarations assign NIL to the variable as its initial value. An initial value can be assigned to a variable using the inline assignment:

```
LOCAL cChoice := S_QUIT
```

Where you place a variable declaration in a program file helps determine its scope. For instance, STATIC declarations, that are made before the first procedure or function definition, have a filewide scope (that is, are visible to all routines in the file), whereas those made within a routine definition are local to that routine. See the Variables section in this chapter for more information on variable declaration and scoping.

Function and Procedure Definitions

Function and procedure definitions are also part of a typical program. In ListBox.prg, there is only one function definition which is shown below:

```
FUNCTION CreateListBox(cChoice)
   CLEAR
   @ 1, 31, 6, 45 GET cChoice LISTBOX {S_ADD, ;
      S_EDIT, ;
      S_REPORT, ;
      S_QUIT}
   SET MESSAGE TO 23 CENTER
   READ
   CLEAR
   RETURN cChoice
```

This definition is typical, consisting of the declared function name with a parameter list in parentheses, followed by the function body, and ending with a return statement that passes a value to the calling routine. All function definitions must return a value in CA-Clipper.

The definition of a procedure is like that of a function except for these differences: the keyword PROCEDURE precedes the procedure name and parameters, and a return value is unnecessary since procedures automatically return NIL.

Normally, routines defined within a program file are only used by other routines in that file. More generic routines are usually stored in a separate file, often called a procedure or function file, so they can be shared by many programs. For example, in ListBox.prg the functions AddRecs(), EditRecs(), and Reports() are called but not defined in the program. Instead, you should create these functions and place them in a separate program file, which you should then compile and link with ListBox.prg.

Preprocessor Directives

In addition to statements, preprocessor directives can be part of a CA-Clipper program. These directives are instructions to the compiler rather than statements that are compiled. Preprocessor directives always begin with a hash symbol (#).

In ListBox.prg, there are several manifest constant definitions at the beginning of the file. These directives assign constant values to identifier names that can be used in statements throughout the program. Whenever the compiler encounters the identifier name, it substitutes the associated value:

```
#define S_ADD        " Add"
#define S_EDIT       " Edit"
#define S_REPORT     " Report"
#define S_QUIT       " Quit"
```

Comments

You can formulate program comments in several ways, depending on the effect that you want. For long comments that span several lines, use the /*...*/ to delimit the comment block. All text following the slash-asterisk (/*), including carriage returns, is ignored by the compiler until an asterisk-slash (*/) is encountered to indicate the end of the comment block. The following example illustrates:

```
/* ListBox.prg
   Display a menu and process a menu choice
*/
```

For single-line comments, use the double-slash (//) to introduce the comment. All text following this comment symbol until the next carriage return/linefeed is treated as a comment to be ignored by the compiler. In our example, there are several comments formulated in this way throughout the code, one of which follows:

```
// Continuous loop redisplays ListBox after each
// function return
```

You can also use the asterisk (*) symbol for single-line comments, as in dBASE programs.

Comments need not be in a block or on a line by themselves. The double-slash can also be used to place a comment at the end of another statement line as in the following example:

```
EXIT                    // Break out of continuous loop
```

You can also use the double-ampersand (&&) for inline comments. With both inline comment symbols, the compiler ignores all text that follows the symbol until the next carriage return/linefeed.

Continuation

Since the CA-Clipper compiler interprets the carriage return/linefeed pair as marking the end of a program statement, it might appear that each statement must be on a single line and, furthermore, that no more than one statement can be on a line. However, you can use the semicolon (;) character to specify multiline statements and multistatement lines.

To continue a statement on more than one line (that is, form a multiline statement), place a semicolon at the end of the first line before the carriage return/linefeed, and put the remainder of the statement on a new line. When the compiler encounters a semicolon followed by a carriage return/linefeed, it continues reading the statement on the next line. The following @...GET LISTBOX command uses semicolons to put each list box item on a new line:

```
@ 1, 31, 6, 45 GET cChoice LISTBOX {S_ADD, ;
   S_EDIT, ;
   S_REPORT, ;
   S_QUIT}
```

You can repeat this method of line continuation several times to break a single statement into several lines. Using this feature, you can format very long expressions to make them more readable.

To form a multistatement line, simply place a semicolon, without a carriage return/linefeed, between the statements. When the compiler encounters such a semicolon, it knows to read the next statement from the same line. For example, the case structure in ListBox.prg could have been coded as follows:

```
DO CASE
CASE cChoice = S_ADD; AddRecs()
CASE cChoice = S_EDIT ; EditRecs()
CASE cChoice = S_REPORT ; Reports()
OTHERWISE ; EXIT
ENDCASE
```

Reserved Words

In CA-Clipper there are several words that are reserved for internal use by CA-Clipper itself. These are listed in the "Reserved Word" appendix of the *Error Messages and Appendices Guide*. Reserved words cannot be used as identifier names in program statements. In addition to these reserved words, it is illegal for an identifier to start with an underscore.

Functions and Procedures

Functions and procedures are the basic building blocks of CA-Clipper programs. Every program is made of one or more procedures and/or functions. These building blocks consist of a group of statements that perform a single task or action. They are similar to functions in C, Pascal, or other programming languages.

The visibility of function and procedure names falls into two classes. Functions and procedures visible anywhere in a program are referred to as *public* and are declared with FUNCTION or PROCEDURE statements. Functions and procedures that are visible only within the current program (.prg) file are referred to as *static* and are declared with STATIC FUNCTION or STATIC PROCEDURE statements. Static functions and procedures are said to have filewide scope.

Static functions and procedures are quite useful for a number of reasons. First, they limit visibility of a function or procedure, thereby restricting access to the function or procedure. Because of this, subsystems defined within a single program (.prg) file can provide an access protocol with a series of public functions or procedures, and conceal the implementation details of the subsystem within static functions and procedures.

Second, since the static function or procedure references are resolved at compile time, they preempt public functions and procedures which are resolved at link time. This assures that within a program file, a reference to a function or procedure always executes the static routine if there is a public routine of the same name.

Defining Functions and Procedures

Function and procedure definitions are quite similar with slightly different requirements for return values.

Functions

Functions can be defined anywhere in a program (.prg) file, but definitions cannot be nested. A function definition has the following basic form:

```
[STATIC] FUNCTION <identifier>[(<parameter list>)]
    [<variable declarations>]
    .
    . <executable statements>
    .
    RETURN <exp>
```

A function definition consists of a function declaration statement with an optional list of declared parameters. Following the function declaration are a series of optional variable declaration statements such as LOCAL, STATIC, FIELD, or MEMVAR. As mentioned before, a function must return a value, therefore a RETURN statement with a return value must be specified somewhere in the body of the function. A function definition begins with the FUNCTION declaration statement and ends with the next FUNCTION or PROCEDURE statement or end of file.

If the function declaration begins with the STATIC keyword, the function is visible only to procedures and functions declared within the same program (.prg) file.

The following is a typical example of a function definition:

```
FUNCTION AmPm(cTime)
    IF VAL(cTime) < 12
        cTime += " am"
    ELSEIF VAL(cTime) = 12
        cTime += " pm"
    ELSE
        cTime := STR(VAL(cTime) - 12, 2) + ;
            SUBSTR(cTime, 3) + " pm"
    ENDIF
    RETURN cTime
```

Procedures

Procedures are identical to functions with the exception that no return value is required and, therefore, no RETURN statement is required. Like functions, procedures can be defined anywhere in a program (.prg) file, but definitions cannot be nested. The following is the general form of a procedure definition:

```
[STATIC] PROCEDURE <identifier>[(<parameter list>)]
    [<variable declarations>]
    .
    . <executable statements>
    .
    [RETURN]
```

A procedure definition begins with a PROCEDURE declaration statement and ends with the next FUNCTION or PROCEDURE statement, or end of file.

If the procedure declaration begins with the STATIC keyword, the procedure is visible only to procedures and functions declared within the same program (.prg) file.

Calling Functions and Procedures

Although functions and procedures are quite similar they can be called in different ways.

Functions

All functions are called in the same way, regardless of whether they are defined in your application or in a library that you are linking. You can specify functions either in expressions or as statements.

For example, the following are all legitimate function calls:

```
// Expression
? "This is the " + Ordinal(DATE()) + "day"
Report()                              // Statement
result := Report("Quarterly")         // Expression
```

When a function is called, it must always be specified including the open and close parentheses. If arguments are to be passed to called functions, they are specified between the parentheses and separated by commas. For example:

```
? MajorFunc("One", "Two", "Three")
```

Procedures

A procedure can be called using function-calling syntax by specifying the procedure call as a statement. Here, you call the procedure as you would a function, specified as a statement:

```
<procedure>([<argument list>])
```

The second way is using the DO...WITH command. This method is not recommended since it passes arguments by reference as a default.

Unlike a function, however, a procedure cannot be called as part of an expression. This is because procedures cannot return values as functions can and, therefore, cannot be evaluated within the context of an expression.

Passing Parameters

When you invoke a procedure or function, you can pass values and references to it. This facility allows you to create black box routines that can operate on data without any direct knowledge of the calling routine. The following discussion defines the various aspects of passing parameters in CA-Clipper.

Arguments and Parameters

Passing data from a calling routine to an invoked routine involves two perspectives, one for the calling side and one for the receiving side. On the calling side, the values and references passed are referred to as *arguments* or actual parameters.

For example, the following function call passes two arguments, a constant and a variable:

```
LOCAL nLineLength := 80
? Center("This is a string", nLineLength)
```

On the receiving side, the specified variables are referred to as *parameters* or formal parameters. For example, here the variables are specified to receive the string to center as well as the line length:

```
FUNCTION Center(cString, nLen)
   RETURN PADC(cString, nLen, " ")
```

The specified receiving variables are place holders for values and references obtained from the calling routine. When a procedure or function is called, the values and references specified as arguments of the routine's invocation are assigned to the corresponding receiving parameter in the invoked routine.

In CA-Clipper, there are two ways to specify parameters, depending on the storage class of the parameter you want to use. Parameters specified as a part of the procedure or function declaration are declared *parameters* and are the same as local variables. They are visible only within the called routine and have the same lifetime as the routine. Parameters specified as arguments of the PARAMETERS statement are created as private variables and hide any private or public variables or arrays of the same name inherited from higher-level procedures and functions.

Note: In CA-Clipper, you cannot mix declared parameters and a PARAMETERS statement within the same procedure or function definition. Attempting this generates a fatal compiler error.

In CA-Clipper, parameters are received in the order passed and the number of arguments does not have to match the number of parameters specified. Arguments can be skipped within the list of arguments or left off the end when a routine is invoked. Received parameters without corresponding arguments are initialized to NIL. For example, the following function call skips three arguments (two from within the list and one left off the end), thereby initializing them to NIL values:

```
? TestProc("Hello",,,"There")

FUNCTION TestProc(parm1, parm2, parm3, parm4, parm5)
    ? parm1, parm2, parm3, parm4, parm5
    // Result: Hello NIL NIL There NIL
    RETURN NIL
```

When a routine is called, the PCOUNT() function is updated with the position of the last argument specified within the list of arguments. This includes skipped arguments, but not arguments left off the end of the list. In the example above, PCOUNT() returns 4.

Passing by Value

Passing an argument by value means the argument is evaluated and its value is copied to the receiving parameter. Changes to a received parameter are local to the called routine and lost when the routine terminates. In CA-Clipper, all variables, expressions, and array elements are passed by value as a default if the function-calling syntax is used. This includes variables containing references to arrays, objects, and code blocks.

As an example, the following code passes a database field and a local variable to a procedure by value:

```
LOCAL nNumber := 10
USE Customer NEW
SayIt(Customer->Name, nNumber)
? nNumber                          // Result: 10
RETURN

PROCEDURE SayIt(fieldValue, nValue)
   ? fieldValue, ++nValue          // Result: Smith 11
   RETURN
```

The importance of pass-by-value is that the called routine cannot change the caller's data by changing the value of the received parameter. This increases modularity by relegating the responsibility to the calling routine to determine whether it allows its data to be changed by a called routine.

Passing by Reference

Passing an argument 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.

Variables other than field variables and array elements can be passed by reference if prefaced by the pass-by-reference operator (@) and the function or procedure is called using the function-calling syntax. For example:

```
LOCAL nNumber := 10
USE Customer NEW
SayIt(Customer->Name, @nNumber)
? nNumber                          // Result: 11
RETURN

PROCEDURE SayIt(fieldValue, nValue)
   ? fieldValue, ++nValue          // Result: Smith 11
   RETURN
```

In this example, the change to the *nValue* parameter in the called routine is reflected in the *nNumber* argument after SayIt() is called.

As you can see, passing arguments by reference can be quite dangerous if parameters are inadvertently assigned new values in the called routine. Because of this, passing by reference should only be used in special cases such as returning a modified value from a procedure or function. A good example of this is the FREAD() function which takes a buffer variable passed by reference, fills the variable with characters read from a binary file, and returns the number of bytes read.

Note that arguments passed to routines called with the DO...WITH statement are passed by reference as a default. Note also that other dialects commonly pass arguments by reference as a default. In CA-Clipper, this practice is strongly discouraged and therefore use of the DO statement is not recommended. All DO invocations should be replaced with function-calling syntax, and variables passed by reference should be prefaced with the pass-by-reference operator (@).

Passing Arrays and Objects as Arguments

When using function-calling syntax, variables containing references to arrays and objects are passed by value as are variables containing values of any other data type. This may seem confusing since it reveals that passing references involves a level of indirection. But when a variable containing a reference is passed to a routine, a copy of the reference is passed instead of the actual reference.

If, as an example, an array is passed to a routine by value and the called routine changes the value of one of the array elements, the change is reflected in the array after the return, since the change was made to the actual array via the copied reference to it. If, however, a reference to a new array is assigned to a variable passed by value, the new reference is discarded upon return to the caller, and the array reference contained in the original variable is unaffected.

Conversely, if the variable containing a reference to an array or object is passed by reference by preceding the argument variable with the pass-by-reference operator (@), any changes to the reference are reflected in the calling routine upon return.

For example:

```
// Change the value of an array element
a := {1, 2, 3}
ChangeElement(a)
? a[1]                              // Result: 10

// Change an array; does not work
a := {1, 2, 3}
ChangeArray(a)
? a[1]                              // Result: 1

// Change an array by passing by reference
ChangeArray(@a)
? a[1]                              // Result: 4

FUNCTION ChangeElement(aArray)
   aArray[1] := 10
   RETURN NIL

FUNCTION ChangeArray(aArray)
   aArray := {4, 5, 6}
   RETURN NIL
```

Argument Checking

In CA-Clipper, there is no argument checking, and therefore the number of parameters does not have to match the number of arguments passed. Arguments can be skipped or left off the end of the argument list. A parameter not receiving a value or reference is initialized to NIL. If arguments are specified, PCOUNT() returns the position of the last argument passed.

Since arguments can be skipped, PCOUNT() is not always an accurate gauge of what arguments have been specified. To ascertain this information, compare the parameter in question to NIL. If it is equal, you can either supply a default value or generate an argument error. The following example demonstrates this concept:

```
FUNCTION TestParms(parm1, parm2, parm3)
    IF parm2 = NIL
       ? "Parameter was not passed"
       parm2 := "default value"
    ENDIF
    .
    . <statements>
    .
    RETURN NIL
```

In addition to the NIL test, VALTYPE() can be used to test for a parameter receiving a value as well as the proper data type:

```
FUNCTION TestParms(cParm1, nParm2)
    IF VALTYPE(nParm2) != "N"
       ? "Parameter was not passed or invalid type"
    ENDIF
    .
    . <statements>
    .
    RETURN NIL
```

Note: Remember to use VALTYPE() instead of TYPE() to test the data type of declared parameters. A TYPE() applied to a declared parameter always returns "U."

Passing Arguments from the DOS Command Line

Arguments can be passed directly from the DOS command line to a program's main procedure. Parameters are specified either in a PARAMETERS statement or declared as a part on the main procedure's declaration. Remember, if you declare the main procedure, you must compile with the /N option or, from the Workbench, uncheck Create Default Procedure in the Compiler Options dialog box.

Arguments passed are all received as character strings. Specify multiple arguments by separating each argument with a space. If an argument contains an embedded space, it must be enclosed in quotation marks in order to be passed as one string. For example, the following DOS command line passes two arguments when invoking PROG.EXE:

```
PROG "CA-CLIPPER COMPILER" 5
```

The main procedure receiving parameters from the DOS command line should test the number of passed parameters with PCOUNT() to assure that critical parameters have been specified.

Returning Values from Functions

As defined, a function must return a value. To do this, it must contain a RETURN statement with an argument. A RETURN statement performs two actions: first, it terminates processing of the current routine by transferring control to the calling routine; and second, if the current routine is a function, it returns any value specified as the argument of the RETURN statement to the calling routine. The return value can be a constant or an expression. For example:

```
RETURN ("Today is " + DTOC(DATE()))
```

In CA-Clipper, a function can return a value of any data type including arrays, objects, code blocks, and NIL. For example, the following function returns an array to a calling routine:

```
FUNCTION NumArrayNew()
   LOCAL aNumArray := AFILL(ARRAY(10), 1, 1)
   RETURN aNumArray
```

RETURN statements can occur anywhere in a function definition, allowing processing to be terminated before the function definition ends. For example, the following code fragment terminates processing if a condition is true (.T.) and continues if the condition is false (.F.):

```
FUNCTION OpenFile(cDbf)
    USE (cDbf)
    IF NETERR()
        RETURN .F.
    ELSE
    .
    . <statements>
    .
    RETURN .T.
```

Note that RETURN is limited by the fact that it can return only one value. More than one value can be returned if arguments are passed by reference, although this is not a preferred solution. An aggregate data structure can be defined as an array containing other arrays, and arrays, as mentioned above, can be passed throughout a program as a single value.

If the function's return value is not used, it is discarded, as is the case when you specify a function as a statement.

Recursion

A procedure or function is recursive if it contains an invocation of itself. This can be either direct or indirect when a function calls another function that again calls the original function.

For example, the following example is a function that uses recursion to calculate the factorial of a specified number. A factorial is the product of all positive integers from one to a specified number:

```
FUNCTION Factorial(nFactorial)
    IF nFactorial = 0
        RETURN 1
    ELSE
        RETURN nFactorial * Factorial(nFactorial - 1)
    ENDIF
```

The factorial of 3 is, therefore, 1 * 2 * 3 which is 6.

Control Structures

CA-Clipper supports several control structures that let you change the sequential flow of program execution. These structures allow you to execute code based on logical conditions and to repeatedly execute code any number of times.

In CA-Clipper, all control structures can be nested inside of all other control structures as long as they are nested properly. Control structures have a beginning and ending statement, and a nested structure must fall between the beginning and ending statements of the structure in which it is nested. This section summarizes all of the control structures in the CA-Clipper language, giving examples and suggested uses for each.

Tip: There are also control structures to determine which statements in a program to compile. These are the preprocessor directives #ifdef...#endif and #ifndef...#endif. Refer to the *Reference Guide* for more information on these.

Looping Structures

Looping structures are designed for executing a section of code more than once. For example, you may want to print a report four times. Of course, you could simply code four REPORT FORM commands in sequence, but this would be messy and would not solve the problem if the number of reports was variable.

FOR...NEXT

The FOR...NEXT control structure, or FOR loop, repeats a section of code a particular number of times. To solve the printing problem, you could do this:

```
FOR i := 1 TO 4
   REPORT FORM Accounts TO PRINTER
NEXT
```

FOR...NEXT is typical of all CA-Clipper control structures. It consists of a statement that defines the conditions of the structure and marks its beginning, followed by one or more executable statements that represent the body of the structure and, finally, a statement to end the structure.

FOR...NEXT sets up a loop by initializing a counter variable to a numeric value. The counter is incremented (or decremented) each time through the loop after the body statements are performed until it reaches a specified value. By default, as in the previous example, the increment value is one, but the increment can also be specified with the FOR statement's STEP clause. Furthermore, all of the FOR parameters can be numeric expressions allowing for a variable number of loop iterations. FOR...NEXT is commonly used to process arrays on an element by element basis.

DO WHILE...ENDDO

Another type of loop is one that is based on a condition rather than on a particular number of repetitions. For example, suppose you wanted to perform a complex process using records in a database file with a particular customer number.

Without a conditional looping structure, it would be very difficult, if not impossible, to solve this problem.

The DO WHILE...ENDDO control structure (also called a DO WHILE loop) processes a section of code as long as a specified condition is true (.T.). To solve the database problem, you could do the following:

```
USE Accounts INDEX Accounts
SEEK 3456
DO WHILE Accounts->AcctNum = 3456
    .
    . <processing statements>
    .
    SKIP
ENDDO
```

This example illustrates a typical use of DO WHILE to process database file records. Note that SKIP is used as part of the loop body to advance the pointer to the next record. In a DO WHILE loop, the loop body must contain some statement that alters the loop condition; otherwise, the loop will execute forever.

EXIT and LOOP

EXIT and LOOP are special statements that can only be used inside a DO WHILE or FOR loop. EXIT transfers control out of the loop, and LOOP transfers control to the beginning of the loop. These statements are used as part of a conditional control structure to control looping behavior under unusual circumstances.

```
DO WHILE .T.
    IF EOF()
        EXIT
    ELSEIF DELETED()
        SKIP
        LOOP
    ELSE
        <processing statements>
    ENDIF
ENDDO
```

Decision-Making Structures

Decision-making structures allow you to execute one or more program statements based on a condition. For example, you may want to execute a different function based on a menu choice. CA-Clipper has two such structures, IF...ENDIF and DO CASE...ENDCASE, but they are identical in functionality.

IF...ENDIF

The IF...ENDIF control structure executes a section of code if a specified condition is true (.T.). The structure can also specify alternative code to execute if the condition is false (.F.).

The following example illustrates IF...ENDIF in a function that tests for an empty array:

```
FUNCTION ZeroArray(aEntity)
   IF VALTYPE(aEntity) = "A" .AND. EMPTY(aEntity)
      RETURN .T.
   ENDIF
   RETURN .F.
```

The next example uses IF...ENDIF to process a menu choice:

```
IF nChoice = 1
   Func1()
ELSEIF nChoice = 2
   Func2()
ELSEIF nChoice = 3
   Func3()
ELSE
   QUIT
ENDIF
```

ELSEIF specifies an alternative condition to test if the previous condition was not met, allowing multiple levels of control. It replaces nested IF...ENDIF structures. ELSE is an alternative path to execute if none of the other conditions in the structure is met.

DO CASE...ENDCASE

DO CASE and IF are equivalent structures with slightly different syntax representations. For example, the following DO CASE code segment is functionally equivalent to the previous IF example:

```
DO CASE
CASE nChoice = 1
   Func1()
CASE nChoice = 2
   Func2()
CASE nChoice = 3
   Func3()
OTHERWISE
   QUIT
ENDCASE
```

Neither decision-making structure has advantages over the other. Which one you use is purely a matter of personal preference.

Error Handling Structures

BEGIN SEQUENCE...END is a specialized control structure often used for runtime error and exception handling, as illustrated below:.

```
DO WHILE .T.
   BEGIN SEQUENCE
      .
      . <operation that may fail>
      .
   RECOVER
      IF PrintRecover()
         LOOP       // Repeat SEQUENCE block
      ENDIF
   END
   EXIT             // Escape from the operation
ENDDO
```

Variables

Variables are place holders for values and references that have a defined lifetime and visibility, and a given name. When a variable name is referenced, the value it contains is returned.

In CA-Clipper, variables are organized into *storage classes* that determine how the variable is stored, how long it lives, and where in a program its name can be used. Each storage class has a statement that either declares the variable names or creates the variable at runtime.

Lexically Scoped Versus Dynamically Scoped Variables

Public, private, and field variables have what is referred to as *dynamic scope*. Dynamically scoped variables, including names, are created and maintained completely at runtime. Nothing is resolved at compile time, and there is automatic inheritance of variables by called procedures and functions. These classes of variables are characteristic of interpreted systems and exist in CA-Clipper in order to be code compatible with interpreted dialects.

In CA-Clipper, two classes of variables have what is called *lexical scope*. These classes of variables are resolved completely at compile time and have rigidly defined scoping rules, as described below.

- Dynamically scoped variables violate the principle of modularity since to understand the operation of a routine, you must understand the operation of all the routines that call it. Each routine is a modularly constructed program that should be understood in the context of its own definition, and the program (.prg) file in which it is located.

- Lexically scoped variables are more efficient and much faster than dynamically scoped variables since they are resolved completely at compile time. Dynamically scoped variables, by contrast, must be resolved each time they are referenced in a program.

We highly recommend the use of lexically scoped variables and their substitution for all uses of dynamically scoped variables.

Lifetime and Visibility of Variables

During execution, a variable, even though declared at compile time or referred to in an executable statement, does not actually exist until some portion of the computer's internal memory is allocated to contain its value. This is known as *creating* (or *instantiating*) the variable. Some variables are created automatically, while others must be explicitly created.

Lifetime

Once created, a variable continues to exist (and possesses a value, if one has been assigned) until its memory is released. Some variables are released automatically, while others must be explicitly released. Some variables are never released. The duration of a variable's life is referred to as its *lifetime*.

Visibility

Visibility refers to the conditions under which a variable is accessible to the program during execution. Some variables, even though they have been created and assigned a value, may not be visible under certain conditions.

During execution, a single variable name can be associated simultaneously with several different variables, some or all of which may be visible. Declarations can be used to ensure that occurrences of a particular name in the source code refer to the desired variable.

The visibility and lifetime of each variable class are described later in this chapter under the sections entitled Local Variables, Static Variables, Private Variables, and Public Variables.

Declaring Variables

Variable declarations are not executable statements. Instead, they *declare*, at compile time, the names of program variables and inform the compiler of assumptions that can be made about them.

Although the declarations are not themselves executable, the assumptions they produce affect the object code generated by the compiler for references to the variables declared.

You can declare variables by naming them in STATIC, LOCAL, MEMVAR, and FIELD declaration statements. They can also be declared by naming them as declared parameters (that is, listing them in parentheses) in PROCEDURE or FUNCTION statements, or in code block definitions.

Declarations are optional for some kinds of variables (private, public and field variables) and mandatory for others (static and local variables, and declared parameters).

Compile with the /W option or, from the Workbench, check Enable Warnings in the Compiler Options dialog box to enable compile-time checking for undeclared variables.

All declaration statements must occur before executable statements.

The Scope of a Declaration

A declaration's *scope* determines the parts of the program to which the declaration applies. Declaration scope is determined lexically. That is, declarations apply only to the compilation of certain sections of source code. (Note that the scope of a declaration is not necessarily the same as either the *visibility* or the *lifetime* of the variable it declares.)

For the purpose of determining declaration scope, source code is viewed as a series of discrete lexical *units*: files, procedures and code blocks.

- A file is all of the source code within a single disk file submitted for compilation.

  Note: Code included via the #include preprocessor directive is considered part of the file containing the #include directive.

- A procedure is all source code from a PROCEDURE or FUNCTION statement until the next PROCEDURE or FUNCTION statement, or until the end of the file.

- A code block is all source code within the block delimiters that define the code block.

Some units can be nested inside of others; procedures occur within files, for example, and code blocks occur within procedures or other code blocks.

A declaration's scope is the lexical unit in which it occurs and any nested units. For example, a declaration that occurs within a procedure definition applies to that procedure and any blocks within it. A declaration that occurs within a code block (a block parameter), applies to that code block and any code blocks nested within it.

STATIC, FIELD, and MEMVAR declarations can also appear outside of any procedure or code block by specifying them before any PROCEDURE or FUNCTION statements in the source file. In this case, the declaration applies to the entire source file. Variables declared this way are said to have filewide scope. Note that to declare variables with filewide scope, you must compile with the /N option or, from the Workbench, uncheck Create Default Procedure in the Compiler Options dialog box.

A declaration in an inner unit can declare a variable with the same name as one declared in an outer unit. In this case, the inner declaration supersedes the outer, but only within the inner unit. For instance, a declaration which applies to an entire file can be superseded by a declaration in one of the procedures within that file. The superseding declaration affects only the procedure in which it occurred.

Referring to Variables

In the program source code, variables are referred to by name. When the compiler sees an occurrence of a particular name, it generates object code to assign or retrieve the value of the variable with that name.

Ambiguous Variable References

During execution, it is possible for a single name to be associated simultaneously with several different variables, some or all of which may be visible. In the absence of a declaration or an explicit qualifying alias, it can be impossible for the compiler to tell which variable a particular name will actually refer to when the program is executed. The occurrence of such a name is known as an *ambiguous variable reference.*

When the compiler encounters an ambiguous variable reference, it generates special object code (*lookup code*) which, when executed, checks to see if a variable exists with the specified name. If the name exists, it checks what kind of variable it is and whether it is visible. The order in which the various possibilities are checked depends on how the variable was referred to.

Avoiding Ambiguous Variable References

Avoiding ambiguous variable references at compile time allows the compiler to generate more efficient object code. It also increases the dependability of the program, since ambiguous references behave differently at runtime depending on a variety of circumstances.

Declarations can be used to ensure that occurrences of a particular name in the source code refer to the desired variable. Alternatively, references to variables can be explicitly *qualified* by preceding them with the alias operator.

To generate warnings of ambiguous variable references, compile with the /W option or, from the Workbench, check Enable Warnings in the Compiler Options dialog box.

Qualifying Variable References

The alias operator (->) can be used to explicitly qualify a variable reference. The following combinations are legal:

`<alias>-><name>`

Refers to a field in the work area designated by *<alias>*. A runtime error occurs if *<name>* is not a field in the designated work area.

`FIELD-><name>`

Refers to a field in the currently selected work area. A runtime error occurs if *<name>* is not a field in the currently selected work area.

`MEMVAR-><name>`

Refers to either a PRIVATE variable (if one exists) or a PUBLIC variable. The reference can be an attempt to assign a value, in which case a PRIVATE variable is created at the current activation level. Otherwise, if the reference attempts to retrieve a variable that does not exist, a runtime error occurs.

Creation and Initialization

You can create variables and supply initial values in the same statement. The following statements allow initializers:

- STATIC
- LOCAL
- PRIVATE
- PUBLIC

Note: The FIELD and MEMVAR statements do not allow initializers since they do not specify the creation of variables.

If you specify no initializer, a variable is given an initial value of NIL, except for public variables, which are given an initial value of false (.F.).

Initializers for static variables must be compile-time constant expressions. That is, the expressions must consist entirely of constants and simple operators (no variables or function calls). These are computed at compile time and are assigned to the static variables before the beginning of execution.

Initializers for other variables can be any valid CA-Clipper expression. These are computed just prior to the creation of the variables, and assigned to the variables immediately afterward. Note that this allows a private variable to be initialized to the value held by an existing private or public variable with the same name.

The following examples illustrate variable initialization with various statements:

```
STATIC lFirstTime := .T.
LOCAL i := 0, j := 1, nMax := MAXROW()
PRIVATE cString := "This is a string"
PUBLIC lFinishedYet                  // Defaults to .F.
```

Local Variables

Local variables must be declared explicitly at compile time with the LOCAL statement, as shown in the example below:

```
FUNCTION SomeFunc()
   LOCAL nVar := 10, aArray[10][10]
   .
   . <statements>
   .
   NextFunc()
   RETURN .T.
```

In the above example, the variable *nVar* has been declared as LOCAL and defined as having a value of 10. When the function NextFunc() is executed, *nVar* still exists but cannot be accessed. As soon as execution of SomeFunc() is completed, *nVar* is destroyed. Any variable with the same name in the calling program is unaffected.

Local variables are created automatically each time the procedure in which they were declared is activated. They continue to exist and retain their values until the end of the activation (that is, until the procedure returns control to the code which invoked it). If a procedure is invoked recursively (for example, calls itself), each recursive activation creates a new set of local variables.

The visibility of a local variable is identical to the scope of its declaration. That is, the variable is visible anywhere in the source code to which its declaration applies. If a procedure is invoked recursively, only the locals created for the most recent activation are visible.

Static Variables

Static variables work much like local variables, but retain their values throughout execution. Static variables must be declared explicitly at compile time with the STATIC statement.

Their scope depends on where they are declared. If they are declared within the body of a function or procedure, their scope is limited to that routine. If they are declared outside of any routine, their scope is filewide. Remember that to declare variables with filewide scope, you must compile with the /N option or, from the Workbench, uncheck Create Default Procedure in the Compiler Options dialog box.

In this example, the variable *myVar* is declared as STATIC and initialized to a value of 10:

```
FUNCTION SomeFunc()
    STATIC myVar := 10
    .
    . <statements>
    .
    NextFunc()
    RETURN .T.
```

When the function NextFunc() is executed, *myVar* still exists but cannot be accessed. Unlike variables declared as LOCAL or PRIVATE, *myVar* continues to exist and retain its current value when execution of SomeFunc() completes; however, it can be accessed only by subsequent executions of SomeFunc().

The initialization of static variables (in this example to a value of 10) occurs only once, at application startup. This prevents static variables from being initialized each time the declaring routine is invoked. Static variables continue to exist and retain their value throughout the execution of the application. They cannot be released.

Private Variables

Declarations are optional for private variables. If desired, they can be declared explicitly at compile time with the MEMVAR statement.

Private variables are created dynamically at runtime with the PRIVATE and PARAMETERS statements. Additionally, assigning a value to a previously uncreated variable automatically creates a private variable. Once created, a private variable continues to exist and retain its value until termination of the procedure activation in which it was created (that is, until the procedure which created it returns to the invoking procedure). At that time, it is automatically released. Private variables can also be explicitly released using the RELEASE statement.

It is possible to create a new private variable with the same name as an existing private variable. However, the new (duplicate) variable can only be created at a lower activation level than any existing private variable with the same name. The new private *hides* any existing privates or publics (see Public Variables below) with the same name.

Once created, a private variable is visible to the entire program until it is released (either explicitly or automatically, see above) or hidden by the creation of a new private variable with the same name. In the latter case, the existing private becomes inaccessible until the new private is released.

In simpler terms, a private variable is visible within the creating procedure and any procedures called by the creating procedure, unless such a called procedure creates its own private variable with the same name.

For example:

```
FUNCTION SomeFunc()
    PRIVATE myVar := 10
    .
    . <statements>
    .
    NextFunc()
    RETURN .T.
```

In this example, the variable *myVar* is created as a private variable and initialized with a value of 10. When the function NextFunc() is executed, *myVar* still exists and, unlike a local variable, can be accessed by NextFunc(). When SomeFunc() terminates, *myVar* is released and any previous *myVar* definition becomes accessible.

Public Variables

Declarations are optional for public variables. If desired, they can be declared explicitly at compile time with the MEMVAR statement.

You can create public variables dynamically at runtime with the PUBLIC statement. They continue to exist and retain their values until the end of execution unless they are explicitly released with the RELEASE statement.

It is possible to create a private variable with the same name as an existing public variable. You cannot, however, create a public variable with the same name as an existing private variable.

Once created, a public variable is visible to the entire program until it is explicitly released or hidden by the creation of a private variable with the same name. Then the new private variable *hides* the existing public variable of the same name, and the public variable becomes inaccessible until the new private is released. For example:

```
FUNCTION SomeFunc()
    PUBLIC myVar := 10
    .
    . <statements>
    .
    NextFunc()
    RETURN .T.
```

In this example, *myVar* is created as a public variable and initialized with a value of 10. When the function NextFunc() executed, *myVar* still exists and can be accessed. Unlike LOCAL and PRIVATE variables, *myVar* still exists when execution of SomeFunc() is completed.

Declaring a variable PUBLIC when the variable does not already exist creates a new logical variable with an initial value of false (.F.).

Field Variables

Declarations are optional for field variables. You can declare them explicitly at compile time with the FIELD statement.

Field variables are really just synonyms for database fields of the same name. Thus, they typically exist and possess values even before the program begins execution, and they continue to exist after the program terminates. A field variable's value depends on which record of the associated database is the *current* record.

Once a database file is opened, the corresponding field variables are visible to the entire program. They remain visible until the database file is closed.

Field variables are initialized with empty values when records are appended to the database file.

When a variable is declared as a FIELD, all subsequent unqualified references to that variable cause the program to look for a database field in the current work area (or to a specified work area if the IN clause is used) with that name.

Note: Avoid issuing compiler directive ALIASing statements within executable code.

Expressions

In CA-Clipper, all data items are identified by type for the purpose of forming expressions. The most basic data items (that is, variables, constants, and functions) are assigned data types depending on how the item is created. For example, the data type of a field variable is determined by its database file structure, the data type of a constant value depends on how it is formed, and the data type of a function is defined by its return value.

These basic data items are the simplest expressions in the CA-Clipper language. More complicated expressions are formed by combining the basic items with operators. This section defines all of the valid CA-Clipper data types and operators that you will use to form expressions, as well as any special rules you need to know.

Data Types

The data types supported in the CA-Clipper language are:

- Array
- Character
- Code Block
- Numeric
- Date
- Logical
- Memo
- NIL

In CA-Clipper, an array structure is a separate data type, and there are distinct operations for arrays that are invalid for other data types. Code blocks are also a data type, but the operations that you can perform using them is limited. Separate sections in this chapter discuss Arrays and Code Blocks.

This section discusses each of the simple data types, providing you with the following information: when to use the data type; how to form its constants, or literals; what limitations apply; and what operations are available.

Character

The character data type identifies data items you want to manipulate as fixed length strings. The CA-Clipper character set is defined as the entire printable ASCII character set (that is, CHR(32) through CHR(126)), the extended ASCII graphics character set (that is, CHR(128) through CHR(255)), and the null character, CHR(0).

Valid character strings consist of zero or more characters in the defined character set with a maximum of 65,535 characters per string (that is, 64 KB minus one character used as a null terminator).

Character string literals, or constants, are formed by enclosing a valid string of characters within a designated pair of delimiters. In the CA-Clipper language, the following delimiter pairs are designated:

- two single quotes (for example, 'one to open and one to close')
- two double quotes (for example, "one to open and one to close")
- left and right square brackets (for example, [left to open and right to close])

Note: To express a null string, use only the delimiter pair with no intervening characters—including spaces. For example, "" and [] both represent a null string.

Since all designated delimiter characters are part of the valid character set, the delimiter characters themselves can be part of a character string. To include any of these characters in a character string literal, you must use an alternate character for the delimiter.

For example, the following string:

- I do not want to go.

could be expressed as:

- "I do not want to go."

Similarly, the string:

- She said, "I have no idea."

can be represented with:

- 'She said, "I have no idea."'

This constraint should not be too limiting since you have three pairs of delimiters from which to choose.

Note: The memo data type discussed later in this section represents variable length character data. It is a true variable length data type that can only exist in the form of a database field. Any operation that is valid for character strings is also valid for memo fields, and vice versa.

Character strings are compared according to the ASCII collating sequence. Memo fields can also be compared to character strings. When SET EXACT is OFF two character strings are compared according to the following rules; assume two character strings A and B where the expression to test is (A = B):

- If B is null, return true (.T.).
- If LEN(B) is greater than LEN(A), return false (.F.).
- Compare all characters in B with A. If all characters in B equal A, return true (.T.); otherwise return false (.F.).

With SET EXACT ON, two strings must match exactly, except for trailing blanks. For complete details on SET EXACT, see the *Reference Guide, Volume 2.*

Memo

The memo data type is used to represent variable length character data. It is a true variable length data type that can only exist in the form of a database field. A memo field uses ten characters in the database record, space that is used to store a pointer to the actual data, which is stored in a separate (.dbt) file.

The contents of a memo (.dbt) file are handled in blocks of 512 bytes. Each memo field in the database (.dbf) file contains the block number in ASCII which identifies the memo field data location. If the memo field contains no data there is no number in the database (.dbf) file. When the user writes to a memo field, the next available block is used and its number is stored in the block number field.

In CA-Clipper, when you change a memo field with less than 512 bytes, the existing block is used until it is filled. Once full, the block is discarded and copied to a new location.

Besides the fact that the length can vary in a memo field from one record to another, memo fields are identical to character fields. The character set is identical, the 65,535 maximum character limitation stands, and comparisons are performed in the same way.

Since it can apply only to a field, there is no literal representation for the memo data type. However, you can assign character string literals to memo fields with REPLACE.

Date

The date data type is used to identify data items that represent calendar dates. In CA-Clipper, you can manipulate dates in several ways, such as finding the number of days between two dates and determining what the date will be ten days from now. The date character set is defined as the digits from zero to nine and a separator character specified by SET DATE.

You can use SET EPOCH to change the default century assumption. See the entry for this command in the "Language Reference" chapter of the *Reference Guide* for more information.

Form dates by stringing together digits and separators in the order that is defined by the current SET DATE value. By default, SET DATE is AMERICAN and dates are of the form *mm/dd/[cc]yy*:

- *mm* represents the month and must be a value between 1 and 12
- *dd* represents the day which must fall between 1 and 31
- *cc*, if specified, represents the century and must be between 0 and 29
- *yy* represents the year which must fall between 0 and 99; and / is the separator

CA-Clipper supports all valid dates in the range 01/01/0100 to 12/31/2999 as well as a null, or blank, date. You can specify the century as part of a literal date regardless of the status of SET CENTURY; however, you will not be able to enter non-twentieth century dates in @...GET variables, nor will you be able to display the century, unless SET CENTURY is ON.

Dates do not have a straightforward literal representation like character strings and numbers. Instead, you must use the CTOD() function to convert a character string constant expressed in date form into an actual date value. To do this, you form the date according to the rules described above and enclose it in one of the character string delimiter pairs. Then, you use the character date as the CTOD() argument as illustrated in the examples below:

```
CTOD("1/15/89")
CTOD('03/5/63')
CTOD([09/28/1693])
CTOD("01/01/0100")
```

CTOD() checks its argument to be sure it is a valid date. For example, 11/31/95 and 02/29/97 would not be valid dates because November has only 30 days, and 1997 is not a leap year. CTOD() converts an invalid date to a null date.

Note: To express a blank, or null date, use a null string as the argument for CTOD() (for example, CTOD("")).

Dates are compared according to their chronological order. Any non-blank date always compares as greater than a blank date. Less than and/or equality comparisons of a non-blank date to a blank date always returns false (.F.).

Numeric

The numeric data type is used to identify data items that you want to manipulate mathematically (for example, perform addition, multiplication, and other mathematical functions). The CA-Clipper numeric character set is defined as the digits from zero to nine, the period to represent a decimal point, and the plus and minus to indicate the sign of the number.

With the exception of fields, CA-Clipper stores numeric values using the IEEE standard double precision floating point format, making the range of numbers from 10^-308 to 10^308. Numeric precision is guaranteed up to 16 significant digits, and formatting a numeric value for display is guaranteed up to a length of 32 (that is, 30 digits, a sign, and a decimal point). This means that numbers longer than 32 bytes may be displayed as asterisks, and digits other than the 16 most significant ones are displayed as zeros.

When a value is stored in a numeric field of a database (.dbf) file, it is converted from IEEE format to a displayable representation. When a numeric field is retrieved from a file, it is converted back to IEEE format before any operations are performed on it. Since the displayable format of a numeric value is guaranteed up to a length of 32 bytes, this is the largest recommended numeric field length.

Form numeric literals by stringing together one or more of the following:

- A single leading plus or minus sign
- One or more digits to represent the whole portion of the number
- A single decimal point
- One or more digits to represent the fractional part of the number

Some valid numeric constants are shown below:

- 1234
- 1234.5678
- -1234
- +1234.5678
- -.5678

Note: Unlike character strings, literal numeric values are not delimited. If you enclose a number—or any other string of characters—in delimiters, it becomes a character string.

Numbers are compared according to their actual numeric value. Note that the = operator and the == operator behave identically when comparing numbers, they both check for equality to the maximum precision allowed by the IEEE 8-byte floating point format.

Numeric operations, including comparisons, are completely unaffected by any SET command. If numeric operations or comparisons appear to produce unexpected results, it is probably because of the automatic display formatting of floating point values. The display formatting is affected by the SET FIXED and SET DECIMALS commands.

Logical

The logical data type identifies data items that are Boolean in nature. Typical logical data items are those with values of true or false, yes or no, or on or off. In CA-Clipper, the logical character set consists of the letters y, Y, t, T, n, N, f, and F.

Though only two logical values are possible, in CA-Clipper there are several ways to represent these two values. To form a literal logical value, enclose one of the characters in the defined logical character set between two periods. The periods are delimiters for logical values just as quote marks are for character strings.

The literal values .y., .Y., .t., and .T. represent true. The literal values .n., .N., .f., and .F. represent false. The preferred literal representations are .T. and .F.

For the purpose of comparing logical values, false (.F.) is always less than true (.T.).

NIL

NIL is a data type that allows you to manipulate uninitialized variables without generating a runtime error. It is a data type with only one possible value, the NIL value.

The literal representation for NIL is the string of characters "NIL" without delimiters. When referenced by a display command or function, this is how NIL values display. Note also that NIL is a reserved word in CA-Clipper.

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

There are several declaration statements that automatically assign a NIL value to variables and array elements. These statements are listed in the following table and discussed below:

Operation	Description
DECLARE	Assign NIL to array elements
LOCAL	Assign NIL to local variables and array elements
PARAMETERS	Assign NIL to missing parameters
PRIVATE	Assign NIL to private variables and array elements
STATIC	Assign NIL to static variables and array elements

All variables, with the exception of PUBLIC and FIELD variables, are initialized to NIL when declared or created without an initializer. PUBLIC variables are initialized to false (.F.) when created, and FIELD variables cannot contain NIL values at all. When you create an array with a declaration statement, including PUBLIC, all elements are initialized to NIL.

When you invoke a function or procedure, if you omit an argument either by leaving an empty spot next to a comma or by leaving an argument off the end, a NIL value is passed for that argument. Note, however, that the function or procedure cannot distinguish between a NIL value that is passed explicitly (for example, an expression in the argument list that evaluates to NIL) and one that is passed as the result of omitting an argument.

Operators

Along with functions, constants, and variables, operators are the basic building blocks of expressions. An operator is like a function in that it performs a specific operation and returns a value. This section gives a general discussion of operators, and categorizes and describes all of the operators available in the CA-Clipper language.

Terminology

The following paragraphs discuss, briefly, several terms used regarding operators in order to help you understand the remaining material in this section.

Overloading

In CA-Clipper, the meaning of certain operators changes, depending on the data type of the operand(s). For example, the minus operator (-) can be used to concatenate character strings as well as subtract numeric or date values. Using the same symbol, or operator, for different operations is called *overloading*.

Unary and Binary Operators

All operators in CA-Clipper require either one or two arguments, called *operands*. Those requiring a single operand are called *unary operators*, and those requiring two operands are called *binary operators*.

Binary operators use *infix* notation which means that the operator is placed between its operands. Unary operators use *prefix* or *postfix* notation where the operator is placed either before or after the operand, depending on the operator and how you want to use it.

An example of a unary prefix operator is the logical negation (!) operator:

```
lTrue := .T.
? !lTrue                // Result: .F.
```

The postincrement operator (++) is an example of a unary postfix operator. This operator increments its operand by a value of one:

```
nCount := 1
nCount++                // Result: nCount is now 2
```

The multiplication operator (*) is an example of a binary operator which demonstrates infix notation:

```
? 12 * 12               // Result: 144
```

Precedence

Precedence rules determine the order in which different operators are evaluated within an expression. These rules define the hierarchy of all of the operators within an expression. Parentheses can be used to override the default and heirarchical orders and to make complicated expressions more readable.

Error Handling

Operators are used to form expressions and must, therefore, adhere to certain rules. For example, you cannot use a unary operator that allows only prefix notation as a postfix operator and, except in a few well-defined circumstances, you cannot use any binary operator on operands with different data types. All of the rules that apply to operator usage are described in this section, and using any operator incorrectly results in a runtime error.

String Operators

The following table lists each string operator and the calculation that it performs. These operators are binary, requiring two character and/or memo type operands, and return a character value:

Symbol	Operation
+	Concatenate
-	Concatenate without intervening spaces

The minus (-)concatenation operator moves the trailing spaces of the first operand to the end of the resulting string, so that there are no intervening spaces between the two original strings. The plus (+) concatenation operator leaves spaces intact.

Date Operators

The + and - mathematical operators discussed in the next section can be used to add or subtract a number of days from a date value. The result of such an operation is a date that is a number of days before (subtraction) or after (addition) the date value. The order of the operands in the case of subtraction is significant—the date value must come first. This is an exception to the rule against operations using mixed data types.

In the precedence rules defined at the end of this section, this type of mixed operation is considered to be mathematical rather than date.

You can also subtract one date from another using the mathematical subtraction operator. The result of this type of operation is a numeric value that represents the number of days between the two dates. Subtraction of one date from another is the only true date operator.

Mathematical Operators

The following table lists each mathematical operator and the calculation it performs. As a general rule, these operators require numeric type operands (see Date Operators above for exceptions) and return a numeric value. Except where noted in the table, the mathematical operators are binary. The unary operations use prefix notation:

Symbol	Operation
+	Addition or unary positive
-	Subtraction or unary negative
*	Multiplication
/	Division
%	Modulus (integer remainder of division)
** or ^	Exponentiation

Relational Operators

Below is a list of relational operators and their purpose. All of them are binary operators requiring either two operands of the same data type or at least one NIL operand.

The result of a relational operator is a logical value:

Symbol	Operation
<	Less than
>	Greater than
=	Equal
==	Exactly equal for character; equivalence for arrays and objects; equal for other data types
<>, #, or !=	Not equal
<=	Less than or equal
>=	Greater than or equal
$	Is contained in the set or is a subset of

SET EXACT affects both the $ and the = operator when comparing strings. If SET EXACT is ON, both strings must be identical in content and length (except trailing spaces) in order for these operations to return true (.T.).

The == operator compares strings (that is, character and memo types) for exact equality in length and content, including trailing spaces. For arrays, it checks that both arrays refer to the same area of memory. For all other data types, the == operator is equivalent to the = operator.

Logical Operators

The following table lists the logical operators:

Symbol	Operation
.AND.	Logical and
.OR.	Logical or
.NOT. or !	Logical negate

All of the logical operators require logical operands and, except for negate, they are binary. .NOT. and ! are unary prefix operators. The result of a logical operation is always a logical value.

The quick way to define these operators is to tell when they return true: .AND. returns true if both operands are true; .OR. returns true if either operand is true, and .NOT. returns true if its operand is false. Truth tables defining all of the logical operators are shown below:

.AND.	.T.	.F.
.T.	.T.	.F.
.F.	.F.	.F.

.OR.	.T.	.F.
.T.	.T.	.T.
.F.	.T.	.F.

.NOT.	.T.	.F.
	.F.	.T.

Assignment Operators

CA-Clipper has several operators that assign values to variables and these are summarized in the table below:

Symbol	Operation
=	Assign
:=	Inline assign
+=	Inline add (or concatenate) and assign
-=	Inline subtract (or concatenate) and assign
*=	Inline multiply and assign
/=	Inline divide and assign
^=	Inline exponentiate and assign
%=	Inline modulus and assign

All assignment operators are binary and require a single variable as the first operand. The second operand can be any valid expression, including NIL, provided that the data type is suitable for the operation. For the compound operators (that is, all except = and :=), the variable used as the first operand must exist and must be the same data type as the second operand, except for some date and numeric operations. These exceptions are described after the following table which summarizes the assignment operators by data type:

Operator	Valid Data Types
=	All
:=	All
+=	Character, Date, Memo, Numeric
-=	Character, Date, Memo, Numeric
*=	Numeric
/=	Numeric
**= or ^=	Numeric
%=	Numeric

In addition to these data types, -= and += allow a date expression as the first operand and a numeric expression as the second operand. The result is a number of days added or subtracted from the date and the new value assigned.

The assignment operators in CA-Clipper always assume that the assigned variable is a MEMVAR unless it is explicitly declared as a different storage class or explicitly qualified with an alias. The "field first" rule does not apply when assigning.

Note: With respect to the storage class of variables, there is no difference between := and =. (The only difference between := and = is that := can be used as part of an expression since it is not confused with an equality test.) Any assignment operator assumes MEMVAR if the variable reference is ambiguous. If an attempt is being made to assign to a field, the field variable must be either declared with the FIELD statement or referenced with an alias—either the assigned alias or the FIELD-> alias.

Simple Assignment

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. If used within an expression it is interpreted as the equality operator. For example:

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

are all valid simple assignment statements. The first operand, as with all other assignment statements, can be a database field as well as any other variable. The result is that the field is replaced with the new value.

For example, the following two lines of code perform exactly the same function of replacing the current value in the *CustAge* field with the numeric value 20:

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

Inline Assignment

You can use the assignment operator (:=) interchangeably with the simple assignment operator. For example:

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

duplicates the previous example.

This operator, however, is an *inline* operator which means it can be used in any command or function whose syntax allows the use of an expression. The data type of an inline assignment operation is the same as the data type of the second operand.

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 demonstrates multiple assignments using the inline assignment operator as you would use 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())
```

Compound Assignments

The compound assignment operators perform an operation between the two operands then assign the result to the first operand. These operators are summarized and defined in the following table:

Operator	Example	Definition
+=	a += b	a := (a + b)
-=	a -= b	a := (a - b)
*=	a *= b	a := (a * b)
/=	a /= b	a := (a / b)
%=	a %= b	a := (a % b)
^=	a ^= b	a := (a ^ b)

Note that the definitions for these operators use the inline assignment operator. This means that all of the compound operators can be used as inline operators. The data type for these operations is determined by the second operand using the definitions in the table above. If the assignment operation is formed correctly, this should be the data type of the first operand in the original compound assignment statement.

Note: It is not advisable to use compound assignment operators with REPLACE or UPDATE since the WITH clause provides the assignment function as part of the command syntax.

Increment and Decrement Operators

The increment (++) and decrement (--) operators are summarized in the table below:

Symbol	Operation
++	Prefix or postfix increment
--	Prefix or postfix decrement

Both are unary operators you can use with either a numeric or a date operand. Unlike other operators which can operate on more complicated expressions, the operand here must be a single, nonfield variable. The resulting data type is the same as that of the operand.

The ++ operator *increments*, or increases the value of, its operand by one, and the -- operator *decrements*, or decreases the value of, its operand by one. Thus, both operators perform an operation on, **as well as an assignment to**, the operand. The following shows how these operators might be defined in terms of addition (and subtraction) and assignment operators:

- *value*++ is equivalent to *value* := *value* + *1*
- *value*-- is equivalent to *value* := *value* - *1*

Both operators can be specified as prefix or postfix: the prefix form changes the value of the operand before the assignment is made, whereas the postfix form changes the value afterwards. In other words, the postfix form delays the assignment portion of the operation until the rest of the expression is evaluated, and the prefix form gives the assignment precedence over all other operations in the expression.

The following code illustrates the preincrement operator in an assignment statement. Since the increment occurs before the assignment takes place, both variables have the same value:

```
nValue := 1
nNewValue := ++nValue
? nNewValue                          // Result:   2
? nValue                             // Result:   2
```

The next example demonstrates the postdecrement operator. Because the assignment takes place before the original variable is decremented, the two values are not the same:

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

The next example further illustrates the difference in the prefix and postfix forms of these operators:

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

Here, the postincrement operator is used to increase 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. Thus, 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 postincrement assignment is reflected in its new value, 11.

In this last example, the predecrement operator is used to decrease the first operand of the multiplication by one, making its value 9.

Additionally, 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
```

Special Operators

The following table lists all other symbols that have special meaning in the CA-Clipper language. These are special operators that often appear in expressions:

Symbol	Operation
( )	Function or grouping
[ ]	Array element
{ }	Constant array definition
->	Alias identifier
&	Compile and execute
@	Pass by reference

Parentheses

Parentheses are used in expressions either to group certain operations in order to force a particular evaluation order or to indicate a function call. When specifying the grouping operator, the item that falls within the parentheses must be a valid expression. Subexpressions can 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.

Subscript

The subscript operator ([]) is used to reference a single array element. The name of a previously declared array must precede the left bracket and the array element index must appear as a numeric expression within the brackets.

Curly Braces

Curly braces ({}) are used to create and reference a literal array. The array elements must be within the braces and separated by commas.

Curly braces are also used to create code blocks. The code block arguments are further delimited within the curly braces with vertical bars (| |), and the expressions defining the code block are separated by commas. Though the vertical bars are required delimiters, they need not contain an argument.

Alias Identifier

The alias identifier (->) is used to make an explicit reference to a field or variable. If the name of an alias precedes the operator, it can be followed either by a field name from that database file or any other expression enclosed in parentheses. Additionally, the keywords FIELD and MEMVAR can precede the operator followed by a valid field or variable identifier.

Macro

The macro symbol (&) is the compile-and-run operator. It is a unary prefix operator whose only valid operand is a character variable. This operator is discussed in full detail under The Macro Operator later in this chapter.

Pass-by-Reference

The pass-by-reference operator (@) is valid only in the argument lists of function or procedure calls using the function-calling syntax. It is a unary prefix operator whose operand can be any variable name. It works by forcing the argument to be passed by reference rather than by value to the function or procedure.

Operator Precedence

When evaluating expressions with two or more operations that are not explicitly grouped together with parentheses, CA-Clipper uses an established set of rules to determine the order in which the various operations are evaluated. These rules, called precedence rules, define the hierarchy of all of the operators discussed so far in this section.

Note: All function calls and other special operators in an expression are evaluated before any other operators.

Precedence of Categories

For the most part, expressions in CA-Clipper are operations that manipulate a single data type. For example, an expression might concatenate several character strings or perform a series of mathematical operations on several numbers. There are, however, expressions in which the evaluation of several different types of operations is necessary. For example, a complex logical expression can involve several related operations, on different data types, that are connected with logical operators as shown in this example:

```
cString1 $ cString2 .AND. nVal1++ > nVal2 * 10
```

When more than one type of operator appears in an expression, all of the subexpressions are evaluated for each precedence level before subexpressions at the next level are evaluated. Except for multiple inline assignments, all operations at each level are performed in order from left to right; multiple inline assignments are performed from right to left.

The heirarchical order of precedence for the operators, by category, is as follows:

1. Preincrement and Predecrement
2. Mathematical
3. Relational
4. Logical
5. Assignment
6. Postincrement and Postdecrement

The nonassignment portion of the compound assignment operators (for example, the multiplication portion of *=) exists at level 2, and the assignment portion exists at level 5.

Precedence within a Category

Within each category, the individual operators also have an established precedence, which is critical—especially in mathematical operations.

Take the following two expressions:

- 5 * 10 + 6 / 2
- 6 / 2 + 5 * 10

Algebraically, these are equivalent expressions. Though worded differently, the result of both expressions is 53. If CA-Clipper did not establish an order of precedence for evaluating mathematical operators, you could not evaluate this expression correctly without using parentheses.

Without precedence among operators, all mathematical operators would be performed in order from left to right, and the two forms of the expression shown above would be evaluated as follows: the first would multiply 5 * 10, add the resulting 50 to 6, and divide the resulting 56 by 2 to obtain 28; the second would divide 6 by 2, add the resulting 3 to 5, and multiply the resulting 8 by 10 to obtain 80. Neither result is correct, and the two results are not the same.

To save you from having to use parentheses in complicated expressions and to ensure that equivalent expressions give the same result, CA-Clipper uses an established order for evaluating operations in each category.

Preincrement and Predecrement

As mentioned earlier, the prefix and postfix forms of the increment and decrement operators are considered as separate categories because they have distinct precedence levels. This category refers to the prefix form of the increment and decrement operators.

Both operators in this category (++ and --) exist at the same precedence level and are performed in order from left to right.

Mathematical

When more than one mathematical operator appears in an expression, all of the subexpressions are evaluated for each precedence level before subexpressions at the next level are evaluated. All operations at each level are performed in order from left to right. The order of precedence for the mathematical operators is as follows:

1. Unary positive and negative (+, -)
2. Exponentiation (**, ^)
3. Multiplication, division, and modulus (*, /, %)
4. Addition and subtraction (+, -)

Relational

All of the relational operators exist at the same precedence level and are performed in order from left to right.

Logical

Like the mathematical operators, the logical operators also have an established order of precedence. When more than one logical operator appears in an expression, all of the subexpressions are evaluated for each precedence level before subexpressions at the next level are evaluated. All operations at each level are performed in order from left to right. The order of precedence for the logical operators is as follows:

1. Unary negate (.NOT.)
2. Logical and (.AND.)
3. Logical or (.OR.)

Assignment

All of the assignment operators exist at the same precedence level and are performed in order from right to left. For the compound operators, the nonassignment portion (for example, addition or concatenation) of the operation is performed first, followed immediately by the assignment.

Note: The increment and decrement assignment operations exist at their own level of precedence, and are not part of assignment category.

Postincrement and Postdecrement

Both operators in this category (++ and --) exist at the same precedence level and are performed in order from left to right.

Parentheses

You can override the order of precedence for expression evaluation using parentheses. When parentheses are present in an expression, all subexpressions within parentheses are evaluated first using the precedence rules described in this section, if necessary. If the parentheses are nested, the evaluation is done starting with the innermost pair and proceeding outward.

Note that although the CA-Clipper language provides a specific order of precedence for evaluating expressions, it is better programming practice to explicitly group operations for readability and to be certain that what executes meets your expectations.

The Macro Operator

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 referred to as the macro compiler that can compile expressions, but not statements or commands.

You can specify a macro using a variable containing a character string or an expression that returns a character string. If a character variable is used to specify a macro, it is referred to as a *macro variable* and specified as follows:

```
&<macroVar>.
```

The period (.) is the macro terminator and indicates the end of the macro variable. The macro terminator is optional unless you need it to distinguish the macro variable from the adjacent text in the statement.

You can also apply the macro operator (&) to an expression, referred to as a *macro expression*, if the expression evaluates to a character value and is enclosed in parentheses:

```
&(<macroExp>)
```

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.

The macro operator (&) can perform text substitution or runtime compilation of an expression, depending on how it is specified.

Text Substitution

Whenever a reference to a private or public macro variable is encountered embedded within a character string, the contents of the macro variable are substituted for the variable reference:

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

When you use the macro operator for text substitution in this manner, the operand must be a public or private variable whose data type is character, and the macro operator must immediately precede the variable. If you fail to follow these guidelines, the macro variable will not be expanded properly.

For example, attempting to use the macro operator on a local variable results in a compiler error:

```
cMacro := "there"
? "Hello &cMacro.!"      // Compiler error!
```

Enclosing the macro variable in parentheses, on the other hand, causes the macro variable to be treated as literal text:

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

Text substitution is not limited to macro variables, however. You can use macro expressions for this purpose as illustrated below:

```
ABCXYZ := "TEST"
cMacro1 := "ABC"
cMacro2 := "XYZ"

? &(cMacro1 + cMacro2)   // Result: TEST
```

Keep in mind, though, that you still cannot substitute the contents of a declared (that is, lexically scoped) variable, because declared variable names are not known at runtime. In this example, the error cannot be caught at compile time, but at runtime the error message "Variable does not exist: ABCXYZ" is displayed:

```
LOCAL ABCXYZ := "TEST"
cMacro1 := "ABC"
cMacro2 := "XYZ"

? &(cMacro1 + cMacro2)   // Expected result: TEST
```

Compile and Run

When CA-Clipper encounters a macro variable or macro expression within an expression, the macro symbol behaves as the compile-and-run operator.

If the macro is specified as a macro variable, the contents of the macro variable is compiled by the macro compiler, then the compiled code is executed and discarded:

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

If an expression is specified enclosed in parentheses and prefaced by the macro operator (&), the expression is evaluated and the resulting character string is compiled and run just as a macro variable is:

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

One of the interesting effects of macro expressions is that you can compile a character string containing a code block definition. In this case, the character string containing the code block is compiled and the run portion of the operation returns the code block as a value. The resulting code block is saved and evaluated later with the EVAL() function.

```
bBlock := &("{|exp| QOUT(exp)}")
.
.
.
EVAL(bBlock, DATE())
```

Relationship to Commands

Because of the long history of the macro operator (&) in CA-Clipper and its antecedents, it is important to understand the precise nature of the relationship between commands and macros.

Using with Command Keywords

First, the macro operator (&) *cannot* be used to substitute or compile command keywords. In CA-Clipper, this is because commands are preprocessed into statements and expressions at compile time.

Note: Redefinition of command keywords, however, can be accomplished 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, redefinition of a command keyword can occur only at compile time—not at runtime.

Using with Command Arguments

Second, in prior versions of CA-Clipper as well as other dialects, macro variables were often used to specify the arguments of commands requiring literal text values. This included all file command arguments as well as SET commands with toggle arguments. In these instances, you can now use an *extended expression* in place of the literal argument if the expression is enclosed in parentheses. For example:

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

can be replaced with:

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

It is important to use extended expressions, especially if you are using local and static variables. Commands are generally preprocessed into function calls with command arguments translated into function arguments as legal CA-Clipper values. With file commands, for instance, filenames are *stringified* using the smart stringify result marker and passed as arguments to the functions that actually perform the desired actions. If a literal or macro value is specified 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. For example, when the following specifications of the RENAME command are preprocessed, the result text is written as shown in bold:

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

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

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 you might expect. If this seems somewhat confusing, refer to the Variables section in this chapter for more information about the difference between local and static variables, and private and public variables.

Note: Extended expressions are denoted in command syntax with metasymbols prefaced by a lowercase x.

Using with Lists

The macro operator (&) will not fully substitute or compile a list as an argument of most commands, particularly those commands in which 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])
```

Macros and Arrays

You can refer to arrays and array elements using both macro variables and macro expressions, with the restriction that the subscript references cannot be made in a PRIVATE or PUBLIC statement. A further restriction is that the macro operator (&) cannot be specified in a LOCAL or STATIC declaration statement. Attempting this generates a fatal compiler error.

For example, valid references to array elements using macro variables include:

```
cName := "aArray"
nElements := 5
cNameElement := "aArray[1]"

//Creates "aArray" with 5 elements
PRIVATE &cName.[nElements] // Creates aArray[5]
&cNameElement. := 100      // aArray[1] is now 100
&cName.[3] := "abc"        // aArray[3] is now "abc"
```

You can successfully apply a macro operator (&) to an array element if you use 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
```

Note: Because arrays are very powerful in CA-Clipper, you may not find the need to use the macro operator (&) to make variable references to arrays. Without the aid of the macro operator, you can assign array references to variables, return array references from functions, and nest array references within other arrays. In addition, arrays can be created by specifying literal arrays or using the ARRAY() function. For more information on arrays, refer to the Arrays section of this chapter.

Macros and Code Blocks

The macro operator (&) can be applied 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. You cannot use a complex expression (an expression that contains an operator and one or more operands) that includes the macro operator within a code block. If this is attempted, a runtime error occurs.

Note that this restriction has important implications on 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 when the command is preprocessed. 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.

Using with Database Command 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 as follows:

- 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 less the number of conditions that can be specified)

Invoking Procedures and Functions

Procedure and function calls can be referenced using macro variables and expressions. With DO, the macro variable reference to the procedure can comprise 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.

Instead of using the macro operator to invoke a procedure or function in this manner, however, it is much cleaner and more efficient to use a code block. For example, if you have code similar to the following:

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

You could easily replace it with code such as this:

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

The clear advantage of a code block in place of a macro evaluation is that the compilation of a string containing a code block can be saved and, therefore, must be compiled only once, whereas macro evaluations have to be compiled each time they are referenced.

External References

Procedures and functions used in macro expressions and variables, but not referenced elsewhere, must be declared external using the REQUEST statement. Otherwise, the linker will not include them in the executable file (.EXE).

Nested Macro Definitions

The processing of macro variables and expressions in CA-Clipper allows 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. For example:

```
cOne = "&cTwo"
cTwo = "cThree"
cThree = "hello"

? &cOne                    // Result: "hello"
```

Arrays

An array is a collection of related data items that share the same name. Each value in an array is referred to as an element. Array elements can be of any data type except memo, which is limited to fields. For example, the first element can be a character string, the second a number, the third a date, and so on. Arrays can also contain other arrays and code blocks as elements.

Arrays are references. This means that a variable to which an array is assigned (or which is declared as an array) does not actually contain the array. Instead, it contains a reference to the array. In addition, if a variable containing an array is passed as an argument to a procedure or function, a copy of the reference is passed. The array itself is never duplicated.

TYPE() and VALTYPE() return "A" for an array. Because array is a distinct data type in CA-Clipper, you can create complex array expressions.

Creating Arrays

Arrays can be specified with runtime statements, such as PRIVATE and PUBLIC, and with compile-time declaration statements, such as LOCAL and STATIC. The syntax for specifying an array with any of these statements is either:

```
<identifier>[<nElements1>, <nElements2>,...]
```

or:

```
<identifier>[<nElements1>][<nElements2>]...
```

Unlike other syntax representations, the square brackets are part of the array definition and must be included. Other than the first dimension, *<nElements1>*, all other dimensions are optional.

With the exception of static array declarations, whose dimensions must be completely specified at compile time, arrays in CA-Clipper are dynamic, allowing the size to be completely determined at runtime. Thus, you can create an array whose dimensions are specified as expressions.

For example:

```
LOCAL i := 12, j := 4
LOCAL myArray[i, j]
```

You can also use the ARRAY() function to create an array. With this function, you specify the dimensions as arguments, and the return value is an array reference. In fact, each of the array declaration statements translates into two parts: the declaration of the array name and the subsequent creation of an array and assignment of a reference. For example, PUBLIC myArray[12][4] is the same as:

```
PUBLIC myArray
myArray := ARRAY(12, 4)
```

Other functions that create arrays are DBSTRUCT() and DIRECTORY().

Addressing Array Elements

After an array is created, its elements are accessed using an integer index, commonly referred to as a *subscript*. To address an array element, place the subscript of the element in square brackets following the array name. For example, suppose that *myArray* is a one-dimensional array with ten elements. To address the first element, you would use:

```
myArray[1]
```

Note that subscript numbering begins with one.

To specify more than one subscript (that is, when using multidimensional arrays), you can either enclose each subscript in a separate set of square brackets, or separate the subscripts with commas and enclose the list in square brackets. For example, if *myArray* is a two-dimensional array, the following statements both address the second column element of the tenth row:

```
myArray[10][2]
myArray[10, 2]
```

It is illegal to address an element outside of the boundaries of the array. Attempting to do so results in a runtime error.

When making reference to an array element using a subscript, you are actually applying the subscript operator ([]) to an array expression. An array expression, of course, is any expression that evaluates to an array. This includes function calls, variable references, subscripting operations, or any other expression that evaluates to an array. For example, the following are all valid:

```
{"a", "b", "c"}[2]
x[2]
ARRAY(3)[2]
&(<macro expression>)[2]
(<complex expression>)[2]
```

Assigning Values to Array Elements

When you create an array with one of the declaration statements or with ARRAY(), each element is set to NIL until the array is initialized to some other value. Array initialization is the assignment of values to its elements with the assignment (=) or inline assignment operator (:=).

You can create an array and initialize it with a single function. The DBSTRUCT() function creates an array and initializes it with database file structure information. Similarly, DIRECTORY() initializes the array that it creates with disk directory information. You can assign these function results to variables which can, in turn, be referenced as arrays.

It is also possible to initialize arrays at the time they are created using declaration statements. To accomplish this, do not define the array with dimensions in the declaration statement. Instead, just give it a name. After the array name, place an inline assignment operator followed by an expression that returns an array. For example, the following statement creates a local array and initializes it with directory information:

```
LOCAL myArray := DIRECTORY("*.*", "D")
```

The only limitation on this ability is with the STATIC statement, which allows only constants and simple expressions in assignments.

The AFILL() function initializes all the elements of an existing array to a single value. For example, the following example leaves you with a local numeric variable whose value is one:

```
LOCAL myArray[30]
myArray := 1
```

The next example, on the other hand, assigns a value of one to each element in *myArray*:

```
STATIC myArray[30]
AFILL(myArray, 1)
```

You can also use AFILL() to initialize a range of elements. For example, the following lines of code initialize the first ten elements to one, the second ten elements to two, and the last ten elements to three:

```
AFILL(myArray, 1, 1, 10)
AFILL(myArray, 2, 11, 10)
AFILL(myArray, 3, 21, 10)
```

Array elements can also be initialized on an individual basis. For example:

```
myArray[1] := 1, myArray[2] := 2, myArray[3] := 3
myArray[4] = SQRT(4)
```

You can initialize array elements using any expression that results in a valid CA-Clipper data type, including code blocks and other arrays. Once initialized, array elements can be used as variables anywhere in the language that is appropriate for their data type.

Multidimensional Arrays

In CA-Clipper, you can create a multidimensional array by declaring it with more than one dimension parameter or by assigning an array to an existing array element.

You can create and maintain traditional multidimensional arrays that have a fixed number of elements in each dimension. For example, a two-dimensional array is often used to represent the rows and columns of a table.

To declare a two-dimensional array with ten rows and two columns, you might use the following:

```
PUBLIC myArray[10][2]
```

Usually, arrays that are created in this manner are expected to adhere to certain rules. For example, each column contains the same type of information for each row in the array (for example, column one might be a character string and column two a number). Since CA-Clipper does not enforce any rules on what can be stored in an array, you must implement this level of control programmatically.

The fact that you can assign an array to an existing array element allows you to dynamically change the structure of an array. For example, after an array is created, there is nothing to prevent you from doing something like this:

```
myArray[1][2] := {1, 2, 3, 4, 5}
```

This assignment changes *myArray* significantly so it can no longer be thought of as a two-dimensional array in the traditional sense because one of its elements is an array reference. In particular, a reference to myArray[1][2][1] is now valid (it returns 1) where it was not before. References to myArray[1][1][1] and myArray[2][2][1], however, result in runtime errors.

This feature of assigning an array reference to an array element is handy in many applications. The thing to remember is that you, as the programmer, must exercise whatever control you think is necessary for storing and addressing array elements. You cannot make any assumptions about the structure of an array. You must enforce that structure.

Literal Arrays

Literal arrays, sometimes called constant arrays, can be created at runtime using the following syntax:

```
{[<exp> [, <exp> ... ]]}
```

For example:

```
x := {1, 2, 3}
y := {"Hello", SQRT(x[2]), MyFunc(x)}
```

Creating an array like this is the same as declaring the array (for example, with PUBLIC or LOCAL) then assigning the values to each element individually. You can use literal arrays anywhere an array can be specified, including as literal array elements. For example, to create a two-dimensional literal array, you could do the following:

```
aTwoD := {{1, 2, 3}, {"a", "b", "c"}, {.t., .t., .f.}}
```

1	2	3
"a"	"b"	"c"
.t.	.t.	.f.

If expressed as a character string, a literal array can be compiled with the macro operator. For example:

```
cArray := "{1, 2, 3}"
aNew := &(cArray)
? VALTYPE(aNew)              // Returns: A
```

This is useful since you could not otherwise store arrays as database fields or as character values in text files. Note, however, that the macro compiler can process strings up to 254 characters in length only.

Arrays as Function Arguments and Return Values

Arrays can be passed as arguments to procedures and functions. When you specify an array as an argument to a routine, it is passed by value, as a default, if the routine is called using a function-calling convention. This means a copy of the reference to the array is passed to a receiving parameter variable. Although a copy of the reference is passed, any changes made to the referenced arrays are reflected in the original automatically. For example:

```
LOCAL myArray[10]       // All elements set to NIL
AFILL(myArray, 0)       // Elements initialized to 0
MyFill(myArray)         // Elements incremented by 1

? myArray[1]            // Result: 1

FUNCTION MyFill(tempArray)
    FOR i = 1 TO LEN(tempArray)
        tempArray[i]++
    NEXT
    RETURN NIL
```

When the function MyFill() executes, a copy of the reference to *myArray* is passed to *tempArray*. Within MyFill() the values in the referenced array are incremented by one. When the function returns, the reference held by *tempArray* is discarded, but the changes to the referenced array appear after the call.

Arrays can be returned as values from functions. For example:

```
myArray := MakeArray(10, "New Value")

FUNCTION MakeArray(nElements, fillValue)
    LOCAL tempArray[nElements]
    AFILL(tempArray, fillValue)
    RETURN tempArray
```

Traversing an Array

There are two ways to traverse an array in CA-Clipper.

FOR...NEXT

The first and most easily understood traverse method is a FOR...NEXT loop. This construct lets you step through the array one element at a time using the array subscript of the current dimension as the control variable. In the following example, each element in *myArray* is assigned its array position as a value:

```
LOCAL myArray[10]
FOR i := 1 TO 10
   myArray[i] := i
NEXT
```

To traverse a multidimensional array, nest FOR...NEXT constructs, one level for each dimension. The following example displays each element in a two-dimensional array:

```
FUNCTION ArrDisp(myArray)
   LOCAL i, j

   FOR i := 1 TO LEN(myArray)
      FOR j := 1 TO LEN(myArray[1])
         ? myArray[i][j]
      NEXT j
   NEXT i
   RETURN NIL
```

This method of array traversal is quite traditional and is probably familiar to you if have experience with a high-level language such as C, Pascal, or BASIC.

AEVAL()

The other method, AEVAL(), requires an understanding of code blocks. This function evaluates a code block for each element of an array, passing the element value as a block parameter, and returns a reference to the array. For example, the following invocation of AEVAL() displays each element in a one-dimensional array:

```
AEVAL(myArray, {|element| QOUT(element)})
```

For a multidimensional array, you must specify nested AEVAL() functions to traverse each dimension. The following example initializes and displays a two-dimensional array:

```
LOCAL myArray[10][2], i := 1
// Fill each element with row number
AEVAL(myArray, {|element| AFILL(element, i++)})
// Display each element
AEVAL(myArray, {|element| AEVAL(element, ;
   {|value| QOUT(value)})})
```

Empty Arrays

An empty array is an array with zero elements and, therefore, no dimensions. To create an empty array, declare an array with zero elements or assign an empty literal array to a variable.

For example, the following statements are equivalent:

```
LOCAL aEmpty[0]
LOCAL aEmpty := {}
```

You can use empty arrays whenever you do not know in advance what size array you will need. Later, you can add elements to the array with AADD() or ASIZE().

To test for an empty array, use the EMPTY() function. For example:

```
FUNCTION ZeroArray(entity)
   IF VALTYPE(entity) = "A" .AND. EMPTY(entity)
      RETURN .T.
   ENDIF
   RETURN .F.
```

You cannot, on the other hand, test for an empty array by comparing it to NIL. This comparison always evaluates to false (.F.).

Determining the Size of an Array

You can determine the number of elements in an array using the LEN() function. This returns the number of element slots in the array, or the last element position in the array. The following code fragment demonstrates:

```
LOCAL myArray[10][12]
? nArraySize := LEN(myArray)      // Returns: 10
```

Note that this example returns the number of elements in the first dimension of *myArray*. Remember that CA-Clipper creates multidimensional arrays by nesting subarrays within a containing array. LEN() returns the number of elements in the specified containing array.

To determine the number of elements in a subarray, you apply LEN() to the first element in the containing array:

```
? nNestedSize := LEN(myArray[1]) // Returns: 12
```

Note that LEN() returns zero for an empty array.

Note: When there is not enough memory to hold the structure you are creating, you will receive the error message "UE332: String/Array memory overflow." In this case, you need to create a smaller structure for the array. For more information see "CA-Clipper Technical Specifications" appendix in the *Error Messages And Appendices Guide*.

Comparing Arrays

The == operator compares two arrays for *equivalence*. Arrays are equivalent if they are references to the same array (that is, they point to the same location in memory). For example:

```
LOCAL myArray := {1, 2, 3}
sameArr := myArray          // Create a new reference
                            // to myArray
? sameArr == myArray        // Result: .T.
newArr := ACLONE(myArray)   // Create a new array
? newArr == myArray         // Result: .F.
```

Note that the = operator is not valid for comparing arrays.

There is no single operator that can check whether all elements are the same in two distinct arrays. To accomplish this, you must traverse the arrays and compare them element by element. The following function accomplishes this for one-dimensional arrays:

```
FUNCTION ArrComp(aOne, aTwo)
    IF LEN(aOne) <> LEN(aTwo)   // Sizes are different
       RETURN .F.
    ENDIF
    FOR i := 1 TO LEN(aOne)
       IF (aOne[i] <> aTwo[i]) // At least one
    RETURN .F.                 // element is
    ENDIF                      // different
    NEXT
    RETURN .T.                 // Sizes and elements
                               // are identical
```

Changing the Size of an Array

There are two functions, ASIZE() and AADD(), that change the size of an existing array. ASIZE() specifies the array to change and the new size as function arguments. With this function, you can make an existing array either larger or smaller. For example:

```
LOCAL myArray[10]
ASIZE(myArray, 25)   // Increase by 15 elements
? LEN(myArray)       // Returns: 25
ASIZE(myArray, 5)    // Decrease by 20 elements
? LEN(myArray)       // Returns: 5
```

When you enlarge an array, the new elements are added to the end of the array and are set to NIL. Shrinking an array (that is, making it smaller) removes elements beyond the specified new array size. ASIZE() updates the array and returns a reference to it.

AADD() enlarges an array by one element and initializes the new element to a particular value. The new element is added to the end of the array. For example:

```
LOCAL myArray[10]
AADD(myArray, 500)   // Add one element
? LEN(myArray)       // Returns: 11
? myArray[11]        // Returns: 500
```

Inserting and Deleting Array Elements

The AINS() function inserts an element into an existing array at a specified location, pushing all subsequent elements down one position. The last element in the array is lost.

ADEL() deletes an element at a specified location, and all subsequent elements are moved up one position. The last element becomes NIL.

Both ADEL() and AINS() update the original array and return a reference to it. Unlike AADD() and ASIZE(), these functions do not change the size of the array.

Copying Elements and Duplicating Arrays

ACOPY() copies elements from one array to another. The target array must exist prior to invoking the function. With this function, you can copy all elements or a particular range of elements. If the source array contains nested arrays, the target array will contain only references to the subarrays, not actual copies. ACOPY() returns a reference to the target array.

For multidimensional or nested arrays, use ACLONE() if you want to duplicate the entire array. If the source array contains a subarray, ACLONE() creates a matching subarray and fills it with copies of the values in the original subarray, as opposed to copying a reference. This function creates the target array and returns a reference to it.

The following example illustrates the difference between these two functions:

```
LOCAL aOne[4], aTwo[14]
aOne[1] := {1, 2, 3}
ACOPY(aOne, aTwo)           // Copies elements from
                            // aOne to aTwo
? aOne[1] == aTwo[1]        // Returns: .T.
aThree := ACLONE(aOne)      // Duplicate aOne in
                            // its entirety
? aOne[1] == aThree[1]      // Returns: .F.
```

The first element in *aOne* is an array. Thus, after ACOPY() is invoked to copy elements to *aTwo*, the first elements in these arrays are equivalent (that is, are references to the same subarray). ACLONE() creates *aThree* as a duplicate, and the equivalence test fails because the subarrays being compared are not the same. If, however, aThree[1] and aOne[1] were compared on an element by element basis, all elements would be the same.

Sorting an Array

The ASORT() function sorts an array and returns a reference to the newly sorted array. With this function, you can sort an entire array or a specified portion. This example sorts the entire array:

```
ASORT(myArray)
```

This example sorts elements ten through fifteen, leaving all other elements in their original place:

```
ASORT(myArray, 10, 5)
```

An additional feature of ASORT() lets you specify the sort criteria as a code block. For example, the following statement does a descending sort of an entire array:

```
ASORT(myArray,,, {|x , y| x < y})
```

Searching an Array

ASCAN() searches an array for a particular value specified as an expression or a code block. This function operates on the entire array or a specified range of elements. If used with a simple search expression, the function searches the array until it finds a matching value and returns the subscript, or zero if the value was not found. For example:

```
LOCAL aArray := {"Tom", "Mary", "Sue", "Mary"}
? ASCAN(aArray, "Mary")          // Result: 2
? ASCAN(aArray, "mary")          // Result: 0
```

If the search criteria is specified as a code block, the function is much more powerful. In the example above, the second ASCAN() could not find the name because the search was case-sensitive. The following example uses a code block to perform a search that is not case-sensitive:

```
? ASCAN(aArray, {|x| UPPER(x) == "MARY"})
// Result: 2
```

Since the function lets you specify a range of subscripts to include in the search, it can resume the search with the next element as in the following example:

```
LOCAL myArray := {"Tom", "Mary", "Sue", "Mary"}
LOCAL nStart := 1
nEnd := LEN(myArray)       // Set boundary condition
DO WHILE (nPos := ASCAN(myArray,"Mary",nStart)) > 0
   ? nPos, myArray[nPos]
   // Test boundary condition
   IF (nStart := ++nPos) > nEnd
      EXIT
   ENDIF
ENDDO
```

Code Blocks

Code blocks provide a means of exporting small pieces of executable program code from one place in a system to another. You can think of them as *assignable unnamed functions*. They are assignable because, except when executing them, CA-Clipper treats them as values. That is, they can be stored in variables, passed as arguments, and so forth.

Code blocks bear a strong resemblance to macros, but with a significant difference. Macros are character strings which are compiled *on the fly* at runtime and immediately executed. Code blocks, on the other hand, are compiled at compile time along with the rest of the program. For this reason code blocks are more efficient than macros, while offering similar flexibility.

The difference between code blocks and macros becomes especially important with declared variables. Variable declarations act at compile time and, thus, have no effect within runtime macros. Since code blocks are compiled at compile time, declarations can be used to control access to variables in code blocks. Variables that require compile-time declarations (static and local variables, and declared parameters) are never visible within runtime macros, whereas they can be accessed freely in blocks.

Defining a Code Block

The syntax of a code block is as follows:

```
{|[<argument list>]| <exp list>}
```

Both *<argument list>* and *<exp list>* are comma-separated. The expressions must be valid CA-Clipper expressions—they cannot contain commands or statements, such as control structures and declarations.

Note: The vertical bars that delimit a code block argument list must be present, even if there are no arguments, to distinguish a code block from a literal array.

Some examples of code blocks follow:

```
{|| "just a string"}
{|p| p + 1}
{|x, y|  SQRT(x) + SQRT(y)}
{|a, b, c|  MyFunc(a) , MyFunc(b) , MyFunc(c)}
```

Executing a Code Block

The EVAL() function executes a code block. EVAL() takes a block as its first argument, followed by a list of parameters. The function then executes the code block, passing the specified parameters as arguments. The expressions are evaluated in order from left to right. The returned value is the result of the last (or only) expression in the list. For example:

```
bBlock := {|nValue| nValue + 1}
? EVAL(bBlock, 1)                    // Result:  2
```

The AEVAL() and DBEVAL() functions also execute code blocks. These are iterator functions that process an array and a database file, respectively, executing a code block for each element or record.

Scope of Variables Within a Code Block

When a code block is executed, local and static variable references (other than arguments) are scoped to the procedure or function in which the code block was defined. A local variable referenced in a code block is still visible after the declaring function is no longer active, as long as there is an active reference to the code block. For example:

```
bBlock := MyFunc()
FOR i := 1 TO 10
   ? EVAL(bBlock)
NEXT
RETURN

FUNCTION MyFunc()
   LOCAL nNumber
   LOCAL bBlock
   nNumber := 10
   bBlock := {|| nNumber++}
   RETURN bBlock
```

In this example, each time the code block returned by MyFunc() is evaluated, it increments *nNumber* and returns the new value.

nNumber is accessible through the code block as the function defining the code block's activation record becomes *detached* from the activation stack when the function returns. The function's local variables then become, in essence, a free-floating array. The detached activation lives in object memory and stays alive as long as any of the blocks referring to it are alive.

This facility has significant ramifications for the capabilities of a code block. In particular, it solves the problem in which a code block instance needs long term ownership of some values. It means that you must macro-compile a code block only if the actual code to be executed is not known until runtime.

Note also that every call to the function generates unique instances of the local variables. Those variables continue to exist as long as there is a code block that refers to them.

Using this fact, along with the fact that a code block can be passed as a parameter to another program, you can export static and local variables. For example:

```
FUNCTION One()
   LOCAL myVar
   myVar := 10
   bBlock := {|number| number + myVar}

   NextFunc(bBlock)

   RETURN NIL

FUNCTION NextFunc(bBlock)
   RETURN EVAL(bBlock, 200)
```

When *bBlock* is evaluated in NextFunc(), *myVar*, which is local to function One(), becomes visible even though it is not passed directly as a parameter.

Note that if the specified variable is a private or public, the variable is not exported from the defining routine. The currently visible copy of the variable is used instead.

Using Code Blocks

Code blocks are particularly useful when creating *black box* functions which can operate without knowing what type of information is passed to them. Consider the following:

```
FUNCTION SayData(dataPassed)
   // Display dataPassed as a string
   DO CASE
   CASE VALTYPE(dataPassed) = "C"     // Character
      @ 10,10 SAY dataPassed
   CASE VALTYPE(dataPassed) = "N"     // Numeric
      @ 10,10 SAY LTRIM(STR(dataPassed))
   CASE VALTYPE(dataPassed) = "L"     // Logical
      @ 10,10 SAY IF(dataPassed, "True", "False")
   ENDCASE
   RETURN .T.
```

In this example, an error occurs if data of any type other than those specified is passed to SayData(). If, for example, a date needs to be displayed, SayData() must be modified.

In the following example, the code to be executed is stored in a block:

```
charCode := {|data| data}
numCode  := {|data| LTRIM(STR(data))}
logCode  := {|data| IF(data, "True", "False")}
dateCode := {|data| DTOC(data)}
```

SayData() can now be defined as a black box routine which can act on any type of data. The decision of which block to pass to SayData() is made in the calling program:

```
FUNCTION SayData(dataPassed, codeBlock)
   // Display dataPassed as a string
   @ 10,10 SAY EVAL(codeBlock, dataPassed)
   RETURN .T.
```

Code blocks are also used to pass an expression to a function or procedure that you do not want to execute until sometime later. A good example of this can be found in STD.CH with the definitions of the SET KEY and SET FUNCTION commands. Both commands translate into a call to the same function, although the operation performed is not the same in both cases: SET KEY assigns the name of a procedure or function to call when the specified key is pressed, while SET FUNCTION assigns a string to stuff into the keyboard buffer.

Evaluation of Macros in Code Blocks

When a code block contains a macro, an ambiguity arises regarding when the macro is evaluated. There are two possibilities, early or late evaluation.

Early Evaluation

Early evaluation means that the macro is expanded at the time the code block is created, and the expanded value remains constant for all subsequent evaluations of the block. CA-Clipper uses early evaluation by default, with an alternate method available for late evaluation.

As an example of early evaluation, consider a SET FILTER command in which the filter expression is a macro:

```
SET FILTER TO &cFilter
```

A filter expression, specified by a SET FILTER command, is executed each time a SKIP command is issued. The macro string contained in *cFilter* above, however, should be expanded only once, when the command is first issued. Otherwise, if the macro is expanded each time the filter block is evaluated, the macro variable (*cFilter* above) must be preserved for the duration of the filter.

In CA-Clipper, the SET FILTER command is implemented by use of a code block. The code for the above command, after preprocessing, is similar to the following:

```
DBFILTER({|| &cFilter})
```

Early evaluation of the macro within the code block results in the desired behavior: *cFilter* is evaluated when the block is defined (as part of the creation of the block). The block is created via the macro system and remains constant through all subsequent block evaluations, regardless of any change to the value of *cFilter*.

Late Evaluation

Late evaluation means that the macro is evaluated each time the block is evaluated.

As an example of a situation where late evaluation is required, consider a block which must perform a macro operation on one of its own parameters.

The following block attempts to assign a value to a variable by use of a macro:

```
bAssignByName := {|vName, value| &vName := value}
```

Because CA-Clipper implements early evaluation, this code block would not have the desired effect: instead of evaluating the macro each time the block is run, CA-Clipper would evaluate the macro when the block was first created. The initial value of *vName* would become part of the block.

Late evaluation is available by using a macro expression. The above code block will function correctly if defined as follows:

```
bAssignByName := {|vName, value| &(vName) := value}
```

The use of a macro expression (the macro operator applied to an expression in parentheses) prevents the early evaluation from taking place, and the code block will evaluate the macro string contained in *vName* during each block evaluation.

Storing and Compiling Code Blocks at Runtime

Although you cannot store code blocks directly to database fields or (.mem) files, you can store them as character strings. When you want to use a code block definition stored as a character string in a database field, compile the field using the macro operator then assign the resulting code block to a variable. For example:

```
USE DataDict NEW

// Create a block string and assign it to a field
variable
APPEND BLANK
REPLACE BlockField WITH "{|| Validate()}"
.
.
.
// Later compile and execute the block
bSomeBlock := &(BlockField)
EVAL(bSomeBlock)
```

Instead of storing the compiled code block to a variable and evaluating it, you could compile and evaluate in a single step with EVAL(&(BlockField)).

There is, however, a speed advantage to storing the compiled code block to a variable, since the compile step only has to be performed once. Once compiled, the variable can be passed throughout the system and evaluated at the same speed as other CA-Clipper-compiled code.

Note: The macro operator is appropriate only for use with character expressions. Thus, you cannot compile a code block directly with the macro operator.

Objects and Messages

CA-Clipper offers a limited implementation of a special data type called *object*. The concept of an object is part of the object-oriented programming paradigm.

CA-Clipper objects are complex data values which have a predefined structure and set of behaviors. A small number of predefined object types, called classes, are part of the system. These classes support particular operations in CA-Clipper. Creating new classes and subclassing of existing classes are not possible in CA-Clipper.

Note: CA-Clipper is not an object-oriented language. The CA-Clipper implementation of objects represents only a small subset of the object-oriented paradigm.

Classes

In CA-Clipper, there are several different types of objects. An object's type is known as its *class*. The information contained in an object, and the operations that can be applied to it, vary depending on the class of the object.

For each available class, there exists a special function called a *create function*. The create function facilitates the creation of new objects of the associated class.

Instances

You create a new object with a call to the create function for a particular class. The object will have the attributes and behaviors specified in the description of the class. The new object is known as an *instance* of the class.

Like CA-Clipper arrays, objects are handled referentially. This means that a program variable cannot actually contain an object. It can, however, contain a reference to an object. The variable is then said to *refer to* the object, and operations are performed on the object by applying the send operator (:) to the variable. Object references can be freely assigned, passed as parameters, and returned from functions. Two or more variables can contain a reference to the same object (if, for example, a variable containing an object reference is assigned to another variable). In this case, both variables refer to the object, and operations are performed on the object by applying the send operator to either variable.

Objects continue to live as long as there are active references to them somewhere in the system. When all references to an object are eliminated, the system automatically reclaims the space occupied by the object.

When you create a new object, the create function returns a reference to the new object. You can then assign this reference to a variable, and access the object through that variable.

Instance Variables

An object contains within it all of the information necessary for it to perform the operations specified for its class. This information is stored inside the object in special storage locations called *instance variables*.

When you create a new object, it receives its own dedicated set of instance variables. The new instance variables are automatically assigned initial values.

Most instance variables are invisible to the programmer. However, some instance variables are accessible. These are known as *exported* instance variables. Exported instance variables can be inspected and, in some cases, assigned using the send operator (:).

Sending Messages

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

The syntax for sending a message to an object is as follows:

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

For example:

```
oGet:left()
```

In this example, *oGet* is the name of a variable that contains an object reference. left() is the message being sent; it specifies the operation to be performed. The available operations (and corresponding messages) vary depending on the class of the object.

Parentheses are optional if no parameters are supplied in the message send. By convention, however, parentheses are used to distinguish a message send with no parameters from an access to an exported instance variable.

Executing a message send produces a return value, much like a function call. The return value varies depending on the operation performed. Often, this return value is a reference to the object that was just operated on (referred to as *self*). This allows multiple messages to be sent in a chain:

```
oGet:left():right()
```

Accessing Exported Instance Variables

Some classes allow access to the instance variables within objects of that class. These variables are called *exported instance variables*. The definition for a class specifies which, if any, instance variables are exported. Some exported instance variables are said to be *assignable*. An assignable instance variable can be modified using a special variant of the message send operator.

You access exported instance variables using a variant of the normal message sending syntax. The syntax for retrieving the value of an instance variable is:

```
<object>:<instance variable>
```

The syntax for assigning a new value to an assignable exported instance variable is:

```
<object>:<instance variable> := <value>
```

For example:

```
nRow := oBrowse:rowPos
oBrowse:cargo := xNewCargo
```

Conceptually, it is easiest to consider an instance variable reference (of the form *<object>:<instance variable>*) as if it is a single variable—both are valid in all of the same contexts. Instance variables, however, are much more powerful than variables. Internally, assignments and retrievals to instance are mapped to methods, which can enforce validation rules and perform other actions.

For example, in the Error class, assigning a value to the Error:genCode instance variable sets the Error:description instance variable to the default description for that type of error. This helps ensure consistency in the messages the user receives when errors occur.

Though this is useful, there is still a need to act upon the data stored in the object, independently of setting instance variables. This is accomplished by sending messages to the object to invoke the desired methods.

The Database System

The CA-Clipper database system is composed of work areas that are used to manage database and other related files, and operations designed to manipulate those files. A database file is a collection of related information that is stored in the form of a table. The design of the table, known as the structure, is also stored in the database file.

The structure is defined and added to the database file when the file is created. It consists of one or more field definitions describing the name, width, and data type attributes for each column in the table. The table rows, or records, are added to the file using append operations which understand and enforce the structure on a field by field basis. Records are added to a database file in a particular physical order that is used, by default, when accessing the file.

An index file is an ancillary file that is created separately from the database file. It lets you define and maintain a logical ordering for a database file without affecting the physical file order.

Several database files that have related structures and data can be associated, along with their index files, using the SET RELATION command. SET RELATION lets you establish relationships between several files and operate on them as a single entity known as a database, or view.

The CA-Clipper package contains a database utility, DBU.EXE, that lets you perform most database and index operations using a menu-driven system. It is an excellent tool for designing database file structures for your applications and for entering test data. Refer to the "Database Utility—DBU.EXE" chapter in the *Programming and Utilities Guide* for further information regarding DBU.

This section describes the various facets of the database system and gives a brief overview of all database and index operations.

Work Areas

The CA-Clipper database system defines 255 work areas. 250 of them are available for your use; with the remaining five reserved for internal use. A work area is essentially an area in memory in which you can manage a single database file along with an optional memo file and up to 15 index files. A work area is *occupied* or *unoccupied*, depending on whether or not it contains an open file. At application startup all work areas are unoccupied, and work area number one is the current, or active, work area.

Accessing Work Areas

To open a database file, you must first access the work area you want to use. The most common way to do this is with the SELECT command. SELECT is designed specifically to move between work areas by changing the current work area to the one you specify.

The general usage rules for SELECT are:

- To access a specific unoccupied work area, specify SELECT with a number between one and 250
- To access the next available unoccupied work area, use SELECT 0
- To access an occupied work area (that is, an area in which the database file is in USE), specify SELECT with the alias name rather than the work area number

In CA-Clipper, the USE command supports a NEW clause that SELECTs the next unoccupied work area before performing the open operation. This feature makes the SELECT 0 command obsolete, so that any code of the form:

```
SELECT 0
USE <xcDatabase>
```

can be replaced with:

```
USE <xcDatabase> NEW
```

After a USE, the selected work area remains active.

If no ALIAS clause is specified, as in the above example, the work area in which the database file is opened is assigned an alias name that is the same as the database filename. This name is used with SELECT to access the work area.

Aliased Expressions

Aliased expressions are used to temporarily access a work area for the purpose of expression evaluation. To form an aliased expression, enclose the expression in parentheses and prefix it with the desired alias name and the operator (->) (for example, <idAlias>->(<exp>)). The work area is selected, the expression evaluated, and finally the original work area restored.

You can also evaluate simple database functions and more complicated expressions in unselected work areas this way. For example:

- Cust->(EOF()) evaluates the end of file status in the Cust work area.
- Cust->(HEADER() + (LASTREC() * RECSIZE())) computes the size of the file in the Cust work area.
- Cust->Amount + total adds the value of the *Amount* field in the Cust work area to the variable *total*.

Notice that in the final example above, *Amount* is referenced explicitly with its alias name, no parentheses are used within the actual aliased expression. This explicit field reference avoids any ambiguity that may exist among field and variable names and illustrates the following point: using an aliased expression does not override any compiler assumptions about variables within the expression, whereas using an alias as an explicit field reference does.

In other words, the compiler does not assume that everything within an expression refers to a field simply because the expression is preceded by an alias. Instead, the standard precedence rules resolve ambiguities when the expression is evaluated at runtime. You can, however, explicitly reference a field with an alias, thereby removing any ambiguity that might exist at compile time.

To illustrate, suppose you have the following code that uses a local variable *amount* and a field called *Amount* in the Cust work area. This aliased expression accesses the local variable named *amount* because when the expression is evaluated, the variable is visible and takes precedence over the field with the same name:

```
LOCAL total := 0, amount := 0
total := Cust->(Amount + total)
```

Essentially, evaluating Cust->(Amount + total) is the same as selecting the Cust work area, and evaluating the expression Amount + total.

Changing this line of code as follows accesses the field *Amount* rather than the local variable. The reference to Amount using the Cust alias is an explicit field reference, which overrides the local variable of the same name:

```
total := Cust->Amount + total
```

As a good programming practice, you should make all identifier references explicit and avoid naming conflicts whenever possible. Otherwise, you may not agree with decisions made by the compiler to resolve ambiguities.

Work Area Attributes

A work area provides a means for logically grouping attributes pertaining to a database file. These are called work area attributes and are listed in the table below:

Command/Function	Attribute
SET DELETED	Delete record filter
SET FILTER	Logical record filter
ALIAS()	Database alias name
BOF()	Beginning of file flag
DBFILTER()	SET FILTER condition
DBRELATION(), DBRSELECT()	SET RELATION information
EOF()	End of file flag
FCOUNT()	Number of fields
FOUND()	Search flag
FLOCK(), RLOCK()	Lock status flags
INDEXKEY()	Index key expression
INDEXORD()	SET ORDER value
LOCATE, CONTINUE	LOCATE condition
LASTREC()	Number of records
RECNO()	Record number
SELECT()	Work area number

Database Files

The database file is the main file type and has a default extension of (.dbf). A database file, often called a *table*, consists of one variable length header record that defines the structure in terms of its field definitions, and zero or more fixed length records that contain actual data for nonmemo fields and pointer information for memo fields. Each record has one additional byte for the record delete status flag. The CA-Clipper database file format is compatible with dBASE III PLUS.

Database files can be thought of as tables, where each field in the structure represents a column of the table and each record represents a row.

Memo Files

If a database file structure has definitions for one or more memo fields, there is one memo file associated with that database file. The memo file has a default extension of (.dbt) and the same root filename as the associated database file.

By default, the database and memo files are created and maintained in the same directory.

The data for all memo fields is stored in the same memo file, no matter how many memo fields are defined in the database file structure. Each memo field in the database file contains a pointer into the memo file, so the database system can quickly locate the associated data.

The memo file is automatically opened and maintained in the same work area as the database file.

Database File Attributes

Like work areas, database files have associated attributes as listed in this table:

Function	Attribute
DELETED()	Record delete status flag
FIELD()	Field name
HEADER()	Header size in bytes
LUPDATE()	Date of last update
RECSIZE()	Record size in bytes

Database Operations

There are several operations (database operations) that can be performed on an open database file.

The following table categorizes all of the database operations. Note that all of these operations require an open database file in the currently selected work area:

Category	Command/Function
Add	APPEND BLANK, APPEND FROM, BROWSE()
Close	CLOSE, USE
Compute	AVERAGE, COUNT, SUM
Create	COPY STRUCTURE, COPY STRUCTURE EXTENDED, COPY TO, CREATE, CREATE FROM, JOIN, SORT, TOTAL
Delete	DELETE, PACK, RECALL, ZAP
Display	DISPLAY, LABEL FORM, LIST, REPORT FORM, BROWSE(), DBEDIT()

Continued

Continued

Category	Command/Function
Filter	INDEX...UNIQUE, SET DELETED, SET FILTER
Information	AFIELDS()
Iterate	DBEVAL()
Lock	SET EXCLUSIVE, UNLOCK, FLOCK(), RLOCK()
Navigate	CONTINUE, FIND, GO, LOCATE, SEEK, SKIP
Open	USE
Order	INDEX, REINDEX, SET INDEX, SET ORDER, SORT
Relate	SET RELATION
Update	@...GET <idField>/READ, REPLACE, UPDATE, BROWSE(), DBEDIT()

Record Scoping

Many database operations can process subsets of records within a work area using scope and conditional clauses. For any database command that allows a <*scope*> as part of its syntax, the syntax of the <*scope*> is as follows:

```
[ALL | NEXT <nCount> | RECORD <nRecord> | REST]
```

- ALL processes all records
- NEXT processes the current record and the specified number of records
- RECORD processes the specified record
- REST processes all records from the current record to the end of file

Commands specified without a scope default to the current record (NEXT 1) or ALL records, depending on the command. For example, DELETE and REPLACE typically process the current record by itself and, therefore, default to NEXT 1. Other commands, such as REPORT and LABEL FORM, typically process a set of records and, therefore, default to ALL records. Specifying a scope changes this default by indicating how many records to process and where to begin.

The set of records processed can also be restricted using a conditional clause which specifies a subset of records based on a logical condition. The two conditional clauses are FOR and WHILE.

A FOR clause defines a condition that each record within the scope must meet in order to be processed. If no other scope is specified, FOR changes the default scope to ALL records.

A WHILE clause defines another condition that each record processed must meet; as soon as a record is encountered that causes the condition to fail, the command terminates. If no other scope is specified, WHILE changes the default scope to REST.

Specifying a scope, a FOR clause, and a WHILE clause within the same command syntax can raise questions regarding the order in which the clauses are processed. The scope is evaluated first in order to position the record pointer. Then, the WHILE clause is evaluated and, if the condition is not met, the process terminates. If the WHILE condition is met, the FOR clause is evaluated. If the FOR condition is also met, the record is processed; otherwise, it is not. Either way, the record pointer is moved to the next record within the scope until the scope is exhausted.

The Record Processing Primitive—DBEVAL()

The iterate operation, DBEVAL(), is relatively new and, therefore, deserves further explanation. This function is the database primitive in STD.CH that defines database commands that allow record scoping, and it is sometimes called the record processing primitive.

DBEVAL() evaluates a code block for each record matching the specified scope and conditions. It creates user-defined database processing commands. The DBEVAL() entry in the "Language Reference" chapter of the *Reference Guide* has more information and examples. For additional information on code blocks, see the Code Blocks section earlier in this chapter.

Index Files

Database files are maintained and processed in a particular physical order identical to the order in which records are added to the file. Each record is numbered sequentially beginning with one, and the number of the current record can be accessed with the RECNO() function. Indexing provides a means for logically ordering the database file according to the data it contains. In CA-Clipper, you can index by creating index files in a work area with a database file.

Note: Although the default method for creating and maintaining index files is unique and incompatible with other dialects, CA-Clipper supplies several Replaceable Database Drivers (RDDs) for using various index file formats. Some RDDs support multiple orders per index file, while others, like the default RDD, support only a single order per index file. This section applies specifically to the default RDD, DBFNTX, but can be generalized to other RDDs. Refer to the *Drivers Guide* for more specific information about a particular RDD.

Creating

You create index files for the database file in the current work area with the INDEX command in which you specify an order key expression. The maximum length of the key expression is 250 characters, and its type can be character, date, or numeric.

You determine the index key value for each record in the database file using the index key expression. This expression is stored in the index file in such a way that the database file can be processed in order according to the key values which can be searched very quickly.

The UNIQUE clause of the INDEX command and the SET UNIQUE ON flag create an index file in which only the first occurrence of any given key value is maintained in the index file. Uniqueness is an attribute that is stored in the index file.

Opening

You can create several index files for a single database file so the data can be accessed in many different ways. Once the necessary index files are created, you must open them along with the associated database file in order to use them. You can open up to 15 index files at the same time with the INDEX clause of the USE command or with SET INDEX.

Like all other database file opening operations in CA-Clipper, USE and SET INDEX first search the current SET DEFAULT drive and directory for the file to open. If the file is not found, the SET PATH list is searched in the order specified. Only after all paths are searched is an open operation considered a failure.

Ordering

With both USE and SET INDEX you list the index files you want to open, and the first one in the list becomes the *controlling* index. The controlling index defines the order in which the database file is processed and is the only index that can be searched.

The SET ORDER command changes the controlling index by specifying a position in the index file list. SET ORDER TO 0 specifies the physical order as if no index files were in use.

Searching

SEEK and FIND locate a record based on its index key value in the controlling index file. SEEK is the preferred search command; FIND is an obsolete command that is provided for compatibility only. Searching an index file using either of these commands is much faster than performing a sequential search.

CA-Clipper uses one of two index search methods depending on a global flag setting, SET SOFTSEEK. If SET SOFTSEEK is OFF (the default), a failed index search moves the record pointer to the end of file, setting EOF() to true (.T.). If SET SOFTSEEK is ON, a failed index search moves the record pointer to the record with the next higher index key value.

The FOUND() function tests the success or failure of a search operation in either case.

Updating

As long as an index file is open, any changes made to the database file are automatically reflected in the index file. This is true of all open index files, not just the controlling index. Changes to a database file are made using any of the database update operations mentioned earlier in this section.

Additionally, database operations that move the record pointer update the index file before moving the record pointer if the index key value has changed. A COMMIT or an index close operation also updates the index file.

The REINDEX command recreates all open index files in the current work area by reindexing each record in the database file.

Closing

Close index files with SET INDEX TO with no parameters or CLOSE INDEX. Any of the database close operations also closes open index files in the same work area.

The Input/Output System

The CA-Clipper language provides a complete input/output system allowing data entry and data display on the screen and the printer. The system includes built-in data validation that is extensible through the use of functions. Included facilities let you create simple columnar reports and mailing labels, and, with the provided commands and library functions, you can create reporting mechanisms to suit your own needs. This section introduces you to CA-Clipper's input/output system, giving you basic information on controlling the console and printer. CA-Clipper has the ability to display output to the screen in one of two modes: text mode and graphic mode.

Text Mode

In text mode, all characters that are displayed must be one of 255 characters that make up the ASCII character set. In addition, you can place only one character at one particular location on the screen.

These locations are typically sixteen pixels high by eight pixels wide. A pixel is the smallest unit that can be displayed on a computer screen that represents exactly one color.

Graphic Mode

In graphic mode, however, in addition to the regular ASCII character set, you now have control over what is displayed in pixel coordinates instead of row and column coordinates as in text mode. Using graphic mode, bitmaps can be displayed, smooth lines can be drawn, and applications look smoother and more elegant than ever before.

For compatibility with previous versions of CA-Clipper, text mode is the default, but switching from text mode to graphic mode is easy. Simply use the SET VIDEOMODE command as shown in this example:

```
// Header file with graphic constants
#include "Llibg.ch"

SET VIDEOMODE TO LLG_VIDEO_VGA_640_480_16
// All output to the screen will now be in graphic
mode

SET VIDEOMODE TO LLG_VIDEO_TEXT
// Now all output will be in text mode
```

Using SET VIDEOMODE with the constant LLG_VIDEO_VGA_640_480_16 first clears the screen then switches to a 640 x 480 pixel screen in VGA mode with sixteen color support. Your video card must be VESA 1.2 compatible to use graphic mode. See the SET VIDEOMODE command in the *Reference Guide* for more details.

LLG_VIDEO_VGA_640_480_16 and LLG_VIDEO_TEXT are the only display modes built into CA-Clipper.

For more detailed information about displaying output in graphic mode, see the "Programming in Graphic Mode" chapter later in this guide.

Console Operations

Commands and functions that display output on the screen without reference to row, column, or pixel position are called *console operations*. Used in conjunction with SET ALTERNATE and SET PRINTER, these operations can simultaneously send output to the screen, a text file, and the printer. However, the majority of the console commands have TO PRINTER and TO FILE clauses that supersede SET PRINTER and SET ALTERNATE. The following table lists all the console operations in the CA-Clipper language:

Operation	Output
? | ??	The result of one or more expressions
ACCEPT	A prompt. Waits for character input.
DISPLAY | LIST	The contents of a database file
INPUT	A prompt. Waits for input of any type.
LABEL FORM	The contents of a database file as labels
REPORT FORM	The contents of a database file as a report
TEXT...ENDTEXT	A block of text
TYPE	The contents of a file
WAIT	A prompt. Waits for a single character input.
QOUT() | QQOUT()	The result of one or more expressions

SET CONSOLE is an important command to console operations since it lets you temporarily suppress the screen output without affecting file and printer output. For example, if you are printing a report, it can be disconcerting to see the report scrolling up the screen while it is printing. To control this, SET CONSOLE OFF before printing the report, as in this code segment:

```
USE Accounts NEW
SET CONSOLE OFF
REPORT FORM Accounts TO PRINTER
SET CONSOLE ON
CLOSE Accounts
```

Full-Screen Operations

The *full-screen operations* address the screen (and sometimes the printer) directly, usually by addressing a specific row and column. These operations are distinguished from console operations in that they ignore SET ALTERNATE, SET CONSOLE, and SET PRINTER:

Operation	Result
@...BOX	Draw a box
@...CLEAR	Erase the screen
@...SAY	Display the result of an expression
@...TO	Draw a box
CLEAR	Erase the screen
COL()	Return current column position
ROW()	Return current row position
MAXCOL()	Return the maximum column position
MAXROW()	Return the maximum row position
SAVESCREEN()	Save a screen region
RESTSCREEN()	Restore a screen region

Note: In graphic mode, the SAVESCREEN() and RESTSCREEN() functions should be used with caution. Please see these entries in the *Reference Guide* for more information.

Of all the full-screen operations, only @...SAY can be redirected to the printer with SET DEVICE and written to a file with SET PRINTER. Printing is discussed later on in this section.

Controlling Screen Color

SETCOLOR() allows you to save the current color setting and optionally set new colors for subsequent screen painting. A color string contains several color settings, each corresponding to a different logical region of the screen.

All console and full-screen operations (with the exception of @...GET) use the standard color setting when displaying to the screen. The GET area (@...GET) uses the enhanced setting, and the unselected setting causes all GETs other than the current one to be displayed using a different color attribute.

For more information regarding color, including a list of the available color codes, see SETCOLOR() in the "Language Reference" chapter of the *Reference Guide*.

Controlling Output Destination

In CA-Clipper, the screen is always the default output destination, but you can direct output to the printer or a disk file.

Directing Output to the Printer

The way to direct output to the printer depends on what operation you are using. Many console commands have a TO PRINTER clause designed to echo output to the printer. For instance, to print a set of labels:

```
LABEL FORM Ship TO PRINTER
```

For console commands that do not support this clause, use the SET PRINTER command:

```
SET PRINTER ON
DO WHILE .NOT. EOF()
   QOUT(Name)
   SKIP
ENDDO
SET PRINTER OFF
```

You can use SET PRINTER instead of the TO PRINTER clause, if you prefer. As stated earlier, you may want to SET CONSOLE OFF before directing console output to the printer.

For @...SAY, use the SET DEVICE command:

```
SET DEVICE TO PRINTER
DO WHILE .NOT. EOF()
   @ 1, 1 SAY Name
   SKIP
ENDDO
SET DEVICE TO SCREEN
```

Unlike console operations, @...SAY displays either to the screen or to the printer, but never to both devices simultaneously.

CA-Clipper allows other printer controls, including access to the current printhead position with PROW() and PCOL(). These functions allow relative printer addressing with @...SAY in much the same way as ROW() and COL() do relative screen addressing.

When printing with @...SAY, a page eject is issued anytime you address a row that is less than the current PROW() value. SETPRC() resets PROW() and PCOL() to prevent an automatic page eject where it is not desired.

SET MARGIN sets the left printer margin. With console output, the margin setting indents output whenever there is a new line. @...SAY adds the margin setting to the specified column value.

Directing Output to a File

The SET PRINTER command has a second form that is used to change the output device, allowing you to access more than one printer and to direct printed output to a file. SET PRINTER TO sets the output device for all printed output, including console and full-screen operations.

You cannot send @...SAY output directly to a file. Instead, you must reroute its printed output to a file with SET PRINTER TO as in this example:

```
SET PRINTER TO AtOut.prn
SET DEVICE TO PRINTER

// <@...SAY commands go here>

SET DEVICE TO SCREEN
SET PRINTER TO
```

Redirecting printer output to a text file with SET PRINTER TO causes all printed output to go to the file, not just @...SAYs. Thus, it can be used to send @...SAY and console output to the same file, as long as both SET DEVICE TO PRINTER and SET PRINTER ON are in effect:

```
SET PRINTER TO AtOut.prn
SET DEVICE TO PRINTER

// <@...SAY commands go here>

SET PRINTER ON

// <console operations go here>

SET PRINTER OFF
SET DEVICE TO SCREEN
SET PRINTER TO
```

You can direct output to a file without rerouting printer output. The easiest way is to use the TO FILE option if it is supported. If not, use SET ALTERNATE:

```
SET ALTERNATE TO OutFile.prn
SET ALTERNATE ON

// <console operations go here>

SET ALTERNATE OFF
CLOSE ALTERNATE
```

The Keyboard System

By default, CA-Clipper applications automatically save the characters typed on the keyboard in a buffer called the keyboard, or typeahead, buffer. The characters in the keyboard buffer are then removed on a first in, first out basis and used as input when required by a wait state or INKEY(). The wait states include ACCEPT, INPUT, READ, WAIT, ACHOICE(), DBEDIT(), and MEMOEDIT().

The buffering of keystrokes lets the user continue typing without having to wait for the application to catch up, and it is very convenient, especially for applications that are data entry intensive where the user happens to be a very fast typist. This feature may never affect the way you program your applications, but there may be cases when you want to control the keyboard buffer. This section describes the commands and functions available in CA-Clipper that affect the keyboard buffer.

Changing the Keyboard Buffer Size

SET TYPEAHEAD controls the number of characters the keyboard buffer can hold. The minimum size of the keyboard buffer is zero characters, a setting which essentially disables the keyboard buffer so the user can type only when an operation requiring input is actively executing. The maximum keyboard buffer size is 32,767 characters. Issuing a SET TYPEAHEAD command in an application clears the current contents of the keyboard buffer before changing the buffer size.

Putting Characters in the Keyboard Buffer

KEYBOARD is used to programmatically put characters into the keyboard buffer. This command clears all pending keystrokes before placing one or more ASCII characters in the keyboard buffer. You can use the CHR() function with KEYBOARD for nonprintable keystrokes such as a carriage return (CHR(13)).

Characters placed in the keyboard buffer this way are treated just as if they were typed from the keyboard—they are not executed until extracted by a wait state or by INKEY().

KEYBOARD is useful for developing self-running demonstration programs in CA-Clipper, and it is also commonly used with the ACHOICE() user function to force a particular menu selection.

Refer to the "CA-Clipper INKEY() Codes" appendix in the *Error Messages and Appendices Guide* for a complete list of key codes.

Reading Characters from the Keyboard Buffer

There are three functions that access the keyboard buffer. These are INKEY(), LASTKEY(), and NEXTKEY(). Each function returns a numeric value between -39 and 386, identifying the INKEY() code of keys including function, Alt+function, Ctrl+function, Alt+letter, and Ctrl+letter key combinations.

INKEY() is the only keyboard function that actually extracts a character from the keyboard buffer. In this sense, it is like a wait state, but it may or may not pause for input depending on its argument. INKEY() extracts the next pending keystroke from the keyboard buffer and returns the ASCII value of that key. This function is useful for polling the keyboard or pausing program execution.

NEXTKEY() returns the ASCII value of the next pending keystroke without removing it from the keyboard buffer.

LASTKEY() returns the ASCII value of the last keystroke extracted from the keyboard buffer. Keystrokes are extracted from the buffer either by INKEY() or by a wait state. This function has several applications, including determining the key used to terminate a READ.

Controlling Predefined Keys

There are several commands and functions that control the action of specific, predefined keys. These are summarized in the following table:

Command/Function	Purpose
SET ESCAPE	Toggle ability to terminate READ with Esc
SET WRAP	Toggle circular menu navigation with Up arrow and Down arrow
ALTD()	Toggle ability to invoke the debugger with Alt+D
READEXIT()	Toggle ability to terminate READ with Up arrow or Down arrow
SETCANCEL()	Toggle ability to terminate application with Alt+C

In addition to these limited controls, all predefined keys can be completely redefined with SET KEY.

Reassigning Key Definitions

SET KEY lets you redefine any key on the keyboard by specifying a procedure to be executed when the key is pressed during a wait state. Three parameters, PROCNAME(), PROCLINE(), and READVAR(), are automatically passed to the procedure. As with INKEY(), keys are identified by a numeric ASCII code.

Another command, SET FUNCTION, redefines function keys just as SET KEY redefines other keys. This command, however, is preprocessed into SET KEY and KEYBOARD commands, causing SET FUNCTION to clear a SET KEY for the same key number. This is incompatible with previous releases of CA-Clipper which maintained a separate list for SET FUNCTION and SET KEY definitions.

A total of 32 keys can be SET at one time. At startup, the system assumes that F1 is set to Help. If you link a procedure called Help into the current program, pressing F1 from a wait state executes it.

All SET KEY definitions take precedence over the standard key definitions. This includes predefined keys such as Insert and Delete that are used in editing, as well as keys such as Esc and Alt+C that you can control with built-in commands and functions.

Clearing the Keyboard Buffer

You can clear the keyboard buffer of all pending keystrokes with CLEAR TYPEAHEAD. This command is useful when you want to be sure the user does not bypass important messages on the screen. Simply displaying the message and waiting for a response quickly flashes the message and uses the next pending character in the keyboard buffer for the response. However, issuing a CLEAR TYPEAHEAD before displaying the message ensures that there are no pending characters and forces the user to read the message before entering a response.

The Low-Level File System

Several functions are provided that allow you to manipulate DOS binary files directly from a CA-Clipper application. These include the capability to open existing binary files, to create new files, and to read from and write to open files. This section describes these functions and their capabilities. Note that the functions described herein allow low-level access to DOS files and devices. They require a thorough knowledge of the operating system and should be used with care. Refer to the "Language Reference" chapter of the *Reference Guide* for more information on a particular function.

Opening a File

There are two functions available to open a binary file. The first, FOPEN(), opens an existing file, while the second, FCREATE(), creates a new file, leaving it open for use. Both functions return a number representing the DOS file handle for the open file. The file handle must be saved in a variable to identify the file to other functions. FCREATE() lets you specify a DOS file attribute, and FOPEN() lets you specify the DOS open mode.

Since these functions are dealing with files at the operating system level, they do not respect the SET DEFAULT and SET PATH settings. Instead, unless a filename is unambiguously identified with a path, the current DOS drive and directory are assumed. Also, no file extension is assumed unless one is explicitly specified.

Reading from a File

Once a binary file is open, you can read its contents with one of two functions, FREAD() or FREADSTR(). Both functions require that you identify the file using its file handle. Specifying a particular file is necessary since more than one binary file can be open at a time.

FREAD() and FREADSTR() are very similar in functionality. The main difference is that FREAD() requires you to specify where to save the characters read from the file, whereas FREADSTR() returns the characters read as a character string.

In either case, the data read from a binary file is in binary form. Thus, several functions are provided to convert binary data to numeric so that you can manipulate the information in your CA-Clipper application.

All of the conversion functions require that the argument be represented as a character string, which makes them work easily with FREAD() and FREADSTR().

Writing to a File

Once a binary file is open, you can write to it with FWRITE(). This function requires that you identify the file using its file handle. It is necessary to specify a particular file since more than one binary file can be open at a time. You cannot use FWRITE() on a file that was opened in read-only mode.

With FWRITE() you specify a character string to write to the file. The I2BIN() and L2BIN() functions are provided to convert CA-Clipper numeric values to binary form and return the results as character strings. These character strings can then be written to a binary file.

Manipulating the File Pointer

When a binary file is initially opened for use, the file pointer is positioned to the beginning of the file. The read and write functions move the file pointer, as needed, to perform their operation, and leave the pointer in its new position in the file.

Another function, FSEEK(), lets you move the pointer directly to a specific position in the file. This function is particularly useful if you must read or write characters somewhere in the middle of a file but are not interested in the information preceding those characters. Instead of performing a read operation to bypass the unwanted data, you would perform an FSEEK()—which is much more efficient—to move the file pointer to the correct position in the file. Then, you would read or write the necessary characters.

FSEEK() is very flexible in its ability to position the file pointer. You specify an argument that tells the function to begin from the beginning of file, from the current file position, or from the end of file, and another argument that indicates the number of positions to move.

Closing a File

FCLOSE() closes an open binary file and writes the associated DOS buffers to disk. This function requires that you identify the file you want to close by its file handle. Specifying a particular file is necessary since more than one binary file can be open at a time.

Error Detection

When using the low-level file functions, you can test for errors with the individual function return value or with the FERROR() function. Which method you use depends on the level of detail you want. The following table summarizes the return values of the low-level file functions that indicate an error condition:

Function	Error Condition
FCLOSE()	Returns false (.F.)
FCREATE()	Returns -1
FOPEN()	Returns -1
FREAD()	Returns zero or a value less than the number of bytes to read
FREADSTR()	Returns a null string
FSEEK()	No error condition
FWRITE()	Returns zero or a value less than the number of bytes to write

FERROR() returns a DOS error number for the last low-level file function executed. If there is no error, FERROR() returns zero. After determining that a function has failed by checking its return value, you can use FERROR() to narrow down the cause of the error. This function retains its value until the next execution of a low-level file function.

Chapter 3
The Runtime Environment

In This Chapter

After you have designed, written, debugged, and configured a CA-Clipper application, the next step is to install the program on the end user's machine. This process consists of preparing the user's workstation. Before doing this, however, there are a number of issues to understand in order to guarantee that your application will run correctly in the user's computing environment.

This primarily consists of setting the workstation operating system environment so the program will have enough resources to run, configuring the application's environment, and configuring the user's network environment to ensure that software installed on the network will operate correctly and have access to shared resources.

In this chapter, the following major topics are discussed:

- Setting the workstation environment
- Setting the application environment
- Setting the network environment
- The application batch file

Setting the Workstation Environment

The workstation's operating environment is configured by two facilities at system initialization: settings defined in CONFIG.SYS and commands executed in AUTOEXEC.BAT.

CONFIG.SYS

CONFIG.SYS is an ASCII text file that DOS processes during system initialization. It is used to configure the workstation's operating system environment by setting the number of internal buffers and open files and installing device drivers to enable RAM disks, terminal emulation, extended or expanded memory, and other peripheral devices. Of particular interest when executing CA-Clipper programs are the FILES and BUFFERS settings, although there are others.

When the user's computer is started, CONFIG.SYS must be located in the root directory of the user's startup disk. After the operating system is configured the system batch file AUTOEXEC.BAT is called.

AUTOEXEC.BAT

AUTOEXEC.BAT is a special batch file located in the root directory of the workstation startup disk. After the command processor (COMMAND.COM) is loaded, it looks for AUTOEXEC.BAT and executes all of the commands found in the file. Typically, an AUTOEXEC.BAT would perform one or more of the following operations:

- Define the PATH environment variable to tell COMMAND.COM where to look for executable files if they are not located in the current drive/directory
- Execute a series of SET commands to define common environment variables
- Use the MODE command to configure the serial port and redirect printed output to it
- Load TSR programs, such as the network shell or a local print spooler
- Invoke a default application, such as a DOS shell or user menu

For more information on the full range of configurations that can be made in CONFIG.SYS and AUTOEXEC.BAT, refer to your DOS manual.

Files and Buffers

In order for a CA-Clipper application to run, it must have enough file handles and buffers, which you configure using the FILES and BUFFERS commands in the CONFIG.SYS file.

The FILES command uses *file handles* to set the number of files or devices that can be open at one time. The default value is eight, but FILES can be set as high as 255 if the user's workstation is running DOS 3.3 or higher. If the number of file handles is greater than 20, you must use the CLIPPER environment variable (discussed later on in this chapter) to tell CA-Clipper how many handles to use.

The actual number of file handles to specify depends on your application. As a general principle, you should allow one handle for each database and index file in your application. Note, however, that some operations, such as indexing and sorting, create temporary files. This increases the number of file handles required for the application to run.

If the user's workstation is running a version of DOS earlier than version 3.3, the maximum number of files that can be open at one time is 20. If this is the case, you may want to upgrade to a more recent version of DOS.

The BUFFERS command allocates the number of buffers (disk cache) DOS uses to keep copies of the sectors most recently read from or written to disk. If not specified, the default value is two for IBM PC/XTs and three for IBM ATs and beyond. The recommended number of buffers for CA-Clipper is eight.

For example, the CONFIG.SYS file for versions of DOS prior to 3.3 should contain the following two statements:

```
FILES=20
BUFFERS=8
```

Changing the DOS Environment Size

When setting up your application on a user's machine, if you add more environment variables than will fit in the default environment space (160 bytes in version 3.2), a DOS error message is displayed, and the variable is not assigned.

In DOS versions 3.2 and above, you can change the default environment size by specifying a SHELL command in CONFIG.SYS, like this:

```
SHELL=COMMAND.COM /E:<nBytes> /P
```

<nBytes> is the initial allocation of environment space and can be any value between 160 and 32,768. The recommended value is 1024 bytes.

Specifying the Location of Executable Files

With a CA-Clipper program, there are normally two types of files associated with the application's executable image:

- Executable files (.EXE)
- Overlay files (.OVL)

Note: Overlay files are applicable only if the application was linked with a real mode linker. If you link with a protected mode linker such as CA-Clipper/Exospace, the application will not have associated .OVL files because they are not necessary. Protected-mode applications overlay code as needed when the application is running—but it is completely automatic and transparent, and you never need to worry about it.

In the *runtime environment*, there are some specific rules as to how these files are located when invoked or requested.

The executable file (.EXE) is the program file that can be invoked directly from the DOS command prompt and is searched for in the current directory. If not found, the current PATH is searched.

Overlay files (.OVL) are files associated with the application .EXE file and contain either dynamic or static overlays. They are searched for only in the directory where the .EXE is located. No other directories are searched. If not found, a runtime error is generated. This means you must install all overlay files in the same location as the executable (.EXE) file.

To assure executable files (.EXE) and associated files can be located, add the location of the .EXE file to the PATH environment variable in the user's AUTOEXEC.BAT file:

```
SET PATH=C:\APPS\MYAPP
```

Temporary Files

In the process of some operations, such as sorting and indexing, CA-Clipper creates temporary files. These files are created either in the current directory or in the directory specified by the environment variable TMP. To assign a TMP variable, place a command line like the following in the user's AUTOEXEC.BAT file:

```
SET TMP=C:\TEMP
```

You can override the TMP setting using the TEMPPATH environment setting. See the Setting the Application Environment section later in this chapter for more information.

Specifying the Location of COMMAND.COM

When you use the RUN command to execute external programs, the application runtime system needs to load COMMAND.COM to execute the specified program. To do this successfully, the environment variable COMSPEC must point to the location of COMMAND.COM. Normally, DOS sets the value of this variable to the root directory of the startup disk.

If for some reason this is not the case, add an explicit assignment of COMSPEC in the user's AUTOEXEC.BAT or application batch file (discussed later in this chapter), as follows:

```
SET COMSPEC=C:\DOS
```

Configuring the Serial Port

If the workstation is printing to a local printer attached to a serial port, you must configure the port with the MODE command. MODE is used to configure the serial port with the appropriate baud rate, parity, word length, and stop bits for the attached printer. A subsequent MODE command can also be used to redirect output from the default parallel port (LPT1, LPT2, or LPT3) to a serial port (COM1 or COM2).

For CA-Clipper programs running on a workstation with a serial port and a local printer attached, the recommended procedure is to first run MODE with the configuration options within the application batch file (discussed later in this chapter). To redirect printed output from the default parallel port, you have two choices: you can either redirect output using the MODE command in the application batch file, or you can use the SET PRINTER command within the application.

Using MODE

For example, the following application batch file shows how to use the MODE command to configure a serial port and redirect output to it:

```
REM Application batch file
MODE COM1:=1200,8,1,P
MODE LPT1:=COM1
MYAPP.EXE
MODE LPT1
```

Using SET PRINTER

If SET PRINTER is used, the reference to the port can be a literal or variable value, but a literal value designation is somewhat restrictive and environment specific. The printer port can be specified to the SET PRINTER command as an extended expression. This allows you to pass the print destination to an application via an environment variable or as a command line argument of the application.

For example, this application batch file configures the serial port then defines an environment variable designating the printer port as COM1:

```
REM Application batch file
MODE COM1:=1200,8,1,P
SET CLIPPORT=COM1
MYAPP.EXE
MODE LPT1
```

In the CA-Clipper application, the GETENV() function is used to obtain the port. If the environment variable exists, SET PRINTER directs output to the specified port:

```
IF !EMPTY((cDest := GETENV("CLIPPORT")))
   SET PRINTER TO (cDest)
ELSE
   SET PRINTER TO LPT1
ENDIF
```

Setting the Application Environment

Once you have set the workstation environment, you may need to set the application environment. CA-Clipper provides runtime configuration control of CA-Clipper-compiled programs with a number of settings that you can specify in a DOS environment variable named CLIPPER or on the application command line.

The following table lists the syntax and a brief description of each setting:

Setting	Meaning
BADCACHE	Save/restore EMM page frame on each EMM access
CGACURS	Prevent use of EGA/VGA extended cursor capability
DYNF:<*nHandles*>	Specify number of file handles for dynamic overlay system use
E:<*nExpandedKBytes*>	Configure amount of expanded memory
F:<*nHandles*>	Set maximum number of file handles
INFO	Display memory configuration details at startup

Continued

Continued

Setting	Meaning
NOIDLE	Prevent detection of idle time during execution of compiled applications
SWAPK:<*nBytes*>	Specify maximum size of disk swap file used for VM system
SWAPPATH:'<*path*>'	Specify location of VM swap file
TEMPPATH:'<*path*>'	Control placement of temporary sort and index files
X:<*nKBytes*>	Exclude available memory

Note: If you link your application with CA-Clipper/Exospace, there are several additional runtime environment settings that you can use to control the VM system. See the EXOSPACE EXECUTABLE CLIPPER command in the "CA-Clipper Protected Mode Linker—EXOSPACE.EXE" chapter of this guide for a table of these settings.

The following rules apply when specifying environment settings, regardless of whether you specify them on the application command line or in the CLIPPER environment variable:

- Preface each setting with a double-slash
- Place a single blank space between settings
- Place a colon between setting and argument, with no intervening space

It is possible to specify a setting twice, once in the CLIPPER environment variable and then again on the application command line. If you do this, settings on the command line override like settings in the environment variable. This feature provides an easy way to define those settings that are more or less permanent using the CLIPPER environment variable and to override those settings on an application-by-application basis.

Application Command Line

To specify environment settings on the application command line, use the following syntax:

```
<app> [//<setting> ... ] [<app arguments>]
```

Note that all environment settings must come before any application arguments on the command line.

Tip: If you wish to use the command line method for specifying environment settings but also wish to hide the command line complexity from your end user, use an application batch file (discussed later in this chapter) to invoke the application with the configuration settings.

The CLIPPER Environment Variable

The CLIPPER variable can be entered at the DOS prompt or included in the user's AUTOEXEC.BAT or application batch file (discussed later in this chapter). If the user will be accessing the application from a network, you can also add the SET CLIPPER command to the user's login script.

To specify environment settings using the CLIPPER environment variable, use the following syntax:

```
SET CLIPPER=[//<setting> ...]
```

Note: When linking your your application, you can specify application environment settings using an environment variable name other than CLIPPER.

Saving/Restoring EMM Page Frame—BADCACHE

BADCACHE causes the Virtual Memory Manager (VMM) to preserve and restore the state of the Expanded Memory Manager (EMM) page frame before and after every EMM access (the EMM page frame is an area in real address space through which EMM data is accessed). This setting can be used to correct problems when there are conflicts with other programs that use EMM.

Note that on some EMM systems, the BADCACHE setting may adversely affect VMM performance. It should be used only if you experience disk or file corruption because of a conflict with a disk cache or other resident software.

Preventing Extended Cursor Use—CGACURS

CGACURS prevents the use of the extended cursor capability of the EGA/VGA. Specifying this setting may preclude some cursor modes by some display adapters. CGACURS prevents the unusual cursor behavior when a CA-Clipper program is executed in some multitasking and TSR environments.

Specifying Number of Dynamic Overlay File Handles—DYNF

DYNF specifies the number of file handles the dynamic overlay system is allowed to use. Valid settings range from 1 to 8, inclusive. If not specified, the default is 2.

Configuring Expanded Memory—E

CA-Clipper-compiled programs can use expanded memory to speed up processing, as well as some disk-based operations. CA-Clipper can use memory configured as expanded memory according to the Lotus-Intel-Microsoft (LIM) Expanded Memory Specification (EMS) version 4.0 or higher.

Expanded memory is automatically allocated in its entirety at startup—there is no dynamic allocation as execution proceeds. The maximum amount allocated is limited by four factors:

- The amount of expanded memory available
- The //E:<*nExpandedKBytes*> environment setting
- A theoretical maximum of 32 MB
- The amount of conventional memory available

The E parameter can be used to restrict the amount of expanded memory automatically allocated to <*nExpandedKBytes*>. For example, specifying the following E parameter restricts the expanded memory CA-Clipper will allocate to 2 MB:

```
SET CLIPPER=//E:2000
```

Note that a certain amount of conventional memory must be used to contain management tables for the virtual memory system—the more total memory (both conventional and expanded), the more space taken up by this control information. The amount of expanded memory used may be less than the amount available if there is insufficient conventional memory to hold the tables.

Specifying the Number of Files—F

If you have an application that uses more than 20 files and are running under DOS 3.3 or greater, you must use the F parameter to inform CA-Clipper of the maximum number of file handles to use. This parameter is used in combination with the value you specified with the FILES setting in CONFIG.SYS.

When specified, CA-Clipper determines the number of files that can be opened using the smaller of the F parameter and the CONFIG.SYS FILE setting. For example, if the FILES command is set to 120 and the F parameter is set to 50, the maximum number of files that can be opened is 50. The ideal *<nHandles>* is an odd number that is 5 less than the number specified with the FILES setting.

Keep in mind that the F parameter should be set judiciously. It must be large enough to allow the use of all needed files and yet not so large that valuable memory space is used unnecessarily.

Displaying Memory Configuration Details at Startup—INFO

INFO provides the following information about CA-Clipper's memory usage at application startup:

- The first line describes the general product version, revision, and international version.
- *DS=<offset>:0000* is the address for the data segment or DGROUP. This value has no application functionality.
- *DS avail=<memory>KB* reflects the amount of DGROUP available which is used for the allocation of the processor stack, CA-Clipper statics, CA-Clipper environment variables, and the evaluation stack.
- *OS avail=<memory>KB* represents the amount of conventional memory available for VMM swap space. If the reported value is too low, there is a strong possibility the program will terminate with a "Conventional Memory Exhausted" unrecoverable error.

- *EMM avail=<EMM memory>KB* shows the amount of EMM (expanded memory) allocated to the current application. A "P" following the *EMM avail* value appears when the application is loaded with the BADCACHE configuration setting.
- *Fixed Heap=<fixed heap>KB/<number of fixed segments>* shows the size of the fixed heap in kilobyte increments in addition to the number of fixed segments. Fixed heap is used for symbols at runtime (macros), static allocations (extend system), the EMM table, and the symbol table.

Preventing Detection of Idle Time—NOIDLE

NOIDLE prevents CA-Clipper from detecting and taking advantage of idle time during execution of compiled applications.

If NOIDLE is not specified, CA-Clipper detects idle states (for example, keyboard wait states) during execution of compiled applications. When an idle condition is detected, the system uses the slack time to perform garbage collection, file updates, and other routine housekeeping duties. This increases system performance by doing this work while the application is waiting for user input.

Note: NOIDLE is provided for applications in which idle time processing is unacceptable. Since it reduces overall system performance, its use is generally not recommended.

Specifying Maximum Swap File Size—SWAPK

SWAPK specifies the maximum allowable size of the disk swap file used for the virtual memory (VM) system. Settings are specified in KB increments. Valid settings range from 256 to 65,535, inclusive. If this setting is not specified, the default is 16,384 (16 MB).

Note: Swap space is allocated only as needed—a particular setting does not guarantee that the swap file will get that big. Suppressing or restricting disk swapping may cause an application to fail.

Specifying Swap File Location—SWAPPATH

SWAPPATH specifies the location of the virtual memory swap file. If not specified, the swap file is created in the current DOS drive and directory.

Specifying Temporary File Location—TEMPPATH

TEMPPATH controls the placement of temporary files created during sorting and indexing. By default these files are placed in the current DOS directory or in the directory specified by the SET TMP environment variable, if it is defined.

Note: Temporary files created during sorting and indexing can be quite large. Setting TEMPPATH to a small volume (for example, a ram disk) may cause these operations to fail. In general, the volume where these temporary files will be written should have an available capacity at least twice the size of the largest index to be created or database file to be sorted.

Excluding Available Memory—X

In certain situations, especially debugging, you may need to test how a program operates in a memory environment less than the current machine offers. You can do this using the X parameter to exclude *<nKBytes>* of memory from being allocated, except for the RUN command. For example, if you specify //X:64 with a RAM configuration of 640 KB, memory is allocated as though the computer contains only 576 KB (that is, 640 - 64) of memory.

Valid values for *<nKBytes>* range from zero to 64, inclusive. The default value is zero.

Note: Memory—X is excluded before other memory is allocated. The amounts allocated for other purposes are diminished accordingly.

The BLINKER Environment Variable

The BLINKER environment variable can be used to alter settings quickly while testing an application, and once the application is in production it provides unique settings for individual workstations.

Settings in the BLINKER environment variable should be separated by spaces.

This example disables Blinker's use of the EMS pageframe, and sets the size of the overlay pool to 30KB:

```
SET BLINKER=/OP- /0030
```

The environment variable name 'BLINKER' can be changed with the use of the command BLINKER ENVIRONMENT NAME, which will avoid conflict with other Blinker linked programs that are present on the same computer.

This example changes the name to 'MYAPP', causing Blinker to ignore the 'BLINKER' variable, using 'MYAPP' instead:

```
SET MYAPP=/OP- /OO30 /CX128
```

Note: For all settings, the Blinker overlay manager obtains control before the program begins execution. As a result the system resources controlled by these settings are allocated before the main program gets control.

Controlling Amount of EMS Memory—/CEnnn,nn

/CEnnn,nn is an override of the link-time command BLINKER CACHE EMS nnn which controls the amount of EMS memory used by the overlay manager for overlay caching. For details of the usage of these values, see the command BLINKER CACHE EMS.

Controlling Amount of XMS Memory—/CXnnn,nn

/CXnnn,nn is an override of the link-time command BLINKER CACHE XMS nnn which controls the amount of XMS memory used by the overlay manager for overlay caching. For details of the usage of these values, see the command BLINKER CACHE XMS.

Controlling Operating Size of Blinker Overlay Pool—/OOnnn

/OOnnn is an override of the link time Blinker command BLINKER OVERLAY OPSIZE n which controls the operating size of the Blinker overlay pool. For details of the usage of this value, see the command BLINKER OVERLAY OPSIZE.

Controlling Allocation of Overlay Pool—/OPc

/OPc (where c=- is disabled, and c=+ is enabled) is an override of the link-time Blinker command BLINKER OVERLAY PAGEFRAME which controls the allocation of the overlay pool using the EMS (3.2+) pageframe. For details of the usage of this value, see the command BLINKER OVERLAY PAGEFRAME.

Controlling Allocation of Overlay Pool—/OUc

/OUc (where c=- is disabled, and c=+ is enabled) is an override of the link-time Blinker command BLINKER OVERLAY UMB which controls the allocation of the overlay pool using XMS 2.0 upper memory blocks. For details of the usage of this value, see the command BLINKER OVERLAY UMB.

Setting the Network Environment

If your application is installed on a network, there are some basic requirements that must be fulfilled for the application to operate properly.

Network Hardware Requirements

In order for a CA-Clipper-compiled program to operate on a network, there are some basic hardware and operating system requirements:

- The workstation operating system must be DOS 3.3 compatible
- The workstation must have enough memory to run the network operating system shell, the workstation operating system, and the CA-Clipper application
- The network operating system must support the DOS 3.3 networking protocol

Note: The amount of memory used by the network shell varies from one network operating system to another.

Assigning Rights

Most network environments govern the access of users to various locations on the file server disk as a part of the network security system. Network security is necessary in a shared environment to provide suitable levels of protection to other people's work, as well as privacy. This access control is maintained through a system of rights to directories and files, which are granted to users by a network supervisor.

With most PC-based network operating systems, rights are pessimistically granted. Users are given a minimal set of rights as a default and must be explicitly granted subsequent rights. If the user attempts to access a file and perform some operation without the required rights for that operation, an error is generated from the network operating system to the application. This means that for an application to operate properly in a network environment, the user must be granted the appropriate level of rights.

With a CA-Clipper-compiled program, the minimum set of rights required to guarantee that the program will successfully run is as follows:

- Open rights—the user has the right to open existing files
- Read rights—the user has the right to read from or execute files
- Search rights—the user has the right to search the directory's file list
- Write rights—the user has the right to write to existing files

Remember when the application attempts to open a file in read-write mode and the user does not have write privileges to that directory or file, an error is generated. As a default, CA-Clipper opens database and index files as read-write unless specified with the READONLY clause of the USE command.

Note: If the application creates files for any reason, the user will also require create rights. This could include indexing, copying, or writing output to a print file.

Setting Up Network Devices

Part of the resource a network offers is shared access to peripherals and devices. For most applications these are network printers.

In a network environment, output to a network printer is captured by the network shell (resident in the workstation's memory) and redirected to the network's printing system. Typically, captured output is spooled to a file until the print job terminates. The spool file is then placed in a print queue and printed when the network printer becomes ready.

With a CA-Clipper program, one of the workstation's ports must be redirected to the specified network printer or queue. This is usually done by executing a network utility prior to executing the application. This can be done either in the user's login script or in the application batch file (discussed later in this chapter).

Refer to the "Network Programming" chapter in this guide for more information on programming considerations for network printing. Refer to your network documentation for more information about accessing and setting up network printers.

The Application Batch File

If your application has specific invocation requirements, you can create a batch file to invoke it. This batch file can be quite simple, changing the directory to the application home directory then invoking the application. For example:

```
ECHO OFF
REM Accounts Receivable Module
CD \APPS\ACCOUNT
AR
CD \
```

Additionally, you can set the CA-Clipper-specific environment variables for the duration of the run then release them when the application quits. For example:

```
ECHO OFF
REM Accounts Receivable Module
SET CLIPPER=//F:54 //E:2000
SET TMP=C:\TEMP
CD \APPS\ACCOUNT
AR
CD \
SET CLIPPER=
SET TMP=
```

Additionally, the application batch file can run network configuration utilities to redirect output to the network printer, among other workstation specific actions.

Depending on the standards of the user's site, such as corporate or company standards for invoking programs, you can put the batch file in the user's root directory, batch file directory, or \DOS directory. Most importantly, the batch file must be in the PATH so it can be invoked from anywhere on the user's disk.

Chapter 4
Network Programming

In This Chapter

CA-Clipper provides the following features that allow you to take advantage of Local Area Network (LAN) capabilities to develop shared applications:

- Shared access allows two or more users to open the same file simultaneously
- Exclusive access prevents other users from opening the same file at the same time
- Logical file locking prevents two or more users from updating the same file at the same time
- Logical record locking prevents two or more users from updating the same record at the same time
- Multiple record locking allows a range of records to be locked simultaneously
- Lock status checking, built into both the file and record locking mechanisms, returns the current lock status as a logical value
- Record and file lock release returns to shared access mode
- Printer redirection allows use of a shared network printer
- Network error checking tests the success or failure of certain operations

This chapter discusses the commands and functions used to implement these features in detail. It contains conceptual information as well as detailed instructions for developing applications in CA-Clipper to run on a LAN. For more specific reference material on any of the commands and functions discussed in this chapter, see the *Reference Guide*.

The following topics are covered:

- LAN requirements for CA-Clipper
- Using shared mode
- Locking
- Overlays on a network
- Update visibility
- Network printing

Note: If your application does not require access to a shared database file, even though your computer system has a LAN, the information in this chapter does not apply to you. By default, a CA-Clipper application assumes a single user and opens all files in exclusive mode; therefore, no file sharing takes place and no locking is necessary.

Important! *Even though your CA-Clipper application uses an RDD that supports the database and index file formats of another software product, the application may not be able to share data simultaneously with an application written using that software product. The interoperability of two applications depends on the locking schemes defined in the individual RDDs, and compatibility is not always achievable. Refer to the Drivers Guide for more specific information.*

LAN Requirements for CA-Clipper

CA-Clipper uses DOS calls exclusively for all network-related operations and, consequently, applications that are compiled with CA-Clipper can run on any LAN designed to the DOS standard. You should be familiar with the nature and design of your LAN before attempting to develop applications that use the CA-Clipper network features described in this chapter.

The requirements for running a CA-Clipper application on a LAN are as follows:

- The LAN must adhere to DOS 3.3 or higher function calls
- Each workstation must be running under DOS version 3.3 or higher in order to use files in shared mode

Using Shared Mode

When you write an application that will run on a LAN, you should open database files in shared mode unless exclusive mode is required.

You can use either of these strategies:

- Change the status of the global SET EXCLUSIVE flag from ON to OFF in the main routine for your application. This will cause database files to be opened in shared mode by default. It also means that you must specify exclusive use of database files when you require it (via USE...EXCLUSIVE).
- Specify shared access when you open a database file that requires it (via USE...SHARED).

Note: The low-level file open function, FOPEN(), allows you to specify the open mode directly and is unaffected by SET EXCLUSIVE. See the *Reference Guide* for more information on this function.

When to Obtain Exclusive Use

Certain operations require exclusive use of a database file to function properly.

CA-Clipper enforces this requirement with an error message if you attempt to use any of the following operations with a database file opened in shared mode:

- PACK
- REINDEX
- ZAP

If SET EXCLUSIVE is OFF, attempt to open the file with USE...EXCLUSIVE before performing any of these operations. Because exclusive use is precluded if any other users are sharing the file, it may be necessary to provide logic for communication among workstations to request another user to close the file.

Other File Open Operations

In addition to USE and SET INDEX, several other CA-Clipper operations open one or more files in the course of operation and determine the open mode (either shared or exclusive) automatically. As a programmer you do not have control over this, but knowing it helps you make the proper allowance in your network applications.

There are two general rules that will help you to decide how a given operation works: if it writes to the file, the open mode is exclusive; if it only reads the file, the open mode is shared. For example, SAVE writes to a (.mem) file and opens it exclusively; RESTORE reads a (.mem) file and opens it shared. Therefore, your programs must anticipate possible conflicts when working with .mem files and provide mechanisms for avoidance (for example, using interworkstation communication) or recovery (for example, using the error system).

The following table shows the open modes for all operations that automatically open files. In cases where there are two possible file open operations, both files are shown; the ordering of the files refers to the order in which they appear in the syntax.

Operation	File1	File2
APPEND FROM	Shared	
COPY FILE	Shared	Exclusive
COPY STRUCTURE	Exclusive	
COPY STRUCTURE EXTENDED	Exclusive	
COPY TO	Exclusive	
CREATE	Exclusive	
CREATE FROM	Exclusive	Shared
DISPLAY...TO FILE	Exclusive	
INDEX	Exclusive	
JOIN	Exclusive	
LABEL FORM...TO FILE	Shared	Exclusive
LIST...TO FILE	Exclusive	
MEMOREAD()	Shared	
MEMOWRIT()	Exclusive	
REPORT FORM...TO FILE	Shared	Exclusive
RESTORE	Shared	
SAVE	Exclusive	
SET ALTERNATE	Exclusive	
SET PRINTER	Exclusive	
SORT	Exclusive	
TEXT...TO FILE	Exclusive	
TOTAL	Exclusive	
TYPE...TO FILE	Shared	Exclusive
UPDATE	Shared	

Retrying After an Open Failure

In network programming, the failure of a file open operation becomes a normal possibility; therefore, you should always check NETERR() immediately after a USE to make sure the file was successfully opened.

Opening Index Files

For this reason, you should never include the INDEX clause in a USE command when programming in a network environment. Although USE...INDEX is a single command, it performs several distinct open operations. Instead, open the index files with SET INDEX after checking NETERR() to determine that the USE succeeded. Using this method, you can prevent possible runtime errors and be guaranteed that the index files will be opened successfully and in the same mode as the associated database file.

The following code illustrates the correct way to open database and index files in a network program:

```
USE MyFile SHARED
IF NETERR()     // Returns true (.T.) if USE failed
   ? "File not available in shared mode."
   BREAK
ENDIF
SET INDEX TO MyIndex1, MyIndex2
```

The NetUse() Function

NetUse() is defined in the sample program Locks.prg. The function demonstrates a flexible and comprehensive file opening scheme that attempts to open a database file in either shared or exclusive mode and continues trying to open the file for a given amount of time or until it succeeds.

The syntax for NetUse is as follows:

```
NetUse(<cDatabase>,<lOpenMode>,<nSeconds>) → lValue
```

<cDatabase> is the name of the database file to open; *<lOpenMode>* is the open mode; and *<nSeconds>* is the number of seconds to retry the operation—a value of zero retries forever. NetUse() returns true (.T.) if the file was opened successfully and false (.F.) if it was not.

The next example is very similar to the previous one, except that it uses NetUse() to open the database file:

```
#define DB_SHARED    .F.
#define DB_EXCLUSIVE       .T.

IF NetUse("File", DB_SHARED, 5)
   SET INDEX TO MyIndex1, MyIndex2
ELSE
   ? "File not available in shared mode."
   BREAK
ENDIF
```

Tip: To use any of the functions defined in Locks.prg (located in your \CLIP53\SOURCE\SAMPLE directory), compile Locks.prg according to the instructions at the top of the program file and link it with your application. Note that you can use any function defined in this program file as is, or simply use it as a starting point for your own customized version.

Note: In a network environment, your application must always anticipate that a file may not be available, regardless of the open mode. When exclusive use is requested, another user accessing the file (in either shared or exclusive mode) would make the file unavailable. When shared mode is requested, another user with exclusive use would make the file unavailable.

Locking

If you allow file sharing in your application, you must determine when file or record access by more than one user must be prevented. Then, you can incorporate lock functions into your application with accompanying logic to control execution in accordance with the lock results. Use these basic guidelines to determine when locking is necessary:

- Locking is *required* whenever you are going to write to a database file
- Locking is *optional* at all other times

There are two levels of locking: record and file. The one you use will depend on the operation required. If you are updating one record at a time, a record lock will be sufficient; however, if you are performing a mass update, you will need to lock either the entire file or all of the records involved in the operation. If you neglect to obtain a lock before attempting to write to a shared file, the operation will result in an error message.

The following table summarizes the operations that require locks in a network environment:

Operation	Requirement
@...GET	Record lock
DELETE (single record)	Record lock
DELETE (multiple records)	Exclusive use, file lock, or multiple record locks
RECALL (single record)	Record lock
RECALL (multiple records)	Exclusive use, file lock, or multiple record locks
REPLACE (single record)	Record lock
REPLACE (multiple records)	Exclusive use, file lock, or multiple record locks
UPDATE	Exclusive use, file lock, or multiple record locks

Note: CA-Clipper does not require a lock for read-only operations, such as COUNT, LIST, REPORT FORM, and SUM—even if they have multiple record scopes. The automatic buffer management technique built into CA-Clipper ensures that these operations have access to the most up-to-date information in the database files; however, you may wish to obtain a file lock before performing these operations to make sure that none of the data changes during the process. Whether to lock the file depends on your application requirements.

File Locking

The FLOCK() function attempts to place a file lock on the current database file and returns a logical value indicating the success or failure of the lock.

FLOCK(), however, has the drawback that it only tries once to lock the file. Because of the possibility of a failure (a file lock will fail if another user has a file or record lock for the same file), you will probably want to make several attempts before admitting defeat.

The FilLock() Function

The FilLock() function, defined in the sample program Locks.prg, allows you to retry the file lock for a specified time limit. The syntax for FilLock() is as follows:

```
FilLock(<nSeconds>) → lValue
```

<nSeconds> specifies the number of seconds to retry the file lock—a value of zero retries forever. Like FLOCK(), FilLock() returns true (.T.) if the file lock was obtained and false (.F.) if it was not.

Record Locking

Record locking is similar to file locking and is accomplished with RLOCK() or DBRLOCK(). RLOCK() attempts to lock the current record, releasing all other locks currently held, whereas DBRLOCK() allows you to lock a range of records, called a *lock list*, by specifying a particular record to lock. Both functions make a single attempt to lock the specified record and return a logical value indicating success or failure.

Like FLOCK(), however, RLOCK() and DBRLOCK() try to lock the record only once. Because of the possibility of a failure (a record lock will fail if another user has a file lock for the same file or a record lock for the same record), you will probably want to make several attempts before admitting defeat.

The RecLock() Function

The RecLock() function, defined in the sample program Locks.prg, allows you to retry the record lock for a specified time limit. The syntax for RecLock() is as follows:

```
RecLock(<nSeconds>) → lValue
```

<nSeconds> specifies the number of seconds to retry the record lock—a value of zero retries forever. Like RLOCK() and DBRLOCK(), RecLock() returns true (.T.) if the record lock was obtained and false (.F.) if it was not.

Tip: RecLock(), in its default implementation, does not allow for multiple record locks. Slight modifications to the source code to allow a record number argument and to call DBRLOCK() instead of RLOCK(), however, will add this functionality.

Automatic Locking

When you append a blank record to a shared database (using DBAPPEND() or APPEND BLANK), CA-Clipper automatically tries to lock the new record, setting NETERR() to indicate its success or failure. Unless the new record can be locked, it is not added. Thus, there is no need to attempt a record lock before editing the new record, but it is necessary to check NETERR() so that you will know whether the record was actually added.

The AddRec() Function

AddRec(), defined in the sample program Locks.prg, is similar to FilLock() and RecLock() in that it allows you to retry the append operation for a specified time limit. This function illustrates how to properly test for a successful append operation in a network environment. The syntax is as follows:

```
AddRec(<nSeconds>) → lValue
```

<nSeconds> specifies the number of seconds to retry the append—a value of zero retries forever. AddRec() returns true (.T.) if the new record is added and locked and false (.F.) if it is not.

Tip: AddRec(), in its default implementation, does not allow for multiple record locks. Slight modifications to the source code to allow a logical argument and to call DBAPPEND() instead of APPEND BLANK, however, will add this functionality.

Unlocking

Once a lock is in place, an application can write to the file. Other users' attempts to lock the same record or file will fail, but they will have read access to the data.

The lock remains in place until the application releases it by:

- Closing the locked file
- Explicitly releasing the lock (for example, DBRUNLOCK() releases a single record lock, DBUNLOCK() and UNLOCK release all locks for the current work area, and DBUNLOCKALL() and UNLOCK ALL release all locks in all work areas)
- Issuing another lock function (successful or not) for the same file

  Note: DBRLOCK() will not release existing locks, unless specified without an argument.
- Terminating the program normally

Because efficient network applications seek to minimize the duration of the locks they impose, you should release locks as soon as possible after they have served their purpose.

Resolving a Failure

The code for opening files and attempting locks is straightforward and, as suggested already, you may want to try these operations several times before giving up. If, however, a file cannot be opened or a lock cannot be obtained, the application has to abandon its original intentions and come up with substitute plans.

Communicate the Failure to the User

It is essential to communicate any lock or open failure to the user. You can make the communication transient (a two second duration, for example) or follow it by an INKEY(0) pause to be sure the user sees it. It can be a small message displayed in the corner of the screen or a whole screen displayed between SAVESCREEN() and RESTSCREEN().

Offer Choices

As part of the communication, you may also want to provide alternatives and make a branching choice based on the user's selection. A typical menu might offer these choices:

- Retry

  This choice simply allows the user to extend the lock or open effort, without seeking to resolve the failure.

- Try same activity, different data

  This choice makes sense for lock failures in many applications. If the target record is not available, use another instead. For example, a data entry operator working on a stack of credit adjustment slips will not mind putting Smith's slip at the bottom of the pile and moving on to Jones and Brown. When the operator gets back to Smith an hour later, that record will probably be free. Therefore, make the code branch to the record selection entry point.

- Abort current activity and go back to higher menu

  The choice implies abandoning not only the record in question, but the activity itself. For example, the user leaves the credit adjustment section of the program altogether and goes back to the main menu.

Overlays on a Network

Linkers sometimes produce overlays containing compiled program code. By default, all compiled CA-Clipper code is placed in dynamic overlays. Additionally, static overlaying may be requested for C or Assembler code.

By default, overlays are appended to the end of the application's .EXE file (the code may be written to a separate overlay file by using the INTO clause in the overlaying directive). During execution, the runtime system opens the .EXE file and reads overlays as required.

When the runtime system needs to read overlays, it opens the .EXE file for shared, read-only access. On a network, this allows multiple processes to read the file at the same time.

On some networks, however, a problem can arise if an attempt is made to run the .EXE when another process has opened it to read overlays. The error occurs because the DOS EXEC function (used by COMMAND.COM to execute programs) opens .EXE files using a sharing mode called "compatibility mode." On some networks, this causes a sharing violation if the file is already in use by another process, even if the other process has specified that it wishes to share the file.

If this conflict occurs, it can usually be resolved by marking the affected file read-only on the disk. A small utility program RO.COM, located in \CLIP53\BIN, is supplied for this purpose. RO toggles the read-only status of a specified file.

Update Visibility

When programming in a network environment, it is important to determine when database (and index) updates actually become visible, which differs depending on the *observer*. This section describes the visibility rules for each of the following possible observers:

- The initiator
- The operating system and other processes
- The physical disk

Note: The rules specified in this section are for the DBFNTX and DBFNDX database drivers. Rules for other drivers may differ.

The Initiator

The initiator of an operation is the process which causes an update to occur. All updates appear to the initiator immediately after they are made.

The Operating System and Other Processes

Updates are guaranteed to be visible to other processes (other applications running on the network) when the initiator writes the update to the operating system (although they may become visible sooner). Writing to the operating system can be done in several ways:

Exclusive Mode

If the file is open in exclusive mode, the update is not guaranteed to appear until the file is closed or a commit operation (for example, using DBCommit() or COMMIT) is performed.

File Lock

If the file is open in shared mode with a file lock, the update is not guaranteed to appear until the file is closed or one of the following operations is performed: unlock, record lock, or commit.

Record Lock

If the file is open in shared mode with a record lock or lock list, the update is not guaranteed to appear until the file is closed or one of the following operations is performed: unlock, commit, or any record movement operation (such as skipping to a new record or going directly to a specified record).

The Physical Disk

Writing an update to the operating system does not guarantee that a physical disk write will take place, because the operating system may hold recently written records in memory. These records appear to other processes as if they were on the disk, but if a failure occurs (such as a hardware problem), the records may never be physically written to disk. In this case, updates are lost, and processes to which the updates were visible may be proceeding with erroneous data.

To ensure that updates are written to disk, commit the changes (for example, with DBCOMMIT() or COMMIT) or close the database file (which automatically commits the changes). Any of these actions sends a request to the operating system to perform a solid-disk write immediately.

Caution! *Even if you commit changes to disk, there is no guarantee that a physical disk write will actually occur. Some cache programs and network server software will intercept the commit request and postpone the physical disk write. This is a violation of DOS protocol unless the underlying system can guarantee that its method of committing updates is as reliable as writing them to the disk controller hardware.*

Abnormal Termination

When an application terminates normally, all files are closed and all updates are committed to disk.

If the application terminates abnormally (for example, with an unrecoverable database error), some updates may be lost. The missing updates may potentially include every update to a file since updates to that file were last made visible to the operating system.

If the operating system fails during an application, missing updates may potentially include every update to a file since the last point at which updates to that file were committed to disk.

Network Printing

A network by nature is a system that provides shared access to common resources. Printers are among several hardware devices to which a network can provide shared access. From a general point of view, network operating systems provide services to make access to shared printers as transparent as possible to the workstation. Because of this, printing to a local printer appears no different to an application than does printing to a network printer.

The actual difference between printing to a local printer and a network printer is that the output to the printer port is captured by the network shell and redirected to the network file server. As the output is received by the file server, it is *spooled* to a file stored on the file server.

When a print job is complete, the file is closed. If the network printer is busy, the spooled file is placed in a print queue and sent to the printer whenever it becomes ready. Network print queues are generally first in, first out, although most network operating systems provide print queue management facilities to change the order of print jobs.

When sending printed output to a network printer from a CA-Clipper program, there are some considerations and approaches to consider.

Setting Up the Network Printer

In order to print to a network printer, one of the workstation ports must be redirected to the network printer. This is usually done with a network utility that configures the network shell.

This configuration can be accomplished:

- In the user's login script
- In the workstation AUTOEXEC.BAT file, but after the user has logged into the network
- In the application batch file
- By running the configuration utility under program control

Program Design Considerations

With most PC-based networks, a print job begins when the application opens the PRN device. Printed output is captured and redirected to the spooler until the PRN device is closed by the application. At that point, the print job is submitted to the print queue and is printed when the printer becomes available.

In CA-Clipper, you can explicitly control this process with the SET PRINTER TO command. SET PRINTER TO specified with the device name begins the print job. After all of the printed output has been sent to the printer, SET PRINTER TO with no argument closes the print job and places the output in the print queue.

For example, the following code fragment starts and ends a print job with SET PRINTER TO:

```
USE Customer NEW
SET PRINTER TO LPT1
SET PRINTER ON
DO WHILE !EOF()
   ? Customer, Amount
   SKIP
ENDDO
SET PRINTER OFF
SET PRINTER TO
CLOSE Customer
```

Note: Some network operating systems support a time-out feature that terminates a print job if the network shell has not received output for a specified number of seconds. Refer to your network documentation for more information.

Printing to a File

One of the great problems of printing is the strong likelihood that something will happen during the printing process that could abort a critical print job. In large scale installations with many users and a large volume of transactions, a printer problem can have system-wide consequences. Accounting systems may, for example, print a report of transactions as they are posted at the end of an accounting period. If the report is sent directly to the printer, a print job failure could compromise database integrity. In this instance, reprinting the report would require a rollback of all posted transactions, and all print jobs would then have to be resubmitted.

A better approach is to print everything to a file then, sometime later, submit the print file to the network printer. If there is a printer or network failure before the report is successfully printed, the print file need only be resubmitted to the network printer. Once the report has been successfully printed, the print file can be deleted.

After printing to a file, you need to place the file in the print queue. There are several approaches to this. Perhaps the easiest method is to use the COPY FILE command to copy the file to the PRN device.

Chapter 5
Programming in Graphic Mode

In This Chapter

CA-Clipper 5.3 provides you with the capability of switching easily from text mode to graphic mode. In fact, its GUI controls, such as push buttons and check boxes, are designed to take advantage of graphic mode, thereby minimizing the code changes that are needed to make the transition to graphic mode.

This chapter explains the benefits and techniques of switching programming modes. The following topics are discussed:

- Overview of graphic mode
- Invoking graphic mode
- Using graphic mode
- Compiling and linking
- Managing bitmap and font resources
- Troubleshooting

Overview of Graphic Mode

As mentioned earlier in the discussion of the input/output system in the "Basic Concepts" chapter, CA-Clipper has the ability to display output to the screen in one of two modes: text mode and graphic mode. Unlike text mode, which limits your display options to the 255 characters in the ASCII character set and allows you to place only one character at one particular location on the screen, graphic mode allows you to use pixel coordinates for your screen displays. This means you can draw smoother lines and utilize bitmaps for icons, toolbar buttons, and as wallpaper for the background of your screens.

Programming Similarities

In some ways, text mode and graphic mode programming are very similar. For example, the task of changing the size of a window containing a TBrowse object involves the same basic approach, no matter which mode you are using.

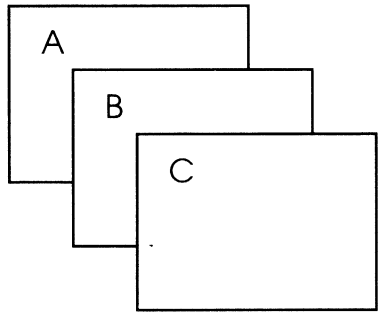
In either mode, you would include the following steps:

1. Establish how the user will indicate a desire to change the window size, probably using some form of the INKEY() function.
2. Remove the existing window frame and its contents by using RESTSCREEN() to restore the previously saved background.
3. Redisplay the window frame using DISPBOX().
4. Update the TBrowse object's coordinate instance variables.
5. Repaint the TBrowse data using TBrowse:refreshAll().

There are, however, many differences between programming in the two modes. In particular, because graphic mode presents you with more display options and flexibility, it entails greater programming complexity than text mode.

Programming Differences

For example, consider the techniques needed to prevent output from being displayed in a defined region of the screen. Imagine a cascading window system with three windows, A, B, and C:



Let us compare the differences between text mode and graphic mode techniques used to display information in window A while it is covered by windows B and C.

The Text Mode Technique

Using 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. Remove each screen by individually saving it and immediately restoring its previous background. For example, to uncover window A you would remove windows C and B as follows:

```
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, A is the only visible window, making its contents easy to manipulate.

4. After making the necessary display changes to window A, save and redisplay windows B and C:

```
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() to instantly flush *all* of the buffered screen output to the screen.

Using the technique of buffering screen output by placing it between calls to DISPBEGIN() and DISPEND() ensures that 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 and simple, and the memory needed to save a full screen is small (4 - 8 KB).

The Graphic Mode Technique

The situation is totally different in graphic mode. Instead of removing windows one by one until the desired window is uncovered, you use *exclusion areas*. An exclusion area is simply a defined screen region where nothing can be written, a concept that is used in almost every graphical system, including Windows.

In the example above, you would declare windows B and C as exclusion areas to prevent data from being written to these windows. Then, when you modify information in window A, windows B and C are not affected.

Your code would be read as follows:

```
// Window B coordinates
GSETEXCL(nTop, nLeft, nBottom, nRight)

// Window C coordinates
GSETEXCL(nTop, nLeft, nBottom, nRight)
// Manipulate or paint information in window A
.
.
.

// Remove the previous exclusion areas
GSETEXCL(LLG_EXCL_KILL_ALL)
```

Note: GSETEXCL() is probably the most complex function in CA-Clipper graphic mode. Please refer to the *Reference Guide* for further explanation of GSETEXCL(), as well as the other graphic mode commands and functions.

Now that you have had a brief introduction to graphic mode, let us proceed to some of the basics, including: invoking and using graphic mode, compiling and linking, and managing bitmap and font resources with graphic mode utilities.

Invoking Graphic Mode

For compatibility with previous versions of CA-Clipper, text mode is the default video mode, but switching from text mode to graphic mode is easy. Simply include the header file Llibg.ch and use the SET VIDEOMODE command, as shown in this example:

```
// Header file with graphic constants
#include "Llibg.ch"

SET VIDEOMODE TO LLG_VIDEO_VGA_640_480_16
// All output to the screen will now be in graphic
// mode
```

Llibg.ch contains #defines for all of the constants designed for use with the graphic mode commands and functions. Using SET VIDEOMODE with the LLG_VIDEO_VGA_640_480_16 constant first clears the screen, then switches to a 640 x 480 pixel screen in VGA mode with 16-color support.

To switch back to text mode, use the SET VIDEOMODE command again:

```
// Header file with graphic constants
#include "Llibg.ch"

SET VIDEOMODE TO LLG_VIDEO_TEXT
// Now all output will be in text mode
```

You can even switch just part of an application to graphic mode, as long as you use the CLS command before switching modes! See the SET VIDEOMODE command in the *Reference Guide* for more details.

Using Graphic Mode

Switching from text mode to graphic mode in existing CA-Clipper applications should follow the steps enumerated below, while observing the precautions discussed in the next section, entitled Shadowing Restrictions:

1. Make sure that you are using only CA-Clipper functions for all screen displays and that other third-party products are manipulating the screen functions (using the General Terminal (GT) module).
2. Invoke graphic mode as explained in the previous section, remembering to place the SET VIDEOMODE command as early as possible in your application. For example, if you are using an INIT() function, put this command in the first INIT() defined in your main program file.
3. Link the application with the LLIBG.LIB library, as explained later in this chapter under Compiling and Linking.
4. Run your application. It should function in graphic mode the same way it functioned in text mode.

Shadowing Restrictions

Graphic mode is restricted when displaying shadows around windows. However, if you stay in text mode, there is no such restriction since screen saves are compatible with CA-Clipper text mode and are handled as character strings.

For example, the following user-defined function, MakeShade(), remains valid in text mode:

```
MakeShade(nTop,nLeft,nBottom,nRight,nFore,nBack)
   cSave := SAVESCREEN(nTop, nLeft, nBottom, nRight)
   cSave := TRANSFORM(cSave, ;
      REPLICATE('X' + CHR(nFore + 15 * nBack), ;
      LEN(cSave) / 2))
   RESTSCREEN(nTop, nLeft, nBottom, nRight, cSave)
   RETURN NIL
```

This shadowing technique, however, will not work in graphic mode. First of all, a graphic screen is saved in more than 150 KB of VMM memory, whereas it is saved as a 4 – 8 KB character string in text mode. Secondly, it is impossible to process a VMM memory area with functions like TRANSFORM(). Therefore, it is impossible to handle shadows using SAVESCREEN(), TRANSFORM(), and RESTSCREEN() as shown in the previous example.

Instead, when you are using graphic mode, it is much quicker and more elegant to draw a rectangle in XOR mode (that is, using GRECT() with the <*nMode*> parameter set to LLG_MODE_XOR) to achieve the shadowing effect. To eliminate the shadow, just draw another identical rectangle of the same region in XOR mode.

Note: Mastering XOR and SET modes is indispensable for a full understanding of graphical interfaces.

Enhancing an Application

Now that your application is working, it is time to start enhancing it. The simplest thing to do is to replace the usual frame parameters in DISPBOX() by LLG_BOX_GRAY_STD so as to get frames with a three-dimensional (3-D) look.

Next, make sure that you modify the TBrowse:footSep variable, replacing it with a null string (""). Because the lower edge of your frame is now made up of pixels instead of semi-graphic line drawing characters, you do not have to write characters in this area.

Extended character specifications for left and right box edges, such as the following, are also no longer desirable and can be removed from your code:

```
@ nRow, 0 SAY CHR(198)
@ nRow, 79 SAY CHR(181)
```

The more modular your application, the easier and more centralized these modifications will be.

Take advantage of the fact that you now have 30 lines of text (normal 640 x 480 VGA mode) and not just 25. Your users will appreciate it. For example, you may want to increase the size of your TBrowse objects with the additional lines available. You may also want to move your message display area from 24 to 29 and give it a thin frame with a 3-D look using GFRAME().

You will generally use GFRAME() to frame the menu, DISPBOX() with LLG_BOX_GRAY_STD for the windows of scrolling menus, and GWRITEAT() to write characters transparently pixel by pixel. Also, GWRITEAT() and GLINE() allow you to simulate top bar menus with accelerator keys (i.e., the underlining of the active character in each menu).

Tips and Techniques

Below are a few tips and techniques for more sophisticated programming in graphic mode.

Push Buttons

Use push buttons whose size is an integer multiple of the font size and whose position is also an integer multiple of this size. This greatly simplifies the task of incorporating icons into applications.

Bitmaps and Icons

You can use icon (.ICO) files as easily as bitmap (.BMP) files. See the GBMPLOAD() function for more information.

X/Y and Row/Column Coordinates

As a CA-Clipper developer, you are probably familiar with functions and commands that address the screen using row and column coordinates (using @...SAY, DISPBOX(), and DEVPOS(), for example). When specifying text-based screen coordinates in this manner, you have always designated—and will continue to designate—the row followed by the column.

Graphical environments, however, use Cartesian coordinates, which are expressed as an X coordinate followed by a Y coordinate. Therefore, when using the graphic mode functions to address pixel-based screen coordinates, you must use this new convention which is, unfortunately, the opposite of the row/column order convention to which you have become accustomed.

In some cases, the graphic mode functions also return coordinates. When this is the case, the return coordinates are stored as an array. The best way to access the elements in such an array is using the relevant #defines. For example, to retrieve the text-based row and column coordinates, you would use:

```
MSTATE(LLM_COOR_TEXT)[LLM_STATE_ROW]
MSTATE(LLM_COOR_TEXT)[LLM_STATE_COL]
```

To retrieve the pixel-based X and Y coordinates, you would use:

```
MSTATE(LLM_COOR_GRAPH)[LLM_STATE_X]
MSTATE(LLM_COOR_GRAPH)[LLM_STATE_Y]
```

Compiling and Linking

As mentioned earlier, all of the constants utilized by the graphic mode functions are defined in Llibg.ch, so it is important that you include this file in your application—otherwise, any function or command call involving constants beginning with LLG_ or LLM_ cannot be compiled.

Once your application is successfully compiled, you can link it using any linker, including either the default, protected mode linker, CA-Clipper/Exospace; the real mode linker, Blinker; or any other linker. The graphic mode functions are defined in the library, LLIBG.LIB, which is automatically linked if you are using CA-Clipper/Exospace. If you are using any other linker (including Blinker), you must include LLIBG.LIB in your linker script.

Important! *If you choose to ignore default libraries using CA-Clipper/Exospace (using the NODEFLIB option) or use another linker, make sure you link LLIBG.LIB before CLIPPER.LIB. Otherwise, your application will not function properly in graphic mode.*

Managing Bitmap and Font Resources

CA-Clipper 5.3 provides several utilities for managing an application's resources (such as bitmap and font files) using a single bitmap library (.BML) file. This section describes all of these utilities and provides an overview of how .BML files are automatically handled by your application.

RES2BML.EXE

Use RES2BML to create a library file to store the individual resources (such as bitmap and font files) used by your application.

Storing resources in a single .BML file rather than individual files has the advantage of making your application easier to deliver to end users—instead of delivering a single .EXE file with 40 .BMP files, for example, you deliver only two files: an .EXE and a .BML containing all the necessary .BMP files.

To invoke this utility, use the following syntax:

```
RES2BML <inputFileSpec list> <bmlFile>
```

<inputFileSpec list> is a list of file specifications to define the input files, which are limited to .BMP, .FND, and .FNT file formats. Each file specification can contain wildcard characters, such as "?" and "*." If you specify more than one file specification, separate them using a single space.

Note: In CLIPPER.BML, some bitmap files have an extension of .BMU. These files are simply renamed .BMP files, and they work properly with RES2BML.

<bmlFile> is the name of the resulting library file. The file name that you choose should have the same base name as the corresponding .EXE file, but with a .BML extension.

Important! *At runtime, the .BML file must be present in the same directory and must have the same base name as its corresponding .EXE file. Otherwise, the application will not be able to locate the .BML file.*

For example, if your executable file is MYAPP.EXE and you have several .BMP files with "MY" as the prefix, you would type the following:

```
RES2BML MY*.BMP MYAPP.BML
```

Likewise, to include both bitmap and fonts in the resulting library file, you would use syntax similar to the following:

```
RES2BML A*.BMP H*.FNT *.FND MYAPP.BML
```

BML2RES.COM

Use BML2RES to extract the original resource files from a .BML file previously created with RES2BML.

To invoke this utility, use the following syntax:

```
BML2RES <bmlFile>
```

<bmlFile> is the name of the library file whose contents you want to extract. All bitmap and font files contained in this file will be extracted and recreated on disk.

BMLDIR.COM

Use BMLDIR to view the contents of a .BML file previously created with RES2BML.

To invoke this utility, use the following syntax:

```
BMLDIR <bmlFile> [ | more ]
```

<bmlFile> is the name of the library file whose contents you want to view. Use the vertical pipe and more filter if the entries exceed the number of available rows on your screen and you want to pause between groups of entries.

Using Bitmap and Font Library Files

The GBMPLOAD() and GFNTLOAD() functions are designed to work with resources that are stored in individual files or in a .BML file. When you invoke either of these functions, the application automatically searches for the named resource in the following order:

1. In a file with the same name as your executable but with a .BML extension
2. In CLIPPER.BML

   Note: The bitmap library file, CLIPPER.BML, contains all the resources and fonts used by CA-Clipper to display its user interface components. (See Appendix A for descriptions of the bitmaps and fonts supplied in CLIPPER.BML.)
3. As a separate disk file

Important! *All .BML files and resources must be located in the directory where the .EXE is running.*

This strategy is very powerful and easy to use. In the development phase, you can work with individual bitmap and font files. When you are ready to distribute your application, use RES2BML to create a single .BML file from the separate resources. Because the graphic mode functions are designed to work with both library and individual resource files, no source code changes are needed!

Troubleshooting

This section suggests ways to resolve certain problems that can occur when you execute your CA-Clipper application in graphic mode. Examine this section for solutions to common problems before you call for Computer Associates Technical Support.

Common Problems

Listed below are a number of common problems that you may have programming in graphic mode:

- *"My application won't go into graphic mode."*

  Either the application is not being linked with LLIBG.LIB or CLIPPER.LIB is before LLIBG.LIB in the link order.

  This may also occur if your video card is not 100 percent VESA 1.2 compatible. Please check your video card manufacturer's specifications. Also, other add-ons or custom routines which manipulate the video output may be causing the problem, including:

  - Non-VGA standard

    The video card is not supported because it is non-VGA standard.

  - VESA

    Graphic mode is designed to work with 100 percent VESA-compatible graphics cards. If your graphics card is not 100 percent compatible or only SVGA-compatible, you may experience problems when switching to some video modes or even when displaying. We provide UNIVESA, which is a TSR to help make your video card VESA-compatible. See Hardware Compatibility later in this section for further information.

- *"My mouse won't load."*

  This may be caused when using an older mouse driver. Please see the MSTATE() function for further information.

- *"Why am I getting conflicts with INT10/INT25 when using CA-Clipper Tools even though I do not link CTUSP?"*

  This problem occurs only with a few CA-Clipper Tools functions—mainly the network functions. These functions rely on some data segments stored inside the CA-Clipper GT module, even though these data segments are not used. As most graphics libraries replace the GT module, this data segment is no longer available at the required location. It is not important under real mode linking because the CA-Clipper Tools functions will access irrelevant data. However, under protected mode, this invalid data access causes a GPF.

- *"Can I define the button coordinates on the screen with pixel accuracy, as opposed to an 8 x 16 grid defined by regular text mode?"*

  Yes, any bitmap (.BMP) or icon (.ICO) file can be displayed at the pixel level and, additionally, the .BMP file sizes are not limited to 64 KB.

- *"Can I save and restore numerous screen areas to memory for icons and overlaid windows?"*

  The CA-Clipper VMM allows you to save whatever you want. Under protected mode, you will benefit from very fast memory handling. However, saving a screen is a very resource-greedy thing to do when in graphic mode.

  Instead, you can implement an exclusion regions algorithm that allows most of the overlaid window operations to be executed without any need for the SAVESCREEN() and RESTSCREEN() functions.

  For example, imagine you have a window with a TBrowse object and a second window which partially overlaps the TBrowse. By issuing a GSETEXCL() to the second window's coordinates, repainting the TBrowse in the first window will never affect the "excluded" window. See The Graphic Mode Technique earlier in this chapter for details.

Software Compatibility

In regard to third-party software packages, you cannot mix multiple graphic libraries in the same application. Add-on products that might conceivably cause compatibility problems are those that add or change CA-Clipper behavior with respect to terminal-handling functions.

You cannot use another library to switch your CA-Clipper application to 640 x 480 x 16 VGA mode and use the graphic mode GLINE() function. Even though it may result in a line being drawn on the screen, your graphic settings will not be correct.

Graphic mode does not simply switch the video mode when you call SET VIDEOMODE. A series of environment settings, such as SETMODE(), mouse size conversion, and so on, are updated. These values would not be properly set if you use another library to switch to graphic mode instead of calling SET VIDEOMODE. Mixing two graphics libraries is thus very likely to cause problems, and you will usually end up rebooting your PC!

For libraries that claim to "do everything," our advice is less simple. If the library is properly organized, it should be possible to use all of the non-graphic portions, along with graphic mode, without problems.

In general, beware of shadowing functions developed for CA-Clipper using C or Assembly language. These were intended for text mode in which a screen is described in terms of character/attributes. In graphic mode, shadowing is not accomplished in the same way as it is in text mode (see Shadowing Restrictions earlier in this chapter).

Hardware Compatibility

Processors

The graphic mode library was developed in C and Assembly languages. For execution speed, it uses instructions that are only supported by 80286 and later processors. An executable file linked with graphic mode will, therefore, not work on computers using 8088 or 8086 processors.

VGA Drivers

Graphic mode works with video cards that either conform to or perfectly emulate the VGA standard.

VESA Drivers

Graphic mode also works with VESA video modes. It is VESA 1.2 compatible, which means the video hardware must be compatible with the VESA 1.2 standard. However, many video cards (both older and recent) are not fully compatible with VESA 1.2—these may be compatible with VESA 1.0 or the SVGA standard.

You need a fully VESA VBE-compatible video card in order to use the VESA modes. Graphic mode supports resolutions of 640 x 480, 800 x 592, 1024 x 768, 1280 x 1024, and provides color support for 16, 256, 32 KB, 64 KB, and 16 million colors. (Only 640 x 480 resolution with 16 colors is supplied with CA-Clipper.) No driver is required if your video card is compatible with VESA VBE. Otherwise, you should use the driver supplied with your video card. The driver should be optimized for your card by the manufacturer.

If no driver was supplied by the manufacturer, use UNIVESA, which is a TSR shareware program that is included with CA-Clipper. (If UNIVESA is not able to install itself, your card is probably not VESA-compatible.)

When you load the UNIVESA driver, it overcomes some VESA 1.2 shortcomings in the hardware with software algorithms. This allows graphic mode to see the video hardware as fully compatible with VESA 1.2.

UNIVESA is universal, so it should run on all hardware. However, sometimes you may be able to load UNIVESA and access new video modes and still get poor results when switching to high resolution modes. In this case, you should try to use the VESA driver from the card manufacturer.

Memory Shortage

Graphic mode uses the VMM to allocate the memory it needs, taking care to free that memory after each call.

However, there are limits to what is possible in CA-Clipper. For example, a 640 x 480 screen with 16 colors takes more than 150 KB of memory. If you have enough EMM memory, saving the screen will be fast; but whenever the VMM needs to swap to disk, performance will suffer.

When you use TBrowse objects, GET commands, or any other type of display within an application, the display time amounts to a small percentage of the total processing time. Increasing this processing time a little will not be noticed by your user.

Chapter 6
Introduction to TBrowse

In This Chapter

TBrowse is one of several predefined classes included in CA-Clipper designed to give you control and flexibility in the retrieval, display, and editing, the *browsing*, of tabular views of data. It provides a robust, open-architecture browsing mechanism that is not tied to the (.dbf), or any other, file format.

This chapter establishes criteria for effective use of the TBrowse class. It starts with creation of a TBrowse object and a description of the TBrowse class, continues with information about the most common uses of TBrowse and TBColumn objects, and concludes by outlining problems and establishing guidelines for their resolution.

This chapter discusses the following topics:

- TBrowse overview
- Basic browse operations
- Optimization of the browse
- Browsing with Get
- Adding color
- Controlling the scope

Note: This chapter assumes that you are familiar with the concepts and terminology associated with CA-Clipper's predefined classes as discussed in the "Basic Concepts" chapter of this guide.

TBrowse Overview

To understand TBrowse, you must understand what it *does not* do as well as what it *does*. TBrowse is a passive object in the sense that it never initiates an action. It never takes control from the programmer for more than a few moments, never, without explicit direction, handles a single keystroke, and never "ends." You simply stop using it when you are done with it. TBrowse is a data source access tool that can browse any data that is expressible in a table form.

TBrowse differs from the CA-Clipper functions DBEDIT(), MEMOEDIT(), and ACHOICE(), in its degree of *exclusive* control. The functions control all screen and data manipulation without regard to the activation that calls them. TBrowse, on the other hand, operates as a tool of the calling activation. That activation controls and orders the action of the TBrowse object by sending messages to it.

TBrowse is not a replacement for DBEDIT() (though, internally, DBEDIT() is written using TBrowse). TBrowse is less limiting and more powerful than any of the functions. Due to the nature of objects under TBrowse, you can activate several simultaneous browses which can be suspended and restarted at will. TBrowse can be read-only or interactive.

Basic Browse Operations

Regardless of the nature of the data on which you use TBrowse, you should use the same general approach. The first example is the simplest of all cases, a read-only browse of a database. It starts with creation of the TBrowse object and design of the main TBrowse loop, then continues through stabilization and keystroke handling on a monochrome display.

Creating TBrowse Objects

The browse is composed of two different object classes: TBrowse, the main class, which controls the overall browse, and TBColumn, of which each column is an instance, and which is 'owned' by the TBrowse object.

Create a TBrowse Object

There are two functions for creating a TBrowse object: TBrowseNew() and TBrowseDB().

TBrowseNew() creates a completely generic TBrowse object. The following code creates a new TBrowse object and stores it in the variable, *oBrowse*. The coordinates define the rectangular area of the TBrowse object:

```
oBrowse := TBrowseNew(<nTop>, <nLeft>, ;
   <nBottom>, <nRight>)
```

TBrowseDB() has the same syntax as TBrowseNew(). It creates a TBrowse object with default code blocks to locate and position the data source within a database. In other words, the TBrowse object it creates is customized for browsing a (.dbf).

Add Columns

Next, add columns to the browse by creating TBColumn objects. This is a two-step process. First create the TBColumn object, then add the column to the TBrowse object:

```
oCol := TBColumnNew("Record #", {|| RECNO()})
oBrowse:addColumn(oCol)
```

This example, using TBrowseDB(), creates one object for every field in the database in the currently selected area:

```
//   StockBrowseNew()
//
//   Create a stock TBrowse object for the
//   current work area

FUNCTION StockBrowseNew(nTop,nLeft,nBottom,nRight)
   LOCAL oBrowse, n, oCol

   // Start with a new browse object
   oBrowse := TBrowseDB(nTop,nLeft,nBottom,nRight)

   // Add a column for each field
   FOR n := 1 TO FCOUNT()
      // Make a new column
      oCol := TBColumnNew(FIELDNAME(n), ;
         FIELDWBLOCK(FIELDNAME(n), SELECT()))

      // Add a column to the browse object
      oBrowse:addColumn(oCol)

   NEXT
   RETURN oBrowse
```

Main Loop

The main loop of the browse is very straightforward; stabilize the browse (explained below), get a keystroke, act on the keystroke, then repeat the process. If the keystroke happens to be an exit key (in this case, Esc), exit the loop and return. When you exit the loop, the reference to the TBrowse object, since it was held in a local variable, falls out of scope, and the system automatically reclaims the space used by the object.

The relevant code looks like this:

```
// Define exit key
oBrowse:setKey(K_ESC, {|| TBR_EXIT})

// Main loop of the browse object
lMore := .T.
DO WHILE lMore

    // Stabilize the browse
    oBrowse:forceStable()

    // Positioned at top or bottom of browse
    IF oBrowse:hitTop .OR. oBrowse:hitBottom
       TONE(125, 0)
    ENDIF

    // Get a new keystroke
    nKey := INKEY(0)

    // Act on the keystroke
    lMore := (oBrowse:applyKey(nKey) = TBR_EXIT)

ENDDO
```

Stabilization

To permit greater control, TBrowse does not update the screen immediately when messages are sent to it. Instead, it waits until the object is *stabilized* with TBrowse:stabilize(). When that message is received, the browse is stabilized incrementally, displaying one change at a time. For example, when the TBrowse object is created, it is not immediately displayed on the screen. The first time a TBrowse object receives a stabilize message, it displays the headers; then every time stabilize is received, it displays one record until everything is finally displayed. (When everything is displayed, TBrowse:stabilize() returns true (.T.)) Multiple browses are, therefore, displayed "in sync," and the redraw of the screen can be interrupted by a keystroke.

In this example, we do nothing during the stabilization process except maintain a simple loop that forces complete stabilization by calling TBrowse:forceStable(). Internally, this method calls TBrowse:stabilize() until it returns true (.T.):

```
// Stabilize the browse
oBrowse:forceStable()
```

Handling Keystrokes

The most efficient way to handle keystrokes is to use TBrowse:applyKey(). Many of the most commonly used keystrokes are predefined. For example, K_DOWN maps to TBrowse:down(), and K_PGDN maps to TBrowse:pageDown() (refer to the TBrowse class in the *Reference Guide* for a complete listing of the default keystroke mappings).

Using TBrowse:setKey(), however, you maintain total control over these mappings. You can add new ones, as illustrated in the code for the main loop which defines the K_ESC key in this manner, and you can modify or delete the values for predefined keystrokes to suit your own needs.

TBrowse:applyKey() and TBrowse:setKey() eliminate the need to process keystrokes via a procedure or function that is necessarily external to the TBrowse object, giving your browser a greater degree of encapsulation.

Optimization of the Browse

A minimum browse like the example is barely useful and you should consider the values of constructing a more robust 'production' browse. These additional operations optimize a browse for single or multi-user environments and include reduction in response time, creation of calculated fields, picture clauses and custom headers, explicit record pointer control, and use of the TBrowse:cargo and TBColumn:cargo instance variables.

Calculated Fields, Picture Clauses, and Custom Headers

This function adds a record number column to the passed browse and formats it with commas. It illustrates how to use TRANSFORM() in a data retrieval block to apply a picture to a column:

```
//   AddRecNo()
//
//   Insert a column at the left that shows
//   current record number formatted with commas

PROCEDURE AddRecNo(oBrowse)
   LOCAL oCol

   // Create the column object with custom header
   // Use TRANSFORM() to apply picture clause
   oCol := TBColumnNew("Record #", ;
      {|| TRANSFORM(RECNO(), "999,999,999")})

   // Insert it as the leftmost column
   oBrowse:insColumn(1, oCol)

   RETURN
```

Quicker Response Time

Often, you want to let the user page through a database or other table quickly. Previous examples have forced the user to wait until the screen redraws before they can move again. This significantly slows user operations.

Fortunately, you can easily reduce the response time by interrupting the stabilize loop. You do this, as illustrated in the next example, by calling TBrowse:stabilize() in the main loop instead of relying on TBrowse:forceStable(). You should, however, use care with this approach.

Certain TBrowse operations and instance variables are valid only with a stable browse, which you can test for using the TBrowse:stable exported instance variable, also illustrated below:

```
//   Interrupting the stabilize loop

// Define exit key
oBrowse:setKey(K_ESC, {|| TBR_EXIT})

lMore := .T.
DO WHILE lMore
   // Stabilize the browse until it's stable or
   // a key is pressed
   nKey := 0
   DO WHILE nKey = = 0 .AND. !oBrowse:stable
      oBrowse:stabilize()
      nKey := INKEY()
   ENDDO

   IF oBrowse:stable
      // positioned at top or bottom of browse
      // (these values only valid if stable)
      IF oBrowse:hitTop .OR. oBrowse:hitBottom
         TONE(125, 0)
      ENDIF

      // Get a new keystroke
      nKey := INKEY(0)
   ENDIF

   // Act on the keystroke
   lMore := (oBrowse:applyKey(nKey) = TBR_EXIT)

ENDDO
```

Multi-User Issues

TBrowse buffers all cells for the currently displayed row (whether the cells are visible or not). While this gives a good performance boost, it means that what is on the screen may not match what is in the database. For example, in a multi-user application another user may have deleted or otherwise modified one of the displayed records.

Rather than verifying that everything on the screen is current (which might be impossible and would certainly be slow), modify the browse to ensure that the current record is up-to-date using TBrowse:refreshCurrent(). Immediately afterwards, stabilize the browse. The following lines of code, inserted immediately before you get the keystroke in the previous example, do just this. Note that the placement of this code within the IF oBrowse:stable...ENDIF construct is important, because it ensures that the browse is stable; otherwise, the current row is undefined.

```
IF oBrowse:stable
    .
    .
    .
    // Ensure that the current record is showing
    // up-to-date data (e.g., on a network).
    oBrowse:refreshCurrent()
    oBrowse:forceStable()

    // Get a new keystroke
    nKey := INKEY(0)
ENDIF
```

Repositioning the Record Pointer

TBrowse is a general-purpose browser unaffected by the underlying data format. While this is useful in many cases, there are times when it means that a little more work is necessary on the programmer's part. One of these cases is when the record pointer is repositioned outside the control of TBrowse (when seeking a new value, for example).

When a TBrowse stabilizes, it tries to leave the same cell highlighted as before. That is, it tries to keep the highlight at the same position within the browse window unless it is explicitly moved by an up or down message. The TBrowse positions the data source correspondingly. If there are not enough rows left in the data source (that is, end of file is encountered while trying to adjust the database to match the window), the TBrowse moves the cursor upward, leaving it on the correct record, but with part of the window unfilled.

This behavior is appropriate for logical end of file, but a problem occurs when something outside of TBrowse moves the record pointer so close to the logical beginning of file that it is impossible to highlight the correct record while leaving the highlight at its previous position within the window. In this case, TBrowse leaves the highlight in the same position within the window, even though that position no longer corresponds to the same record as before. That is, it repositions the database as far as it will go, then leaves the highlight where it was. The result is the highlight on a different record than the one just edited.

You can correct this behavior by forcing a complete refresh (through TBrowse:refreshAll()), and a full stabilization. CA-Clipper then checks for repositioning of TBrowse to a different record than before. If so, CA-Clipper assumes the old record is somewhere above the current record, and a series of up messages are issued to the browse to move the highlight to the proper position. Determination of proper positioning of the highlight varies depending on what is being browsed. You must have some way of determining the exact row position. In the case of databases, use the record number or the index key, if it is unique.

This function is called from a keystroke handler that prompts for a key value to search for when you type Alt+S. The following example correctly repositions the highlight by using the record number:

```
//    SearchBrowse()
//
//    Search browse on key field with SEEK
//    and adjust highlight
PROCEDURE SearchBrowse(oBrowse)
   LOCAL cSaveScreen, xSearch, nRec, GetList := {}
   // Ensure browse is stable before getting info
   // about current record
   oBrowse:forceStable()
   // Get current key value and record number
   xSearch :=  &(INDEXKEY(0))
   nRec := RECNO()
   // Put prompt online surrounding browse
   cSaveScreen := SAVESCREEN(oBrowse:nBottom + 1, ;
      0, oBrowse:nBottom + 1, MAXCOL())
   SETPOS(oBrowse:nBottom + 1, oBrowse:nLeft + 2)
   DISPOUT("[Search For ")
   @ ROW(), COL() GET xSearch PICTURE "@K"
   DISPOUT("]")
   READ
   RESTSCREEN(oBrowse:nBottom + 1, 0, ;
      oBrowse:nBottom + 1, MAXCOL(), cSaveScreen)
   IF UPDATED() .AND. (LASTKEY() != K_ESC)
      SEEK xSearch
      IF !SET(_SET_SOFTSEEK) .AND. !FOUND()
         // Don't allow search to throw me to
         // end of file
         GOTO nRec
         ALERT(xSearch + " not found.", {"OK"})
      ELSE
         // Record pointer was moved.  Save new
         // location, refresh browse, then move
         // highlight to proper position
         // Ensure I have a valid record
         IF EOF()
            GO BOTTOM
         ENDIF
         nRec := RECNO()
         oBrowse:refreshAll()
         oBrowse:forceStable()
         DO WHILE RECNO() != nRec
            oBrowse:up()
            oBrowse:forceStable()
         ENDDO
      ENDIF
   ENDIF
   RETURN
```

Using TBrowse:cargo and TBColumn:cargo

Sometimes it is desirable to track more information about the browse or its columns than the browse itself tracks. To let you do this, CA-Clipper defines the *cargo* exported instance variable in all of its classes. It is called "cargo" because it is just "along for the ride." Cargo can hold data of any type.

If you have complete control over the objects, you can use cargo in any way you desire. This example makes the preprocessor code easier to understand:

```
// These #defines use the browse's cargo slot to
// hold the "append mode" flag for the browse.
// Having #defines for these makes it easy to
// change later.

#define TURN_ON_APPEND_MODE(b)    (b:cargo := .T.)
#define TURN_OFF_APPEND_MODE(b)   (b:cargo := .F.)
#define IS_APPEND_MODE(b)         (b:cargo)
```

Dictionaries

If, on the other hand, you are not in complete control of the objects (that is, you are writing general purpose routines for others to use), we strongly recommend that you use cargo as a dictionary, as implemented in the Dict.prg sample program (located in you CA-Clipper SOURCE\SAMPLE directory). This allows several programmers to use cargo without conflict, and without any change in the source code.

Dictionaries provide a way of associating a key with a value, as in a real dictionary, where the word you look up (the key) is associated with a particular definition (the value). Note that the entries in data dictionaries have no particular order.

This example uses a data dictionary to store the append mode. Because the dictionary is created and evaluated at runtime, any number of modules can use the dictionary without knowledge of how other modules are using it:

```
//   Using a dictionary
//
//   Use a dictionary to hold cargo information.
//   Requires that Dict.obj be linked in to work.

// Create new dictionary
oBrowse:cargo := DictNew()

// Turn on append mode
DictPut(oBrowse:cargo, "APPEND", .T.)

// Turn off append mode
DictPut(oBrowse:cargo, "APPEND", .F.)

// Check append mode
DictAt(oBrowse:cargo, "APPEND")
```

Browsing with Get

Browsing is more than read-only screens and pulldown menus. You can, for instance, create a popup edit screen or edit the fields in place. Using the Get system, you have unprecedented access to your data, but you are responsible for integrating the system into an application and controlling movement through the data source.

Though, at first, editing the browse fields "in-place" may seem an impossible task, it is actually quite easy. This discussion covers two different methods.

The first, and probably simplest method, employs cargo.

This example, a modification of the basic TBrowse object creation code, places the field name for the created objects into the TBColumn:cargo instance variable:

```
//   Inserting a column's field name into oCol:cargo

#define SET_FIELD_NAME(c, f)      (c:cargo := f)
#define GET_FIELD_NAME(c)         (c:cargo)

//   StockBrowseNew()
//
//   Create a stock TBrowse object for the
//   current work area

FUNCTION StockBrowseNew(nTop,nLeft,nBottom,nRight)
   LOCAL oBrowse, n, oCol

   // Start with a new browse object
   oBrowse := TBrowseDB(nTop,nLeft,nBottom,nRight)

   // Add a column for each field
   FOR n := 1 TO FCOUNT()
      // Make a new column
      oCol := TBColumnNew(FIELDNAME(n), ;
         FIELDWBLOCK(FIELDNAME(n), SELECT()))

      // Store the field name with the column
      SET_FIELD_NAME(oCol, FIELDNAME(n))

      // Add the column to the browse object
      oBrowse:addColumn(oCol)

   NEXT

   RETURN oBrowse
```

By placing this field name into TBColumn:cargo instance variable, the question of when to edit the field becomes a simple matter:

```
//   DoGet()
//
//   Get current cell using cargo macro method

PROCEDURE DoGet(oBrowse)
   LOCAL oCol, cField, nCursSave
   MEMVAR GetList

   // Ensure browse is stable
   oBrowse:forceStable()

   // Get current column
   oCol := oBrowse:getColumn(oBrowse:colPos)

   // Get field name from column
   cField := GET_FIELD_NAME(oCol)

   // Turn cursor on
   nCursSave := SETCURSOR(SC_NORMAL)

   @ ROW(), COL() GET &cField PICTURE oCol:picture ;
      WHEN oCol:preBlock VALID oCol:postBlock
   READ

   // Restore cursor
   SETCURSOR(nCursSave)
   RETURN
```

A more efficient way to accomplish the same thing takes advantage of the fact that the TBColumn:block and the Get class's Get:block can be compatible. Using this method, you can manipulate the Get directly:

```
//   DoGet()
//
//   Get current cell using GetNew() method

PROCEDURE DoGet(oBrowse)
   LOCAL oCol, oGet, nCursSave

   // Ensure browse is stable
   oBrowse:forceStable()

   // Get current column
   oCol := oBrowse:getColumn(oBrowse:colPos)

   // Create Get at current cell position
   oGet := GetNew(ROW(), COL(), oCol:block, ;
      oCol:heading, oCol:picture)

   // Turn cursor on
   nCursSave := SETCURSOR(SC_NORMAL)

   // And do READ
   READMODAL({oGet})

   // Restore cursor
   SETCURSOR(nCursSave)
   RETURN
```

Determining Whether the Record Has Moved

As discussed earlier under Repositioning the Record Pointer, indexes can cause problems in simple code examples. If you change the key field of a particular record, its relative position in the database might also change, requiring you to refresh the display.

It is relatively easy to determine if this is necessary by evaluating the index key before and after the read. If it has, refresh the screen.

Here is the previous example, with the proper code added:

```
//   DoGet()
//
//   Get current cell using GetNew() method and
//   update TBrowse correctly when key field is
//   changed

PROCEDURE DoGet(oBrowse)
   LOCAL oCol, oGet, nCursSave, xOldKey, nRec

   // Ensure browse is stable
   oBrowse:forceStable()

   // Indexes key expression, macro returns the
   // value of the expression for the current
   // record
   xOldKey := &(INDEXKEY(0))
   nRec := RECNO()

   // Get current column
   oCol := oBrowse:getColumn(oBrowse:colPos)

   // Create Get at current cell position
   oGet := GetNew(ROW(), COL(), oCol:block, ;
      oCol:heading, oCol:picture)

   // Turn cursor on
   nCursSave := SETCURSOR(SC_NORMAL)

   // And do READ
   READMODAL({oGet})

   // See if index key has changed
   IF !(&(INDEXKEY(0)) == xOldKey)
      // Force refresh, and reposition highlight if
      // necessary
      oBrowse:refreshAll()
      oBrowse:forceStable()
      DO WHILE RECNO() != nRec
         oBrowse:up()
         oBrowse:forceStable()
      ENDDO
   ENDIF

   // Restore cursor
   SETCURSOR(nCursSave)
   RETURN
```

Tip: To adapt this code to a multi-user environment, apply a record lock immediately after the first call to forceStable() and unlock after the call to READMODAL().

Adding Color

Though the previous examples have been for monochrome displays, the TBrowse class also gives control over color. The simplest way to add color to a TBrowse object is to SET COLOR TO the desired standard and enhanced colors. The headers, footers, and columns will be displayed in the standard color, and the highlight will be in the enhanced color. However, a more complex use of color requires an understanding of the relationships between the various exported instance variables that control color.

TBrowse:colorSpec

Just as CA-Clipper maintains a color table using SETCOLOR() to determine the colors of display objects, TBrowse maintains a color table using TBrowse:colorSpec for its own use.

When unspecified, TBrowse:colorSpec defaults to the value of SETCOLOR() at the time the TBrowse object is created.

A typical TBrowse:colorspec assignment looks like:

```
oBrowse:colorSpec := "N/W, N/BG, B/W, B/BG, R/W"
oBrowse:colorSpec += ", R/BG, G/W, G/BG"
```

TBrowse:colorSpec, unlike SETCOLOR(), implies no specific meaning by the order in which colors are listed. Instead, all of the other color-related instance variables and methods refer to colors as indexes into TBrowse:colorSpec. In the example above, color one is N/W, color two is N/BG, and so on.

In actual usage, colors are expressed as an array of up to four such indexes, called a *color array*. These are the normal, highlighted, header, and footer colors for the particular cells you are working with. The array {3, 4, 3, 3}, assuming the TBrowse:colorSpec listed above, represents a normal color of B/W, a highlighted color of B/BG, and header and footer color of B/W.

TBColumn:defColor

TBrowse lets you set column colors individually through the TBColumn:defColor instance variable. The first, or normal, color in the TBColumn:defColor color array determines the color of the data values; the highlighted color determines the color of the highlight when it falls on this column; the third and fourth colors determine the header and footer colors. TBColumn:defColor defaults to {1, 2, 1, 1}.

This code displays every data type in a different color:

```
//   FancyColors()
//
//   Display each data type in a different color

PROCEDURE FancyColors(oBrowse)
   LOCAL nCol, oCol, xValue

   // Set up a list of colors for the browse to use
   oBrowse:colorSpec := "N/W, N/BG, B/W, B/BG, R/W"
   oBrowse:colorSpec += ", R/BG, G/W, G/BG"

   // Loop through the columns, choosing a
   // different color for each data type
   FOR nCol := 1 TO oBrowse:colCount

      // Get (a reference to) the column
      oCol := oBrowse:getColumn(nCol)

      // Get a sample of the underlying data by
      // evaluating the code block
      xValue := EVAL(oCol:block)

      DO CASE
      CASE VALTYPE(xValue) == "C"
         // Characters in black
         oCol:defColor := {1, 2, 1, 1}

      CASE VALTYPE(xValue) == "N"
         // Numbers in blue
         oCol:defColor := {3, 4, 3, 3}

      CASE VALTYPE(xValue) == "L"
         // Logicals in red
         oCol:defColor := {5, 6, 5, 5}

      OTHERWISE
         // Dates and others in green
         oCol:defColor := {7, 8, 7, 7}
      ENDCASE
   NEXT
   RETURN
```

TBColumn:colorBlock

TBrowse lets you highlight the color of individual cells based on the value retrieved by TBColumn:block (the data retrieval block) using TBColumn:colorBlock. This code block is passed the result of the data retrieval block and returns a color array. TBColumn:colorBlock overrides the TBColumn:defColor for the column. This is valuable in a number of situations, a few of which are illustrated in the examples to follow:

In this example, negative numbers are displayed in red:

```
//   FancyColors()
//
//   Display negative numbers in a different color

PROCEDURE FancyColors(oBrowse)
   LOCAL nCol, oCol, xValue

   // Set up a list of colors for the browse to use
   oBrowse:colorSpec := "N/W, N/BG, R/W, W+/R"

   // Loop through the columns, set up default, and
   // force columns to display negative #s in red
   FOR nCol := 1 TO oBrowse:colCount

      // Get (a reference to) the column
      oCol := oBrowse:getColumn(nCol)

      // Get a sample of the underlying data by
      // evaluating the code block
      xValue := EVAL(oCol:block)

      // Default color
      oCol:defColor := {1, 2, 1, 1}

      IF VALTYPE(xValue) == "N"
         // Negative numbers in red
         oCol:colorBlock := {|n|  IIF(n < 0, ;
            {3, 4}, {1, 2})}
      ENDIF
   NEXT
   RETURN
```

This example displays cells in random colors:

```
//   FancyColors()
//
//   Display cells in random colors

PROCEDURE FancyColors(oBrowse)
   LOCAL nCol, oCol

   // Set up a list of colors for the browse to use
   oBrowse:colorSpec := "B/W, W/B, R/W, W/R, " + ;
      "G/W, W/G, GR/W, W/GR, W/BG, BG/W, N/W, W/N"

   // Loop through the columns, setting up lines to
   // display in alternating colors
   FOR nCol := 1 TO oBrowse:colCount

      // Get (a reference to) the column
      oCol := oBrowse:getColumn(nCol)

      // Default color
      oCol:defColor := {1, 2, 1, 1}

      // Ignore the value of the cell and choose
      // color for column based on the time at
      // which the block is executed
      oCol:colorBlock := {|x| RandColor(6)}
   NEXT
   RETURN

//   RandColor()
//
//   Return color based on time

FUNCTION RandColor(nSets)
   LOCAL nSelector

   // nSets is the number of color arrays available
   nSelector := (SECONDS() * 10) % nSets

   RETURN {(nSelector * 2) + 1, ;
      (nSelector * 2) + 2}
```

Controlling the Highlight

By default, TBrowse turns the highlight on and off. TBrowse gives you complete control over when and where the highlight appears through TBrowse:autoLite, TBrowse:hilite(), and TBrowse:deHilite(). If TBrowse:autoLite is true (.T.), the default setting, TBrowse automatically turns the highlight on and off.

In the example under Repositioning the Record Pointer, you can see the highlight flicker as the necessary adjustment takes place. You can correct this by changing the searching function to turn off automatic highlighting before and turn it on after the movement is done. The following code fragment illustrates:

```
IF !SET(_SET_SOFTSEEK) .AND. !FOUND()
    .
    .
    .
ELSE
    .
    .
    .
    nRec := RECNO()
    oBrowse:refreshAll()

    // Turn off highlighting while movement is
    // taking place
    oBrowse:autoLite := .F.
    oBrowse:deHilite()

    oBrowse:forceStable()

    DO WHILE RECNO() != nRec
        oBrowse:up()
        oBrowse:forceStable()
    ENDDO

    // Turn it back on now that we are done
    oBrowse:autoLite := .T.
ENDIF
```

TBrowse:colorRect()

TBrowse also includes a method, TBrowse:colorRect(), for coloring any rectangular region. This method takes as parameters the four corners of the box (top, left, bottom, and right) in an array, and a color array. The coordinates of the box are relative to the browse window, so that 1, 1 is the first cell of the first row currently visible. The box "1, 1, oBrowse:rowCount, oBrowse:colCount" refers to all currently visible rows, though the column numbers could refer to columns that are not currently visible.

Colors set with TBrowse:colorRect() remain in effect on a particular row until that row's data is refreshed, either because the row scrolled off the screen or due to one of the refresh methods. TBrowse:colorRect() overrides TBColumn:colorBlock and TBColumn:defColor.

You can use TBrowse:colorRect() to highlight an entire row rather than an individual column. This function uses the passed color array to highlight a row. If you add a call to the method immediately after the stabilize portion of the main loop (in an area where the browse is guaranteed stable), the row maintains that color until refreshed, immediately before applying the key to the browse:

```
//   HiliteRow()
//
//   Highlight current row with passed color array

PROCEDURE HiliteRow(oBrowse, aColors)
   LOCAL nCurRow

   nCurRow := oBrowse:rowPos

   // Color row
   oBrowse:colorRect({nCurRow, 1, nCurRow, ;
      oBrowse:colCount}, aColors)

   // And make it happen
   oBrowse:forceStable()

   RETURN
```

Controlling the Scope

TBrowse lets you restrict a browse to a specific set of records without the speed penalty inflicted by a filter or the need to create a temporary database. In CA-Clipper, you restrict the browse by defining the limits of cursor movement within the data source using the goTopBlock, goBottomBlock, and skipBlock instance variables of the TBrowse class. Restricting movement this way is, effectively, the same as defining a filter or redefining the scope of your search, only much, much faster.

To define movements within your redefined scope to TBrowse, you must specify how to move directly to the top, how to move directly to the bottom, and how to move forward and backward through records.

TBrowse:goTopBlock

TBrowse uses TBrowse:goTopBlock to go to the top of the data set being browsed. TBrowseDB() creates a default TBrowse:goTopBlock equivalent to:

```
oBrowse:goTopBlock := {|| DBGOTOP()}
```

TBrowse:goBottomBlock

TBrowse uses TBrowse:goBottomBlock to go to the bottom of the data set being browsed. TBrowseDB() creates a default TBrowse:goBottomBlock equivalent to:

```
oBrowse:goBottomBlock := {|| DBGOBOTTOM()}
```

TBrowse:skipBlock

TBrowse uses TBrowse:skipBlock to move forward and backward the specified number of records through the data set. TBrowse passes a positive number to TBrowse:skipBlock to move forward, and a negative number to move backward.

TBrowse:skipBlock returns only the number of rows it actually moved. Thus, if the current row is two rows from the end of the data set, and TBrowse:skipBlock is passed 4, it returns 2, indicating it could only move forward two rows. TBrowseDB() creates a default TBrowse:skipBlock equivalent to:

```
//   TBrowse:skipBlock created by TBrowseDB()
oBrowse:skipBlock := {|nMove| Skipper(nMove)}

//   Skipper()
//
//   Handle record movement requests from the
//   TBrowse object

STATIC FUNCTION Skipper(nMove)
   LOCAL nMoved

   nMoved := 0

   IF nMove = = 0 .OR. LASTREC() = = 0
      // Skip 0 (significant on a network)
      SKIP 0

   ELSEIF nMove > 0 .AND. RECNO() != LASTREC() + 1
      // Skip forward
      DO WHILE nMoved < nMove
         SKIP 1
         IF EOF()
            SKIP -1
            EXIT
         ENDIF

         nMoved++
      ENDDO

   ELSEIF nMove < 0
      // Skip backward
      DO WHILE nMoved > nMove
         SKIP -1
         IF BOF()
            EXIT
         ENDIF

         nMoved--
      ENDDO
   ENDIF

   RETURN nMoved
```

Viewing a Specific Key Value

You can look at all records associated with a particular key value by defining the movements within the data set, then forcing the TBrowse:skipBlock to remain within the key value. Define movement to the top of the data set as seeking the particular key and movement to the bottom of the data set as seeking the last item matching the current key.

First, write a function to create a block for TBrowse:goTopBlock. Note that this function creates a code block that can only refer to *xKey*, relieving us of the burden of ensuring that the key value remains available:

```
//    FirstKeyBlock()
//
//    Create TBrowse:goTopBlock to seek the first
//    occurrence of the passed key

FUNCTION FirstKeyBlock(xKey)
   RETURN {|| DBSEEK(xKey)}
```

Next, write a function to create a TBrowse:goBottomBlock. This is more complex because the browse should move almost instantaneously to the last matching key. Do this by seeking the next key value, then skip back one record. Calculate the next key value using the TBNextKey() function.

In the following example, LastKeyBlock() returns a block that implements TBrowse:goBottomBlock:

```
//    LastKeyBlock()
//
//    Create TBrowse:goBottomBlock to seek the last
//    occurrence of the passed key

FUNCTION LastKeyBlock(xKey)
    RETURN {|| DBSEEK(TBNextKey(xKey), .T.), ;
        DBSKIP(-1)}

//     TBNextKey()
//
//     Return value of next key in database

//     WARNING! Does not work on logical values or
//     floating point numbers

FUNCTION TBNextKey(xValue)
    LOCAL cType := VALTYPE(xValue), xNext

    DO CASE
    CASE cType == "C"
        // Find next value by incrementing the ASCII
        // value of the last character in the string

        // NOTE: Assume last char is not 255 for
        // simplicity
        xValue := STUFF(xValue, LEN(xValue), 1, ;
            CHR(ASC(RIGHT(xValue, 1)) + 1))

    CASE cType == "N"
        // Since we can only handle integers,
        // increment value by one
        xValue++

    CASE cType == "D"
        // Increment date by one day
        xValue++

    OTHERWISE
        // Couldn't handle it

    ENDCASE

    RETURN xNext
```

Finally, write a function that returns the necessary TBrowse:skipBlock. This function is like the default TBrowse:skipBlock described earlier, except that it checks for moving out of range as well as for beginning and end of file.

One typical use of this function is zooming to child records from a browse of the parent database:

```
//    KeySkipBlock()
//
//    Create TBrowse:skipBlock based on key value

FUNCTION KeySkipBlock(xKey)
    RETURN {|nMove| KeySkipper(nMove, xKey)}

//    KeySkipper()
//
//    Handle record movement requests from the
//    TBrowse object while staying within one key

FUNCTION KeySkipper(nMove, xKey)
    LOCAL nMoved

    nMoved := 0

    IF nMove = = 0 .OR. LASTREC() = = 0
        // Skip 0 (significant on a network)
        SKIP 0

    ELSEIF nMove > 0 .AND. RECNO() != LASTREC() + 1
        // Skip forward
        DO WHILE nMoved <= nMove .AND. !EOF() .AND. ;
                &(INDEXKEY(0)) == xKey
            SKIP 1
            nMoved++
        ENDDO

        // All above cases lead to skipping one extra
        SKIP -1
        nMoved--

    ELSEIF nMove < 0
        // Skip backward
        DO WHILE nMoved >= nMove .AND. !BOF() .AND. ;
                &(INDEXKEY(0)) == xKey
            SKIP -1
            nMoved--
        ENDDO

        // No phantom BOF() record, else above loop
        // leads to skipping back one extra record
        IF !BOF()
            SKIP 1
        ENDIF

        nMoved++
    ENDIF

    RETURN nMoved
```

This example ties the various code blocks and functions defined so far in this section together, giving you a scoped browse based on a key value that you pass as an argument:

```
//   BrowseWhile()
//   Browse database while the key field matches
//   xKey

PROCEDURE BrowseWhile(nTop, nLeft, nBottom, ;
   nRight, xKey)
   LOCAL oBrowse          // The TBrowse object
   LOCAL nKey             // Keystroke
   LOCAL lMore            // Loop control
   LOCAL nCursSave        // Save state

   // Create another generic browse
   oBrowse := StockBrowseNew(nTop + 1, nLeft + 1, ;
      nBottom - 1, nRight - 1)

   // Change the movement methods
   oBrowse:goTopBlock := FirstKeyBlock(xKey)
   oBrowse:goBottomBlock := LastKeyBlock(xKey)
   oBrowse:skipBlock := KeySkipBlock(xKey)

   // Draw a box around the browse
   @ nTop, nLeft CLEAR TO nBottom, nRight
   @ nTop, nLeft TO nBottom, nRight

   // Turn cursor off and save shape while browsing
   nCursSave := SETCURSOR(SC_NONE)

   // Define exit key
   oBrowse:setKey(K_ESC, {|| TBR_EXIT})

   // Main loop
   lMore := .T.
   DO WHILE lMore

      // Stabilize the browse
      oBrowse:forceStable()

      // Positioned at top or bottom of browse
      IF oBrowse:hitTop .OR. oBrowse:hitBottom
         TONE(125, 0)
      ENDIF

      // Get a new keystroke
      nKey := INKEY(0)

      // Act on the keystroke
      lMore := (oBrowse:applyKey(nKey) = TBR_EXIT)

   ENDDO

   SETCURSOR(nCursSave)

   RETURN
```

Browsing Search Results

Before the implementation of TBrowse, when browsing a random collection of records (such as the results of a user's query), it was necessary to create temporary databases or deal with the slow display times of SET FILTER. Because TBrowse gives you complete control over movement through the database, you can execute fast browses of any arbitrary subset of the database.

To accomplish this, run the search through the database once, creating an array of the record numbers that satisfy the query. Use this array to control the browse movement. While this imposes a limit of 4,096 matching records, this is usually sufficient.

The fastest way to create the control array is to use the DBEVAL() function. This function receives a code block equivalent to the FOR condition and returns a control array:

```
//    ArrayFor()
//
//    Create array of record numbers that match
//    passed condition

FUNCTION ArrayFor(bFor)
   LOCAL aRecs := {}

   DBEVAL({|| AADD(aRecs, RECNO())}, bFor)

   RETURN aRecs
```

Using *aRecs*, the array returned by ArrayFor(), it is easy to specify TBrowse:goTopBlock and TBrowse:goBottomBlock (assuming *nCurRow* is the current row of the browse, as in LOCAL nCurRow := oBrowse:rowPos):

```
oBrowse:goTopBlock := {|| DBGOTO( ;
   aRecs[nCurRow := 1])}
oBrowse:goBottomBlock := {|| DBGOTO( ;
   aRecs[nCurRow := LEN(aRecs)])}
```

TBrowse:skipBlock is also defined in terms of the array returned by ArrayFor():

```
oBrowse:skipBlock := {|nMove| SkipFor( ;
   nMove, aRecs, @nCurRow)}
```

The function that TBrowse:skipBlock calls to move through the database uses the end and beginning of the array, rather than the end and beginning of the database:

```
//   SkipFor()
//
//   Move through array-controlled database

FUNCTION SkipFor(nMove, aRecs, nCurRow)
   LOCAL nMoved

   nMoved := 0

   // Put move into an acceptable range
   IF nCurRow + nMove < 1
      // would skip past top...
      nMove :=  -nCurRow + 1

   ELSEIF nCurRow + nMove > LEN(aRecs)
      // Would skip past bottom...
      nMove := LEN(aRecs) - nCurRow

   ENDIF
   nCurRow += nMove
   nMoved := nMove

   DBGOTO(aRecs[nCurRow])

   RETURN nMoved
```

Chapter 7
Introduction to the Get System

In This Chapter

The CA-Clipper Get system is an integral interface tool consisting of commands, objects (and their related data and methods), environments, and behaviors that implement data entry and retrieval. It gives the developer direct and specific control over the interaction between the user and data, while making the retrieval, display, manipulation, and storage of data easy and flexible. This chapter lays the groundwork required to take advantage of the Get system, after which, the reader should understand and be able to extend the system. The chapter assumes knowledge of the @...GET and READ commands, so the focus lies instead on more advanced uses of the system.

This chapter starts with a discussion of the Get class, continuing with the layers of the Get system. It concludes with a discussion of possible modifications to the behavior of the Get system. The chapter contains many examples and recommendations.

This chapter discusses the following topics:

- The Get class
- The Get system structure
- The Read layer
- Extending the Read layer
- Creating a new Read layer
- The Reader layer
- Creating a new Reader layer
- The Get function layer
- Creating a new Get function layer

Note: This chapter assumes that you are familiar with the concepts and terminology associated with CA-Clipper's predefined classes as discussed in the "Basic Concepts" chapter of this guide.

The Get Class

The Get class implements the formatting behavior associated with CA-Clipper's Gets. Like all other classes, the Get class is defined as a set of methods and exported instance variables, as well as a function, GetNew(), used to create new Get objects.

This section discusses important behavior and uses of Get:block and Get:cargo and a few other features of the Get class in preparation for deeper discussions of the Get system itself. Throughout the chapter, however, other instance variables and methods not discussed in this section will be mentioned and discussed. If you need additional information, refer to the Get class in the "Language Reference" chapter of the *Reference Guide*.

Creating Get Objects

GetNew() creates a new Get object. You supply parameters to GetNew() that specify the row and column position of the Get, as well as a block that defines how the value of the Get should be retrieved and stored, the name to associate with the Get (usually the name of a variable or field being edited by the Get), a picture clause, and a color setting. These are stored in the instance variables Get:row, Get:col, Get:block, Get:name, Get:picture, and Get:colorSpec, respectively. Note that not all parameters are required, as illustrated in this example:

```
LOCAL oGet
oGet := GetNew(1, 2, FIELDBLOCK("Zip"), "Zip")
```

Get:block

Get:block holds a code block that stores the value passed, or retrieves the value associated with the Get if no parameter is passed. If the Get refers to a simple variable (*cVar* in this example), the code block is straightforward:

```
oGet := GetNew(1, 1, ;
    {|cNew| IIF(PCOUNT() = = 0, cVar, ;
    cVar := cNew)}, "cVar")
```

The block can also refer to a function call. For example, to create a Get object to modify the _SET_WRAP setting, one would do the following:

```
cName := "_SET_WRAP"
oGet := GetNew(1, 1, ;
    {|lNew| IIF(PCOUNT() = = 0, SET(_SET_WRAP), ;
    SET(_SET_WRAP, lNew)}, cName)
```

Due to the nature of code blocks, arrays create a particular challenge. An array with a constant index works reliably:

```
cName := "aVar[1]"
oGet := GetNew(1, 1, ;
    {|cNew| IIF(PCOUNT() = = 0, aVar[1], ;
    aVar[1] := cNew)}, cName)
```

But when the index is a variable (for example, *aVar*[*x*]), it does not work because the code block then refers to a variable, *x*, that can change before the block is executed. For example, at the time the block was created, *x* may have a value of 3, making the block refer to *aVar*[3]. If *x* is later changed to 1, the code block refers to *aVar*[1]:

```
// This doesn't work
cName := "aVar[" + LTRIM(STR(x)) + "]"
oGet := GetNew(1, 1, {|cNew| IIF(PCOUNT() = = 0, ;
    aVar[x], aVar[x] := cNew)}, cName)
```

To solve this problem, the Get class makes special provisions for arrays. Normally, a Get:block for an array simply returns the array reference. However, since the code block must access the actual array element rather than the reference to the entire array, and Get:subscript is not assignable, you must create the code block *within a function*:

```
cName := "aVar[" + LTRIM(STR(x)) + "]"
oGet := GetNew(1, 1, AGetBlock(aVar, x), cName)
...
STATIC FUNCTION AGetBlock(aArray, x)
   RETURN {|xVal| IIF(PCOUNT() = = 0, aArray[x], ;
      aArray[x] := xVal)}
```

This effectively hard-codes the array subscript, because the local variable *x* has gone out of scope and can no longer be changed. For more information, refer to the two sections, Arrays and Code Blocks, in the "Basic Concepts" chapter of this guide.

Setting and Retrieving Values

Note that while it seems logical to evaluate the Get:block to set and retrieve the value of a Get object, the way arrays are handled makes this impractical. Use the Get:varGet() and Get:varPut() methods instead.

```
// Retrieve the current value of the Get
nVal := oGet:varGet()

//...and assign back one greater
oGet:varPut(nVal + 1)
```

Get:cargo

Sometimes it is desirable to track more information about the Get than the Get object itself tracks. To let you do this, CA-Clipper defines the *cargo* exported instance variable in all of its classes. It is called "cargo" because it is just "along for the ride." Cargo can hold data of any type.

If you have complete control over the objects, you can use cargo in any way you desire. This example makes the preprocessor code easier to understand:

```
// These #defines use the GET's "cargo" slot to hold
// the message displayed at the bottom of the
// screen.  Having #defines for these just makes it
// easy to change later.

#define SET_MESSAGE(g, msg) (g:cargo := msg)
#define GET_MESSAGE(g)      (g:cargo)
```

Dictionaries

If, on the other hand, you are not in complete control of the objects (that is, you are writing general purpose routines for others to use), we strongly recommend that you use cargo as a dictionary, as implemented in the Dict.prg sample program (located in your CA-Clipper SOURCE\SAMPLE directory). This allows several programmers to use cargo without conflict, and without any change in the source code.

Dictionaries provide a way of associating a key with a value, as in a real dictionary, where the word you look up (the key) is associated with a particular definition (the value). Note that the entries in data dictionaries have no particular order.

This example uses a dictionary to store help text. Because the dictionary is created and evaluated at runtime, any number of modules can use the dictionary without knowledge of how other modules are using it:

```
// Use a dictionary to hold cargo information.
// Requires linking the sample program Dict.obj.
//
// Create new dictionary
oGet:cargo := DictNew()
// Assign message to Get object
DictPut(oGet:cargo, "MESSAGE", "Help Text")

// Print message from Get object at the bottom of
// the screen
@ MAXROW(), 0 SAY DictAt(oGet:cargo, "MESSAGE")
```

Get Class Protocol

Once the object is created and the appropriate instance variables are set, it is ready for use. You must first display the Get on the screen using either of two methods, Get:display() or Get:colorDisp(), to accomplish this. Get:display() simply displays the Get in its current color (set through Get:colorSpec), while Get:colorDisp() sets the Get:colorSpec then displays the Get. The Get is initially displayed in its unselected color.

When you perform actual data entry into the Get, you must first prepare the Get to accept characters. Do this using Get:setFocus(). This causes the Get object to create and initialize its internal state information, including the exported instance variables: Get:buffer, Get:pos, Get:decPos, Get:clear, and Get:original. The contents of the editing buffer are then displayed in the GET's selected color.

Keystrokes are intercepted externally to the object. Data is put into the Get using the Get:insert() and/or the Get:overStrike() methods, and movement within the Get is accomplished using any of the movement methods.

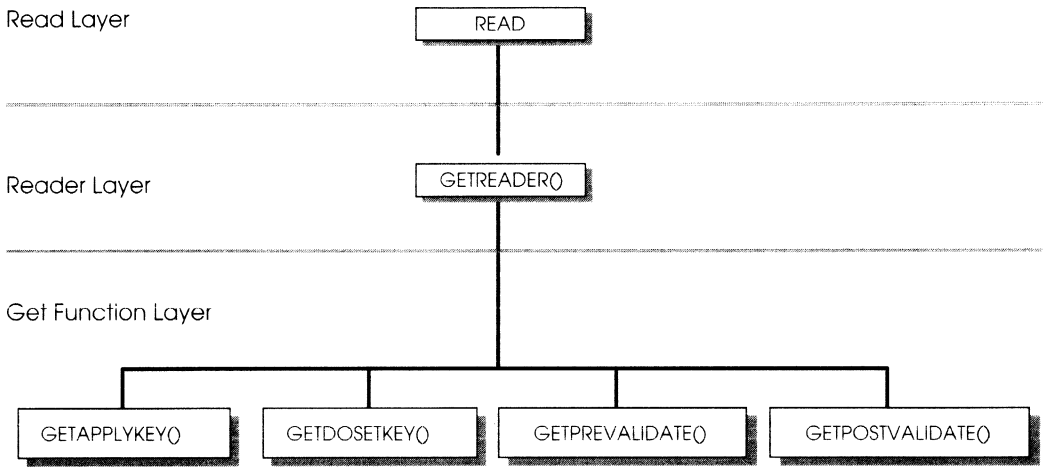
When editing is completed, the new value in the buffer is either accepted by Get:assign() or discarded by Get:undo(). The killFocus message removes input focus from the Get. The Get is then redisplayed in its unselected color and internal state information is discarded.

This description supplies all the information necessary to follow the examples given below. Understand that the Get class is simply a tool used by the Get system to accomplish its goals—there is nothing about the Get class that ties the two together. This will become clearer in the following discussion of the Get system.

The Get System

The Get system is an effective way to accept data input from the user. It is flexible and open enough to allow modifications that greatly enhance the power of the system.

The Get system is composed of several layers. Moving down the chart, the layers increase in both complexity of use and in programmer control. As with all areas of programming, *the highest possible layer should always be used.* This ensures maximum programmer productivity and system reliability.



The Read Layer

The top layer, the Read layer, is composed of the familiar @...GET and READ commands. This is where the bulk of CA-Clipper programming is done.

The Reader Layer

The next layer, the Reader layer, controls the behavior of one Get at a time. This layer is implemented by default through the function GETREADER(), but can be extended through additional custom *readers*.

The Get Function Layer

The bottom layer, the Get function layer, separates the behavior of a single Get down into four distinct actions: prevalidating the Get, handling SET KEYs, applying keystrokes to the Get, and postvalidating the Get.

The GETPREVALIDATE(), GETDOSETKEY(), GETAPPLYKEY(), and GETPOSTVALIDATE() functions perform the default implementation, respectively. This layer can also be extended through custom Get functions.

The Read Layer

One way to see the relationship between layers is to analyze the implementation of the Read layer. This system is written almost entirely in CA-Clipper. The file Getsys.prg (located in your CA-Clipper SOURCE\SYS directory) contains the source code for the system.

A quick look at this source, or at the commands in STD.CH, shows that a screen of Gets is represented internally as an array of Get objects. These Gets are stored in the public array *GetList* by default. This array is always present, just as though the first line of all your programs is:

```
PUBLIC GetList := {}
```

As you can see if you look at its #command definition in STD.CH, @...GET adds a Get object to the GetList array, assigns certain instance variables based on various keywords, then displays the newly created Get object with code similar to the following:

```
AADD(GetList, GetNew(...))
ATAIL(GetList):caption := ...
ATAIL(GetList):capRow := ...
ATAIL(GetList):capCol := ...
ATAIL(GetList):display()   // Display newly created
                           // Get object
```

READ is implemented through the function READMODAL(). As you can see by studying the source code in Getsys.prg, READMODAL() is passed the GetList array to process and returns a logical value that indicates whether any of the Gets were updated.

READMODAL() first initializes all of its variables and implements some Summer '87 compatibility behaviors. It then enters a loop to process all of the Gets. Within this loop, it notifies the system that this is the "active" Get object (making it available through the GETACTIVE() function) and passes control to the Reader layer using the default reader, GETREADER() or a custom reader as defined through the Get:reader code block.

Extending the Read Layer

With understanding of the basic Read layer, it is possible to make significant enhancements to the system without modifying it.

Nested Reads

The ability to nest reads, a most desired enhancement, is easy with CA-Clipper. Because the READ command simply passes an array named GetList to READMODAL(), you can use a different variable, GetList, in other functions. If you call a function from a VALID or WHEN clause or from a SET KEY procedure, and then define a new array, GetList, local to that function, the @...GETS and READs there will refer to the local GetList.

This GetList falls out of scope when the function ends, returning control to the previous READ and its GetList:

```
// Creating nested reads

PROCEDURE Main()
   LOCAL cName := SPACE(10)
   @ 10, 10 GET cName VALID SubForm(cName)
   READ
   RETURN

FUNCTION SubForm(cLookup)
   LOCAL GetList := {}          // Create new GetList
   USE Sales INDEX Salesman NEW
   SEEK cLookup
   IF FOUND()
      // Add Get objects to new GetList
      @ 15, 10 GET Salesman
      @ 16, 10 GET Amount
      // READ from new GetList
      READ
   ENDIF
   CLOSE Sales
   // Release new GetList
   RETURN .T.
```

Using GETACTIVE()

Much of the power of the new Get system lies in its direct manipulation of a Get object after its retrieval with the GETACTIVE() function. This function lets you not only retrieve and change the current value of the Get, but access all the information stored in it.

Changing Gets Through SET KEYs

A perfect example of direct object manipulation is changing the value of a Get through a SET KEY. While you can do this using the KEYBOARD command to stuff the keyboard buffer or by applying the macro operator to the result of READVAR() (if the Get is being done on a field, private, or public variable), it is much cleaner to just send messages to the current Get object.

This example examines the current content of the Get. If the Get is of character data type, it converts the contents to mixed case using the sample function Proper() when the user types Ctrl+K. Proper() is defined in String.prg, which is located in your CA-Clipper SOURCE\SAMPLE directory:

```
//  Converting character input to mixed case
//  Must be linked with String.obj to work

#include "Inkey.ch"
.
. .<statements>
.
SET KEY K_CTRL_K TO GetCapFirst
.
. <statements>
.
@ 1, 1 GET cTest
READ

SET KEY K_CTRL_K TO
RETURN


//  GetCapFirst()
//  Capitalize the first letter of every word
//

STATIC PROCEDURE GetCapFirst()
   LOCAL oGet, cString

   // Grab the current Get object
   oGet := GETACTIVE()

   // Get the current value of the Get
   cString := oGet:varGet()

   IF VALTYPE(cString) == "C"
      // Convert and put the value back
      oGet:varPut(Proper(cString))
   ENDIF

   RETURN
```

Improving Help Routines

You can also improve help systems by storing the help text (or an index to the help text) in Get:cargo. Notice how dictionaries are used to reduce potential conflicts between this and any other modules that implement enhancements to the Get system:

```
// Storing Help Text in Get:cargo
// Dict.obj is necessary for the following example.

PROCEDURE ReadTest()
   LOCAL cVar1
   LOCAL GetList := {}

   cVar1 := SPACE(30)

   CLS
   @ 3, 0 SAY PADC("GETs w/ Help Text in " + ;
      "Get:cargo", MAXCOL())
   @ 4, 0 SAY PADC("(<F1> for help)", MAXCOL())

   // Put the help text in the cargo
   // dictionary manually
   @ 7, 0 SAY "Variable 1" GET cVar1
   IF ATAIL(GetList):cargo = = NIL
      ATAIL(GetList):cargo := DictNew()
   ENDIF

   DictPut(ATAIL(GetList):cargo, "Help Text", ;
      "Help for variable one.")
   READ

   RETURN

//  Help()
//  The system automatically assigns F1 to this
//  procedure, returning help from wherever you are

PROCEDURE Help(cProc, nLine, cVar)
   LOCAL oGet
   LOCAL cMsg := "No help available for this item"

   // Retrieve help text from Get:cargo dictionary
   // In a more robust version of this system,
   // some sort of index into a help file would be
   // stored rather than the help text itself. This
   // would allow easier editing of the help text
   // and permit runtime modification of it.
   IF (oGet := GETACTIVE()) != NIL
      cMsg := DictAt(oGet:cargo, "Help Text")
   ENDIF
   DispMsg(cMsg)

   RETURN
```

The first concern many people have when seeing this is the difficulty of adding the help text. Fortunately, CA-Clipper also provides a graceful way to do this. By defining a command that adds a HELP clause to the standard @...GET command, the interface is made much cleaner:

```
// Adding a help clause to the @...GET Command
// Helpget.ch

command @ <row>, <col> GET <var> [<clauses, ...>] ;
   HELP <help> [<moreClauses, ...>] ;

   => @ <row>, <col> GET <var> [<clauses>] ;
      [<moreClauses>] ;
      ;IF ATAIL(GetList):cargo = = NIL ;
      ;ATAIL(GetList):cargo := DictNew() ;
      ;ENDIF ;
;DictPut(ATAIL(GetList):cargo, "Help Text", <help>)
```

This command, like the earlier code in ReadTest(), checks to see if Get:cargo is NIL before making it a dictionary because this command's definition lets you use it with other extensions to the GET command. It is very plausible that some other extension has already made Get:cargo a dictionary. If so, the command just inserts the new entry into the existing dictionary. If Get:cargo is not NIL, and it is not a dictionary, the command will crash at runtime.

To use this command, include the header file and simply add the HELP clause to any existing Gets, as shown below:

```
//  Using the Help Clause

#include "HelpGet.ch"

PROCEDURE ReadTest()
   LOCAL cVar1
   LOCAL GetList := {}

   cVar1 := SPACE(30)

   CLS
   @ 3, 0 SAY PADC("GETs w/ Help Text in " + ;
      "Get:cargo", MAXCOL())
   @ 4, 0 SAY PADC("(<F1> for help)", MAXCOL())

   // Put the help text using new command
   @ 7, 0 SAY "Variable 1" GET cVar1 PICTURE "@!" ;
      HELP "Help for variable one."

   READ

   RETURN
```

Using Code Blocks for WHEN/VALID

Because the @...GET command accepts a code block as a legal parameter for either clause, it is easier to manipulate a GET from within a WHEN or VALID clause than from a SET KEY.

Dynamic Pictures

The following example changes the PICTURE clause of a Get on-the-fly in the WHEN clause:

```
// Dynamically changing the PICTURE clause of a Get

PROCEDURE Main()
   LOCAL GetList := {}
   LOCAL cName := SPACE(20)
   LOCAL cIDType := "I"
   LOCAL cSSNo := SPACE(9)

   SET CONFIRM ON
   SET SCOREBOARD OFF
```

```
    CLS
    @ 3, 0 SAY PADC("Member Information")

    @ 5, 0 SAY "Name                    " GET cName
    @ 6, 0 SAY "C)ompany or I)ndividual" ;
       GET cIDType PICTURE "!" VALID cIDType $ "CI"

    // Change picture based on whether this is for a
    // person or company
    @ 7, 0 SAY "SS# or Tax ID#         " GET cSSNo ;
       PICTURE StartPic(cIDType) ;
       WHEN {|oGet| ChangePic(oGet, cIDType)}
    READ

    RETURN


//  StartPic()
//  Determine what the initial value for the picture
//  should be.
//

STATIC FUNCTION StartPic(cIDType)

    IF cIDType == "C"
       RETURN "@R 99-99999-99"
    ELSE
       // Default value
       RETURN "@R 999-99-9999"
    ENDIF

//  ChangePic()
//  Change picture on-the-fly from SS# to Tax ID or
//  vice versa (to be called from WHEN clause).

STATIC FUNCTION ChangePic(oGet, cIDType)
    IF cIDType == "C"
       oGet:picture := "@R 99-99999-99"
    ELSE

       // Default value
       oGet:picture := "@R 999-99-9999"
    ENDIF
    RETURN .T.
```

Processing the Entire GetList

While modifying a single Get is useful, there are times when it is desirable to modify all of the Gets. One might want to change their colors or move them as a unit across the screen. Because you can dynamically modify almost everything about a Get, such changes are surprisingly easy.

First write the functions to process the entire GetList. The functions below move the Gets a specified number of rows and columns, and change the colors of the Gets. Both force the Gets to redisplay themselves:

```
//  Moving rows and columns and changing colors
//
//  MoveGets(<aGets>, <nRows>, <nCols>) --> <aGets>
//  Move Get array to new location and redisplay

FUNCTION MoveGets(aGets, nRows, nCols)
   LOCAL nGet

   FOR nGet := 1 TO LEN(aGets)
      aGets[nGet]:row += nRows
      aGets[nGet]:col += nCols
      aGets[nGet]:display()
   NEXT

   RETURN aGets


//  ColorGets(<aGets>, <cColor>) --> <aGets>
//  Change colors of all Gets in Get array
//  and redisplay

FUNCTION ColorGets(aGets, cColor)
   AEVAL(aGets, {|oGet| oGet:colorDisp(cColor)})
   RETURN aGets
```

Creating a New Read Layer

While the previous examples demonstrate that many additions can be made without modifying the Read layer, sometimes it is necessary or cleaner to replace the default Read layer.

Basic Guidelines

How do you know whether to extend the Read layer or to create a new one? There is no absolute rule, but there are some basic guidelines. Create a new Read layer under the following circumstances:

- The new feature would, otherwise, require adding a function to the VALID or WHEN clause of every Get.
- The new feature requires actions between each Get.

Extend the default Read layer, if either of the following is true:

- The new feature only requires modifying the Get at the time it is created (for example, storing information in Get:cargo or one of the other instance variables).
- The new feature does not affect the order in which Gets are traversed, or what tasks are carried out between Gets. This type of feature is usually added by making a modification at one of the lower layers (leaving the Read layer intact).

Note: One set of modifications to the Read layer is generally incompatible with any other set (that is, both can exist in the same application but cannot be used simultaneously). Combining them requires creating a third Read layer that incorporates ideas from both implementations.

Important Implementation Rules

If you determine that a new Read layer is necessary, be mindful of several things. First and foremost is the following rule, important in all aspects of CA-Clipper programming: avoid changing the default behavior of the system.

In the case of the Get system, this means:

- Do not modify Getsys.prg or redefine any of the functions contained within it.
- Do not redefine the basic READ command. You can add clauses and map the new command to a different Read layer.
- Do not redefine the basic @...GET command; add clauses through a new command definition.
- If you add a clause to the @...GET command, map the new command to the basic @...GET command, if at all possible.

In addition to the above rules, if it is necessary to store any information with individual Gets, store it as a dictionary entry in Get:cargo.

If you follow these rules, your Read layer has a much better chance of working with extensions you intend to use with the default Read layer. It also lets you revert to the default Read layer if you have problems with your code. Most importantly, your code will be much more maintainable by other programmers (or by you at a later date) because it will be obvious where you have made modifications to the system.

Implementation Steps

Creating a Read layer that follows the above rules is relatively straightforward:

1. Determine where any initialization code should be located. It is important that your Read layer be *reentrant* (that is, your read function can be called from within itself via a VALID or WHEN clause). Therefore, take care that you save and restore any filewide statics as needed.

   Your initialization code will therefore be split between your read function and the static function ClearGetSysVars(), defined in Getsys.prg.

2. Create your new read function based on READMODAL(), making sure to rename it. After making the appropriate modifications, you have defined the interface.

3. Add any necessary supporting functions, declaring as public any functions that will be called by the user (either directly or through a command), and declaring as static all other functions.

4. Create a header file to define the commands that implement your new Read layer.

READ VALID

The first example of a custom Read layer lets you control exit from the Read with the VALID clause. This is accomplished by writing a new function, ReadValid(), based on the standard READMODAL() function as defined in Getsys.prg.

ReadValid() accepts all parameters required by READMODAL, as well as an additional parameter—a code block that executes when the user attempts to leave the Read. If the code block returns false (.F.), the user is left on the same Get; otherwise the Read is exited. Note that the code block is passed both the GetList and a logical value that indicates whether any of the Gets in the Read were changed.

Because the addition of this capability does not require any initialization and adds no new "states" to the Read layer, you can completely ignore the reentrancy issue, which greatly simplifies the implementation. The code shown below for ReadValid() is an excerpt—any code not shown is the same as READMODAL():

```
// Restricting a read based on a VALID clause
//
// ReadValid()
// Read until condition is true

FUNCTION ReadValid(GetList, ..., bValid)
    .
    .
    .
    LOCAL nLastPos
    .
    .
    .
    WHILE !snPos == 0
        .
        .
        .
        // Save current position
        nLastPos := snPos

        // Move to next Get based on exit condition
        snPos := Settle(aGetList, snPos)

        // If I am about to exit, check to see if
        // VALID condition is true
        IF snPos = = 0
            // VALID clause is passed current GetList,
            // whether Read has been changed
            //
            IF !EVAL(bValid, aGetList, UPDATED())
                // If it isn't valid, don't move
                snPos := nLastPos
            ENDIF
        ENDIF
    ENDDO
    .
    .
    .
    RETURN UPDATED()
```

A new command, READ VALID, makes it easy to use this new Read. Notice that the blockify result marker is used, allowing the code block to be either an expression or a block:

```
//  Translation Rules for READ VALID
//
//  Valid.ch
//  READ VALID

#command READ VALID <valid> SAVE => ;
   ReadValid(GetList, <{valid}>)

#command READ SAVE VALID <valid> => ;
   ReadValid(GetList, <{valid}>)

#command READ VALID <valid> => ;
   ReadValid(GetList, <{valid}>) ;GetList := {}
```

You need only #include Valid.ch and add the VALID clause where desired to use the new Read layer. A simple example follows:

```
//  Using the New Read layer
//

#include "Valid.ch"

MEMVAR GetList

PROCEDURE Main()
   LOCAL cVar1, cVar2, cVar3
   cVar1 := cVar2 := cVar3 := SPACE(20)

   CLS
   @ 10, 10 SAY "Variable 1" GET cVar1
   @ 11, 10 SAY "Variable 2" GET cVar2
   @ 12, 10 SAY "Variable 3" GET cVar3

   // The following insures that none of the Gets
   // are left empty, and is equivalent to:
   //
   //    READ VALID (!EMPTY(cVar1)) .AND. ;
   //      (!EMPTY(cVar2)) .AND. (!EMPTY(cVar3))
   //
   READ VALID {|aGets, lUpdated| ;
      NoneEmpty(aGets, lUpdated)}
   RETURN

//  NoneEmpty()

//  Return .T. if none of the Gets is empty

STATIC FUNCTION NoneEmpty(aGets, lUpdated)
   LOCAL nGet

   // See if there are any empty Gets
   nGet:=ASCAN(aGets,{|oGet| EMPTY(oGet:varGet())})

   // nGet will equal 0 if there were no empty Gets
   RETURN nGet = = 0
```

Gets With Messages

A slightly more complex Read layer example displays a message on the screen for every Get. In this section, the ReadMessage() function is implemented, again based on the standard READMODAL() function as defined in Getsys.prg.

Note: This is presented as an example of how to customize the Get system. A much easier way to display a message for a Get is to use Get:message (which can be assigned using the MESSAGE clause of the @...GET command) in association with the MSG clauses for the READ command.

Implementing ReadMessage() requires that you define a means for specifying and storing the message as well as for actually displaying it. As discussed earlier in the context of implementing context-sensitive help, the easiest way to do this is to store the message as a dictionary entry in Get:cargo.

In the following example, the Read layer retrieves the message from the Get:cargo dictionary and displays it as specified by the SET MESSAGE TO command, saving and restoring that section of the screen. Here again, only an excerpt of ReadMessage() is shown, indicating where it differs from READMODAL():

```
//  Retrieving and displaying a message
//

#include "Set.ch"

// ReadMessage()
//
FUNCTION ReadMessage(aGetList, ...)
    .
    .
    .
    LOCAL cMsg, lCenter, nRow
    LOCAL nSaveRow, nSaveCol
    LOCAL cSaveScr
    .
    .
    .
    // Retrieve settings from SET MESSAGE TO
    nRow := SET(_SET_MESSAGE)
    IF nRow = = NIL
       nRow := 0
    ENDIF
```

```
lCenter := SET(_SET_MCENTER)
IF lCenter = NIL
   lCenter := .F.
ENDIF
.
.
.
WHILE !snPos == 0
   // Get next Get from list and post it
   .
   .
   .
   // If a message is supplied and messages are
   // enabled, display it
   cSaveScr := NIL
   IF nRow > 0 .AND. oGet:cargo != NIL

      // Retrieve message from dictionary
      cMsg := DictAt(oGet:cargo, "cMessage")

      IF cMsg != NIL
      // Save message row and cursor settings
      cSaveScr := SAVESCREEN(nRow, 0, nRow, ;
         MAXCOL()-1)
      nSaveRow := ROW()
      nSaveCol := COL()

      IF lCenter
         SETPOS(nRow, ((MAXCOL()/2) - ;
            (LEN(cMsg)/2)))
         DISPOUT(cMsg)
      ELSE
         SETPOS(nRow, 0)
         DISPOUT(cMsg)
      ENDIF

      // Restore cursor
      SETPOS(nSaveRow, nSaveCol)
      ENDIF

   ENDIF

   // Read the Get
   .
   .
   .
   // If screen line was saved, restore it

      IF cSaveScr != NIL
      RESTSCREEN(nRow, 0, nRow, MAXCOL() - 1, ;
         cSaveScr)
   ENDIF

   // Move to next Get based on exit condition
   snPos := Settle(aGetList, snPos)
ENDDO
.
.
.
RETURN UPDATED()
```

Two new commands, @...GET MYMSG and READ MYMSG, make it easy to access this new Read layer:

```
// Creating commands to access the new Read layer
//
// GetRead.ch

// Translation rule for @...GET MYMSG
#command @ <row>, <col> GET <var> [<clauses, ...>] ;
   MYMSG <msg> [<moreClauses, ...>] ;

=> @ <row>, <col> GET <var> [<clauses>] ;
      [<moreClauses>] ;
      ;IF ATAIL(GetList):cargo = NIL ;
      ;ATAIL(GetList):cargo := DictNew() ;
      ;ENDIF ;
;DictPut(ATAIL(GetList):cargo, "cMessage", <msg>)

// Translation rule for READ MYMSG
#command READ MYMSG SAVE => ReadMessage(GetList)

#command READ SAVE MYMSG => ReadMessage(GetList)

#command READ MYMSG => ;
   ReadMessage(GetList) ;GetList := {}
```

To use this new Read layer, add the MYMSG clause to any Gets that should have a message. SET MESSAGE TO sets the row for the messages and whether they should be centered. Finally, use the READ MYMSG command instead of READ. This example demonstrates:

```
//  Using the new Read layer

PROCEDURE Main()
   LOCAL GetList := {}
   LOCAL cVar1, cVar2, cVar3

   cVar1 := cVar2 := cVar3 := SPACE(20)

   SET MESSAGE TO (MAXROW() - 1) CENTER

   CLS
   @ 10, 10 SAY "Var 1" GET cVar1 MYMSG ;
      "Enter Variable 1"
   @ 11, 10 SAY "Var 2" GET cVar2
   @ 12, 10 SAY "Var 3" GET cVar3 MYMSG ;
      "Enter Variable 3" PICTURE "@!" ;
      VALID (!EMPTY(cVar3))

   READ MYMSG

   RETURN
```

The Reader Layer

As flexible as the ability to create new Read layers is, it allows no control over individual Gets. You gain control over an individual Get through that GET's *reader*. This is how the new GUI-like commands (such as @...GET CHECKBOX and @...GET LISTBOX) are implemented.

A reader intercepts the keystrokes and applies them to the Get; it is completely responsible for implementing the behavior of that particular Get. You can see this most clearly by examining the default reader, GETREADER(), as defined in Getsys.prg.

GETREADER() communicates with the Read layer through the current GET's exitState instance variable. This tells the Read layer how the Get was exited so it can determine which action it should take next. Getexit.ch defines the possible values for Get:exitState. GETREADER() then passes through three phases: prevalidation, assignment, and postvalidation.

Prevalidation

The GETPREVALIDATE() function performs the prevalidation step to evaluate whether the Get fulfills the requirements set in the WHEN clause. If this step fails, the Get is not entered and Get:exitState is set to GE_WHEN.

Assignment

In the assignment phase, INKEY(0) intercepts keystrokes, then GETAPPLYKEY() acts on the ASCII value of the keystroke. When a user strikes an exit key, Get:exitState is set to something other than GE_NOEXIT and control passes to the postvalidation phase.

Postvalidation

The GETPOSTVALIDATE() function performs the postvalidation step to evaluate whether the Get fulfills the requirements set in the VALID clause. If the requirements are met, execution passes back to the Read layer. If they are not met, the assignment phase is reentered.

Creating New Reader Layers

Though GETREADER() implements all of the standard CA-Clipper Get behaviors, you can add new behaviors that fulfill special needs.

Basic Guidelines

As with the decision to extend the default Read layer or to create a new one, it is not always clear when you should implement a feature through a new Read layer and when you should implement it through a new reader. This indecision is especially strong when the feature involves taking some action before and/or after every Get.

The message Read layer created earlier in this chapter is an example of this; you could have implemented it as a reader rather than as a Read layer. Unfortunately, it would then have been incompatible with other readers. Since it is not uncommon to use several readers at a time, this is a larger drawback than incompatibility with other Read layers.

Note: Adding some features requires creating new behavior at all layers. In most cases, the change is so drastic that it counts as a new form system and is no longer part of the Get system. In these cases, you should consider creating a completely new form tool rather than drastically modifying the existing system. The creation of new form systems is outside the scope of this chapter.

Important Implementation Rules

Once you decide that creating a new reader is the correct thing to do, there are a few rules you should follow. As in creation of a Read layer, the default behavior of the system should never be modified. When creating a new reader this means:

- Do not modify Getsys.prg or redefine any of the functions contained within it, including the GETREADER() and GETAPPLYKEY() functions.
- Do not redefine the basic @...GET command; again, you can add clauses through a new command definition.
- If a clause is added to the @...GET command, the new command should map to the supplied @...GET command, if at all possible.

To function well, your reader must adhere to the protocol laid out by the default reader, GETREADER(). It must also store data in a way that will not harm other extensions that may be storing data. This means that the reader will:

- Use Get:exitState as its means of communicating with the controlling Read layer.
- Use GETPREVALIDATE() to ensure that the Get can be entered.
- Use GETPOSTVALIDATE() to ensure that the Get can be exited.
- Use GETAPPLYKEY() for *all* keystrokes, or define a unique "apply key" function that handles all keystrokes. This enables you to use the new reader in non-modal form systems.
- Conform to all of CA-Clipper's defined editing keystrokes unless there is a compelling reason to do otherwise.
- Handle all possible valid picture clauses for the data type being handled.

- Store any data that must accompany the Get as a dictionary entry in Get:cargo. This requires use of the functions found in the sample program Dict.prg supplied with CA-Clipper.
- Check Get:cargo, when storing such data, to see if it is NIL. If it is NIL, create and assign a dictionary to the instance variable. If it is not NIL, assume that Get:cargo is already a dictionary and add the entry.

By following these rules, you ensure that your reader is compatible with the broadest possible range of Read layers, new form tools, and other readers.

Incremental Date Get

Our first example of a custom reader is one in which a date is changed by incrementing it one day with the plus key (+) and decrementing it a day with the minus key (-). All normal editing keys are still allowed. This example, following our guidelines, creates two functions, DateGetReader() and DateGetApplyKey(), to implement this reader.

DateGetReader() is identical to GETREADER() except that it calls DateGetApplyKey() instead of GETAPPLYKEY(). The code shown below is an excerpt—any code not shown is the same as GETREADER():

```
#include "Getexit.ch"

//    DateGetReader()
//
//    Replacement for GETREADER(). Logic is identical,
//    except DateGetApplyKey() processes keystrokes

PROCEDURE DateGetReader(oGet, ...)
    .
    .
    .
    // Apply keystrokes until exit
    WHILE oGet:exitState=GE_NOEXIT .AND. !slKillRead
       DateGetApplyKey(oGet, INKEY(0, .T.), ...)
    ENDDO
    .
    .
    .
    RETURN
```

Because this reader requires special handling for only two keys, GETAPPLYKEY() can handle most of the keystrokes. Thus, DateGetApplyKey() intercepts the plus and minus keys and implements special handling for them, passing control to GETAPPLYKEY() for all other keystrokes. This is the recommended way to add correct support for all the default editing keystrokes:

```
//  DateGetApplyKey()
//
//  Replacement for GETAPPLYKEY()
//  Performs incr/decr on a date get for '+' and '-'
//  keys, standard behavior otherwise.

PROCEDURE DateGetApplyKey(oGet, nKey, ...)
   LOCAL cKey := CHR(nKey)

   DO CASE
   CASE cKey == '+' .AND. oGet:type == 'D'
      oGet:buffer := TRANSFORM(oGet:unTransform() ;
         + 1, oGet:picture)
      oGet:changed := .T.
      oGet:display()

   CASE cKey == '-' .AND. oGet:type == 'D'
      oGet:buffer := TRANSFORM(oGet:unTransform() ;
         - 1, oGet:picture)
      oGet:changed := .T.
      oGet:display()

   OTHERWISE
      GETAPPLYKEY(oGet, nKey, ...)

   ENDCASE

   RETURN
```

To use this new reader, you must create a new command that includes support for it. In particular, the key to getting READMODAL() to call the new reader instead of the default reader is to assign the new reader, as a code block, to the Get object's reader instance variable. This is accomplished using the SEND clause of the @...GET command. Notice that this command is implemented in a way that adheres to the guidelines, calling the default implementation of @...GET in its definition:

```
//    Translation rule for @...GET DATECHANGE
//
//    Dateget.ch


// @...GET DATECHANGE

#command @ <row>, <col> GET <var> [<clauses, ...>];
    DATECHANGE [<moreClauses, ...>];

    => @ <row>, <col> GET <var> [<clauses>];
    SEND reader:={|get,...| DateGetReader(get,...)};
    [<moreClauses>]
```

This example demonstrates the ease of using the new reader. It includes the header file and uses the @...GET DATECHANGE command to create the Gets:

```
//  Using the new reader
//

#include "Dateget.ch"

PROCEDURE Main()
   LOCAL dOne, dTwo

   dOne := CTOD("3/17/65")
   dTwo := CTOD("5/20/91")

   CLS
   @ 10, 10 GET dOne DATECHANGE
   @ 11, 10 GET dTwo DATECHANGE

   READ

   RETURN
```

The Get Function Layer

As you have already seen, the usual way of manipulating the Get function layer is by defining a new reader which calls your functions instead of the default ones. For example, in the previous example, you saw how to intercept keystrokes at the Reader layer and manipulate them using your own apply key function.

The example, however, used the default GETAPPLYKEY() to implement the majority of its keystroke behavior, which is not always the case. Sometimes it is necessary to create a new apply key function from scratch. To do that, it is important to know exactly what GETAPPLYKEY() does.

GETAPPLYKEY() goes through several steps to implement the keystroke behavior associated with Gets:

- Checks and executes SET KEYs, if appropriate
- Checks movement keys and sets Get:exitState to GE_UP, GE_DOWN, GE_TOP, or GE_BOTTOM, if appropriate
- Checks other keys that exit Gets and sets Get:exitState to GE_ENTER, GE_ESCAPE, or GE_WRITE, if appropriate
- Checks and acts upon special purpose keystrokes like Insert and Ctrl+U (undo), if appropriate
- Discards all invalid keystrokes, or inserts/overstrikes them into the Get, based on the state of the insert flag

Note: To gain a complete and thorough understanding of GETAPPLYKEY(), or any of the other Get functions, the best thing to do is study the source code provided in Getsys.prg.

Creating New Get Function Layers

Creating a completely new apply key routine that does not use GETAPPLYKEY() is a complicated task and should not be undertaken lightly. You should do this only when you must define entirely new behaviors that are not "naturally" supported by the Get class (for example, redefining cursor movement, creating new picture template characters, or inserting characters from the right instead of the left).

To create a reader that requires this type of support, start by creating a new module that contains a renamed copy of GETREADER() and GETAPPLYKEY(), as well as the static function ShowScoreBoard() and its associated defines. (Once again, the source code for all these functions can be found in Getsys.prg.) Then throw out most of the code, and rewrite from scratch. Working from the copies of these functions ensures that you have covered all of the default keystrokes.

There are many pitfalls in creating this type of reader, including:

- Failure to accommodate all possible picture clauses, including those produced by the picture function "@R."
- Failure to correctly handle numeric Gets with no picture specified. In most cases it is easiest to handle this by calculating the picture and, if there is not one defined, assigning it upon entry to the reader.
- Use of cursor movement and editing methods provided by the Get class. These methods are specific to the defined pictures and behaviors associated with the default reader. If you can use these methods, you probably should not be creating a Get function layer.
- Failure to obey Get:clear and clear the Get before accepting more keystrokes.
- Failure to correctly handle floating point numbers. This includes failure to truncate insignificant digits and comparison of floating point numbers exactly without rounding to a specific precision.

Chapter 8
Introduction to the Menu System

In This Chapter

The CA-Clipper Menu system is a powerful tool that allows you to create standard menus consisting of a menu bar with pull-down menus and built-in mouse support. The system lets you define all types of standard items, such as menu commands, toggles, cascading menus, and separators. It also lets you define standard features for menu items, such as shortcut and accelerator keys, toggle and cascading menu indicators, and the ability to disable an item.

The Menu system consists of several predefined classes: TopBarMenu, PopUpMenu, and MenuItem. Collectively, these classes provide you with a great degree of control and flexibility in creating and editing custom-designed menus. This chapter discusses these classes in some detail and establishes the criteria for effective use of the Menu system.

This chapter discusses the following topics:

- Menu terminology
- Overview of class components
- Relationships between menu classes
- Menu properties
- Activating the menu

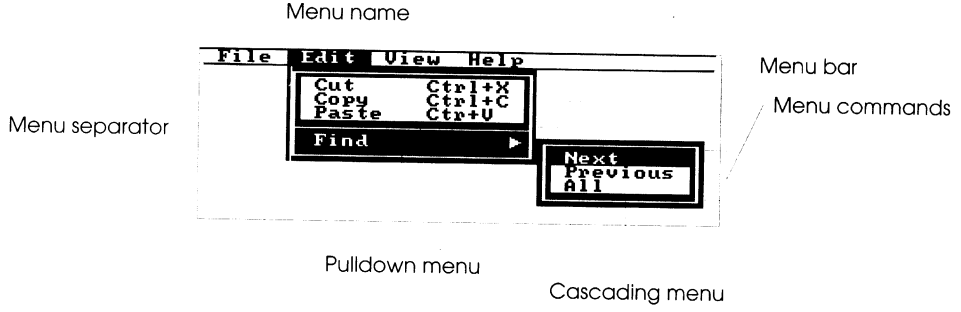
Note: This chapter assumes that you are familiar with the concepts and terminology associated with CA-Clipper's predefined classes as discussed in the "Basic Concepts" chapter of this guide.

Menu Terminology

A *menu* is a user-interface element that presents a list of choices. Menus usually appear in an application's *menu bar*, which generally appears across the top of the screen.

Typically, when a menu is selected in a menu bar (with the mouse or keyboard), it displays a menu of items (menu commands, separators, other menus, and so on). However, it can also immediately perform an action (for example, an Exit menu that quits the application when selected).

The sample menu shown below illustrates some of the components available to you using the Menu system:



In this illustration, the menu bar is created using the TopBarMenu class. The Edit pull-down menu and the Edit Find cascading menu are created using the PopUpMenu class. All menu commands, including the menu separator, and all menu names shown on the menu bar are defined using the MenuItem class.

The MenuItem class allows you to associate a menu item with either a menu bar or menu, define properties for the menu item (such as accelerator and shortcut keys), and associate an action with the menu item (such as displaying a menu or performing an operation against a database).

This example illustrates one way to design a menu, but there are many other ways. For example, it is not necessary to have a menu bar.

You can use the PopUpMenu class to define a pop-up menu based on some other user action. This chapter concentrates on the standard menu setup described here to illustrate various features of the Menu system.

Tip: Regardless of the nature of the menu you want to create, the most important guidelines to follow are ease of use and consistency.

Overview of Class Components

Each class in the Menu system defines a class function, as well as methods and instance variables that you will use when writing code to build and manipulate menus.

Class Functions

You create menu bars, menus, and menu items using functions associated with each individual class. These functions, which are discussed in more detail later in this chapter, are TopBar(), PopUp(), and MenuItem(). Each creates an object and sets up some basic, required properties for the object by assigning argument values to instance variables. For example, TopBar() assigns values to TopBarMenu:row, TopBarMenu:left, and TopBarMenu:right to define the location and width of the menu bar. MenuItem(), among other things, assigns a menu name or code block to the MenuItem:data instance variable to determine the action to take when the item is selected.

Instance Variables

You define other, optional properties for menu objects by assigning values to instance variables. For example, you can disable a menu item by assigning false (.F.) to MenuItem:enabled or specify colors for a menu bar or menu using the colorSpec instance variable, available for both TopBarMenu and PopUpMenu objects.

Methods

You establish the relationships between individual components of the overall menu system using methods defined for the various menu classes. For example, the addItem() method, defined for both the TopBarMenu and PopUpMenu classes, adds an item to the end of a menu bar or menu.

Methods are also used to make the object perform actions. For example, TopBarMenu:display() activates a menu bar by showing it on the screen.

Finally, methods supply information about the current status of object. For example, MenuItem:isPopUp() tells you whether the menu item is associated with a menu (in which case its data instance variable is a PopUpMenu object) or an action (in which case its data instance variable is a code block).

Many, though not all, of the instance variables and methods associated with the Menu system classes are discussed and used in examples throughout this chapter. For a complete list and explanation, refer to the entry for the individual class in the *Reference Guide*.

Relationships Between Menu Classes

Using the sample menu illustrated earlier, you can easily see the relationships between the three menu classes.

TopBarMenu

TopBarMenu defines the menu bar, which controls the overall menu. To create a top bar menu, you use its class function, TopBar(). This function creates a completely generic menu bar by defining the location of the menu bar (that is, its row and starting and ending column positions).

For example, the following code creates a menu bar and stores it in the variable *oTopBar*. When displayed, the menu bar will be placed on the top row and take up almost the entire width of an 80-column screen:

```
// Create a menu bar at the very top of the screen
oTopBar := TopBar(0, 1, 78)
```

MenuItem

Once the menu bar is established, the next step is to define a menu item for it, which you do by first creating a MenuItem object then adding it to the menu bar. In this code, a menu bar item named Exit is added that terminates the application. The MenuItem() class function and the TopBarMenu:addItem() methods are used, which have been mentioned already.

```
oTopBar:addItem(MenuItem("Exit", {|| ShutDown()}))
```

PopUpMenu

It is not typical, however, for an item on a menu bar to be directly associated with an action as in this example. Instead, menu bar items are usually associated with a pull-down menu. The menu item is still added using the construction TopBarMenu:addItem(MenuItem()), but you specify the name of a PopUpMenu object instead of a code block.

To accomplish this, you must first create the pull-down menu. Use the PopUp() function to create a generic PopUpMenu object, associate the menu with the menu bar, then add items to the menu in the same manner as you added them to the menu bar. This example illustrates:

```
#include "Inkey.ch"

// Create a new menu
oPopUp := PopUp()

// Associate oPopUp with oTopBar menu bar item, File
oTopBar:addItem(MenuItem("&File", oPopUp))

// Add some items to the newly created menu
oItem := MenuItem("&New", {|| MyCreateFile()}, ;
   K_CTRL_N, "Create a new file")
oPopUp:addItem(oItem)

oItem := MenuItem("&Open...", {|| MyOpenFile()}, ;
   K_CTRL_O, "Open a file")
oPopUp:addItem(oItem)

// Add Save menu item
oItem := MenuItem("&Save", {|| MySaveFile()}, ;
   K_CTRL_S, "Save a file")
oPopUp:addItem(oItem)
```

The resulting menu looks like this:



More on Menu Items

This code is fairly straightforward, but there are some interesting features about the MenuItem() function that should be noted. The following lines of code, excerpted from the previous example, are used to demonstrate:

```
oTopBar:addItem(MenuItem("&File", oPopUp))
oItem := MenuItem("&New", {|| MyCreateFile()}, ;
    K_CTRL_N, "Create a new file")
```

Accelerator Keys

One of the things you may notice is that the name, or caption, of the both menu items contains an & character ("&File" and "&New"). The & causes the letter immediately following it in the caption to become the accelerator key for the menu item. Accelerator keys are accessed differently depending on whether the menu item is defined as part of a menu bar or menu. For menu bar items, you must press the letter in combination with the Alt key (for example, Alt+F opens the File menu). For menu items, simply press the letter itself (for example, once the File menu is open, press N to select the New command). Thus, in this example, pressing Alt+F, N would be a quick way to execute the File New command.

Tip: Although accelerator keys are not underlined as they are in Windows applications, you can display them using a different color. See Specifying Colors later in this chapter for more information.

Note: Within a menu bar or menu, accelerator keys must be unique. Use the first letter in the menu item's caption unless there is a conflict, in which case you can resort to a secondary letter within the caption. For example, use "&Print" and "P&rint Setup" to distinguish two common commands on the File menu.

Action vs. Menu

A menu item can be associated with an action or it can open another menu from which the user makes a choice. All menu items, however, eventually lead to an action. The MenuItem() function provides for this flexibility by allowing you to specify a code block or the name of a PopUpMenu object as its second parameter.

The most common scenario is illustrated in the example in which the menu bar item is associated with a menu and the individual menu items are associated with actions. In this case, when the user chooses File from the menu bar, the menu associated with *oPopUp* is activated.

When the user chooses the New command, the code block {|| MyCreateFile()} is evaluated, resulting in execution of the function, MyCreateFile().

Shortcut Keys

Besides associating an accelerator key with a menu item, you can associate a key combination, called a shortcut, that executes a menu item directly, regardless of whether its associated menu is active at the time the key combination is pressed. Note, however, that shortcut keys are limited to menu items that are directly associated with a PopUpMenu object and are not available for TopBarMenu menu items.

The shortcut key can be any combination of an Alt, Ctrl, or Shift key and another key on the keyboard, all of which are defined as constants in Inkey.ch. You specify the shortcut key, if any, using the third parameter of MenuItem(). In this example, K_CTRL_N defines Ctrl+N as the shortcut key.

Note: No two menu items associated with the same menu bar can have the same shortcut key. Also, since shortcut keys perform an immediate action, they should be used cautiously. For example, you would probably not want to assign a shortcut key to a menu command whose end result is a destructive database operation, such as pack or zap.

Status Messages

You can also define a message to display when a menu item is selected using the fourth parameter. In this example, the message "Create a new file" is displayed when the File Create menu command has focus.

Menu Properties

Earlier in this chapter, it was mentioned that the properties associated with menu components are defined using instance variables and that the various class functions provided a mechanism for assigning initial values to certain instance variables when you first create an object of that class. This was illustrated in the previous section, although not explicitly, using the MenuItem() class function.

The general syntax for this function is as follows:

```
MenuItem(<cCaption>, <expData>, [<nShortcut>],
[<cMessage>], [<nId>]) → oMenuItem
```

Each parameter maps directly to an instance variable of the MenuItem class as shown below:

Argument	Instance Variable
<cCaption>	MenuItem:caption
<expData>	MenuItem:data
<nShortcut>	MenuItem:shortcut
<cMessage>	MenuItem:message
<nId>	MenuItem:id

In addition to assigning these initial instance variables during object creation, you will often manipulate instance variables directly in your applications. For example, you may want to add color to both the menu bar and its associated pull-down menus, and you do this using the colorSpec instance variable. More importantly, you may want to manipulate menus on an item-by-item basis, including enabling/disabling an item or placing a checkmark next to an item to indicate that it is selected.

The next example builds on the previous one. It demonstrates using various instance variables to create a more effective and visually appealing menu. After the code listing, the enhancements are discussed in more detail.

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

// Create a menu bar at the very top of the screen
oTopBar := TopBar(0, 1, 78)

// Add color to the menu bar
oTopBar:colorSpec := "B/W,GR+/RB,R/W,G/RB,N+/W,W+/B"
.
.
.
// Add color to menu immediately after creating it
oPopUp := PopUp()
oPopUp:colorSpec := "B/W,GR+/RB,R/W,G/RB,N+/W,W+/B"

// Associate menu with oTopBar menu bar item, File
oTopBar:addItem(MenuItem("&File", oPopUp))

// Add some items to the newly created File menu
oItem := MenuItem("&New", {|| MyCreateFile()}, ;
   K_CTRL_N, "Create a new file")
oPopUp:addItem(oItem)
.
.
.
// Disable the Save menu choice immediately after
// creating it
oItem := MenuItem("&Save", {|| MySaveFile()}, ;
   K_CTRL_S, "Save a file")
oItem:enabled := .F.
oPopUp:addItem(oItem)
.
.
.
```

```
// Add a separator and other menu items, such as
// Print and Exit
oPopUp:addItem(MenuItem(MENU_SEPARATOR))
.
.
.
// Create an Edit menu, attach it to menu bar, and
// add some items to it, such as Undo, Cut, and Copy
.
.
.
// Add a separator
oPopUp:addItem(MenuItem(MENU_SEPARATOR))

// Add Edit Go menu item, which will result in a
// cascading menu
oPopUp1 := PopUp()
oPopUp1:colorSpec := "B/W,GR+/RB,R/W,G/RB,N+/W,W+/B"
oItem := MenuItem("&Go", oPopUp1)

// Add some items to the Go cascading menu
oItem := MenuItem("&Go To...", {|| MyGoToLine()}, ;
   K_F5, "Go to a specific line number")
oPopUp1:addItem(oItem)
.
.
.
// Change the style of the Go cascading menu
oItem:style := CHR(251) + ">"
oPopUp:addItem(oItem)

// Add another separator
oPopUp:addItem(MenuItem(MENU_SEPARATOR))

// Add one final menu item to the Edit menu
.
.
.

// Create a third menu called Options and attach it
// to the menu bar
oPopUp := PopUp()
oPopUp:colorSpec := "B/W,GR+/RB,R/W,G/RB,N+/W,W+/B"
oTopBar:addItem(MenuItem("&Options", oPopUp))

// Add the first item to the Options menu
oItem := MenuItem("AutoSave", {|| MyAutoSave()}, ;
    , "Toggle auto save preference")

// Place a check mark next to this new item
oItem:checked := .T.
oPopUp:addItem(oItem)
.
.
.
```

Note: You could turn this code into a function and simply return *oTopBar* to the calling routine which would, in turn, display and activate the entire menu system. This technique is discussed later in this chapter under Activating the Menu and illustrated in the sample program Menu.prg, located in your CA-Clipper SOURCE\SAMPLE directory.

Specifying Colors

The colorSpec instance variable is defined for both the TopBarMenu and PopUpMenu classes, allowing you to define separate colors for the menu bar and each pull-down menu associated with it. In this example, the same color string is used for the menu bar and all other menus in the system:

```
<oMenu>:colorSpec := "B/W,GR+/RB,R/W,G/RB,N+/W,W+/B"
```

colorSpec uses positional parameters to identify areas of a menu for which you can specify colors. If you do not explicitly assign a value to colorSpec, the default colors are determined by SETCOLOR(), as indicated in the table below:

Position	Defines Color for	SETCOLOR() Default
1	Unselected menu items	Unselected
2	Selected menu item	Enhanced
3	Accelerator key for unselected menu items	Background
4	Accelerator key for selected menu item	Enhanced
5	Disabled menu items	Standard
6	Menu border	Border

Menu Separators

The MenuItem:caption instance variable has already been discussed in this chapter. In particular, you have seen how it defines the text associated with a menu item and how it can be used to define the accelerator key for a menu item.

This example introduces another use for MenuItem:caption, which is to define a horizontal line to separate menu choices into logical groups. For example, in the File menu, the Print command is separated from the New, Open, and Save commands.

To define a separator, use the constant value MENU_SEPARATOR (defined in the header file, Button.ch) where you would normally specify a menu caption:

```
oPopUp:addItem(MenuItem(MENU_SEPARATOR))
```

The menu separator is not selectable and cannot have an action, message, shortcut, or id associated with it.

Toggles

Some menu items represent a flag, or *toggle*, that somehow affects the behavior of the overall system. Options AutoSave is such a menu item, determining whether changes being made by the user are automatically saved periodically by the system.

This type of menu item is either on or off, a condition usually indicated by the presence or absence of a checkmark (see Menu Item Indicators below for more information) to the left of the menu item. You specify that a menu item is on (checked) by assigning a value of true (.T.) to MenuItem:checked:

```
oItem:checked := .T.
```

By default MenuItem:checked is false (.F.), which means that menu items are not checked.

You can build program logic that changes the status of MenuItem:checked and controls behavior based on its current status. This technique is not illustrated in the example because all of the action code blocks are over-simplified; however, modifying the example slightly can easily show how it might be done. The code block in the example for Options AutoSave looks like this:

```
{|| MyAutoSave()}
```

Changing it as follows has two effects: switching MenuItem:checked to its opposite value and calling the SetAutoSave() function using this new value:

```
{|| oItem:checked := !oItem:checked, ;
    SetAutoSave(oItem:checked)}
```

SetAutoSave() is a standard set/get function that returns the current setting when called without an argument and changes the current setting when called with an argument:

```
FUNCTION SetAutoSave(lNew)
   STATIC lSetAutoSave := .T.
   LOCAL lOld

   IF lNew = = NIL
      RETURN lSetAutoSave

   ELSEIF VALTYPE(lNew) == "L"
      lOld := lSetAutoSave
      lSetAutoSave := lNew
      RETURN lOld

   ENDIF
```

To control logic in other parts of the application based on this setting, simply call SetAutoSave() without an argument:

```
IF SetAutoSave()
   // Code to do auto saving on periodic basis
ENDIF
```

Finally, to make the menu definition more dynamic, you would call this function to determine the original MenuItem:checked status instead of assigning the constant value of true (.T.) as in the original example:

```
oItem:checked := SetAutoSave()
```

Disabling Menu Items

Menu items that are not meaningful to the current state of the application are usually disabled. For example, the File Save and File Print menu items might be disabled initially because there is no open file to save or print.

Assign false (.F.) to the MenuItem:enabled property to disable a menu item:

```
oItem:enabled := .F.
```

By default, menu items are enabled (that is, MenuItem:enabled is true (.T.)).

In the example, the value of the MenuItem:enabled flag is simply set to a constant value of false (.F.), but in a real menu you would seldom do this. Instead, you would make the initial value contingent on some other logical setting. For example, the USED() function (which determines if there is a database file in use in the current work area) is a valid indicator for whether the File Save and File Print commands should be available each time the menu is displayed:

```
oItem:enabled := USED()
```

Cascading Menus

A cascading menu, or *submenu*, is any menu selection that results in the display of another menu. Submenus simplify the top-level menus by limiting the number of items that are initially displayed. This allows the user to see the most important commands while browsing through the menus without being distracted by the details of submenus.

This is well-illustrated in the example. The Edit menu has a Go command that results in a submenu. Thus, when the user sees the main menu, it is immediately apparent that the menu supports database navigation. The details of Go To, Go Bottom, and Go Top, however, are hidden within the submenu.

The example code shown earlier did not give all the details that are shown below:

```
// Add Edit Go menu item, which will result in a
// cascading menu
oPopUp1 := PopUp()
oPopUp1:colorSpec := "B/W,GR+/RB,R/W,G/RB,N+/W,W+/B"
oItem := MenuItem("&Go", oPopUp1)

// Add some items to the Go submenu
oItem := MenuItem("&Go To...", {|| MyGoToLine()}, ;
   K_F5, "Go to a specific line number")
oPopUp1:addItem(oItem)

oItem := MenuItem("G&o To Top", {|| MyGoTop()}, ;
   K_CTRL_HOME, "Go to top of file")
oPopUp1:addItem(oItem)

oItem := MenuItem("Go To &Bottom", ;
   {|| MyGoBottom()}, K_CTRL_END, ;
   "Go to bottom of file")
oPopUp1:addItem(oItem)
```

Menu Item Indicators

Most, but not all, of the information that appears as part of a menu item is defined through its MenuItem:caption instance variable. For example, when you define a shortcut key, the key name is automatically displayed to the right of the menu item, without explicitly including it in the caption.

On the other hand, if you want to indicate that additional information will be required when a menu item is selected (for example, the user may be required to fill in a dialog box), you must include the ellipsis (...) as part of the caption:

```
oItem := MenuItem("&Open...", {|| MyOpenFile()}, ;
   K_CTRL_O, "Open a file")
```

Two items that are automatically included as part of a menu item's display are the checkmark (√) used to indicate that MenuItem:checked is true (.T.) and the solid arrow (►) used to indicate that a menu item results in a submenu. There is never a need for any code that writes these characters as part of the menu display, but you can control which characters are used via the MenuItem:style instance variable. MenuItem:style is a string containing exactly two characters: the first is the character to display if MenuItem:checked is true (.T.), and the second is the character to use if a submenu results.

In the example code, this is demonstrated for the Go submenu, in which the MenuItem:checked character is left at its default value while the submenu indicator is changed to the greater than symbol (>):

```
oItem:style := CHR(251) + ">"
```

Activating the Menu

Use the MENUMODAL() function to display and activate the menu bar, as shown below:

```
oMenuBar := Menu_Create()
DO WHILE MENUMODAL(oMenuBar,1,24,1,79,"R/W") <> 999
ENDDO
```

This assumes that a TopBarMenu object, *oMenuBar*, is returned by the function Menu_Create(). This is the first argument for MENUMODAL(). The remaining arguments specify the position of the menu bar item that is selected when the menu is first activated (1), the row and left and right column positions of the message line (24, 1, 79), and the color specification for messages ("R/W").

Note: Messages are defined for each menu item using the MenuItem:message instance variable.

Because MENUMODAL() takes only a single menu event from the user then terminates, it is necessary to put the function call in a continuous loop, as in the example. The function returns the MenuItem:id of the selected menu item, so you can set up a condition for breaking out of the loop based on the menu's exit option.

In this case, it is based on the File Exit menu command:

```
oItem := MenuItem("E&xit", {|| .T.}, ;
   K_ALT_F4, "End of application", 999)
```

MenuItem:id is specified as the fifth, and final, MenuItem() parameter. Note that no two menu items associated with the same menu bar can have the same id.

Chapter 9
Error Handling Strategies

In This Chapter

Earlier versions of CA-Clipper responded to runtime errors by passing control to one of six routines you could write in CA-Clipper. Error creation, however, was still done by code deep in the heart of CA-Clipper so that you could only respond to a predefined set of errors. Your encounter with errors was purely reactive and, while you could customize the reaction, you could only do it globally. You could not install and remove error handling code on the fly.

In CA-Clipper applications, the line between internal and your own error handling code has blurred to the vanishing point. Not only can you customize the response to an error, you can customize the error. The mechanisms for custom errors and CA-Clipper errors are the same, and you are free to create and handle your own errors in emulation of system errors.

The first part of this chapter is intended to give you some insight into theoretical error handling strategies without giving you any specifics on how you would implement those strategies in CA-Clipper. The idea is to give you an understanding of the basic design of the error system before the implementation details are discussed.

Next, you are introduced to the specific CA-Clipper language elements designed to implement the described error handling strategies. Each element is presented followed by an example that shows how best to use it.

This chapter is theoretical in nature and is not intended as a tutorial on how to use the CA-Clipper error handling system nor is it intended to compare the error handling systems of previous versions to the new system. It does, however, provide an in-depth analysis of error handling strategies, as well as several examples which, if studied carefully, can be used as models for your own error handling implementations.

This chapter introduces you to the CA-Clipper error system, starting with an overview of general exception handling theory then progressing through the role of exceptions in modular applications. The following major topics are discussed:

- Overview of exception handling concepts
- CA-Clipper exception mechanisms
- Error objects
- A baseline strategy
- Network processing

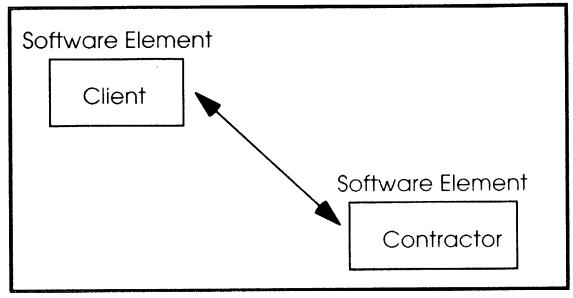
Overview of Exception Handling Concepts

Error processing works well with modular architecture, where applications are broken into component modules. Self-contained, independent modules avoid interapplication dependencies and unforeseen side effects. CA-Clipper architecture promotes use of the modular model.

A module can be a simple function, but more often it is a multi-function subsystem defined in a single source file. In order to scope variables and functions to that file, the module utilizes STATIC declarations. There is, typically, a logical coherency among the functions you include in a subsystem. They all serve some common purpose, which is your rationale for assigning them to a module.

Other modular languages use different terminology, but the concept is the same: portions of an application with common functionality are combined in units. Among these units there is specific division of labor and allocation of responsibility. A complete application is a joining of these subassemblies.

In a modular application, interaction between modules is formal, almost contractual, as illustrated below.



There is an implied contract between a client module and a contractor module. The client assigns the contractor both responsibility and authority for performing the specialized job for which it was designed.

Each module has specific rights and obligations. The client must supply the contractor with the necessary input parameters, or raw materials, in proper form and number. In return, the contractor owes the client the results of operations. The contractor can expect proper input, and the client can expect proper output.

Error Scoping

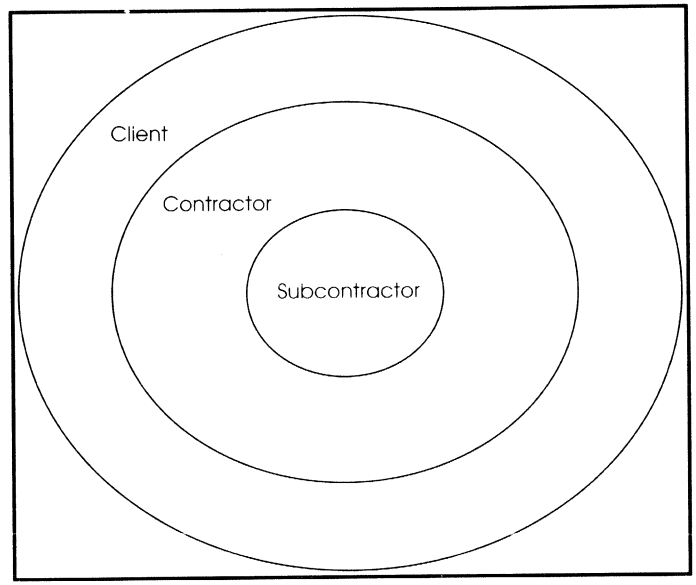
Localization is the key to effective error processing. A module with responsibility for some well-defined part of the application's functionality should also take responsibility for possible malfunctions. The contractor should process departures from normal functionality. A "departure from normal" is called an *exception*, because it is exceptional to expected behavior.

Processing encompasses both sides of a runtime error: generating (detecting, raising, asserting) and responding (reporting, handling). Runtime errors do not merely occur in CA-Clipper—they are explicitly created. The process of detecting an error condition and displaying an alert message is called *raising an error*.

An application should include exception handling as part of its primary functionality rather than patch it in as an afterthought. The right place for code that handles errors pertaining to a certain module of an application is normally within that same module. This is called *localization* of error processing.

In-module coding is not the only way to localize error handling, even though it is the most common way. You can also localize to sub-modular or supra-modular structure. Such localization can, with proper coding, span a function, a statement, or a statement sequence, or the error handling can be invoked over several modules through calling routines or other activations.

Error processing, like other aspects of CA-Clipper, conforms to lexical scoping. That is, error handling procedures are part of the context of the software elements in which they are written, so they have similar scoping. The illustration below shows three software elements, layered like an onion:



Outer layers contain or call inner layers, and some error-processing code lies within each layer. An error raised by the innermost layer falls in that element's context and receives immediate processing by the error handling code scoped to that layer. The error also falls within the context of all surrounding layers and, if not resolved within the inner layer, can be passed to the error handling code scoped to the outer layers. Note that the contexts surrounding an error depend on the calling sequence on any particular activation. Later, if a different set of callers invokes the same inner element, the same error will have a different overall context.

What this does is localize the error handling to the place closest to its raising. It frees clients from involvement with a contractor's difficulties. Should the contractor successfully recover, the client proceeds unaware that an incident occurred. The author of the error handler is likely to intelligently interpret the error and respond appropriately.

Raising Errors

Errors have their roots in assertions. An *assertion* is an inline test of runtime adherence to specification. It can be as simple as an IF statement whose control condition returns true (.T.). Failure of the IF test constitutes an exception. Assertions have the following general pseudocode form:

```
IF .NOT. AsItShouldBe
   REACT
ENDIF
```

Note: Whether all exceptions are errors, or only some, is a matter of opinion. If under some circumstances it is okay for the condition to be false, but you code the assertion just because you still want to know about it, you probably would not label the exception erroneous.

Upon error detection, the reaction is to launch into error processing. This usually means transfer of control to application code designed for that purpose. Executing that code is called *handling* the error.

You can see that errors are not self-generating or self-defining. They do not just happen. Errors are reports of possible, though exceptional, behavior. Error handling is premeditated.

The following pseudocode illustrates use of assertions in the contractual context. The assertions verify conformity to contractual responsibilities. In this simplified case, the module is a function that derives square roots. The function has the right to insist on only non-negative numbers and has the obligation to return a correct value—one that returns the square when squared.

An assertion at the top ensures that the function complains if it receives invalid input. Another at the bottom enforces valid output, testing it independently of the derivation algorithm. If that algorithm involves generating a mathematical series whose terms should converge, another assertion could sound an alarm if they began to diverge:

```
FUNCTION TakeSquareRoot(nSquare)
    // Precondition: at this point nSquare had
    // better be non-negative

    IF !(nSquare >= 0)
        REACT
    ENDIF
    .
    . <statements>
    .
    // Postcondition: at this point nRoot * nRoot
    // had better yield nSquare

    IF !(nRoot * nRoot = nSquare)
        REACT
    ENDIF

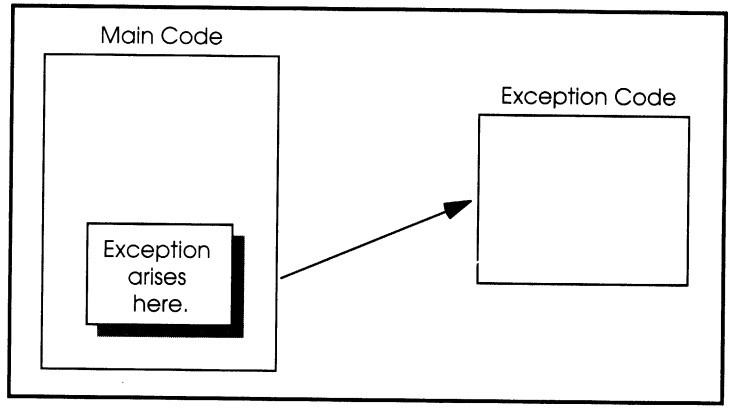
    RETURN nRoot
```

It is natural to expect these errors. You should anticipate them, then insert appropriate assertions in your code. Otherwise, you may encounter strange results without knowing why. Unanticipated errors can cause unpredictable program failures. An error is far preferable.

Handling Errors

As a contingency, an application should generally provide exception handling code. Such code should be strategically set aside in error handlers where it will be available when needed. On one hand, it is separate from the main code; on the other, it is integral to the application. It is as important to an application's design as fire stairs are to a building's design. It may get less use than the main hallway, but it is no less essential.

The location of this code within the source is less important than its availability to the application at runtime. When error-handling code is encountered during runtime, the main code is abandoned as illustrated below:



At the discretion of the error handler, control may or may not return to the main code. If it does, the main code must be written to retake control gracefully (retry, default handling, or substitution options precoded).

When the error-handling code gains control, there are only two theoretical options: retry or terminate. Generally, retrying means finding some way to proceed. This may amount to a simple, brute force retry, but it is usually more elaborate, including attempts to identify and remedy the cause of the error condition.

The retry need not be performed by the error handler. If the error-originating code is written to retry on its own (equipped with a loop, for example), the error handler can signal the original code to give it a try. This necessitates a protocol whereby the originating code indicates its readiness to retry if asked, and the error handler asks by returning some predetermined value.

Variations on the theme are default handling and substitution. Suppose that error-generating code is written so it can proceed even if the error condition is not removed. If it communicates that fact when calling an error handler, the error handler has the option of returning control to the original code for continuation. Similarly, if the original code's purpose is to derive a value but it is written to accept a substitute value if derivation is impossible, an error handler has the option of returning a substitute value to the main code (for example, zero to a routine that tries to divide by zero).

Termination also has a broader meaning than just halting and returning to the operating system. More generally, it means terminating the attempt to perform the current operation and signaling failure to the original invoker (client), with or without prior clean-up (for example, closing files). The invoking process then decides whether it can tolerate the failure and what, accordingly, to do next. This is sometimes called propagating the error to the client or outer context.

CA-Clipper Exception Mechanisms

CA-Clipper imposes no error-processing protocol on applications but gives you the tools to do so in your own way. These tools, and their practical application, are the focus of this section.

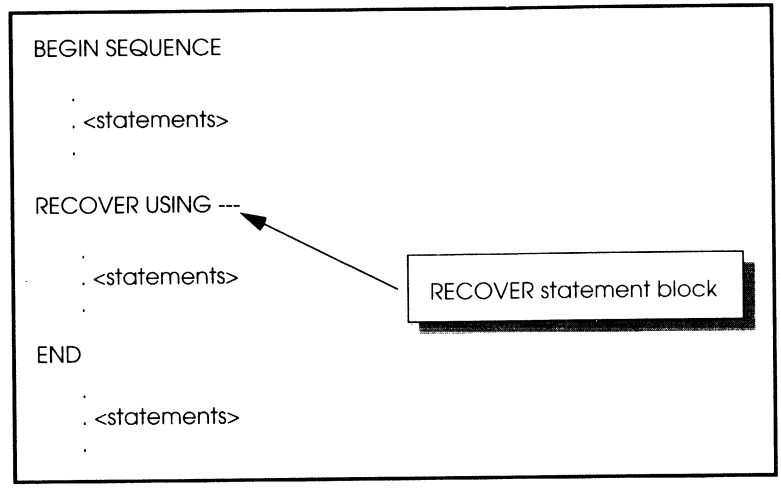
In CA-Clipper, there are two mechanisms for processing exceptions, the SEQUENCE construct, and a posted code block (the *error block*). With these mechanisms as tools, you can build different approaches to error handling. You can, if necessary, use these tools to emulate the fixed, more rigid behaviors of other programming languages.

The two mechanisms differ in the location of error-handling code and how it is called. In the SEQUENCE construct, error-handling code is inline (part of the RAM image of the loaded application) and is called by issuing a BREAK statement. In the error block approach, it is stored in memory (as a code block) and is triggered using EVAL().

In general, use SEQUENCE for application-specific exceptions (for example, "debits do not equal credits", "inventory is negative", "my boomerang will not come back"), and error block for generic, low-level exceptions that usually deal with the computer and could arise across applications (for example, "device not ready", "disk full", "stack overflow").

The SEQUENCE Construct

The basic form of a SEQUENCE construct is shown below:

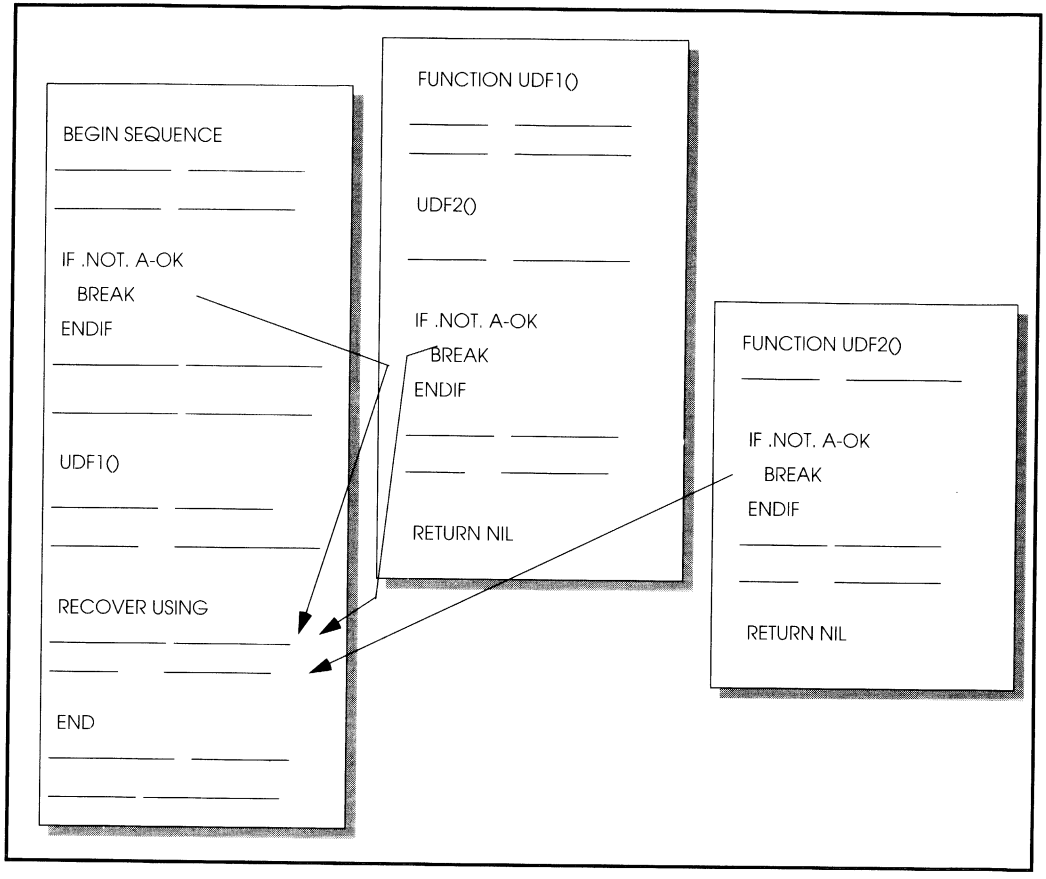


Between the BEGIN SEQUENCE and RECOVER statements is the main code; between RECOVER and END is error-handling code. The error-handling code is called a RECOVER statement block.

A BREAK statement permits the RECOVER statement block to gain control. Without one, the block never executes; instead, control jumps past the END statement when the RECOVER statement is encountered.

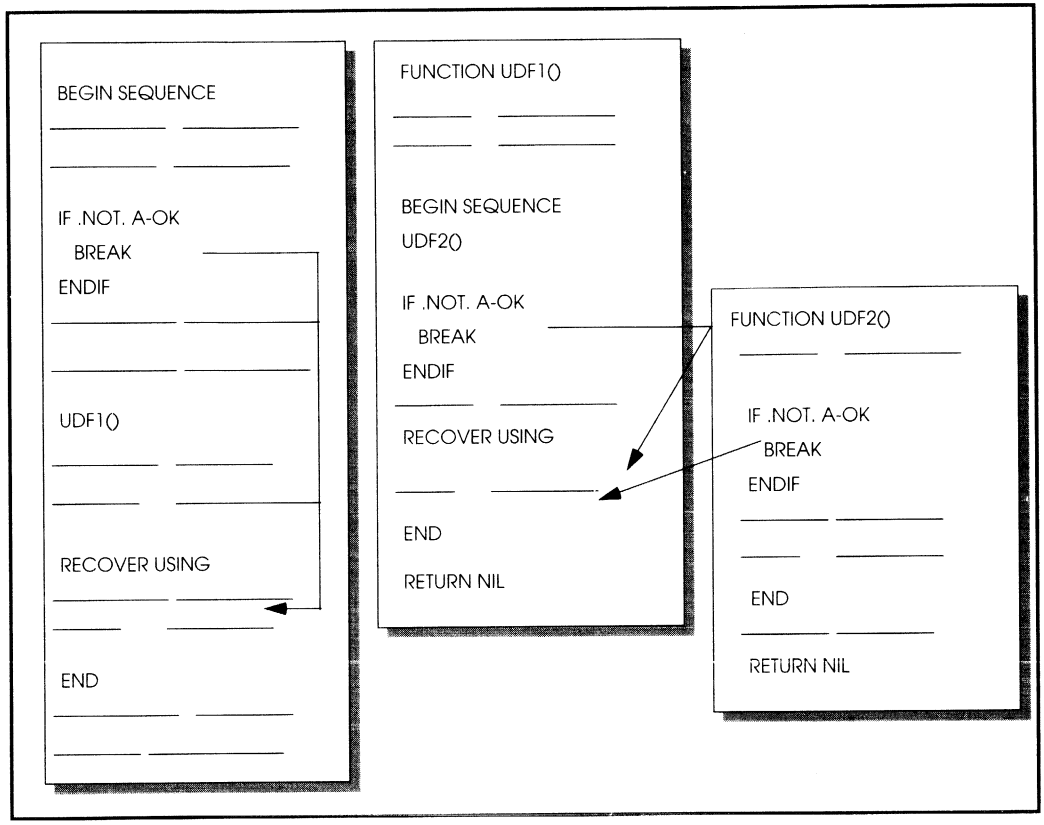
BREAK represents the concrete reaction (pseudocoded using REACT earlier in this chapter). Any assertion in the main code should, when violated, provide a BREAK, abandoning the main code and passing control to the RECOVER code.

BREAKs can be nested, as shown below:



Here an assertion passes control to the RECOVER code whether appearing in the main code itself; one activation removed, as in Udf1(); or two activations removed, as in Udf2().

You can also nest SEQUENCE constructs, as shown below:



Here, Udf1() contains its own SEQUENCE construct, so the assertions in Udf1() or Udf2() now call the RECOVER code local to the context of Udf1(), not that of the original outer sequence. To return the error to the outer sequence would require a BREAK statement issued within Udf1()'s RECOVER statement block.

This illustrates the practical side of error contexts in CA-Clipper. BEGIN SEQUENCE statements delimit context boundaries. Each time you encounter a new BEGIN SEQUENCE statement, you cross into a new context, with its own local (though optional) RECOVER statement block. You exit that context when you leave the main code sequence (whether entering or jumping the RECOVER statement block).

Activation boundaries do not, necessarily, define contexts. For example, the same function can be part of different contexts at different times. If Udf2() is invoked by another caller, it is part of that caller's context. Since you can bracket any code you like with a SEQUENCE construct, CA-Clipper lets you define contexts quite flexibly. Contexts lie wherever you put your SEQUENCE constructs.

The following example shows the suggested general form of such a statement block. A CASE statement responds to whatever errors the programmer anticipated when writing the program. Where control passes next is governed by whether a QUIT, BREAK, or RETURN appears. If you supply no such code, control passes through the END SEQUENCE statement to whatever code follows.

```
//  General form of SEQUENCE construct
//  and RECOVER clause

BEGIN SEQUENCE

   IF lProblem1
      objError := ErrorNew()
      .
      . <stuff objError>
      .
      BREAK objError
   ENDIF
   .
   .
   .
   IF lProblem2
      objError := ErrorNew()
      .
      . <stuff objError>
      .
      BREAK objError
   ENDIF
   .
   .
   .
```

```
RECOVER USING WhatItsAllAbout

    DO CASE
    CASE <obj-related condition>
        <handling>

    CASE ...
        <handling>
        BREAK WhatItsAllAbout // Optional propagation

    CASE ...
        <handling>

    OTHERWISE
        BREAK WhatItsAllAbout // Optional propagation

    ENDCASE

END SEQUENCE
```

There are several special cases you should consider. An application that does not use SEQUENCE constructs has DOS as its implicit outer context. A BREAK anywhere in the application leads to the DOS prompt. One with a single SEQUENCE construct surrounding the initial main menu module has that module as outer context, with any BREAK returning there like RETURN TO MASTER. (Dot.prg in your CA-Clipper SOURCE\SAMPLE directory illustrates this case.)

Ultimately, you can bracket single statements with a SEQUENCE construct, with RECOVER code for that statement alone. If that statement is an invocation of DBEDIT(), ACHOICE(), or MEMOEDIT(), strategically distributed BREAKs in the called user function can be used to exploit the RECOVER code. Otherwise, there may be no apparent opportunity for a BREAK. How, for example, would you issue a BREAK statement from "within" a USE command? In cases of system errors raised within the statement, however, you can accomplish this using the posted code block and indirect breaking mechanism discussed below.

The Posted Error Block

The error block is a code block written to handle errors. It is distinguished by being *posted*, or installed in a designated memory location. Installation and later retrieval is done exclusively through the ERRORBLOCK() function.

ERRORBLOCK() views the block as data, not code. (The hallmark of code blocks is that they are hybrid: data that happens to contain code, or code that can otherwise be handled like data.) ERRORBLOCK() contains, in effect, a static variable, or slot for storing data accessible to itself alone, but ERRORBLOCK() itself is globally accessible.

Because ERRORBLOCK() can install any code block, any part of the application can dynamically install its own new code.

The posted code block is strategically important because CA-Clipper's unconditional behavior when raising an error is to retrieve it with ERRORBLOCK(), then execute it with EVAL(). When you post an error block, you are specifying code for CA-Clipper to execute next time there is an error condition. You get to write CA-Clipper's response. Also, ERRORBLOCK() and EVAL() let you make your application's response the same as CA-Clipper's when one of your assertions is violated.

To illustrate, consider a deliberately erroneous call to a system function, say VAL(99). You know VAL()'s job is to return a numeric from a string of numerals, so it requires a character type argument. Thus, the expression VAL(99) produces the following screen output:

```
Error BASE/1098  Argument error: VAL
Called from MAIN(1)
```

Errorsys.prg (located in your CA-Clipper SOURCE\SYS directory) contains the source code for the standard error handling routine provided as part of CA-Clipper. If you look at this source code, you will see a procedure named ErrorSys(), which is executed automatically at the outset of every application. It contains a single line:

```
ERRORBLOCK({|e| DefError(e)})
```

This line of code creates a code block that calls a function named DefError() (also defined in Errorsys.prg) and calls ERRORBLOCK() to install that block at runtime. Then, when CA-Clipper raises any error, a call is issued to DefError(). If you look at the source for DefError() (also defined in Errorsys.prg), you can identify the code responsible for the error message. (The parameter that is passed to DefError is an Error object, which will be discussed later in this chapter.)

The VAL() function does not call DefError() directly. It does, however, fetch and execute the currently posted error block in case of an error, as do all other runtime errors. It does not know or care about the code block's contents, who posted it, or how long ago.

If the default posted block is still in place, DefError() will get a call. If, however, you replace the block, you replace the error behavior. For example, if you write a function that rings the bell, you would install it as follows:

```
ERRORBLOCK({|e| MyBellRinger(e)})
```

Then, a call to VAL(99) would ring the bell.

This example uses the basic error block form. The ERRORBLOCK() function installs the code block passed to it, retaining the old block by assigning it to a variable. You then have the options of executing the previously installed block and reinstalling it once your subsystem is no longer in control:

```
// ERRORBLOCK() --> {|e| <SomeExp>}
bOldBlock := ERRORBLOCK({|e| MyErrHan(e)})
// ERRORBLOCK() --> {|e| MyErrHan(e)}
// bOldBlock --> {|e| <SomeExp>}
```

This error block retains, and later executes, its predecessor:

```
bOldBlock := ERRORBLOCK({|e| MyFunc(e), ;
   EVAL(bOldBlock, e)})
```

This technique is the basis for error block *chaining*. The installed block's execution of its predecessor as its "final act" is equivalent to terminating and passing control to the predecessor.

Notice the implications. The code block is built at compile time. Its installation and predecessor's assignment to *bOldBlock*, and its execution, take place at runtime. While the block contains a reference to a variable called *bOldBlock*, nonexistent at build time, that is not a problem as long as *bOldBlock* exists at evaluation. Depending on its storage class, *bOldBlock* may or may not exist at any given moment during runtime. Declaring it as static ensures permanent availability, as demonstrated below:

```
STATIC bPrevHandler
    .
    . <statements>
    .

    bPrevHandler := ERRORBLOCK({|e| NewHandler(e, ;
       bPrevHandler)})
    .
    . <statements>
    .

FUNCTION NewHandler(objError, bMyAncestor)
    IF CanHandle
        .
        . <error-handling statements>
        .
    ELSE
        RETURN EVAL(bMyAncestor, objError)
    ENDIF
```

SEQUENCE vs. Error Block

With SEQUENCE and with error block, you can create two parallel chains of command: a group of code blocks that EVAL() one another in a chain; and a series of nested SEQUENCEs that BREAK from innermost to outermost. It is natural to ask about the interrelationship. (See A Baseline Strategy later in this chapter for more information.)

Error code embodied in an error block, can transfer control to error code in the current RECOVER statement block. However, since code blocks cannot execute statements such as BREAK, there is an equivalent BREAK() function that you use for this purpose.

The following example shows how a system-level runtime error can lead to RECOVER code. The error originates in code where a surrounding RECOVER statement block is pending. Like all system errors, it executes the installed error block; however, because the error block calls the BREAK() function, RECOVER code can receive control:

```
bOldBlock := ERRORBLOCK({|e| IIF(!<lAssert>, ;
    BREAK(e), <exp>), <exp>})
.
. <statements>
.

FUNCTION SomeSubsequentFunc()
    BEGIN SEQUENCE
        .
        . <statements>
        .
        VAL(99)
        // Error arises here
        .
        .
        .
    RECOVER USING e
        .
        . <recovery code>
        .
    END SEQUENCE
    .
    .
    .
    RETURN NIL
```

Calling procedures differ between error blocks and RECOVER clauses. The difference is that you can RETURN from the former but not the latter. An error block executes through the EVAL() function, and RECOVER executes after a BREAK from within a BEGIN SEQUENCE statement. Therefore, EVAL() lets you RETURN as from any other function call, whereas BREAK does not. The BREAK's location is not saved as a return point for future reference. A RETURN from RECOVER goes to the most recent calling function, not the BREAK location. This means that in order to return to a chosen point from a RECOVER, you must write code loops. It also means that the discussion of error protocols below applies only to code block error handlers, not RECOVER clause error handlers.

There are several directions for the error block to go when called.

- It can RETURN to the offending code, but this must be done with care to avoid producing unexpected results.
- It can go to its predecessor error block, by EVAL().
- It can branch to the SEQUENCEs in the main application code using BREAK.
- It can QUIT.

Error Objects

Once you understand the mechanics of invoking error code, you must decide how to use it. That decision depends on the error's characteristics, and the vehicle for characterizing it is the Error object.

An Error object is a package of information with several items, called instance variables, in each package. Most of these items simply describe the error, but others tell the error handler about the error-generating code itself. The error-generating code supplies this package to the error-handling code. In a SEQUENCE construct, the error object is passed as part of the BREAK and received by RECOVER USING. In an error block, it is the *e* parameter that is passed to the code block.

The Error object's role is that of go-between in communication between the error-generating and error-handling code. Communication is initiated by the former and can become a two-way dialog if the latter returns. All this implies a third step in the error-generating process. As mentioned earlier, errors are generated by assertions, which have a two-stage detect-react form:

```
IF .NOT. AstShouldBe   // detect exception
   CALL HANDLER        // BREAK or EVAL(ERRORBLOCK())
ENDIF
```

The third step is preparation of an Error object:

```
IF .NOT. AsItShouldBe
   PREPARE ERROR OBJECT
   CALL HANDLER WITH ERROR OBJECT
ENDIF
```

Preparing the object amounts to calling the ErrorNew() function to generate an empty object, then assigning values to its instance variables.

When your error handler gains control, it has the Error object on which to base a particular response. Typically, it uses a CASE statement. The CASE conditions are based on information about the error conveyed in the instance variables. This is called specialization of error handling, and it is the means by which an Error object is passed among error handlers.

The typical CASE statement provides localized handling for the module that contains it and is probably most frequently invoked by errors that originate within the module. The cases are written accordingly, with those errors in mind.

For example, in an accounting module, you are likely to find both assertions and error handler code that address known possibilities like "inventory fell below zero" and "debits and credits unequal." These reflect the specialized conventions of that module's world and the errors they define, but the error handler can also receive incoming Error objects for errors that fit none of the cases.

These objects normally originate in a called module that propagated them to this error handler. They fall into the error handler's OTHERWISE category, where the error handler can further process them. In a RECOVER USING clause, processing amounts to issuing a BREAK *<objError>* where *<objError>* is the same Error object the handler originally received.

In this way, an error raised but not handled to containment in an inner module is passed back up through higher contexts, each of which screens it and has the opportunity to respond. Ultimately, it reaches the outermost context, the application's initial procedure. There, it experiences a hard landing to the DOS prompt if not bracketed by an outermost SEQUENCE construct. On the other hand, if it finds a considerate containing sequence in the main program (a good practice), a graceful and informative message can soften the blow.

A Model Example

Reconsider VAL(99) for a moment as an illustration of the Error object's role in raising an error. You now know that when the error block is called in response to the VAL(99) runtime error, the parameter passed is an Error object. You can intercept and study its contents by writing a diagnostic UDF that displays instance variables, plus a two line main program that first installs a block to call your UDF then issues VAL(99). When VAL(99) fails, it hands control to the block, which hands it to the UDF, which displays the information on the screen.

In this example, a pseudocode of CA-Clipper's VAL() function demonstrates that you can reproduce system error behavior in your own code:

```
//  Pseudocoded CA-Clipper VAL() function

FUNCTION VAL(p)
   LOCAL objVErr

   // Assertion to verify type consistency
   IF VALTYPE(p) != "C"

      // Make Error object
      objVErr := ErrorNew()

      // Stuff Error object
      objVErr:args          := {p}
      objVErr:canDefault    := .F.
      objVErr:canRetry      := .F.
      objVErr:canSubstitute:= .T.
      objVErr:description   := "Argument error"
      objVErr:filename      := ''
      objVErr:genCode       := 1
      objVErr:operation     := "VAL"
      objVErr:osCode        := 0
      objVErr:subCode       := 1098
      objVErr:subsystem     := "BASE"
      objVErr:tries         := 0
      objVErr:severity      := 2

      // Transfer control and ship object
      RETURN EVAL(ERRORBLOCK(), objVErr)

   ENDIF
   .
   . <the string-to-numeric conversion code>
   .
   RETURN <numeric result>
```

The "System Error" Line-Up: Error.ch

"Argument Error" is a commonplace, predictable error because any function can receive bad parameters from its caller. In fact, this error (along with many others) is so reliably expected, that the CA-Clipper runtime support system raises it, explicitly. The header file Error.ch (located in your CA-Clipper INCLUDE directory) assigns a distinguishing (version-dependent) code to each of these anticipated errors and a corresponding (version-independent) name.

The anticipated errors in Error.ch have fairly obvious names like EG_ARG, EG_NOVAR, and EG_OPEN. EG_ARG is the one just demonstrated; refer to a non-existent variable to raise an EG_NOVAR; and USE a made-up filename to raise EG_OPEN. The names are shorthand for corresponding generic codes, one of which appears in Error:genCode, which Error handlers can query to identify which type of error was raised.

To make the pseudocode above version-independent, you would #include Error.ch and assign Error:genCode as follows, instead of assigning the value 1:

```
objVErr:genCode := EG_VAR
```

As indirectly suggested in the example code, if you raise your own errors you must be thorough. Certain application-level errors you specifically foresee will be the ones for which you place assertions in your code and which you depict in their Error objects.

Carrying Error-Processing Protocols

Error object instance variables supply information about not only the error, but also the code that raised it. After dealing with the error, the handler can return control to the failed code for further purposes, but it generally has no internal knowledge of this code. Therefore, it is up to the failed code to tell the error handler whether it is prepared to regain control, and, if so, to what return values it will respond.

That is the purpose of Error:canDefault, Error:canRetry, and Error:canSubstitute. The failed code may or may not be equipped to: proceed despite the error condition; perform a retry; or accept a substitute for a value it failed to derive. Respectively, the three instance variables signify whether the code is so equipped. If it is, it performs the default, retry, or substitution, contingent on regaining control from an error handler and on the value the error handler returns to it. In this sense, the Error object is a protocol carrier.

A look at the pseudo-VAL() shows that it is unequipped to retry or default, but VAL() can substitute. By setting Error:canSubstitute to true (.T.), the VAL() code is saying, "If you return control and a value, I will return that value to my caller." If, for example, you post a code block that returns "Wacky Wacky," then write a program to print VAL(99), it prints "Wacky Wacky."

```
ERRORBLOCK({|e| SomeHandler(e)})

FUNCTION SomeHandler(objMyErr)
    .
    . <statements>
    .
    IF objMyErr:canSubstitute
       RETURN "Wacky Wacky"
    ENDIF
    .
    . <statements>
    .
```

The pseudo-VAL() shows how to write code that can substitute: it embeds its call to the error block in a RETURN statement. Should the error block ever return to VAL() (which is optional, as demonstrated in DefError()), the value returned by the block becomes the value returned by EVAL().

Similarly, there are conventions for holding a retry dialog (via Error:canRetry) or for plowing ahead in spite of the error condition (via Error:canDefault). Note that even though these two conventions can be used together, neither of them can be used in conjunction with the substitution convention discussed previously. In other words, if Error:canSubstitute is true (.T.), both Error:canRetry and Error:canDefault must be false (.F.).

Retry is possible if the operation that fails is surrounded with a looping structure to let it go back and repeat. This example demonstrates a basic retry structure:

```
//  Error handling protocol conventions
//
lFirstTry := .T.
lRetry := .T.
DO WHILE lRetry
    .
    . <body of operation>
    .
    IF .NOT. AsItShouldBe    // Your assertion
        IF lFirstTry
            obj := ErrorNew()
            obj:canRetry      := .T.
            obj:canDefault    := .T.
            obj:canSubstitute := .F.
            obj:tries         :=  0
            lFirstTry         := .F.
        ENDIF
        obj:tries++
        // Raise the error
        lRetry := EVAL(ERRORBLOCK(), obj)
        IF lRetry
            LOOP              // Causes retry
        ELSE
            .
            . <default code> // Time for plan B
            .
        ENDIF
    ELSE
        lRetry := .F.         // No violation, no retry
    ENDIF
    .
    . <continuation code>
    .
ENDDO
```

If Error:canRetry is true (.T.), the error block can seek to retry the original code. On the other hand, the error code may go through a few motions, perhaps altering the original error condition, then return its request for another try.

When it tries the second time, the code encounters the error condition once again. This time, instead of creating a new Error object, it increments the tries instance variable of the existing one and raises the error again. This can happen any number of times. The error block can test for a specified limit and return false (.F.) in order to invoke plan B.

If default response is indicated, *<default code>* has several options: it can drop out of the IF construct and execute the *<continuation code>*; it can exit, short-circuiting the *<continuation code>*; or it can disrupt continued flow with a QUIT, BREAK, or RETURN. The possibilities are numerous and entirely user-definable.

Note: The EG_PRINT error code (indicating that printing has failed) only sets Error:canRetry.

A Baseline Strategy

A hallmark of CA-Clipper's error mechanisms is versatility. The mechanisms are generalized, so you can adapt them to emulate various approaches to error handling. Typically, CA-Clipper does not provide a particular approach. Instead, it provides tools to implement an approach of your choosing. Therefore, for code integrity, you should adopt certain conventions. Here are some suggestions:

- To provide more than 'say so and quit' response to errors, replace DefError() by posting a generic error block that BREAKs rather than quits.

  Retain the initial, special case code for zero division and network processing that appears in CA-Clipper's DefError(). During programming, the standard DefError() may be desirable, but a user in the middle of a completed application, is less appreciative of cryptic error messages on a DOS screen.

- Place substantive processing of application-specific errors you raise in RECOVER clauses. Deal with them in the application, not in the error block.

- Instead of directly invoking RECOVER using BREAK in your assertions, do so indirectly through EVAL(ERRORBLOCK(), <*object*>). This will work only after following the first guideline, which is to post a generic error block that BREAKs.

Allow subsystems to install their own error blocks freely, while assuring that such a subsystem:

- Preserves a copy of the predecessor block upon entry.
- Restores (reinstalls) that block upon exit.
- Provides a call to the predecessor when done, in case there is an error along the way (that is, to chain).

The block should customize low-level handling, not substantively handle application-level exceptions. For example, it may keep track of more information before using BREAK, or it may intercept additional errors for 'say so and quit' treatment instead of using BREAK. For application-level processing, you should use the RECOVER clause local to the error, thereby heading back up the chain of error blocks to reach the top one, which BREAKs.

Provide OTHERWISE BREAK *<objError>* for those situations in a RECOVER clause you cannot handle through a CASE option. That way, you guarantee that an unrecognized error eventually reaches the outermost context.

Provide an outermost RECOVER that gracefully QUITs. It takes over the job of DefError(), which has been replaced with a block that BREAKs. CA-Clipper's ALERT() function is designed for user interface under error conditions. It is sensitive to the presence or absence of the CA-Clipper full-screen I/O system. If the full-screen system is not present and normal I/O is compromised, ALERT() produces a TTY-style display. Do not use ALERT() in other settings, where it may not produce expected screen output.

Finally, keep the Error object near at hand throughout. All assertions should build Error objects in the first place, and subsequent error code invocations should pass it along using EVAL() or BREAK.

These conventions make the application's error behavior uniform. When an error occurs, whether user-defined or the system's, just call the current error block and you are finished.

A Comprehensive Example

The following application is a bare-bones demonstration of the error handling principles described in this chapter. The example is not all-inclusive but shows how control flows among the error handlers. The error handlers are, essentially, empty (for example, the outermost RECOVER clause does not contain "graceful" QUIT code).

VAL(99), in subsystem 2, functions as a practice error. DefError() is supplanted, in subsystem 1, by a replacement that BREAKs. Each subsystem installs, then removes, its own error handler. Upon installation, the predecessor block is always saved. These error handlers chain. Following execution, each calls its predecessor.

You can discover some useful behaviors by compiling and executing variations of this example. For example, you can move VAL(99) among the subsystems and change the mix of subsystems that do and do not install their own error handlers in the chain.

Notice that the block triggered by VAL(99) is always the most recently installed, not necessarily the one that appears in the same subsystem as the VAL() call. Installing a code block globalizes it, even though, as here, it may call static functions. Similarly, when the error block chain BREAKs, notice which RECOVER clause is called. It is the one that belongs to the innermost SEQUENCE surrounding the BREAK, not necessarily the one that appears in the same subsystem as the BREAK.

When the VAL(99) in subsystem 2 leads to the BREAK that appears in subsystem 1, it is the RECOVER clause of subsystem 2, not of subsystem 1, that takes over. Remove the BREAK and watch DefError() kick back in:

```
// Error subsystem architecture

// First subsystem (.prg)
FUNCTION Main()
    STATIC bOldHandler
    // First Handler
    bOldHandler := ERRORBLOCK({|e| Hndlr1(e, ;
        bOldHandler)})
    CLEAR SCREEN

    BEGIN SEQUENCE
        SecondEntryPoint()
    RECOVER
        ? "Recovery in Main"
    END SEQUENCE

    ERRORBLOCK(bOldHandler)
    RETURN NIL

STATIC FUNCTION Sub1()
    RETURN NIL

STATIC FUNCTION Sub2()
    RETURN NIL

STATIC FUNCTION Hndlr1(obj, handler)
    ? "UDF called by First Handler"
    BREAK
    RETURN EVAL(handler, obj)

//  Second Subsystem (.prg)
FUNCTION SecondEntryPoint()
    STATIC bOldHandler
    // Second Handler
    bOldHandler := ERRORBLOCK({|e| Hndlr2(e, ;
        bOldHandler)})

    BEGIN SEQUENCE
        ThirdEntryPoint()
        ? VAL(99)
    RECOVER
        ? "Recovery in SecondEntryPoint"
    END SEQUENCE

    ERRORBLOCK(bOldHandler)
    RETURN NIL

STATIC FUNCTION Sub2()
    RETURN NIL

STATIC FUNCTION Hndlr2(obj, handler)
    ? "UDF called by Second Handler"
    RETURN EVAL(handler, obj)
```

```
// Third Subsystem (.prg)
FUNCTION ThirdEntryPoint()
   STATIC bOldHandler
   // Third Handler
   bOldHandler := ERRORBLOCK({|e| Hndlr3(e, ;
      bOldHandler)})

   BEGIN SEQUENCE

   RECOVER
      ? "Recovery in ThirdEntryPoint"
   END SEQUENCE

   ERRORBLOCK(bOldHandler)
   RETURN NIL

STATIC FUNCTION Sub1()
   RETURN NIL

STATIC FUNCTION Sub2()
   RETURN NIL

STATIC FUNCTION Hndlr3(obj, handler)
   ? "UDF called by Third Handler"
   RETURN EVAL(handler, obj)
```

What Belongs in RECOVER?

Finally, consider application errors and responses to them. These are the conditions you put in your assertion IFs and your RECOVER CASEs.

Determining what is an error, and an appropriate response, is subjective. The error condition should be judged genuinely exceptional, and the response should not be overly complex. Ensure that the responses anticipate specific problems and are deliberately calculated to address those problems. One response may propagate the error, other responses may unwind a series of discrete events that could not be collectively completed. If your RECOVER code calls dedicated functions, consider making them static since they have no outside applicability.

There is a high degree of subjectivity in error processing. Suppose you have a loop to obtain and process numbers from a source (perhaps it takes an average). Some may be ill-formed (contain character values outside the legal range according to their representational format). Is that an error? It depends why you need the numbers and the derived result. Some results are good only if all prescribed inputs are good. Others can tolerate a percentage of throw-away inputs.

A bank, calculating next year's interest rate on a customer's variable mortgage must factor in all 52 weeks of the contractually specified government index values. Omission or inaccuracy of a single value invalidates the whole result. On the other hand, tabulation of survey responses may require a result accurate within a certain margin of error. That may be statistically obtainable even if 10% of gathered responses are omitted.

The following pseudocode characterizes such typical subjectivity:

```
FUNCTION GatherValues()

    BEGIN SEQUENCE
        DO WHILE
            <obtain next value>
            IF <ill-formed>
                BREAK indication-of-error
            ENDIF
            <process value>
        ENDDO
        RETURN result

    RECOVER USING indication-of-error
        CASE this-particular-error
            <clear up>
            QUIT

    END SEQUENCE
```

Beyond raising the error, even if that is necessary, QUIT may be overkill. That requires a subjective decision too. Your source might be an interactive user or a mainframe tape. If a user types "T" instead of "6," it is worth requesting the number again, but if a tape has an unreadable scratch and 100 retries have already failed, another request is futile. The correct response is context-dependent.

Whatever your response, keep it in bounds. Do not let error code proliferate into a parallel application. Remember, localization is the key. Also, do not use the error system to process conditions that are not errors (that is, conditions that are not genuinely exceptional.) By definition, errors are abnormalities.

The following example illustrates the difference between true errors and other conditions. It shows two operations, either of which could be accomplished using the error system or traditional control structures. The first advances a day-of-the-week array. It experiences a system error whenever the current day is Saturday. The second divides two numbers. It experiences an error whenever the divisor is zero.

Either function could raise the error, address it reactively through RECOVER, or avoid the error by anticipating the boundary. While recovery is preferred for division by zero, it is discouraged for day of the week. The former is a mathematical aberration; trying it is a mistake. The latter is a normal boundary condition:

```
//  Abnormalities only, please!
STATIC TheDays := {"Sunday", "Monday", "Tuesday", ;
   "Wednesday", "Thursday", "Friday", "Saturday"}
STATIC i := 0

FUNCTION NextDay()
   LOCAL retval
   BEGIN SEQUENCE
      retval := TheDays[++i]
   RECOVER USING --
      retval := TheDays[i := 1]
   END
   RETURN retval
```

Since, however, this is not really an error condition, it is preferable to code the NextDay() function as follows:

```
FUNCTION NextDay()
   LOCAL retval
   IF i != 7
      retval := TheDays[++i]
   ELSE
      retval := TheDays[i := 1]
   ENDIF
   RETURN retval
```

Now, the Divide() function is shown both ways, but in this case the first, which uses a SEQUENCE construct to handle the error, is preferable:

```
FUNCTION Divide(x, divisor)
    LOCAL retval
    BEGIN SEQUENCE
        retval := x/divisor
    RECOVER USING --
        retval := 0
    END
    RETURN retval

// Alternative, if division by zero was not
// considered an error
FUNCTION Divide(x, divisor)
    LOCAL retval
    IF divisor != 0
        retval := x/divisor
    ELSE
        retval := 0
    ENDIF
    RETURN retval
```

The responses in RECOVER represent conscious anticipation of a specific set of error possibilities, as discussed above and illustrated in the first example. They are not haphazard or catch-all.

Propagation is a common option for errors that fall into the OTHERWISE category. It is also a possibility for those that do not, once desired processing is complete.

Error response can unwind; this is the basis for transaction processing.

The following example outlines unwinding. It also shows the technique for re-executing the SEQUENCE statement block and demonstrates the SEQUENCE construct as the enclosure for coherent, logical units of activity, not random statement groups. The activity is foreseen to have certain, predictable outcomes that call for remedy, and the RECOVER clause addresses those.

This example attempts to perform an entire group of updates on a database. If a problem arises, say with the fourth update, the RECOVER code undoes the successful updates and starts over:

```
//  Unwinding and restarting
//
#command RESTART SEQUENCE => LOOP

DO WHILE .T.
    BEGIN SEQUENCE
        .
        . <do 3 out of 5 things>
        .
        IF <can't do 4th>
            BREAK--
        ENDIF
        .
        . <do 5th>
        .
    RECOVER USING--
        .
        . <undo those 3 things>
        .
        IF HaveAnotherGo
            // Restarts SEQUENCE block as a whole
            RESTART SEQUENCE
        ENDIF
        .
        .
        .
    END SEQUENCE
    EXIT
ENDDO
```

Assuming that the installed error block eventually BREAKs, as discussed previously, your five updates are guaranteed if you drop out the SEQUENCE construct. This is because an error internal to any of the update operations (for example, REPLACE) triggers the error block, which will BREAK to the RECOVER code because all update operations are bracketed in the SEQUENCE construct. Thus, the explicit BREAK for the fourth update shown in the example is not the only way to invoke the recovery mechanism. In the event of a BREAK from the error block, you would back all successful updates out of the database (the error block helps you identify them) and try again.

Network Processing

The CA-Clipper function NETERR() functions differently from earlier versions of CA-Clipper. Two events that in version Summer '87 automatically set NETERR() to true (.T.), no longer do so. They are failure of a USE command due to an "access denied" from the underlying DOS file open call and failure of APPEND BLANK due to the "phantom" record being locked by another workstation, perhaps by simultaneous APPEND BLANK attempts. (The phantom record is just beyond the last record.)

While USE and APPEND BLANK do preset NETERR() to false (.F.), upon failure they merely call the current error block without further attention to NETERR(). Setting NETERR() to true (.T.) is left to the error block—the commands themselves no longer contain code to do so. The default error block, DefError(), takes care of this, so you do not see a difference in the way network code operates from one version to another:

```
// For network open error, set NETERR()
// and subsystem default
IF (e:genCode = = EG_OPEN .AND. e:osCode = = 32 .AND.
;
   e:canDefault)
   NETERR(.T.)
   RETURN .F.
ENDIF

// For lock error during APPEND BLANK, set NETERR()
// and subsystem default
IF (e:genCode = = EG_APPENDLOCK .AND. e:canDefault)
   NETERR(.T.)
   RETURN .F.
ENDIF
```

Note: Formerly, you could not set NETERR(), but in CA-Clipper you do it by passing a true (.T.) or false (.F.) argument to the function.

If you supplant DefError(), you should include similar code in its replacement; otherwise, network activity that hinges on NETERR() will change behavior.

Also note that because USE and APPEND BLANK have indicated their readiness to regain control with e:canDefault true (.T.), DefError() returns false (.F.) in order to allow the original operations to attempt default processing. In both cases, the original operation does nothing visible (that it, the file is not opened and the record is not added). It probably does little but return, but that allows mainline code to retain control and continue. DefError()'s returning false (.F.) merely allows the operation, and by extension, the ensuing code, to run to completion.

On another network issue, CA-Clipper now distinguishes between the two causes (rooted in the DOS function that opens files) of "access denied" that can occur when opening a file. One failure is the result of other workstations' use of the file; the other, of attributes of the file itself. The DOS file open function returns the same error code in both cases, namely 5, obscuring any distinction between causes; however, a secondary error code is available from DOS which does make the distinction. CA-Clipper obtains this secondary code and returns it in Error:osCode. In a clash with other workstations, the result is code number 32, entitled "sharing violation," as opposed to number 5, "access denied." In a network, 32 indicates inter-workstation contention while 5 reflects rights and attributes issues.

Equipped with tools and understanding, you can now enhance your applications with error processing. You will not seek to make them error-proof, but to make them error-systematic. You can raise abnormal conditions as errors, as supplied CA-Clipper functions do. You can provide localized, specialized handling in response. The approach is user-definable and the tools belong to you.

Transact.prg

The example below illustrates the use of CA-Clipper error features in a network environment, including the implementation of the transaction/rollback concept and the restart/retry of a block of statements.

The illustrated transaction supposes a one-to-many relationship between two data files. The transaction consists of applying deletes to all of the child records that correspond to a given parent record. (In this case, all the invoices associated with customer Jones.) If the deletion of any child record fails, rollback is the reversal (that is, RECALL) of those deletions that succeeded.

The installed error block embodies the recommendation that error block code eventually lead to a BREAK, so that local code can be accompanied by recovery specific to local activity. Because of this, all errors reliably transfer control to that recovery. (The error block shown here accomplishes this artificially. Moreover, it is the highest-level block in a possible chain that would perform the BREAK upon which this code would rely.)

Note that there is separate recovery for the startup code and for the multiple-delete code. If an error arises in either context, execution of the localized RECOVER code is due to the same error block, which eventually BREAKs. Once posted, the error block does not "belong" to the routine that posted it but to all routines that execute it while it remains installed:

```
// Transact.prg
//
// Compile with /N/W/A
// Must be linked with Locks.obj to run

#include "Error.ch"

PROCEDURE Main()

    LOCAL aDels, cName, aNots, objErr, cMsg
    FIELD LastName

    // Artificial but prototypical
    ERRORBLOCK({|e| BREAK(e)})

    // Open files, set relation
    IF .NOT. GetReady()
        ? "No, I don't feel like it."
        QUIT
    ENDIF
```

```
IF .NOT. Invoice->(DBSEEK(cName := "Jones"))
   ? 'This example is hard-wired for "Joneses."'
   QUIT
ENDIF

// Parent record lock, gateway to its children
IF Customer->(RecLock(5))

   // Allow for possible retry
   DO WHILE .T.
      BEGIN SEQUENCE
         // Transaction--multiple deletes
         aDels := {}
         DO WHILE TRIM(Invoice->(LastName)) == ;
            "Jones"
            // If unsuccessful, calls
            // error block
            Invoice->(RLOCK())
            Invoice->(DBDELETE())
            // Records deletions already done
            AADD(aDels, Invoice->(RECNO()))
            Invoice->(DBSKIP())
         ENDDO

      RECOVER USING objErr

         IF (objErr:genCode = EG_UNLOCKED)
            // Rollback--recall deletions
            DO WHILE .NOT. EMPTY(aDels)
               aNots := {}
               AEVAL(aDels, {|nRec| ;
                  UnDeleteIt(nRec, aNots)})
               aDels := ACLONE(aNots)
            ENDDO

            cMsg := "Rollback successful. " + ;
               "Restart transaction?"
            IF ErrBox(cMsg, {"Yes", "No"}) = 1
               // Reposition pointer
               Invoice->(DBSEEK(cName := ;
                  "Jones"))
               // ...and start over
               LOOP
            ENDIF
         ENDIF
      END SEQUENCE
      EXIT

   ENDDO
   Customer->(DBUNLOCK())
ENDIF

RETURN
```

```
STATIC PROCEDURE UnDeleteIt(nRec, aNots)
   Invoice->(DBGOTO(nRec))          // either...
   IF Invoice->(RLOCK())
      Invoice->(DBRECALL())         // recall works
      Invoice->(DBUNLOCK())
   ELSE                             // or...
      AADD(aNots, nRec)             // mark for revisit
   ENDIF
   RETURN

STATIC FUNCTION GetReady()
   /*
   Illustrates use of error system to treat network
   file "acquisition" as a transaction.  Dispenses
   with Summer '87 NETERR() check after USE attempt
   as a result.  Streamlines code and makes it more
   readable.

   Need to accomplish 3-steps-in-1, or nothing:
   open 2 files and indexes, and set a relation.
   That defines the transaction.  Relies on
   knowledge that error block will predictably
   BREAK.  Therefore, if any part of the main
   sequence of statements fails the RECOVER
   statement block that gets control can undo the
   parts that did not.  RETURN value reflects
   overall result.

   */

   LOCAL lReturn := .T.
   CLEAR SCREEN
   BEGIN SEQUENCE
      USE Invoice  INDEX Invoice  SHARED NEW
      USE Customer INDEX Customer SHARED NEW
      SET RELATION TO Customer->LastName ;
         INTO Invoice

   // for ANY error; don't grab object,
   // don't check why
   RECOVER
      CLOSE SELECT("Invoice")
      CLOSE SELECT("Customer")
      lReturn := .F.
   END SEQUENCE
   RETURN lReturn
```

```
FUNCTION ErrBox(cErrMsg, aOptions)
   LOCAL nChoice, i, nStrgLen,
   LOCAL cOldColor, cOldScreen

   cOldColor := SETCOLOR("W/R+, W+/B")
   cOldScreen := SAVESCREEN(10, 0, 14, 79)

   @ 10, 0 TO 14, 79 DOUBLE
   @ 11, 1, 13, 78 BOX SPACE(9)
   @ 11, 2 SAY cErrMsg

   nStrgLen := LEN(aOptions) * 3
   AEVAL(aOptions, {|arr| nStrgLen += LEN(arr)})
   nStrgLen := (78 - nStrgLen) / 2

   FOR i := 1 TO LEN(aOptions)
      @ 13, nStrgLen PROMPT aOptions[i]
      nStrgLen += LEN(aOptions[i]) + 3
   NEXT

   nChoice := 1
   MENU TO nChoice
   SETCOLOR(cOldColor)
   RESTSCREEN(10, 0, 14, 79, cOldScreen)

   RETURN nChoice

// EOF - Transact.prg  //
```

Chapter 10
CA-Clipper Compiler—CLIPPER.EXE

In This Chapter

This chapter describes the basic operations of CLIPPER.EXE, the full-featured and versatile CA-Clipper compiler. It can be used to compile a single program file (.prg) or to compile several files at once by taking its input from a script file. The result is an object file (.OBJ) that can be linked with other CA-Clipper-compiled and foreign object files to form an executable file (.EXE).

CA-Clipper has been upgraded to Microsoft C 8.0. This upgrade switches from alternate to emulator floating point support, enabling CA-Clipper to use a math coprocessor if present. With this new release, you still get all the functionality of release 5.2 features (that is, command line compiling and linking, DBU, and so on), so you can mix and match the tool set as your needs require.

The following topics are discussed in this chapter:

- Invoking the CA-Clipper compiler
- Using CLIPPERCMD
- Compiler options
- The compiler script file
- The compiler return code

- How the CA-Clipper preprocessor works
- How CA-Clipper compiles
- The compile and link batch file
- Header files
- Output files
- Changing the size of the environment

Note: The compiler requires a minimum 25 file handles. This requires DOS 3.3 or greater and FILES=25 in CONFIG.SYS.

Invoking the CA-Clipper Compiler

To execute the CA-Clipper compiler from DOS, use the following syntax:

```
CLIPPER [<sourceFile> | @<scriptFile>
    [<option list>]]
```

<sourceFile> is the name of the program file to compile to an object file. If no extension is specified, (.prg) is assumed. The filename can optionally include a drive designator and a path reference.

<scriptFile> is the name of a script file containing a list of source files to compile into a single object file. If an extension is not specified, (.clp) is assumed.

<option list> is a list of one or more options to control the course of the compilation. All available options are discussed under Compiler Options later in this chapter. To get a list of options, specify the compiler command line with no arguments.

The following compiler command line checks the syntax of Main.prg only, without displaying line numbers:

```
CLIPPER Main /S /Q /M
```

The following compiler command line compiles all files in the Account.clp script file. The options in this case cause debugging information to be included in the object file, which will be placed in the C:\CLIP53\OBJ directory and given the name ACCTPAY.OBJ:

```
CLIPPER @Account /B /OC:\CLIP53\OBJ\ACCTPAY
```

The script file, Account.clp, is shown below:

```
AccMain
AccPay
AccRec
AccRep
```

Using CLIPPERCMD

In addition to specifying options on the compiler command line, you can specify them using the CLIPPERCMD environment variable. To define CLIPPERCMD, use the DOS SET command as follows:

```
SET CLIPPERCMD=[<option list>]
```

<option list> is a list of one or more options that will be read and processed each time the compiler is invoked. You specify these options just as you would on the compiler command line.

To save yourself from having to enter this SET command repeatedly, you can place it in your AUTOEXEC.BAT file where it will be processed automatically each time you reset your computer.

The following example causes CA-Clipper to suppress line numbers and use a new standard header file each time the compiler is invoked:

```
SET CLIPPERCMD=/L /UNewstd.ch
```

The next example specifies a directory in which to place output object files:

```
SET CLIPPERCMD=/OC:\CLIP53\OBJ\
```

Compiler Options

Compiler options are switches that control compilation behavior. Besides the preprocessor directives and the standard header file, compiler options are the main mechanism for controlling the compiler and preprocessor.

As already discussed, you can specify options on the CA-Clipper command line and in the CLIPPERCMD environment variable. Command line options take precedence if there is a conflict.

All options are shown in uppercase preceded by a slash (/). Note, however, that options are not case-sensitive and that you can replace the slash with a dash (-) if you prefer.

Some compiler options have arguments. If an option has arguments, specify them after the option, with no space between the option and its first argument.

/A

Any variable included in a PRIVATE, PUBLIC, or PARAMETERS statement is automatically declared as MEMVAR.

/B

Includes line numbers, local and static variable names, source filenames, and other debugging information in the object file. If /L is also specified, line numbers are not included as part of the debugging information.

Using the /B option causes the size of the resulting object file to increase. The amount of the size increase depends on the number of lines in the source code file as well as the number of local and static symbols defined. The number of additional bytes is approximately (4 + 3 per line + size of each static and local symbol + size of source filename symbol). Note that each symbol size includes a null terminator byte (for example, the symbol Cust_Name requires 10 bytes) and that the 3 bytes per line figured into this size increase is eliminated by the use of the /L option.

/D

/D<*identifier*>[=<*text*>]

Defines an identifier to the preprocessor. If <*text*> is not specified, <*identifier*> is given an empty value. Note that if double quote marks are used in the <*text*> as literal characters, they must be preceded by the backslash (\) character.

This option is designed for use with the conditional compilation directives to allow you to define a manifest constant and control whether a section of the source file is compiled. Refer to the #ifdef and #ifndef directives in the *Reference Guide* for more information on conditional compilation.

/ES

/ES[<*severityLevel*>]

Specifies the severity level of warnings.

If <*severityLevel*> is 0 or omitted, the compiler does not set the DOS ERRORLEVEL upon exit if warnings are encountered. This is the default behavior of the compiler.

If <*severityLevel*> is 1, the compiler sets the DOS ERRORLEVEL upon exit if warnings are encountered but still generates an .OBJ file.

If <*severityLevel*> is 2, the compiler sets the DOS ERRORLEVEL upon exit if warnings are encountered and does not generate an .OBJ file. This effectively promotes warnings to error status.

/I

/I<*pathName*>

Adds the directory specified by <*pathName*> to the front of the list of directories to be searched for header files (as specified by the INCLUDE environment variable). Multiple /I options can be specified in the same compiler session; each one causes an additional path to be added to the front of the header file directory search list.

/L

Excludes program source code line numbers from the object file. The effect of this option is to reduce the object file size by three bytes for each line containing a program statement.

It should be used only on completed programs since its use prevents the reporting of source code line numbers when there are runtime errors. Additionally, a program compiled using the /L option cannot be debugged.

If both /L and /B are specified, line numbers are not included in the debug information written to the object file.

/M

Compiles only the current (.prg) file, suppressing the automatic search for (.prg) files to satisfy unresolved external references. If this option is specified, procedures referenced with a DO statement, SET FORMAT, or SET PROCEDURE commands and not found in the current (.prg) are assumed external.

In addition, procedure files specified by SET PROCEDURE commands are not compiled. Header files specified with the #include directive, however, are always compiled, regardless of the /M option.

/M has no effect when compiling with a script file.

/N

Suppresses the automatic definition of a procedure with the same name as the (.prg) file. This option must be used if you have filewide variable declarations in a program (.prg) file.

If the /N option is not specified, each (.prg) file is compiled with an implicit procedure consisting of all code from the top of the program file to the first PROCEDURE or FUNCTION declaration statement.

The first procedure to execute when you invoke an .EXE file, also referred to as the *starting procedure*, is the first procedure or function encountered at link time. When you compile without the /N option, the implicit procedure serves as the startup procedure. When you compile with the /N option, the first explicitly declared procedure or function in the first object file listed at link time becomes the startup procedure.

/O

/O<*objFile*>

Writes the output object file to <*objFile*>. If <*objFile*> contains only a path specification, it must end with a backslash (\) character. The file is written to the specified directory with a base name the same as the source or script filename specified on the compiler command line, but with an .OBJ extension.

/P

Produces a preprocessed output listing. The listing is written to a file with the same name as the source or script filename specified on the compiler command line, but with a (.ppo) extension. The location and name of this file is unaffected by the /O option.

/Q

Prevents line numbers from displaying on the screen when the compilation is in progress. Note that line numbers are still written to the object file unless /L is also specified.

/R

/R[<*libFile*>]

Embeds in the object file a request to the linker to search <*libFile*> for unresolved external references. Multiple /R options for the same compiler session are additive, causing the name of each library referenced to be embedded.

By default, CA-Clipper embeds a request for CLIPPER.LIB, EXTEND.LIB, DBFNTX.LIB, and TERMINAL.LIB in the object file. Specifying /R without a <*libFile*> causes no libraries to be embedded, thereby overriding this default request.

/S

If specified, the syntax of the current (.prg) file and all referenced source code files is checked, but no object file is generated. To check the syntax of the current program file only, use the /M option in conjunction with /S.

/T

/T<*pathName*>

Specifies a directory for temporary files generated during compilation. If this option is specified, it overrides the directory setting specified in the TMP environment variable.

/U

/U[<*userStandardHeaderFile*>]

Identifies an alternate standard header file to preprocess in place of the supplied STD.CH which is used automatically. If the /U option is specified without a <*userStandardHeaderFile*>, no standard header is used.

The user-defined standard header file is a replacement for STD.CH and, therefore, cannot contain anything other than preprocessor directives. CA-Clipper uses the same searching order for this header file as it does for all other header files: the current directory, the /I directory, and the INCLUDE path.

/V

Forces the compiler to assume that all references to undeclared or unaliased variable names are public or private variables. The default is to treat ambiguous references as fields. The /V option has the same effect as using the dynamic variable alias (MEMVAR->).

/W

This option generates warning messages for undeclared or unaliased variable references. It is useful when converting private and public variables to locals and statics.

/Z

This option suppresses shortcutting optimizations on the logical operators .AND. and .OR. It is provided as an aid to isolating code that depends on the behavior of older versions of CA-Clipper. Note that code executed using the runtime macro operator (&) is always optimized with shortcutting—optimization cannot be disabled in the macro system.

The Compiler Script File

A script file (sometimes referred to as a *clip list*) is a text file containing a list of source files to compile into a single object file. The resulting object file has the same name as the script file unless the /O option is specified.

The list can include program, procedure, and format files. Referenced files (including procedure files) are not automatically compiled as they are if the application is compiled on the CA-Clipper command line. This is true regardless of whether the /M option is specified.

Use the following rules to create a script file:

- Source files are separated by a carriage return/linefeed.
- Source files can be specified with or without an extension. If not specified, a (.prg) extension is assumed.
- Drive designators and path references can also be specified as part of any source filename.

The Compiler Return Code

If a fatal error is encountered or the user presses Ctrl+C or Ctrl+Break, the compiler terminates with a DOS return code of 1. If the compilation ends normally, the return code is set to 0. Whether or not warnings set the return code is controlled by the /ES compiler switch, as discussed in Compiler Options earlier in this chapter.

You can test the DOS return code in a batch file using the DOS ERRORLEVEL keyword. A good example of this can be found in the file CL.BAT located in your CA-Clipper BIN directory (this batch file is discussed more thoroughly in The Compile and Link Batch File later in this chapter.)

If an error occurs, an error message is displayed and no output file is generated. For a list of error messages and their meanings, see the "Compiler Error Messages" chapter in the *Error Messages and Appendices Guide*. To direct these messages to a file, use DOS redirection. For example, the following compiler command line directs the compiler output to a file called ERRFILE.TXT:

```
CLIPPER <sourceFile> > ERRFILE.TXT
```

How the CA-Clipper Preprocessor Works

The CA-Clipper compiler has two distinct phases: the preprocessor phase and the compilation phase. The portion of the compiler engine that performs the preprocessor phase is called the preprocessor, and the portion that performs the compilation phase is called the compiler.

Before the compilation phase takes place, the preprocessor scans the source file from top to bottom for certain directives and translates them into regular CA-Clipper source code that can be compiled. The output of the preprocessor is then used as input to the compiler.

The /P compiler option can be used to write the preprocessor output to disk so that you can see the source code that was used as input to the compilation phase. This option is especially useful if you have used the #command and #translate directives to create user-defined commands. You can look at the contents of the (.ppo) file to see whether or not commands translated as you expected.

The following table summarizes the preprocessor directives. For more information on their usage, see the *Reference Guide*.

Directive	Meaning
#command	Specify a user-defined command or translation directive
#define	Define a manifest constant or pseudofunction
#error	Generate a compiler error and display a message
#ifdef	Compile a section of code if an identifier is defined
#ifndef	Compile a section of code if an identifier is undefined
#include	Include a file into the current source file
#stdout	Write text to the standard output device
#undef	Remove a #define definition
#xcommand	Specify a user-defined command or translation directive without abbreviations

How CA-Clipper Compiles

The CA-Clipper compiler converts source code written in CA-Clipper into Intel object code files. The source code programs are identified by the DOS filename extension (.prg). Object code files are identified by the DOS filename extension (.OBJ). These object files can be linked with a linker to produce an executable file, identified by the DOS filename extension (.EXE).

Unless a script file or the /M option is specified, CA-Clipper compiles the specified program file and all program files referenced with DO, SET PROCEDURE, and SET FORMAT. Within each referenced file, any other files referenced with these commands and statements are also compiled until the entire system of source files is processed. This proceeds according to the following rules:

- The program specified on the compiler command line is compiled first.
- Procedure files, procedures referenced with DO, and format files referenced in the current program are compiled in the order specified.
- Procedure references made with DO and not already compiled are compiled if found as (.prg) files in the current directory. If not found, CA-Clipper returns a warning message.
- Format references made with SET FORMAT and not already known are assumed to have a (.fmt) extension unless otherwise specified. If a procedure with the same name as a format reference has already been compiled, the format will refer to the compiled procedure and not to the (.fmt) file on disk.

If a script file is used, each file is compiled in the order listed according to these rules with one exception—no referenced files are compiled.

Note: A DO requires that the procedure be known or available in the current directory as a file of the same name with a (.prg) extension. If the procedure is called with function-calling syntax and not known, CA-Clipper does not search the disk for a program file of the same name, but instead assumes the procedure is external and proceeds without any warning message.

The Compile and Link Batch File

The default installation of the CA-Clipper development system places a sample compile and link batch file, CL.BAT in your CA-Clipper BIN directory. This batch file can be used to compile and link single program files (.prg) to executable files (.EXE) only if there are no errors during compilation. If there is a compiler error, CLIPPER.EXE terminates with a return code of 1, and the batch file in turn terminates without attempting to link.

The compiler batch file is invoked as follows:

```
CL <sourceFile>
```

<sourceFile> is the name of the program file to compile and link. The filename can be specified including both a drive designator and/or a path reference. However, specifying *<sourceFile>* with an extension will cause a linker error.

Header Files

Header files, also referred to as *include files*, contain preprocessor directives and command definitions. Header files have no default extension and are specified using the #include preprocessor directive or the /U compiler option. The INCLUDE environment variable is used to locate header files and is defined using the DOS SET command as follows:

```
SET INCLUDE=<pathSpec>
```

The INCLUDE path is searched if a requested header file cannot be found in the current directory.

In the default installation configuration, INCLUDE is set to \CLIP53\INCLUDE. To change this default, alter the SET INCLUDE command in your AUTOEXEC.BAT file. To search for header files in a particular directory on the D: drive, you might change it to the following:

```
SET INCLUDE=D:\INCLUDE
```

Note: When searching for header files, CA-Clipper searches the directory specified with the /I compiler option after the current directory and before the INCLUDE path. In a sense, this option adds the specified directory to the front of the INCLUDE path specification.

Output Files

There are three types of output files produced by the CA-Clipper compiler: object files, preprocessed output files, and temporary files. Any specified output filename can include a full or partial path name to explicitly indicate where CA-Clipper will write the file on disk. Unless otherwise specified, files are created in the current directory.

Default extensions for output files, with the exception of preprocessor output files, can be overridden with explicit extensions specified as part of the filename.

Object Files

A single object file is created as a result of a successful compile. By default, the name assigned to the file is the same as the source or script filename specified on the compiler command line, but with an .OBJ extension. You can, however, specify the object filename and destination with the /O compiler option.

Temporary Files

Normally, CA-Clipper generates one or more temporary files during the compilation process. You can control where these files are created by setting up an environment variable called TMP with the DOS SET command as follows:

```
SET TMP=<pathName>
```

The compiler normally deletes temporary files after they have served their purpose. Therefore, you may never see them on your disk. The main purpose of the TMP variable is to write the temporary files to another disk drive if you are running short of space on the drive that you normally use to compile. This way, the temporary files will not take up disk space that is better used by permanent files.

For example, if you are working on the C: drive, you could put temporary files in a directory on another disk drive as follows:

```
SET TMP=D:\TEMP
```

Note: The /T compiler option can be used to specify a temporary file directory. If this option is specified, it overrides the TMP directory setting.

Preprocessed Output Listing

The /P compiler option allows you to write a preprocessed output listing to a file. The name assigned to the file is the same as the program or script filename specified on the compiler command line with a (.ppo) extension.

Changing the Size of the Environment

When the environment variables, CLIPPERCMD, INCLUDE, or TMP are specified, it is possible to run out of environment space. In DOS version 3.2 and above, you can change the size of the environment by loading COMMAND.COM with the SHELL directive in CONFIG.SYS.

For example, the following CONFIG.SYS command line sets the environment size to 2048 bytes:

```
SHELL=C:\COMMAND.COM C:\ /P /E:2048
```

Note that in addition to specifying the /E option to define the environment space, you must give the name and location of COMMAND.COM as well as force COMMAND.COM to remain resident with the /P option.

Chapter 11
CA-Clipper Protected Mode Linker—EXOSPACE.EXE

In This Chapter

Welcome to CA-Clipper/Exospace, the protected mode linker and DOS extender for CA-Clipper. This chapter provides all the information you need to link your applications with CA-Clipper/Exospace and explains the benefits of doing so. The following topics are discussed:

- Overview of CA-Clipper/Exospace
- Invoking CA-Clipper/Exospace
- The CA-Clipper/Exospace return code
- The compile and link batch file
- Output files
- How CA-Clipper/Exospace searches for files
- Linker command reference
- Troubleshooting

Overview of CA-Clipper/Exospace

CA-Clipper/Exospace allows your CA-Clipper applications to run in protected mode, thereby eliminating the DOS 640 KB barrier. In protected mode, extended memory is directly available without requiring swapping of any kind. This in itself would be exciting enough, but it also has a number of other important implications.

Say Good-bye to Load Size

Load size concerns are a thing of the past. CA-Clipper/Exospace applications can be loaded in as little as 256 KB (or less) of low DOS memory. As long as enough extended memory is available, your application will run fine. You will have no problems even if available DOS memory is being squeezed by network drivers and the like. Now, you can even link and run your CA-Clipper applications while shelled out from another CA-Clipper application or a memory-hungry editor!

No Overlaying Worries

Similarly, overlaying of code is no longer an issue. You do not need to worry at all about which code to overlay or not to overlay—just specify all your .OBJ and .LIB files, and link! Overlaying of code occurs automatically when your application is running—but it is completely automatic and transparent, and you never need to worry about it.

Although overlaying occurs automatically if necessary, if you have enough RAM, your .EXE will be loaded into memory—as it is used—until the entire .EXE is resident in RAM. Previously, overlaid code had to be swapped in and out of a relatively small overlay area as it was needed, so a large application could never be completely resident in RAM. This had obvious performance implications.

Industry Standard Virtual Memory

CA-Clipper/Exospace replaces CA-Clipper's real-mode DOS/Expanded Memory Specification (EMS)-based Virtual Memory Manager with a robust, transparent, protected-mode virtual memory engine developed by Rational Systems, a leader in protected-mode technology. This same technology is currently used by Lotus, IBM, Quarterdeck, Borland, Symantec, Watcom, and hundreds of others, and has shipped millions of copies of commercial applications since 1987.

Improved Performance

Without all the swapping that CA-Clipper was forced to do within the 640 KB DOS limit, the performance of many applications will improve visibly and substantially, depending on processor and memory configurations.

Resurrect Your 286 PCs

CA-Clipper/Exospace breathes new life into 80286-based machines. With 2 MB or more of RAM, many 286 machines will now run CA-Clipper applications faster than ever before and will be able to run applications that they may not have been able to run before because of restricted DOS memory.

What About 386 and 486 Machines?

Applications running on 386 and 486 machines also benefit, of course. Instead of continually having to swap data from EMS or having to load overlaid code from disk, applications can now use all of a machine's memory directly. Swapping only occurs when all available memory is exhausted, instead of when the first 640 KB is full. In effect, CA-Clipper/Exospace raises the 640 KB RAM ceiling to include all available memory. This will benefit almost any application on any supported machine.

Protected Mode = Protected Code!

Protected mode, as its name implies, provides a greater level of protection to your applications. C or Assembler code that has not yet been fully debugged, whether it is in a third-party library or your own, will usually generate an exception error, detected directly by the CPU, as soon as it does something wrong, rather than corrupting memory and leading to mysterious and difficult-to-trace problems later on. This makes it easier to find the source of the problem, and if a third-party library is the source of a problem, it will usually be clear which library is at fault.

Invoking CA-Clipper/Exospace

To execute CA-Clipper/Exospace from DOS, use the following syntax:

```
EXOSPACE [@<lnkFile>] [<command list>]
```

If you specify no arguments, CA-Clipper/Exospace displays a brief help screen.

Script Files

<lnkFile> is the name of an ASCII text file called a *script file* from which the linker takes some or all of its input. If you specify the file without an extension, .LNK is assumed.

You can optionally specify a drive designator and/or a path reference for the script file. See How CA-Clipper/Exospace Searches for Files later in this chapter for information on how an input file is located if its filename is unqualified.

You can use script files to automate linker sessions that you tend to repeat and sessions that are too long to enter from the command line—the DOS command line limit of 128 characters may be too restrictive for some linker sessions, necessitating the use of a script file.

The only rule governing the format of a script file is that it contain valid linker commands. You need not place the commands one to a line. The new line character, like a space or a tab, is considered to be white space. Therefore, if a command will not fit on one line, continue it on a new line by pressing Enter—no continuation character is necessary.

The # character designates a comment. When encountered, the rest of the line is ignored.

You can nest script files using the @*<lnkFile>* syntax within a script file. The named script file is read and executed by CA-Clipper/Exospace in its entirety. When it terminates, control returns to the current script file.

Note: CA-Clipper/Exospace supports script files for both .RTLink and Blinker. Simply use your existing script files as input to CA-Clipper/Exospace without modification (see the Linker Command Reference later in this chapter for more information). If, on the other hand, you use another linker, you will need to modify it to use CA-Clipper/Exospace commands.

Linker Commands

<command list> is a list of linker commands separated by spaces. Linker commands on the command line either override or augment the same commands in a script file. For example, a FILE command on the command line adds to any FILE commands in the script file, while an OUTPUT command overrides any OUTPUT commands in the script file. See the Linker Command Reference later in this chapter for details on the syntax and behavior of specific commands.

To invoke the linker, you must specify at least one FILE command to tell CA-Clipper/Exospace which object files you want to link. You can specify this command either on the command line or in a script file. If you attempt to invoke CA-Clipper/Exospace with command line arguments and do not specify a FILE command, the linker session will result in an error.

This example links a single object file, HELLO.OBJ. The result is the executable HELLO.EXE:

```
EXOSPACE FILE HELLO
```

The next example links the DBU utility, assuming you are located in the \CLIP53\SOURCE\DBU directory. The result is DBU.EXE:

```
EXOSPACE @DBU
```

A simplified version of the DBU.LNK file is listed below:

```
FILE DBU, DBUUTIL, DBUHELP, DBUVIEW, DBUINDX,
   DBUEDIT, DBUCOPY, DBUSTRU, DBUNET
```

What EXOSPACE.EXE Does

While CA-Clipper/Exospace links your application for you, its primary function is not that of a linker. The most important tasks CA-Clipper/Exospace performs are invisible and occur before and after linking:

- CA-Clipper-compiled applications are enabled to execute without violating any of the restrictions imposed by protected mode.
- CA-Clipper's real-mode VM system is replaced with a protected-mode version.
- Any other operations necessary to allow CA-Clipper to run in protected mode are performed.

Performance Considerations

CA-Clipper/Exospace performs best if given sufficient memory in which to operate. Just before CA-Clipper/Exospace finishes linking, the amount of memory used during the link is displayed (for example, 2314208 bytes used out of 3057472 bytes available).

Best performance will be achieved if you have at least as much RAM available as the number of bytes used when linking. If you have less, swapping to disk will occur, which slows down the link process. Note that the amount of memory required for linking is proportional to the size of the application being linked.

The CA-Clipper/Exospace Return Code

If CA-Clipper/Exospace encounters an error or is interrupted by the user pressing Ctrl+C or Ctrl+Break, it terminates with a return code (DOS ERRORLEVEL) of 1. If the linker session ends normally or if only warning and/or unresolved symbol errors were encountered, the return code is set to 0.

Some examples of typical errors that are encountered during a linker session are the inability of CA-Clipper/Exospace to locate a file or to interpret a command because of incorrect syntax. If an error occurs, an error message is displayed and no output file is generated (unless you specify the EXOSPACE EXECUTABLE NODELETE command). For a list of error messages and their meanings, see the "CA-Clipper/Exospace Error Messages" chapter in the *Error Messages and Appendices Guide.*

The Compile and Link Batch File

The default installation of the CA-Clipper development system places a compile and link batch file, CL.BAT in the \CLIP53\BIN directory. You can use this batch file to compile and link a single program (.prg) file to an executable file (.EXE) if there are no errors during compilation. If there is a compiler error, CLIPPER.EXE terminates with a return code of 1, and the batch file in turn terminates without attempting to link.

To invoke the batch file, use the following syntax:

```
CL <sourceFile>
```

The *<sourceFile>* is the name of the program file to compile and link, specified without an extension. You can specify the filename with a drive designator and/or a path reference.

Warning! *Specifying the <sourceFile> with an extension will cause a linker error.*

Output Files

There are several types of files produced as output by CA-Clipper/Exospace. This section describes each type separately.

Any output filename that you specify can optionally include a drive designator and/or a path reference to explicitly indicate where CA-Clipper/Exospace places the file on disk. Unless otherwise specified, files are created in the current directory.

Except in certain cases pointed out in the ensuing discussion, you can override default extensions for output files with explicit extensions specified as part of the filename. To indicate a file with no extension, include a period at the end of the filename.

Executable Files (.EXE)

CA-Clipper/Exospace produces an executable file as its output. You can specify the name directly using the OUTPUT command or have CA-Clipper/Exospace determine it for you based on the first filename (either script or object) encountered on the CA-Clipper/Exospace command line. CA-Clipper/Exospace assigns a default extension of .EXE to executable files, which you cannot override.

The .EXE extension is used to designate a file that can be executed directly by the operating system shell, COMMAND.COM.

The order in which .EXE files are searched for at runtime is dictated by DOS as follows:

- The current directory
- The path specified in the PATH environment variable

If the file is not found, a runtime error is generated.

Map Files (.MAP)

You can cause CA-Clipper/Exospace to generate a map file using the MAP command. The default extension for this file type is .MAP.

A map file is simply an ASCII text file containing information about symbol and segment addresses within the memory image. You specify the exact content and order of the file contents with MAP command options.

Temporary Files

On occasion, CA-Clipper/Exospace generates temporary files during the linking process. You can control where these files are created by setting up an environment variable called TMP with the DOS SET command as follows:

```
SET TMP=<pathName>
```

Placing this command in your AUTOEXEC.BAT file automatically defines the TMP variable so that temporary files generated by CA-Clipper/Exospace are automatically placed in the desired location. If you do not specify the TMP environment variable, temporary files are created in the current directory.

The linker deletes temporary files after they have served their purpose. Therefore, you may never see them on your disk.

You can use the TMP variable to write the temporary files to another disk drive if you are running short of space on the drive that you normally use to link. To speed up the linking process you can also set TMP to write temporary files on a RAM disk. This way, the temporary files will not take up disk space that is better used by permanent files. For example:

```
SET TMP=D:\TEMP
```

How CA-Clipper/Exospace Searches for Files

There are several types of files that serve as input to CA-Clipper/Exospace. Each type of file is searched for according to a prescribed set of rules as follows:

- The current directory
- The path in a specific DOS environment variable

You can include a full or partial path name to explicitly indicate where CA-Clipper/Exospace should search for a file. If, however, you do not specify the location of a file, the prescribed rules are used to locate the file. If the file is not found after all of the possible locations have been searched, CA-Clipper/Exospace terminates with an error message, the DOS return code is set to 1, and no output file is produced.

This section describes how CA-Clipper/Exospace attempts to locate each possible input file type. The rules stated above apply to all file types, so they will not be repeated. Instead, the specific search environment variable is defined for each file type and any deviations from the general rules are noted.

Library Files (.LIB)

Library files have a default extension of .LIB and you specify them using the LIBRARY command. The environment variable used to locate library files is called LIB, and you define it using the SET command as follows:

```
SET LIB=<pathSpec list>
```

In the default installation configuration, LIB is set to \CLIP53\LIB. To change this default, alter the SET LIB command in your AUTOEXEC.BAT file. For example, to include a directory containing third-party libraries, you might change it to the following:

```
SET LIB=C:\CLIP53\LIB;C:\CLIP53\THIRDPARTY\LIB
```

Object Files (.OBJ)

Object files have a default extension of .OBJ and you specify them using the FILE command. The environment variable used to locate library files is called OBJ, and you define it using the SET command as follows:

```
SET OBJ=<pathSpec list>
```

In the default installation configuration, OBJ is set to \CLIP53\OBJ. To change this default, alter the SET OBJ command in your AUTOEXEC.BAT file. For example, if your object files are located in a directory called OBJ, you might change it to the following:

```
SET OBJ=C:\OBJ
```

Script Files (.LNK)

Script files have a default extension of .LNK and you specify them on the CA-Clipper/Exospace command line or within another script file by preceding the filename with an @ symbol. There is no associated environment variable to specify where to search for script files.

Linker Command Reference

This section provides an alphabetical reference to all the CA-Clipper/Exospace commands. You can specify commands on the CA-Clipper/Exospace command line or in a script file. The commands are not case-sensitive, and you can specify them in whatever order you want. The only command that is required to link is the FILE command—all others are optional.

You can abbreviate any linker command to just a few letters. For example, FILE becomes FI and LIBRARY become LIB (or just LI). Abbreviations are not documented for each command, but as a general rule you must use enough letters to distinguish one command from another and should include enough letters so that the option is readable.

Unless otherwise stated in this section, the following rules apply when the same command is encountered more than once during a link session:

- Any command encountered on the CA-Clipper/Exospace command line overrides like commands in a script file.
- If CA-Clipper/Exospace encounters the same command within script files, the last occurrence of the command is used.

For example, if you specify STACK 8192 on the CA-Clipper/Exospace command line and link with a script file specifying STACK 9672, the stack size for the application will be 8192 bytes. On the other hand, if you link with no STACK command on the command line and specify STACK 8192 at the beginning of your script file and STACK 9672 somewhere near the end, the stack size for the application will be 9672 bytes.

Compatibility with Other Linkers

CA-Clipper/Exospace is geared for compatibility with .RTLink, the linker included with previous versions of CA-Clipper, and Blinker so that you link using script files developed for these linkers.

.RTLink Compatibility

CA-Clipper/Exospace achieves compatibility with .RTLink in two distinct ways.

First, CA-Clipper/Exospace supports a subset of .RTLink commands (using the FREEFORMAT syntax only). In other words, the syntax and semantics for the commands that the two linkers have in common are exactly the same. These commands are documented fully in the alphabetical reference later in this section. For example, commands such as FILE, MAP, and STACK are the same for both linkers.

Second, CA-Clipper/Exospace ignores any .RTLink command that is not listed in the alphabetical reference later in this section. For example, the BEGINAREA....ENDAREA construct is used to define overlay areas. Since CA-Clipper/Exospace overlays code automatically as needed, it ignores these commands.

Thus, when you link a CA-Clipper-compiled application using CA-Clipper/Exospace, you can specify your old .RTLink script file on the CA-Clipper/Exospace command line to build the application. Commands that are compatible will operate as expected, while commands that are incompatible will be ignored. Note that the link script should work properly to produce the desired .EXE since the ignored commands are either performed automatically or do not apply when linking with CA-Clipper/Exospace.

Important! *The .RTLink MODULE command is mostly used to control module placement relative to overlay areas and for this reason, it is ignored by CA-Clipper/Exospace. If, however, you are using the MODULE command to force linkage of a module which would not otherwise be linked, CA-Clipper/Exospace requires that you use the MODULE...FROM command instead, specifying the library in which the module can be found.*

Blinker Compatibility

CA-Clipper/Exospace achieves compatibility with Blinker in three distinct ways.

First, CA-Clipper/Exospace supports a subset of Blinker commands. In other words, the syntax and semantics for the commands that the two linkers have in common are exactly the same. These commands are documented fully in the alphabetical reference later in this section.

Second, if a particular Blinker command has an CA-Clipper/Exospace equivalent but is performed using a slightly different syntax, CA-Clipper/Exospace performs a simple translation of the command to achieve the desired result. For example, these BLINKER commands are simply replaced by their EXOSPACE equivalent:

BLINKER ENVIRONMENT CLIPPER
BLINKER ENVIRONMENT OVERRIDE
BLINKER EXECUTABLE CLIPPER
BLINKER EXECUTABLE NODELETE
BLINKER PROCEDURE DEPTH

Then, the ALLOCATE and SEARCH commands are interpreted as LIBRARY commands, which works in most cases.

Important! *The behavior of the SEARCH command under CA-Clipper/Exospace can lead to differences in link sequence compared to the same command used with Blinker. If you are using the SEARCH command to prioritize a particular library in a Blinker script, you may need to rearrange the order in which the libraries are specified or use the MODULE...FROM command to explicitly control link sequence as necessary.*

Third, CA-Clipper/Exospace ignores any Blinker command that is not listed in the alphabetical reference later in this section and does not have an equivalent as previously mentioned. For example, the BEGINAREA....ENDAREA construct is used to define overlay areas. Since CA-Clipper/Exospace overlays code automatically as needed, it ignores these commands.

Thus, when you link a CA-Clipper-compiled application using CA-Clipper/Exospace, you can specify your Blinker script file on the CA-Clipper/Exospace command line to build the application. Commands that are compatible or that can be translated will operate as expected, while commands that are incompatible will be ignored. Note that the link script should work properly to produce the desired .EXE since the ignored commands are either performed automatically or do not apply when linking with CA-Clipper/Exospace.

EXOSPACE CLIPPER 501

Purpose

Specify that you are linking a CA-Clipper 5.01 or 5.01a application

Syntax

EXOSPACE CLIPPER 501

Description

If you are using CA-Clipper 5.01 or 5.01a, you must include this command when linking with CA-Clipper/Exospace. Otherwise, CA-Clipper/Exospace assumes that you are linking an application created with a later version of the CA-Clipper compiler.

Note that inclusion of this command does not affect which CA-Clipper libraries are linked—you must still ensure that the CA-Clipper 5.01 libraries are linked. This command merely causes CA-Clipper/Exospace to treat the application as a CA-Clipper 5.01 application. If it is actually an application created with a later version of the CA-Clipper compiler, unpredictable behavior will result.

EXOSPACE ENVIRONMENT CLIPPER

Purpose

Specify the environment variable to use for application runtime configuration

Syntax

EXOSPACE ENVIRONMENT CLIPPER <*envName*>

Arguments

<*envName*> is the name of the environment variable.

Description

By default, the CLIPPER environment variable controls application runtime configuration (see "The Runtime Environment" chapter in this guide for details). This command lets you specify an alternative environment variable name in case you want to use a name unique to your application.

Examples

This example causes the application being linked to read its runtime configuration information from an environment variable named MYAPP:

```
# Use the environment variable MYAPP
# instead of CLIPPER
EXOSPACE ENVIRONMENT CLIPPER MYAPP
```

See Also

EXOSPACE EXECUTABLE CLIPPER, EXOSPACE ENVIRONMENT OVERRIDE

EXOSPACE ENVIRONMENT OVERRIDE

Purpose

Cause runtime configuration environment settings to override .EXE defaults

Syntax

EXOSPACE ENVIRONMENT OVERRIDE

Description

By default, when you hard-code runtime configuration settings into a CA-Clipper executable using the EXOSPACE EXECUTABLE CLIPPER command, settings specified in the CLIPPER environment variable (or the variable specified by the EXOSPACE ENVIRONMENT CLIPPER command) are ignored. This ensures that multiple applications with different settings can run on the same machine without changing any environment settings.

If you specify EXOSPACE ENVIRONMENT OVERRIDE, however, runtime configuration settings specified in the CLIPPER environment variable (or the variable specified by the EXOSPACE ENVIRONMENT CLIPPER command) override corresponding settings hard-coded in the .EXE.

Note: If you use the EXOSPACE ENVIRONMENT CLIPPER command to specify a unique runtime configuration environment variable for an application, you may want to allow the .EXE settings to be overridden.

See Also

EXOSPACE EXECUTABLE CLIPPER, EXOSPACE ENVIRONMENT CLIPPER

EXOSPACE EXECUTABLE CLIPPER

Purpose

Hard-code default runtime configuration settings in the .EXE

Syntax

EXOSPACE EXECUTABLE CLIPPER <*settings*>

Arguments

<*settings*> is one or more environment settings. In addition to the settings listed in the table under Setting the Application Environment in "The Runtime Environment" chapter of this guide, the following settings are available for controlling the CA-Clipper/Exospace protected-mode VM system:

Setting	Meaning
LOWMEM:<*nKBytes*>	Set the amount of low DOS memory to reserve for allocation by third-party libraries for interrupt routines. The default is 0. Ignore this unless your third-party library documentation specifically recommends that you set it.
MAXMEM:<*nKBytes*>	Set the maximum amount of physical memory to use before swapping to disk. The default is 8096 KB, which is ample for most applications. In a task switching environment, you may want to lower this value.
MINMEM:<*nKBytes*>	Set the minimum amount of physical memory that must be available to run the program. If there is not enough memory available, an error message is generated and the application returns to DOS. The default is 1024 KB, which is the recommended minimum.
VMSIZE:<*nKBytes*>	Set the amount of virtual memory the VM system should provide. The default is 8096 KB, which is ample for most applications.

Note: You can also specify these settings in the CLIPPER environment variable (or the variable specified by the EXOSPACE ENVIRONMENT CLIPPER command) and on the application command line as described under Setting the Application Environment in "The Runtime Environment" chapter of this guide.

The following rules apply when specifying environment settings:

- Preface each setting with a double-slash
- Place a single blank space between settings
- Place a colon between setting and argument, with no intervening space

Description

By default, applications that you create using the CA-Clipper compiler and the CA-Clipper/Exospace linker rely on settings in the CLIPPER environment variable (or the variable specified by the EXOSPACE ENVIRONMENT CLIPPER command) to determine certain runtime configuration parameters.

The EXOSPACE EXECUTABLE CLIPPER command lets you hard-code runtime configuration settings in the .EXE, eliminating the need to rely on an environment variable for the information.

Normally, if you use EXOSPACE EXECUTABLE CLIPPER command to link, the runtime configuration environment settings are ignored. You can, however, offer the flexibility of overriding the hard-coded defaults with similar settings in the environment variable using the EXOSPACE ENVIRONMENT OVERRIDE command .

Examples

This example sets the number of file handles to 40 and prevents the use of EGA/VGA extended cursor capability:

```
EXOSPACE EXECUTABLE CLIPPER //F:40 //CGACURS
```

This examples sets various CA-Clipper/Exospace VM system settings:

```
EXOSPACE EXECUTABLE CLIPPER //VMSIZE:16384
   //MAXMEM:2048 //MINMEM:1536
```

See Also

EXOSPACE ENVIRONMENT CLIPPER, EXOSPACE ENVIRONMENT OVERRIDE

EXOSPACE EXECUTABLE NODELETE

Purpose

Create an .EXE even if errors occur during linking

Syntax

EXOSPACE EXECUTABLE NODELETE

Description

By default, CA-Clipper/Exospace will not create an .EXE if errors occur. This command allows you to override that behavior, although the resulting .EXE is not guaranteed to operate correctly.

EXOSPACE PACKAGE

Purpose

Cause specified protected-mode support packages to be included in the .EXE

Syntax

EXOSPACE PACKAGE <*package list*>

Arguments

<*package list*> is a comma-separated list of packages as shown in the table below to be included in the resulting .EXE:

Package	Description
8514	IBM 8514 Display Adapter. Allows direct calls from protected mode, with no mode switching.
DOS25	Absolute Disk Read/Write compatibility package. Services INT 25h and 26h.
INT10	Video BIOS compatibility package. Services INT 10h.
IPX	IPX/SPX compatibility package for Novell Networks. Services INT 7Ah.
IPXCT	Compatibility package for CA-Clipper Tools.
NET5C	NETBIOS compatibility package. Replaces NETBIOS?.OBJ. Services INT 5Ch.
NOVM	Disables the CA-Clipper/Exospace VM system. The resulting application will run only on systems that have sufficient RAM to contain the application's entire .EXE and all its data. (See Notes below for more information about this option.)

Description

Specifies one or more *packages* to be included in the .EXE. Packages provide protected-mode support for certain low-level (C or Assembler) operations. Specifically, packages provide CA-Clipper programs transparent access to software interrupts that are unsupported by the protected-mode kernel. Packages are more suitable than libraries for this purpose because they are much more automatic, much less error prone, and require no source changes.

Important! *You never need to use these packages in an application consisting purely of CA-Clipper code that does not use any third-party libraries. However, some third-party libraries that perform low-level operations—such as direct screen I/O or certain types of network calls—may require one or more of these packages to be included. If you are using a third-party library that does not operate correctly, consult the vendor of that library to determine whether it requires any packages to be included when linking with CA-Clipper/Exospace.*

If you specify more than one EXOSPACE PACKAGE command in the same linker session, either on the CA-Clipper/Exospace command line or within one or more script files, each *<package list>* is added to the previous one, causing all packages in all EXOSPACE PACKAGE lists to be linked.

Notes

Disabling VM. You can also use the EXOSPACE PACKAGE command to prevent the default Virtual Memory (VM) package from being included. Normally, the VM package is automatically included in an CA-Clipper/Exospace application. This ensures that CA-Clipper/Exospace applications can run on a wide variety of target machines, even those that do not have enough physical RAM—in such cases, the VM system will automatically swap data to disk as necessary.

However, if you are certain that a particular application will always be run on a machine with sufficient RAM, you can exclude the VM package using the EXOSPACE PACKAGE NOVM command. Applications linked in this way may load and initialize somewhat faster, but they will fail if there is not enough RAM on the target machine to hold the entire .EXE and all of its data.

Determining how much RAM a particular application requires may require some trial and error. In general, it is recommended that you limit use of this option to special circumstances.

Examples

This example includes the video BIOS package and excludes the VM package:

```
# Include video BIOS package and exclude VM package
EXOSPACE PACKAGE INT10, NOVM
```

EXOSPACE PROCEDURE DEPTH

Purpose

Set the stack size of an application in terms of maximum procedure call depth

Syntax

EXOSPACE PROCEDURE DEPTH <*maxDepth*>

Arguments

<*maxDepth*> is the maximum depth to which you can nest procedure calls.

Description

This command has the same effect as the STACK command, but conveniently allows you to set the stack size in terms of procedure depth rather than number of bytes. To set the stack size to a specified number of bytes, use the STACK command instead.

Examples

The following command allows a maximum of 100 nested procedure calls in the application being linked:

```
EXOSPACE PROCEDURE DEPTH 100
```

See Also

STACK

FILE

Purpose

Specify one or more object files to be linked

Syntax

FILE <*objFile list*>

Arguments

<*objFile list*> is a comma-separated list of object or library files to link. If you specify a file without an extension, .OBJ is assumed. If you specify a library, all object files within the library are linked. Note that in this case, you must specify the .LIB extension.

You can optionally specify a drive designator and/or a path reference for each file in the list. See How CA-Clipper/Exospace Searches for Files earlier in this chapter for information on how an input file is located if its filename is unqualified.

Description

If you specify more than one FILE command in the same linker session, either on the CA-Clipper/Exospace command line or within one or more script files, each <*objFile list*> is added to the previous one, causing all files in all FILE lists to be linked.

Examples

This example links three object files to create an executable file named MAINMENU.EXE:

```
FILE MAINMENU, INVOICE, REPORTS
```

The next example links the same three object files and the CA-Clipper debugger library, CLD.LIB, to create an executable file named MAINMENU.EXE that has access to the debugger:

```
FILE MAINMENU, INVOICE, REPORTS, CLD.LIB
```

See Also

LIBRARY

LIBRARY

Purpose

Specify one or more library files to search during linking

Syntax

LIBRARY *<libFile list>*

Arguments

<libFile list> is a comma-separated list of libraries and/or object files to search in order to resolve any undefined symbols. If you specify a file without an extension, .LIB is assumed.

You can optionally specify a drive designator and/or a path reference for each file in the list. See How CA-Clipper/Exospace Searches for Files earlier in this chapter for information on how an input file is located if its filename is unqualified.

Description

CA-Clipper/Exospace searches certain libraries (EXOSPACE.LIB, LLIBG.LIB, CLIPPER.LIB, EXTEND.LIB, DBFNTX.LIB, and TERMINAL.LIB) automatically without the need for you to specify them as part of a LIBRARY command. These library names automatically supplement any libraries that you name using the LIBRARY command. If you use the NODEFLIB command, however, CA-Clipper/Exospace does not automatically search these library files; therefore, you may need to specify one or more of them using the LIBRARY command, depending on your needs.

If you specify more than one LIBRARY command in the same linker session, either on the CA-Clipper/Exospace command line or within one or more script files, each *<libFile list>* is added to the previous one, causing all libraries in all LIBRARY lists to be searched.

Examples

This example specifies that the application being linked needs access to three libraries: PWFUNCS.LIB, CLASSY.LIB, and PWKERNEL.LIB:

```
LIBRARY PWFUNCS, CLASSY, PWKERNEL
```

See Also

NODEFLIB

MAP

Purpose

Cause a map to be generated

Syntax

MAP [=<*mapFile*>] [<*mapOption list*>]

Arguments

<*mapFile*> is the name of the file to generate. If you specify the file without an extension, .MAP is assumed.

You can optionally specify a drive designator and/or a path reference for the file. Otherwise, the map file is created in the current directory.

If you specify the MAP command without a filename, the map file is given the same name as the resulting .EXE file (see the OUTPUT command), but with a .MAP extension.

<*mapOption list*> is a comma-separated list of map options that determines which reports are written to the specified map file. If not specified, all possible reports are written. You can specify any combination of the options shown in the following table:

Option	Reports
A	Public symbols sorted by address
N	Public symbols sorted by name
S	Segments with assigned addresses

Examples

In this example, even though no map filename is explicitly specified with the MAP command, the map file is generated with a default name of TEST1.MAP:

```
EXOSPACE FILE TEST1, TEST2 MAP S, N
```

On the other hand, you could specify the map filename as part of the MAP option. In this example, the file will be called MYMAP.MAP:

```
EXOSPACE FILE TEST1, TEST2 MAP=MYMAP S, N
```

See Also

OUTPUT

MODULE...FROM

Purpose

Search for the named module in the specified library only

Syntax

MODULE <*moduleName*> FROM <*libName*>

Arguments

<*moduleName*> is the name of a module or symbol whose location you want to specify to the linker.

<*libName*> is the name of the library in which <*moduleName*> is defined. This is not an actual filename (that is, it does not include a path or extension) but the root portion of a library that is already known to the linker (via the LIBRARY command or the fact that the linker knows it by default).

Description

Normally, when CA-Clipper/Exospace encounters a reference to an unknown module or symbol during the linking process, it searches default libraries (unless you specify NODEFLIB) and libraries that you have specified using the LIBRARY command to resolve the reference. CA-Clipper/Exospace searches the libraries in certain predefined order (called the link sequence) and stops as soon as it resolves the reference.

Using the MODULE...FROM command, you can tell CA-Clipper/Exospace exactly which library contains a certain module or symbol, thereby eliminating the automatic searching that it would normally perform. This feature is useful, for example, if you have the same procedure defined in two different libraries and want to specify which of two should be linked.

Examples

This example forces the linker to link the OrdEntry module from MYOLDLIB.LIB, even though it may be defined in other libraries being linked:

```
MODULE OrdEntry FROM MYOLDLIB
```

See Also

FILE, LIBRARY, NODEFLIB

NODEFLIB

Purpose

Do not search default libraries

Syntax

NODEFLIB | NODEFAULTLIBRARYSEARCH

Description

By default, CA-Clipper/Exospace searches libraries whose names are embedded in the object modules being linked. With CA-Clipper-compiled object files, the embedded libraries are CLIPPER.LIB, DBFNTX.LIB, EXTEND.LIB, and TERMINAL.LIB. In addition to these embedded libraries, CA-Clipper/Exospace also includes the EXOSPACE.LIB and LLIBG.LIB libraries. If you use the NODEFLIB command, CA-Clipper/Exospace ignores these library files; therefore, you may need to specify one or more of them using the LIBRARY command, depending on your needs.

Important! *If you specify NODEFLIB, you must explicitly specify the CA-Clipper/Exospace library, EXOSPACE.LIB, using the LIBRARY command. EXOSPACE must be the first library in the first LIBRARY command encountered. Otherwise, CA-Clipper/Exospace will be unable to successfully link a protected-mode executable.*

Notes

- **Renaming EXOSPACE.LIB:** Do not rename the CA-Clipper/Exospace library, EXOSPACE.LIB. CA-Clipper/Exospace requires that the library have this name and will not operate correctly if it is renamed.

  Similarly, do not rename the CA-Clipper debugger library, CLD.LIB. CA-Clipper/Exospace performs special processing on this library to allow it to operate in protected mode, so the debugger will not function correctly if the library is renamed.

- **Using Graphic Mode Functions:** If you use both, make sure you link LLIBG.LIB before CLIPPER.LIB.

Examples

This example shows how to specify selected system-supplied libraries if you choose the NODEFLIB option. In this case, the DBFMDX driver is used instead of the default DBFNTX driver, and the TERMINAL library is not used at all:

```
NODEFLIB
FILE MYAPP
LIBRARY EXOSPACE, CLIPPER, EXTEND
LIBRARY DBFMDX, MYLIB
```

OUTPUT

Purpose

Specify the name of the executable file to be generated

Syntax

OUTPUT *<exeFile>*

<exeFile>

The name of the executable file (.EXE) to generate. If you do not specify an OUTPUT file, the name of the first filename (either script or object) encountered on the CA-Clipper/Exospace command line is used, but with an .EXE extension—you cannot override this default extension.

You can optionally specify a drive designator and/or a path reference for the file. Otherwise, the executable file is created in the current directory.

Examples

This example specifies that the executable file be named MYDBU.EXE. If the OUTPUT command was not specified in this case, the result would be DBU.EXE, after the name of the script file:

```
EXOSPACE @DBU OUTPUT MYDBU
```

See Also

FILE

STACK

Purpose Set the stack size of an application in bytes

Syntax STACK <*sizeBytes*>

Arguments <*sizeBytes*> is the number of bytes to allocate in decimal. The value you specify can be any positive number up to 65,535.

Description The stack size determines how deep the procedure call stack can be. In most cases, it is more convenient to use the EXOSPACE PROCEDURE DEPTH command instead, but the STACK command provides more accurate control over the size of the stack.

Examples This example sets the stack size at 8192 bytes:

```
STACK 8192
```

See Also EXOSPACE PROCEDURE DEPTH

Troubleshooting

This section suggests ways to resolve certain problems that can occur when you execute your CA-Clipper/Exospace application, such as:

- How to determine the cause of an application failure
- What to do about memory problems
- What to look for if your program terminates abnormally under DOS Protected-Mode Interface (DPMI)
- How to deal with third-party library problems
- How to ensure maximum compatibility for your application under various operating environments

Examine this section for solutions to your problems before you call Computer Associates Technical Support.

If the Application Does Not Run

If your application does not run on a particular machine or with other software that uses extended memory, answer these questions to help determine what the problem is:

1. In what way does the application fail?

   Can you load the application at all? If so, what behavior does the failing program exhibit? Does your application display any error messages? If so, what message and register values appear on the screen?

2. Does PMINFO run?

   If you cannot run your program at all, try running PMINFO, supplied with CA-Clipper. If it runs, the problem is not likely to be a BIOS incompatibility.

3. What resident programs are loaded on your system?

   You may have a device driver or some other software (like network software) that is incompatible with CA-Clipper/Exospace. To determine whether the problem is caused by a software incompatibility, try running your application after unloading your network drivers or removing DEVICE= statements from your CONFIG.SYS. If the CA-Clipper/Exospace program runs when the other program is removed, there is a compatibility problem.

   CA-Clipper/Exospace follows the DPMI, Virtual Control Program Interface (VCPI), and Extended Memory (XMS) protocols. If the product that prevents CA-Clipper/Exospace from running follows one of these protocols, call Computer Associates Technical Support. If it does not, refer to Software Compatibility later in this chapter.

4. What kind of a machine are you running on?

   If PMINFO does not run on your machine after you have removed other resident programs, your system or BIOS is not compatible with CA-Clipper/Exospace.

   If you are running on a machine that you think is AT-compatible, your system BIOS is probably not compatible with CA-Clipper/Exospace. Run RMINFO to get the information about your system that you could not get from PMINFO (RMINFO does everything that PMINFO does except attempt to switch to protected mode).

5. How much available memory does your system contain?

   If the application fails with the message "not enough memory," your system may not have enough memory or the memory may not be available to CA-Clipper/Exospace. See Memory Shortage later in this chapter.

How Does It Fail?

Protection Fault

If your CA-Clipper/Exospace application terminates with a General Protection Fault (Error 35), look at the selector that CA-Clipper/Exospace reports as the location of the problem.

The selector is the number printed before the colon. For example, in the error message below, the fault is reported as occurring "at 016F:0022." In this case, the selector is 016F:

```
Error [35]: General Protection Fault in
MYAPP.EXE at 026F:0022
code=0000 ss=07FF ds=07FF es=0000
ax=0004 bx=0016 cx=0000 dx=0000
sp=7B50 bp=7B56 si=0000 di=0001
```

Unless your application is running under DPMI (for example, in a Microsoft Windows DOS box), a selector value less than 200h indicates an error in the CA-Clipper/Exospace code. Selector values greater than 200h indicate a problem in the application code—either in a third-party library, or other C or Assembler code. The error message shown above was generated by a NULL pointer error in a third-party library or other C or Assembler code.

When you link an application, you can generate a .MAP file using the MAP command. Unless your application is running under DPMI, you can find out which module and/or routine is at fault by finding the selector number and closest offset in the map file.

Another way to determine where the problem is occurring is to use the CA-Clipper debugger to trace up to the point where the problem occurs.

Not a DOS/16M Executable

If you get the message "not a DOS/16M executable," your copy of the application may be corrupted. This indicates that a virus may exist on your system. Use a virus scanning program to determine if there is a virus on your system; if there are no viruses, try relinking the application.

If you encounter this error on any of the CA-Clipper distribution files, try reinstalling CA-Clipper from the distribution diskettes.

System Hang

It can be difficult to determine the cause of a system hang, because the system halts before you can obtain any significant information about the problem.

Try running the program on a different machine to determine whether the problem is hardware-related. If it runs on another machine, compare the configurations of the two machines to determine the source of the problem.

If the application does not run on any machine you have, try loading the program under the CA-Clipper debugger and looking at the code that executes prior to when the system hangs. This may give you an indication of the cause or source of the problem.

If you call us for support, we will ask more questions: What happens when the program terminates? Is the keyboard disabled, or does your machine spontaneously reboot? Is there disk activity? When the system hangs, does Ctrl+Alt+Delete work to reset it?

When you call us, we may also ask you to try reproducing the problem in a small test case.

Running PMINFO

If PMINFO runs on your machine, you can usually rule out most hardware or BIOS incompatibilities and conflicts with other software that uses extended memory. PMINFO is a small, simple program, so your problem could be one that only occurs when a program uses a lot of memory or attempts something fairly complicated.

If PMINFO fails, then you have a hardware or BIOS problem, or a problem with other software that uses extended memory. If removing resident software does not remedy the problem, run RMINFO and send us the following information so we can resolve the problem:

- Output from RMINFO (redirect to a file)
- The fact that PMINFO would not run on your machine
- Your name, address, phone number
- A description of your computer

You may also want to call us for help in determining whether you have a hardware problem, or the dealer or manufacturer of your machine to see if there is a newer BIOS available to support protected-mode products. Such a BIOS could enable you to run CA-Clipper/Exospace on your machine.

AT&T 6300 Plus

If you are running your CA-Clipper/Exospace application on an AT&T 6300 Plus, which runs MS-DOS but is not completely AT-compatible, use the DOS16M environment variable to specify the machine type. (Unless VCPI or DPMI is active, this also specifies the switch mode.)

For an AT&T 6300 Plus, set DOS16M as follows:

```
SET DOS16M=6
```

Software Compatibility

Sharing Extended Memory—Compatibility Notes

Two CA-Clipper/Exospace applications can share extended memory without conflict. However, if an CA-Clipper/Exospace application and a protected-mode application that does not use CA-Clipper/Exospace are running concurrently, the two applications may have trouble sharing the extended memory.

The protected-mode DOS extender technology on which CA-Clipper/Exospace relies is supplied by Rational Systems, Inc., a leader in protected-mode technology. Many commercial applications that run in protected mode rely on the Rational Systems technology, including applications from Lotus, IBM, Quarterdeck, Borland, Symantec, and Watcom. Such applications should successfully coexist with CA-Clipper/Exospace applications.

However, when running other protected-mode applications alongside CA-Clipper/Exospace applications, it is possible that the first CA-Clipper/Exospace application which starts up may reserve all available extended memory, unless it is running under a DPMI host, a VCPI host, or an XMS manager.

Without a memory manager, if you try to start up a protected-mode application that was not built with CA-Clipper/Exospace after your CA-Clipper/Exospace program, the non-CA-Clipper/Exospace application will not find any extended memory available.

There are several ways to avoid this problem:

- Use a memory manager whose protocol both programs adhere to.

  CA-Clipper/Exospace follows the DPMI, VCPI, and XMS protocols. If the other program also follows one of these protocols, the memory manager will ensure that the programs properly share memory.

- Start the non-CA-Clipper/Exospace application first.

  If the non-CA-Clipper/Exospace, protected-mode application reserves only the memory that it needs, the CA-Clipper/Exospace program uses extended memory that the first does not reserve.

- Use the DOS16M environment variable to limit the amount of memory that the CA-Clipper/Exospace application can use.

  For example, to limit the CA-Clipper/Exospace application to 1 MB, use this setting:

  ```
  SET DOS16M= :1M
  ```

- Use the DOS16M environment variable to set the range of memory that CA-Clipper/Exospace uses.

If you can specify a range of memory for the non-CA-Clipper/Exospace application, you may be able to establish a private range for each application so that they do not conflict. The DOS16M environment variable allows you to specify a range for the CA-Clipper/Exospace application.

For example, if you can make your non-CA-Clipper/Exospace application utilize addresses from 1 MB to 2 MB, you can specify the memory from 2 MB to 3 MB for CA-Clipper/Exospace. To specify this range of memory, use one of these settings in your DOS16M environment variable:

```
SET DOS16M= @2M - 3M
SET DOS16M= :1 @2M
```

If setting a range in the DOS16M environment variable does not solve the problem, remove the setting.

A20 Problems

If the fifth digit of the address displayed in the GPF error message, when your machine abnormally terminates, is an odd number (for example, from 0x10000 to 0x1FFFF, or from 0x30000 to 0x3FFFF), you may be running software that requires rigorous management of the A20 line. Try setting the A20 option in the DOS16M environment variable:

```
SET DOS16M= A20
```

Memory Shortage

If you encounter a "not enough memory" error when you attempt to execute your application, run PMINFO to determine how much memory is available for CA-Clipper/Exospace applications.

Extended Memory Not Available

CA-Clipper/Exospace uses extended memory, not expanded memory. If you can configure your memory either way, and you are not running under an expanded memory manager, be sure to configure the memory as extended memory.

If you are using an expanded memory manager that follows the VCPI protocol (such as EMM386, QEMM, or 386MAX), your CA-Clipper/Exospace application allocates its memory from the VCPI host (unless you also have an XMS manager). Therefore you do not have to configure it as extended memory.

If you have a VCPI host and an XMS manager (such as HIMEM.SYS), CA-Clipper/Exospace allocates memory from both the VCPI pool and the XMS pool.

If you have another program that uses extended memory, see Software Compatibility earlier in this chapter.

Using the High Memory Area

If you are running CA-Clipper/Exospace on an older machine (manufactured before 1989) and PMINFO reports less memory than your hardware documentation specifies, your manufacturer may be counting extra memory as extended memory. The High Memory Area (HMA) is the area below 1 MB but above the 640 KB conventionally used by DOS.

For example, an AST machine may be documented to have 2 MB of extended memory, but PMINFO reports only about 1.7 MB because AST utilizes HMA. Some manufacturer-specific RAM drives also use the HMA.

CA-Clipper/Exospace automatically utilizes this memory on a Compaq 386 machine, but it does not recognize it on other machines. You can set the DOS16M environment variable to test the memory between 640 KB and 1 MB and use whatever it finds that is not read-only memory. To try this, use the + option in your DOS16M environment variable setting:

```
SET DOS16M= +
```

If implementing this setting does not solve the problem or makes it worse, remove the setting and reboot your machine.

Bad Memory

If your system is configured with adequate extended memory, and you still encounter a "not enough memory" error, some of your configured memory may be bad. Another indication of bad memory is when your program consistently terminates at the same address.

A simple way to check for bad memory is to set the memory range at different memory locations and see if your application still does not run. For example, if you have 4 MB of memory on your machine, and your application takes less than 2 MB, you could try the following setting:

```
SET DOS16M= @2M
```

With this setting, CA-Clipper/Exospace starts loading the executable at 2 MB, leaving extended memory below that location free. If the application runs with this setting, some portion of the memory below 2 MB is bad.

Problems When Running Under DPMI

If you have a program that runs well under DOS but abnormally terminates when it runs under a DPMI host, such as Windows 3.1 in enhanced mode or a Virtual DOS Machine under OS/2 2.0, check for the following problems:

- How much memory does your application require?

  Ensure that enough extended memory is available from the DPMI host.

- Does your program shell out to another program?

  In the DPMI 0.9 specification (which enhanced-mode Windows 3.x follows), only one protected-mode application can run on a virtual machine at a time. That means that you cannot run two protected-mode programs in the same DOS box under Windows 3.1 in enhanced mode.

  Modify your program so that it does not shell to a protected-mode program. DPMI hosts that follow the 1.0 specification allow more than one protected-mode application to run. Some 0.9 DPMI hosts have extensions that allow two protected-mode applications to run on the same virtual machine, but it is difficult to detect this capability.

Problems in Third-Party Libraries

Before linking your CA-Clipper application with CA-Clipper/Exospace, you need to ensure that any third-party libraries you are using are compatible with CA-Clipper/Exospace.

CA-Clipper/Exospace is compatible with most third-party libraries for CA-Clipper because most of them are general purpose function libraries that consist mostly of CA-Clipper code. However, incompatibilities will arise if a third-party library performs operations that are not valid in protected mode. Typically, only libraries that perform very low-level or hardware-dependent operations present problems.

If you use third-party libraries in your program, you may not have access to the source code for the library. What if there is a protected-mode violation in the library? Here are some solutions:

- Seek assistance from the library vendor. Problems are usually easy to correct and rarely require maintenance of a distinct version for protected mode.
- If the violation is in a small module, you may be able to write a replacement for that module. A good example of this is the CA-Clipper/Exospace library, EXOSPACE.LIB, which includes replacement modules for violations in CA-Clipper and the Microsoft C 8.0 runtime libraries. Remember that if you redefine a module, you may need to include a MODULE...FROM command to ensure that your version of the module is linked.
- Execute the library in real mode. You can either link the library as a separate program and communicate with interrupts, or you can use other special functions from the CA-Clipper/Exospace library for managing memory and performing inter-mode calls.

Environment Compatibility Issues

This section provides a detailed discussion of possible environment-related compatibility issues, specifically in the areas of:

- PC/AT compatibility
- Extended memory
- DOS versions

PC/AT Compatibility

CA-Clipper/Exospace works on all IBM PC/AT-compatible computers. There are, however, some machines that claim AT compatibility, but fail in certain important respects—an 80286 CPU does not make a machine compatible. The BIOS code must also be compatible.

Because the 80286 does not have a direct way to switch from protected mode to real mode, CA-Clipper/Exospace forces the switch by requesting a reset (much like power on or rebooting). When it receives the reset, your ROM BIOS startup code must be able to distinguish a restart from a power up, in order to return to the CA-Clipper/Exospace kernel.

The 80386 and 80486 processors have a direct way to switch from protected mode to real mode, If you have an 80386 or 80486, you will not encounter this problem.

CA-Clipper/Exospace will run on any PC that:

1. Has an 80286, 80386, or 80486 CPU.
2. Can run MS-DOS or compatible operating system, Version 3.0 or later.
3. Supports IBM-PC/AT BIOS Interrupt 15h function 88h (get extended memory size).

4. Is compatible with the IBM PC/AT in these respects:
   - CMOS RAM (for saving time/date, configuration data, and the restart code) accessible via I/O ports 70h and 71h
   - Intel 8259-compatible Programmable Interrupt Controller, accessible via I/O ports 20h and 21h (additional controllers may be cascaded to the first)
   - Intel 8042 auxiliary processor, with control port at I/O 60h, and status port at I/O address 64h
   - "AT-compatible" ROM BIOS code that supports some form of restarting

Extended Memory

The CA-Clipper/Exospace program, EXOSPACE.EXE, as well as applications linked with CA-Clipper/Exospace, require extended memory while running. Extended memory is memory addressed by the CPU above 1 MB.

CA-Clipper/Exospace does not work with EMS, as defined by the Lotus/Intel/Microsoft bank-switching scheme, although it can use memory managed by products that use the 386 VCPI standard, as described below.

In its default configuration, the CA-Clipper/Exospace loader utilizes about 90 KB, 18 KB of which is in low memory. Under a VCPI host, the VCPI page tables are also stored in low memory, adding about 15 KB to the low memory footprint. Under DOS DPMI, the stack is also locked in low memory. Under some circumstances, you can reduce the amount of memory that CA-Clipper/Exospace requires.

In most cases, no special action needs to be taken to make extended memory available to CA-Clipper/Exospace. However, to ensure that CA-Clipper/Exospace and applications linked with CA-Clipper/Exospace are running optimally, you may wish to ensure that enough extended memory is available.

If you are running MS-DOS Version 5 or later, you can execute the MEM command to find out how much conventional and extended memory is available.

Two items listed by the MS-DOS MEM command are important to CA-Clipper/Exospace: "available contiguous extended memory" and "available XMS memory." Depending on your machine's configuration, one of these items may be missing or its value may be zero, but as long as one of the two items has a value of 1 MB or greater, you should be able to run CA-Clipper/Exospace and CA-Clipper/Exospace applications successfully.

If neither of these items are listed when you execute MEM, or their values are both zero or less than 1 MB, you may need to change the machine's configuration in some way, perhaps by installing more extended memory or by reducing the amount of memory allocated to any other resources that utilize XMS.

In general, the manner in which a machine is configured via CONFIG.SYS should not be of concern, as long as enough extended memory is available. Software drivers for extended and expanded memory are described in the following section.

Extended Memory Protocols

CA-Clipper/Exospace works with the following protocols that use extended memory:

- DOS Protected-Mode Interface (DPMI)
- Virtual Control Program Interface (VCPI)
- Extended Memory Standard (XMS)

The following table shows examples of products that use these protocols and lists the versions that CA-Clipper/Exospace works with:

Protocol	Products	Versions
DPMI	Microsoft Windows, 386 Enhanced mode	3.0 and later
	OS/2 Virtual DOS Machine	2.0 and later
VCPI	QEMM	4.1 and later
	EMM386	MS-DOS 5 or MS-Windows 3.1 and later
	386MAX	4.01, 4.02, 4.06 and later
XMS	HIMEM.SYS	2.10 and later

When your CA-Clipper/Exospace application starts up, CA-Clipper/Exospace searches the environment for memory managers, in the order listed below. In most cases, CA-Clipper/Exospace uses the first memory manager that it finds:

1. A previously-loaded CA-Clipper/Exospace program with a non-private memory pool

   If it finds a loaded CA-Clipper/Exospace application, the application uses the same memory pool as the first application. (You can create CA-Clipper/Exospace programs that use private memory; this private memory will not be available to subsequently loaded CA-Clipper/Exospace applications.)

   If a previous loaded CA-Clipper/Exospace application is not found, it continues searching.

2. A DPMI host

   If CA-Clipper/Exospace finds a DPMI host, it follows the DPMI protocol. The DPMI host manages memory.

   If it does not find a DPMI host, it continues searching.

3. A VCPI host

   If CA-Clipper/Exospace finds a VCPI host, it follows the VCPI protocol. It also searches for an XMS driver.

4. An XMS driver

   If CA-Clipper/Exospace finds an XMS driver, it follows the XMS protocol to allocate memory from the XMS pool. If both a VCPI host and an XMS driver are found, it allocates memory from the XMS pool and from the VCPI pool.

   With VCPI only, the VCPI protocol is followed. With XMS only, the XMS protocol for allocating memory is followed.

If CA-Clipper/Exospace does not find any memory managers that use one of these protocols, it uses BIOS Interrupt 15h, function 88h on ATs and compatibles to determine how much memory is available to it. On other non-AT-compatible machines, your users must use the DOS16M environment variable to specify the exact extended memory range.

Except under VCPI and DPMI, CA-Clipper/Exospace allocates all available memory, unless you restrict the amount of memory your program can utilize.

Device drivers that allocate their memory from a DPMI, VCPI, or XMS pool pose no problems for CA-Clipper/Exospace. In addition, CA-Clipper/Exospace works with the following device drivers, even when they do not allocate memory from a standard memory manager:

Device Driver	Versions
RAMDRIVE.SYS	2.0 and later
SMARTDRIVE.SYS	2.0 and later
VDISK.SYS	All IBM versions

Other application programs, device drivers, or TSRs (like disk caches) that do not use DPMI, VCPI, XMS, or Interrupt 15h, function 88 for managing extended memory will not work with CA-Clipper/Exospace programs if they start before CA-Clipper/Exospace.

CA-Clipper/Exospace applications will work under environments such as Windows (including Windows 3.0 or later in enhanced mode), Carousel, DoubleDOS, DESQview, and others.

DOS Versions

This section describes CA-Clipper/Exospace compatibility issues with specific versions of DOS.

DOS 3.x to 4.x

CA-Clipper/Exospace runs with DOS Version 4.0, but it does not work with XMAEM.SYS, which supports no extended memory protocol.

DOS 5.0 and 6.0

CA-Clipper/Exospace runs under DOS Version 5.0 and 6.0. These DOS versions allow you to put most of the DOS code in the high memory area (HMA), the 64 KB above 1 MB, saving about 50 KB of conventional memory. They also allow you to put TSRs and device drivers in the upper memory area, between 640 KB and 1 MB, with the LOADHIGH and DEVICEHIGH commands.

To use the features offered in DOS 5.0 or 6.0, however, you must also use a memory manager. These DOS versions come with EMM386, which is a VCPI host, and with HIMEM.SYS, an XMS memory manager upon which EMM386 depends. You can use QEMM or 386MAX instead of EMM386.

CA-Clipper/Exospace programs run well with DOS=HIGH, because they do not use memory in the HMA. In fact, management of the A20 line works better with HIMEM.SYS when you set DOS=HIGH.

To use the LOADHIGH command with EMM386, set the RAM switch to make it an "upper memory block provider." To use the DEVICEHIGH command, set DOS=UMB in your CONFIG.SYS.

For more information about running CA-Clipper/Exospace programs in environments that have memory managers, see Extended Memory earlier in this chapter.

DOS 5.0 Task Switcher

If you use EMM386 and HIMEM.SYS, you should be able to run multiple CA-Clipper/Exospace applications under the DOS 5 task switcher, DOSSHELL, without any problems.

When you use other memory managers (like QEMM or 386MAX), the DOS 5 task switcher takes all the XMS memory, leaving none for the memory manager. To avoid this problem, set the options that reserve EMS memory for the memory manager:

- With QEMM, set the EMB parameter to indicate the maximum amount of XMS memory that will be available to the task switcher
- With 386MAX, set the EMS or the EXT option to indicate the amount of EMS memory that the memory manager will control

This table summarizes the requirements for running CA-Clipper/Exospace programs with the task switcher:

Memory Manager	Requirement for Success
386MAX	Use EMS or EXT parameter
EMM386 + HIMEM	Defaults should work
QEMM	Use EMB parameter

OS/2 Virtual DOS Machines

CA-Clipper/Exospace programs can run in a Virtual DOS Machine within OS/2, Version 2.0 or later; this environment is a DPMI host. Users may need to set one or both of the following parameters for the Virtual DOS Machine:

- DPMI_MEMORY_LIMIT

  This is set to 2 MB by default. If your program needs more than 2 MB, have your user specify a larger limit.

- HW_TIMER

  In order to get an accurate reading for PMINFO, turn the timer ON.

Other parameters may affect your program behavior; this has more to do with the way OS/2 emulates DOS than it does with DOS extension.

To access these parameters, follow this procedure:

1. From the System/Command Prompt window, select the icon for the Virtual DOS Machine.
2. Bring up the menu (Shift+F10).
3. Select Open >Settings from the menu.
4. When the settings box opens, select Session from the tabbed options at the right.
5. Select the DOS settings button. The list of parameters will appear.

For information about these parameters, please refer to your OS/2 documentation.

Chapter 12
CA-Clipper Real Mode Linker—BLINKER.EXE

In This Chapter

Welcome to Blinker, the real mode linker for CA-Clipper. This chapter provides all the information you need to link your applications with Blinker, and explains the benefits of doing so. The following topics are discussed:

- Overview of Blinker
- Upgrade offer
- Blinker command reference
- Blinker function reference

Overview of Blinker

CA-Clipper 5.3 contains a limited version of the full Blinker product produced by Blink*inc*. This real mode CA-Clipper 5.3 version of Blinker features:

High-Speed Linking

High-speed linking of real mode CA-Clipper 5.3 programs.

Dynamic Overlaying

Dynamic overlaying of CA-Clipper, C/C++ and Assembler code within real mode CA-Clipper 5.3 programs to overcome the 640KB memory limit. Dynamic overlaying is the process of breaking a program into small pieces and only loading one piece into memory at a time, instead of keeping the entire program in memory all the time. Once the programmer has specified which part(s) of the program to overlay, Blinker automatically manages the overlays at runtime to make the best use of available real mode memory.

Caching of Program Overlays

Caching of program overlays to EMS/XMS to optimize runtime performance of the overlaid .EXE, plus the utilization of UMBs and the EMS pageframes to execute overlays in the real mode memory above 640KB.

Compression of the Symbol Table

Compression of the CA-Clipper symbol table to reduce memory requirements and the size of the runtime .EXE.

Support for the Paging System

Support for the CA-Clipper paging system to optimize memory usage and improve runtime performance.

Support for "burned in" CLIPPER Environment Variables

Support for "burned in" CLIPPER environment variables in the .EXE for ease of configuration and distribution. Once a CLIPPER environment variable has been "burned" into the .EXE, it cannot be inadvertently overridden by a change of the CLIPPER environment variable on the runtime machine.

Upgrade Offer

Blink*inc* is making an exclusive offer to CA-Clipper 5.3 customers to upgrade to the full version of Blinker. Please see the file entitled BLINKER.TXT in the directory in which CA-Clipper is installed for details and pricing information.

The full version of Blinker is the only product that features incremental linking of both real and protected mode CA-Clipper programs. Incremental linking substantially reduces link times by requiring that only the parts of a program which have changed be relinked.

Blinker also includes a royalty-free DOS extender which can create dual mode programs. This unique dual mode feature allows the creation of a single program which will automatically run in either real or protected mode, depending on the configuration of the runtime machine, on any processor from the 8086 to the Pentium.

Other reasons to upgrade include the following:

- Linking of Windows and OS/2 programs and DLLs.
- Support for C/C++, FORTRAN, BASIC, Assembler and earlier versions of CA-Clipper.
- An integrated swap package to swap out the currently executing program to EMS/XMS or disk and run other large programs from within the main program.
- Full Microsoft CodeView debugger support, creation of demonstration versions, a serialization feature and comprehensive technical documentation.

Blinker Command Reference

Script file(s) are specified on the command line and default to an extension of .LNK. They may be nested, and more than one may be used on the command line, e.g.,

```
BLINKER @Defaults @Other LIB Mylib VERBOSE.
```

This command line instructs Blinker to read the script file DEFAULTS.LNK followed by the script file OTHER.LNK.

It also instructs Blinker to use the library MYLIB.LIB and to display a VERBOSE listing of the .OBJ files being processed. The full names of the commands, or any unique abbreviation of the words which comprise the commands, may be used although it is recommended that at least three letters be used for clarity. For example, the command BLINKER OVERLAY OPSIZE 40 can be abbreviated to BLI OVE OPS 40.

Most Blinker commands may be specified in any order, at any point in the link script file. Typically the first FILE statement should specify the .OBJ containing the main program procedure.

(comment)

Purpose

Indicate text as a script file comment

Syntax

...

Description

Any line in the script file beginning with the character # is treated as a comment and is ignored by Blinker. If the # character occurs within a line, the rest of the line is ignored.

Examples

This example indicates the following text as a script file comment:

```
# To link or not to link, that is the question.
```

// (comment)

Purpose Indicate text as a script file comment

Syntax // ...

Description Any line in the script file beginning with the characters // is treated as a comment and is ignored by Blinker. If the // characters occur within a line, the rest of the line is ignored.

Examples This example indicates the following text as a script file comment:

```
// The main file
```

@ (nested script)

Purpose Specify the name of a script file

Syntax @<*scriptname*>

Arguments <*scriptname*> is the name of the specified script file.

Description Blinker supports nested script files of any size to a depth of 5 levels. The processing of nested script files is similar to a program function call. When Blinker encounters a nested script command, the current script file is suspended, the second script is processed, and then Blinker returns to the first script and resumes processing. When searching for a script, Blinker first looks in the current directory, and then in the path specified in the LIB environment variable. This allows frequently used scripts which are rarely modified, (such as scripts for overlaying a particular library), to be stored in a common directory for convenience.

Examples This example illustrates the use of @ (nested script):

```
# Process the link script 'Myscript' here
@Myscript
```

BEGINAREA

Purpose

Specify the start of a dynamic overlay area

Syntax

BEGINAREA

Description

The BEGINAREA command is used to specify to the linker the start of an overlay area in the output file. Each BEGINAREA should have a corresponding ENDAREA to indicate the end of the overlay area.

With static overlays, BEGINAREAs, and therefore overlay areas, may be nested within one another as long as there is a corresponding ENDAREA for each BEGINAREA. This allows the developer to create a structure of overlays within an area to use the minimum amount of memory for the maximum number of overlay routines.

With Blinker dynamic overlays, as opposed to static overlays, the mutual dependency of each overlay procedure or file is unimportant. Also, the multiple nested overlay areas associated with static overlays are no longer necessary although multiple overlay areas may be helpful in ordering the included libraries. The BEGINAREA and ENDAREA commands are simply treated as markers to the beginning and end of code which is to be overlaid.

For most compilers other than CA-Clipper the .OBJ files listed between a BEGINAREA and ENDAREA are managed at runtime on a segment basis, i.e., the whole CODE segment for each .OBJ is loaded into memory whenever any function or procedure within that .OBJ file is called. The CA-Clipper 5.3 dynamic paging system automatically breaks procedures down into 1K pages for improved execution speed., so this is not a concern for CA-Clipper code.

See Also

ENDAREA

BLINKER CACHE EMS

Purpose

Specify amount of EMS to be used for overlay caching

Syntax

BLINKER CACHE EMS <*nuMax[%]*> [, <*nuMinLeave[%]*>]

Arguments

<*nuMax[%]*> is the maximum number of KB or percentage amount of the total available EMS to be allocated to the overlay cache.

<*nuMinLeave[%]*> is the minimum number of KB or percentage amount of the total available EMS to be left for program use.

Description

The default for BLINKER CACHE EMS is 100 for <*nuMax[%]*>. This command specifies the amount of EMS (LIM version 4.0 and higher) memory to be used for the overlay cache.

Two numeric parameters are used to specify the amount of EMS memory to be allocated and the amount to be left for program use. Both parameters may be specified as either a number of KB, or as a percentage of the total available EMS. The second parameter always overrides the first. For example, if the first parameter calls for 256KB and the second requires 50% of EMS to be left free, then half of EMS will be left free and the cache will fit itself into the remaining memory.

Blinker will never allocate a cache that is larger than the total size of all the overlays contained within the program.

Examples

This example defines the maximum size of the EMS cache as being 50% of available EMS while leaving a minimum of at least 1024KB available for the program:

```
# use a maximum of 50% of available EMS,
# but leave at least 1024KB available
BLINKER CACHE EMS 50%, 1024
```

See Also

BLINKER CACHE XMS, BLINKER ENVIRONMENT NAME

BLINKER CACHE XMS

Purpose

Specify amount of XMS to be used for overlay caching

Syntax

BLINKER CACHE XMS <*nuMax[%]*> [, <*nuMinLeave[%]*>]

Arguments

<*nuMax[%]*> is the maximum number of KB or percentage amount of the total available XMS to be allocated to the overlay cache.

<*nuMinLeave[%]*> is the minimum number of KB or percentage amount of the total available XMS to be left for program use.

Description

The default for BLINKER CACHE XMS is 100 for <*nuMax[%]*>. This command specifies the amount of XMS (version 2.0 and higher) memory to be used for the overlay cache.

Two numeric parameters are used to specify the amount of XMS memory to be allocated and the amount to be left for program use. Both parameters may be specified as either a number of KB, or as a percentage of the total available XMS. The second parameter always overrides the first. For example, if the first parameter calls for 256K and the second requires 50% of XMS to be left free, then half of XMS will be left free and the cache will fit itself into the remaining memory.

Blinker will never allocate a cache that is larger than the total size of all the overlays contained within the program.

Examples

This example defines the maximum size of the XMS cache as being 50% of available XMS while leaving a minimum of at least 1024KB available for the program:

```
# use a maximum of 50% of available XMS,
# but leave at least 1024KB available
# BLINKER CACHE XMS 50%, 1024
```

This example defines the maximum size of the XMS cache as being 512KB while leaving a minimum of 25% of the available XMS memory for the program:

```
# use a maximum of 512KB XMS,
# but leave at least 25% of the total free
BLINKER CACHE XMS 512, 25%
```

See Also

BLINKER CACHE EMS, BLINKER ENVIRONMENT NAME

BLINKER CLIPPER PAGE

Purpose Enable/Disable automatic CA-Clipper 5.3 code paging

Syntax BLINKER CLIPPER PAGE ON | OFF

Description This command can be used to disable the CA-Clipper 5.3 dynamic paging system for CA-Clipper code. The only reason for ever wishing to do this is if a CA-Clipper 5.3 program with no overlays is required, for example, to create an installation program.

BLINKER ENVIRONMENT CLIPPER

Purpose Change the name of the CA-Clipper environment variable

Syntax BLINKER ENVIRONMENT CLIPPER <*cName*>

Arguments <*cName*> is the name of the CA-Clipper environment variable.

Description This command allows the developer to change the name of the CLIPPER environment variable from the default setting of "CLIPPER" to any string of up to 16 characters, and thus avoid the remote site settings for the CLIPPER environment variable.

Used in conjunction with the BLINKER ENVIRONMENT OVERRIDE command, this allows the settings burnt in with the BLINKER EXECUTABLE CLIPPER command to be overwritten at runtime by this newly created variable. The choice of a unique name avoids any conflict with previous settings for the CLIPPER environment variable.

For example, if a script file contains the following commands:

- BLINKER EXECUTABLE CLIPPER F31; X100
- BLINKER ENVIRONMENT OVERRIDE
- BLINKER ENVIRONMENT CLIPPER ACCSYS

Even though the following statement may already have been set from DOS:

```
SET CLIPPER=F21
```

This CLIPPER environment variable will be ignored by the application, and the BLINKER EXECUTABLE CLIPPER setting can be overridden by the following DOS command:

```
SET ACCSYS=F99
```

Examples

This example illustrates the use of BLINKER ENVIRONMENT CLIPPER:

```
# Change the name of CLIPPER variable to Newclip
BLINKER ENVIRONMENT CLIPPER Newclip
```

See Also

BLINKER ENVIRONMENT NAME, BLINKER ENVIRONMENT OVERRIDE, BLINKER EXECUTABLE CLIPPER

BLINKER ENVIRONMENT NAME

Purpose

Specify the name of the environment variable for Blinker runtime options

Syntax

BLINKER ENVIRONMENT NAME *<cName>*

Arguments

<cName> is the name of the environment variable for Blinker run time options.

Description

By default, if the argument is not specified, the default setting is BLINKER. This command is used to change the name of the BLINKER environment variable.

At program runtime the Blinker overlay manager checks the DOS environment for the BLINKER environment variable, and if this is present then the variable is examined for certain runtime parameters.

For a listing of the BLINKER environment variable's valid parameters, see the "Runtime Environment" chapter in the *Programming and Utilities Guide*.

Examples

This example illustrates the use of BLINKER ENVIRONMENT NAME:

```
# Rename the Blinker variable
BLINKER ENVIRONMENT NAME TESTAPP
```

Following linking with the above command, the overlay manager will check the environment for the TESTAPP variable.

Use the following command at the DOS prompt to alter the OPSIZE at program runtime without having to relink.:

```
SET TESTAPP=/0030
```

BLINKER ENVIRONMENT OVERRIDE

Purpose

Specify that environment settings may override defaults

Syntax

BLINKER ENVIRONMENT OVERRIDE

Description

As a default, the variables set using the BLINKER EXECUTABLE CLIPPER command will not be overridden at application runtime. This allows multiple applications with different requirements to run on the same machine without changing the CLIPPER environment settings.

This command is used to tell Blinker to allow the CA-Clipper environment variables set using the BLINKER EXECUTABLE CLIPPER command to be overridden at application run time by the settings of the CLIPPER environment variable. When doing this you may want to consider renaming your CLIPPER environment variable with the command BLINKER ENVIRONMENT CLIPPER.

See Also

BLINKER ENVIRONMENT CLIPPER, BLINKER EXECUTABLE CLIPPER

BLINKER EXECUTABLE CLIPPER

Purpose

Specify default CLIPPER environment settings

Syntax

BLINKER EXECUTABLE CLIPPER *<cEnvironmentstring>*

Arguments

<cEnvironmentstring> is the default CLIPPER environment settings.

Description

This command is used to burn default values for the CA-Clipper runtime environment variables into the executable file. Those settings set using the BLINKER EXECUTABLE CLIPPER command take precedence over those set with the SET CLIPPER command. Parameters not specified with the BLINKER EXECUTABLE CLIPPER command will retain values originally assigned by the SET CLIPPER command. However, if you want the values in the SET CLIPPER command to take precedence, use the command BLINKER ENVIRONMENT OVERRIDE.

The maximum string length that can be burned into a 5.3 application is 128 bytes, and Blinker will ignore all but the LAST environment burn-in command encountered, so all parameters that are to be burned in should be specified in a single BLINKER EXECUTABLE CLIPPER command.

The burned-in environment values are not returned by the function GETENV().

The runtime parameters listed here are applicable to CA-Clipper 5.3. See the "Runtime Environment" chapter of the *Programming and Utilities Guide* for a complete list of these parameters, correct syntax and default values.

Runtime Parameters	Meaning
E:nnnn	Expanded Memory in KB
F:nnn	Number of files
X:nn	EXclude memory in KB
BADCACHE	Protect the EMS pageframe
CGACURS	Disable EGA cursor
DYNF:nn	Number of handles for dynamic paging
NOIDLE	Disable background garbage collection
SWAPK:nnnn	Size of disk swap file
SWAPPATH:'path'	Path for swap file
TEMPPATH:'path'	Path for temporary files

CA-Clipper 5.3 environment settings can always be overridden (regardless of the BLINKER ENVIRONMENT OVERRIDE setting) on the application command line, using the command line switch comment (//) as in:

```
MYAPP //SWAPPATH:'D:\'
```

This would cause CA-Clipper to create the VMM swap file in the root of drive D. Remember that for the application to be able to use 41 file handles, corresponding values have to be set in the CONFIG.SYS, and, if necessary, in the appropriate network configuration.

Examples

This example illustrates the use of BLINKER EXECUTABLE CLIPPER:

```
BLINKER EXECUTABLE CLIPPER F35
```

See Also

BLINKER ENVIRONMENT CLIPPER, BLINKER ENVIRONMENT OVERRIDE

BLINKER EXECUTABLE NODELETE

Purpose Create .EXE regardless of unresolved externals

Syntax BLINKER EXECUTABLE NODELETE

Description By default, Blinker will not create an .EXE if errors occur. This command allows you to override that behavior, although the resulting .EXE is not guaranteed to operate correctly. This can be useful during the development cycle, when not all routines have been defined.

Note: At runtime the program will fail if an unresolved external is called.

BLINKER LINK EMS

Purpose Specify link-time EMS memory use

Syntax BLINKER LINK EMS ON | OFF

Description By default, BLINKER LINK EMS is ON. This command is used to control whether or not EMS memory is used at link time by the virtual memory system within Blinker. There is usually no reason to use this command unless a conflict with some other software is suspected. This command also sets BLINKER LINK PAGEFRAME OFF.

Examples This example disables EMS due to a conflict with a TSR:

```
# disable EMS due to conflict with a TSR
BLINKER LINK EMS OFF
```

See Also BLINKER LINK XMS

BLINKER LINK PAGEFRAME

Purpose Specify link time pageframe use

Syntax BLINKER LINK PAGEFRAME ON | OFF

Description By default, BLINKER LINK PAGEFRAME is ON. This command is used to control the use of the EMS pageframe by Blinker at link time by the virtual memory system within Blinker. There is usually no reason to use this command unless a conflict with some other software is suspected.

Examples This example disables the pageframe due to a conflict with a disk cache:

```
# disable pageframe due to conflict
# with a disk cache
BLINKER LINK PAGEFRAME OFF
```

See Also BLINKER LINK EMS

BLINKER LINK XMS

Purpose Specify link-time XMS memory use

Syntax BLINKER LINK XMS ON | OFF

Description By default, BLINKER LINK XMS is ON. This command is used to control whether or not XMS memory is used at link time by the virtual memory system within Blinker. There is usually no reason to use this command unless a conflict with some other software is suspected.

Examples This example disables XMS use:

```
# disable XMS use
BLINKER LINK XMS OFF
```

See Also BLINKER LINK EMS

BLINKER MESSAGE DUPLICATES

Purpose

Enable warnings for duplicate modules within libraries

Syntax

BLINKER MESSAGE DUPLICATES

Description

By default, BLINKER MESSAGE DUPLICATES is disabled. This command causes Blinker to generate the 1004 link-time warning message for all duplicate symbols encountered at link time. This command can be a very useful debugging tool for identifying duplicate modules among the files and libraries used in linking a program. The first module listed in the Blinker message 1004 is the module which is actually linked. The second module in the 1004 message is the duplicate, and is ignored unless overridden with the MODULE FROM command.

The following modules (and included symbols) are processed at link time and listed in the first half of the VERBOSE output. Each of these symbols will be listed in the generated 1004 messages if another processed symbol has the same name.

- All modules specified in a FILE command.
- Those modules extracted from a library to resolve external references.
- All modules within a SEARCHed library.

Note: There may be numerous duplicated modules/symbols in the included libraries which are not reported when BLINKER MESSAGE DUPLICATES is used, since one or both of the modules are not processed.

To obtain a list of all duplicated modules relevant to your program by using this command in conjunction with the VERBOSE command:

- List all possible duplicates: The first half of the VERBOSE output will identify all modules processed during linking. The duplicates among these modules will be reported by the BLINKER MESSAGE DUPLICATES command. Since you can force all modules within a library to be processed by SEARCHing that library, SEARCHing all included libraries will cause all modules within the program to be processed, and hence cause all duplicates to be listed.

  Linking with all these SEARCHes is only intended to produce the desired duplicate messages, not a viable .EXE.

- Identify only those modules used in the program: The second half of the VERBOSE output will identify only those modules actually included in the final executable file. The symbols highlighted as duplicates whose modules appear in the second half of the VERBOSE listing, are the ones that should be examined. Generally all duplicates between third-party libraries or files which are provided by the same vendor are intended and can be ignored; also those duplicates between a third-party product and a compiler library can be assumed to be intended.

BLINKER MESSAGE NOBLINK

Purpose Disable display of blinking eyes

Syntax BLINKER MESSAGE NOBLINK

Description By default, BLINKER MESSAGE NOBLINK is disabled. This command is used to switch off the blinking eyes which are displayed while Blinker is running.

BLINKER MESSAGE NOWARNING

Purpose Disable display of link-time warning messages

Syntax BLINKER MESSAGE NOWARNING

Description By default, BLINKER MESSAGE NOWARNING is disabled. This command is used to suppress the display of Blinker link-time warning messages 1001 to 1100.

BLINKER MESSAGE WINK

Purpose Enable single eye winking during linking

Syntax BLINKER MESSAGE WINK [LEFT]

Description This command is used to enable the winking of only one eye while Blinker is running. If no parameter is entered, the right eye will wink (from Blinker's point of view, i.e., from inside the screen looking out). The parameter LEFT causes the left eye to wink.

Examples This example displays a winking right eye during linking:

```
# display a winking right eye
BLINKER MESSAGE WINK
```

See Also BLINKER MESSAGE NOBLINK

BLINKER OVERLAY OPSIZE

Purpose Request a size for the overlay pool

Syntax BLINKER OVERLAY OPSIZE *<nuSize>*

Description If the size of the overlay pool is not specified, the default size is 40KB. This command is used to set the operating size of the overlay pool at run time in KB. It can be set to any value between 12KB and 128KB. The default operating size is 40KB. This may be made smaller to save memory or larger to reduce the number of disk accesses made by the overlay manager to re-load overlaid procedures. When overlays are specified in the link script file, Blinker will determine the size of the smallest possible overlay pool by adding approximately 8KB to the size of the largest overlay unit (a segment for most compilers). This minimum value will be displayed along with the value specified in the BLINKER OVERLAY OPSIZE command. If the minimum value is greater than the value specified, the minimum value will override the value specified. The overlay opsize may be adjusted at runtime through the use of the BLINKER environment variable. See the Environment Variable section in this chapter.

Examples This example requests the size of the overlay pool to be increased to 60Kb:

```
# increase the overlay pool size to 60Kb
BLINKER OVERLAY OPSIZE 60
```

See Also BLINKER ENVIRONMENT NAME

BLINKER OVERLAY PAGEFRAME

Purpose

Enable use of the EMS pageframe for overlays

Syntax

BLINKER OVERLAY PAGEFRAME ON | OFF

Description

Warning! Please read this section fully before using this command.

By default, BLINKER OVERLAY PAGEFRAME is OFF. This command will allow the overlay manager to utilize the expanded memory pageframe for the overlay pool (maximum size 64KB), on machines equipped with LIM EMS (3.2 or higher) expanded memory boards or emulators.

Note: This command will be ignored at runtime in the absence of an EMS driver (or pageframe). To use this option there must be 64KB or more of available expanded memory, and the program's overlay OPSIZE must be less than 64KB.

At runtime, if expanded memory is available and this option is enabled, the overlay manager will allocate 4 expanded memory pages, map them into the pageframe, and use the pageframe for overlay execution. The overlay OPSIZE will automatically be expanded to use all 64KB available in the pageframe, resulting in increased execution speed of overlaid code.

If both this command and the UMB command are used, and both the pageframe and a UMB are available, Blinker will use the pageframe in preference. If either or both of these are specified and are not available on the end user's system, Blinker will automatically ignore these commands and allocate the overlay area from conventional memory.

When this command is activated with CA-Clipper 5.3, Blinker will automatically burn in the BADCACHE setting to the CLIPPER environment variable, causing CA-Clipper 5.3 to save the pageframe mapping state between each access to expanded memory. This will protect the overlay area in the pageframe from corruption.

Examples This example enables the use of EMS pageframe for overlays:

```
# use the EMS pageframe
BLINKER OVERLAY PAGEFRAME ON
```

See Also BLINKER ENVIRONMENT NAME, BLINKER OVERLAY UMB

BLINKER OVERLAY THRESHOLD

Purpose Set size of the smallest overlaid segments

Syntax BLINKER OVERLAY THRESHOLD <*nuLimit*>

Arguments <*nuLimit*> is a numeric value representing the size of the smallest overlaid segments.

Description If the size of the smallest overlaid segment is not set, the default size is 32 bytes. This command sets the threshold size below which code specified to be overlaid will be moved back to the root section of the program. Very small routines (less than 32 bytes) take up more root space when they are overlaid, due to overlay manager table entries, than they would if they were simply left in the root. Additionally, very small overlaid routines execute disproportionately slower, since the time overhead in loading/calling them may be large in comparison with the actual execution time of the routine. If it is the case that very small routines are very frequently called, the developer may wish to specify a threshold large enough to cause them to be moved back to the root, improving the performance of the program.

Note: This command has no effect on CA-Clipper 5.3 code overlaid through the paging system but could be used for example, for C functions within a CA-Clipper program.

Examples This example sets the minimum size of overlaid segments to 64 bytes:

```
# only overlay segments larger than 64 bytes
BLINKER OVERLAY THRESHOLD 64
```

BLINKER OVERLAY UMB

Purpose

Enable use of the XMS UMBs for overlays

Syntax

BLINKER OVERLAY UMB ON | OFF

Description

By default, BLINKER OVERLAY UMB is ON. This command used in conjunction with a suitable XMS driver allows the Blinker overlay manager to execute overlays in upper memory blocks above 640KB. Blinker's use of UMBs is dependent on there being a suitably large contiguous block available. The size of the UMB allocated will be the same as the program's overlay opsize.

Use of this command requires a 386 processor (or better), and an XMS 2.0 extended memory driver that supports UMB's (upper memory blocks). UMB's are blocks of memory which are mapped into the area above 640Kb, using the memory management capabilities of the 386/486 family of processors. Upper memory blocks are sometimes referred to as "high memory" or LOADHI areas.

If your XMS driver supports the loading of drivers into high memory, it is likely that you can use this feature. Blinker will need a UMB at least as large as the overlay area, so you may need to alter the placement of your programs in high memory to facilitate this. Most 386 memory managers provide utilities to help optimize this area. If both this command and the PAGEFRAME command are used, and both the pageframe and a UMB are available, then Blinker will use the pageframe. If either or both of these are specified and are not available on the end user's system, Blinker will automatically ignore these commands and allocate the overlay area from conventional memory. This use of UMB's may be adjusted at runtime through the use of the BLINKER environment variable (see the Environment Variable section in Chapter 2).

Examples

This example illustrates the use of BLINKER OVERLAY UMB:

```
# do not use XMS UMB's for the overlay area
BLINKER OVERLAY UMB OFF
```

See Also

BLINKER ENVIRONMENT NAME, BLINKER OVERLAY PAGEFRAME

BLINKER PROCEDURE DEPTH

Purpose

Specify maximum depth of CA-Clipper procedure nesting

Syntax

BLINKER PROCEDURE DEPTH <*nuDepth*>

Arguments

<*nuDepth*> a numeric value representing the maximum depth of CA-Clipper procedure nesting.

Description

If the maximum depth of CA-Clipper procedure nesting is not specified, the default is 50. This command is provided as a convenience to CA-Clipper programmers to allow the setting of the program stack based on the maximum procedure nesting depth which will occur in the application at runtime. This depth is the number of procedure calls that are executed before a procedure return. For example, if procedure A calls procedure B which calls procedure C, the maximum depth is 3. But, if procedure B returns to procedure A which then calls procedure C, the maximum depth is 2. The default depth is 50 procedures, but this may be increased if it is possible for more nesting to occur. For instance, a recursive procedure (one that calls itself) will often require a greater depth. The size of the stack set is approximately 1kb per 10 levels. Thus the default of 50 procedures has an overhead of approximately 5kb, and this will rise proportionally as the PROCEDURE DEPTH is increased. Since both this command and the STACK command control the size of the application stack, only one of these commands should be used (the last one specified is the one that is processed). Some applications making use of third party libraries linked into the root, especially in a network environment, may require an increase in the PROCEDURE DEPTH. Symptoms indicating this include erratic hanging of the machine, spurious characters on the screen and other fatal errors which do not happen regularly.

Examples

This example illustrates the use of BLINKER PROCEDURE DEPTH:

```
# increase stack size for 3rd party libraries
BLINKER PROCEDURE DEPTH 60
```

See Also

STACK

DEFINE

Purpose

Specify symbols to be excluded at link time

Syntax

DEFINE <*symbol*>[, <*symbol*> ...]

Arguments

<*symbol*>[, <*symbol*> ...] are the symbols to be excluded at link time

Description

The DEFINE command is used to "stub out" the named symbols and point them to a dummy routine in the overlay manager. Instead of including the modules containing the named symbols in the final .EXE file, Blinker maps each DEFINEd symbol to a routine within the overlay manager which will display the runtime error 1213 if the routine is ever called. This command can be used (with care) to prevent routines which you know will never be called from being linked in.

Note: In order to prevent a module from being linked in you must DEFINE all its public symbols. Take care to ensure that only code symbols are stubbed using DEFINE. If DEFINEd data items are written to, the overlay manager may be corrupted. When MIXCASE is mentioned prior to any DEFINE commands the symbols named will then be processed as case sensitive.

Examples

This example illustrates the use of DEFINE:

```
DEFINE Alpha, Beta
```

See Also

MODULE, MAP

DEFLIB

Purpose Use default library search records

Syntax DEFLIB

Description The DEFLIB command is used to specify that Blinker should locate and link libraries which are specified within an object module. Some compilers insert information into each generated object module, specifying the names of libraries which the linker may need at link time. This command is enabled by default.

See Also NODEFLIB

ECHO

Purpose Display script or text on the screen

Syntax ECHO [ON | OFF |] [<*cString*>]

Arguments <*cString*> is the character string echoed to the screen.

Description By default, this command is OFF. ECHO <*cString*> causes <*cString*> to be echoed to the screen regardless of, and without affecting, the ECHO status. ECHO ON <*cString*> turns on echoing of the script file to the screen, with the first line displayed being <*cString*> if it is specified. ECHO OFF turns off echoing to the screen of the script file and ignores the rest of the line.

ENDAREA

Purpose Specify the end of an overlay area

Syntax ENDAREA

Description The ENDAREA command is used to specify to the linker the end of an overlay area, as described in the BEGINAREA section.

See Also BEGINAREA

EXTDICTIONARY

Purpose Enable use of library extended dictionaries if they exist

Syntax EXTDICTIONARY

Description By default, EXTDICTIONARY is enabled. This command specifies that Blinker should use the extended dictionary that is appended to library files by some librarian utility programs. The extended dictionary is a cross reference of the library contents, and is used by Blinker to speed up the processing of libraries. Not all librarians provide an extended dictionary—in this case Blinker will process the library slightly slower than if it were present.

See Also NOEXTDICTIONARY

FILE

Purpose

Specify object file(s)

Syntax

FILE [d:][path]<*filename*> [, [d:][path]<*filename*> ...]

Arguments

<*filename*> is the specified object file.

Description

The FILE command is used to specify to the linker the names of one or more .OBJ files to be included in the output file at that point. The default file extension of .OBJ need not be specified. The first file specified must be the program's main module.

Several filenames may be specified within a single FILE command, separated by commas. Blinker will interpret a carriage return/linefeed combination as the end of a FILE command, therefore the FILE command must be restated at the beginning of each line.

All files listed in FILE commands are included in the linked program, whether they are required or not. To include only the files needed by the specific program, a library (.LIB) file should be created using a librarian such as Microsoft LIB.EXE. This library can then be linked into the program using the LIBRARY command, and only the required .OBJ files from the .LIB file will be linked to the program.

Blinker looks for each file first in the current directory and then in each directory path specified in the OBJ environment variable in the order they appear. Alternatively, a full drive and directory path may be specified with any or each of the file names.

The FILE command can be used on a library file to force all the modules within the library to be included in the .EXE. In this case the .LIB extension and full path will probably need to be specified, e.g. FILE path\CLD.LIB. Please use this method with caution.

Examples

This example illustrates the use of FILE:

```
FILE Myapp1, D:\APPS\OBJ\Myapp2
```

See Also

LIBRARY

LIBRARY

Purpose

Specify one or more program libraries

Syntax

LIBRARY [d:][path]<*libname*> [, [d:][path]<*libname*> ...]

Arguments

<*libname*> is the name of the specified program library.

Description

By default, this command specifies compiler defined default libraries. The LIBRARY command is used to specify to the linker the names of one or more program library files required by the program. The default file extension of .LIB is assumed, and need not be specified. If more than one library is specified in a single LIBRARY command, each library name should be separated by a comma. Library files are processed differently from FILE commands in that a LIBRARY command causes only modules which are required by the program to be included. i.e., modules are included from a library if a public definition in the library module resolves an external reference. Object files named in FILE commands are included regardless of whether they are required or not.

The LIBRARY command may be used for both root and overlaid libraries. If the library is to be overlaid, the LIBRARY command should be placed within a BEGINAREA / ENDAREA sequence.

Examples

This example illustrates the use of LIBRARY:

```
LIB C:\LIBS\Mylib1, Mylib2, D:\OTHER\Mylib3
```

See Also

DEFLIB, NODEFLIB, SEARCH

MAP

Purpose Request a segment map of the executable

Syntax MAP [=[d:][path]<*filename*>] [S][,A]

Arguments <*filename*> is the name of the MAP file of the executable that is required.

Description By default, this command is disabled. The MAP command is used to specify to the linker that a map file of the linked .EXE file is required. The map file defaults to the same name as the .EXE file with an extension of .MAP, or the name can be specified in the MAP command. The map file can be listed with varying levels of detail, as required. The two options to the command are as follows: S—All segments—lists the address, length and name of each segment in the executable image. This is the default option, if none is specified. A—Public symbols in address order—lists the public symbols and their memory addresses.

Examples This example illustrates the use of MAP:

```
# generate a map of all public symbols
MAP=C:\TEMP\Mymap A
```

See Also BLINKER MESSAGE DUPLICATES, VERBOSE

MIXCASE

Purpose Specify case sensitivity

Syntax MIXCASE

Description By default, MIXCASE is disabled. This command causes Blinker to treat all publics and externals as case sensitive at link time, rather than converting them to UPPERCASE. This command is of primary interest to developers using C, who may require case sensitivity.

See Also UPPERCASE

MODULE

Purpose

Specify placement of individual library modules

Syntax

MODULE <*module*> [,<*module*> ...] [FROM <*libname*>]

Arguments

<*module*> is the specified module within the library.

<*libname*> is the name of the library specified.

Description

The MODULE command allows individual modules to be forced into the root or the overlay area regardless of whether the originating libraries or object modules are specified within an overlay area or not. The module name refers to an object module that exists within a library which has been specified elsewhere in the link script with a LIB or SEARCH command. Blinker's VERBOSE command output lists all modules that are processed at link time. The position of the module command within the link script determines the placement of the named module within the executable; if the MODULE command is in the root the named modules will be forced into the root; if it is in the overlay area, the named modules will be overlaid. Multiple modules can be specified on the same command line separated by a comma (,).

The library and module names should be specified without path names; if included, they will be ignored. The VERBOSE command can be used to determine the correct module name to use, since it displays each module name as it is processed from the library or object module. This is the same name as used by the Microsoft librarian (LIB.EXE).

MODULE commands are case insensitive, (upper or lower case will be treated the same) unless the MIXCASE command has been specified in the script file before the MODULE command.

The MODULE command has a second mode of operation, whereby a source library is specified using the FROM *<libname>* option. This allows modules from specific libraries to be used in preference to others, and at the same time performs the normal MODULE function of changing the placement of routines in the root or overlay area.

This is useful to resolve a conflict between two libraries which contain modules with the same name. It is advisable to use the FROM option only when necessary, since it modifies the normal linking sequence.

Examples

- This example illustrates the use of MODULE:

```
# Move the module COLOR to the current location
MODULE COLOR
```

- This example forces the modules DBEDIT and MEMOEDIT from EXTEND to be used in preference to modules of the same name which may have been found previously in other libraries or files, AND moves those modules to the current location in the script file:

```
MODULE DBEDIT, MEMOEDIT FROM EXTEND
```

- This example forces the module MEMOEDIT not to be linked at all, since Blinker will be looking for a library called NUL, which cannot exist. If, however, something in MEMOEDIT is explicitly referenced by another module, this will cause an unresolved external. To override this, use the DEFINE command rather than the MODULE command, and DEFINE the public symbols within the module which are unresolved:

```
MODULE MEMOEDIT FROM NUL
```

See Also

BLINKER MESSAGE DUPLICATES, LIBRARY

MURPHY

Purpose

Create a worst case overlay situation

Syntax

MURPHY

Description

The MURPHY command causes the overlay manager to attempt to force any inadvertent overlay errors (programming errors) to manifest themselves during testing or debugging. This command is disabled by default.

When MURPHY is enabled, the overlay manager will only allow a single overlay to be in memory at any given time, allowing the developer to detect 'badly behaved' dependencies between overlays and correct them.

Note: The use of the MURPHY command may slow down the execution of the program significantly.

NOBELL

Purpose

Suppress the beep upon completion of link

Syntax

NOBELL

Description

The NOBELL command is used to specify to the linker that no beep is to sound when the link has completed. This command is disabled by default.

NODEFLIB

Purpose Ignore default library search records

Syntax NODEFLIB

Description The NODEFLIB command is used to specify that Blinker should ignore default library search records which are encountered within object modules. CA-Clipper inserts information into each generated object module, specifying the names of libraries which the linker may require. Blinker defaults to using these records unless NODEFLIB is specified.

See Also DEFLIB

NOEXTDICTIONARY

Purpose Disable processing of extended library dictionaries

Syntax NOEXTDICTIONARY

Description This command specifies that Blinker should ignore the extended dictionary which is appended to library files by some librarian utility programs. The extended dictionary is a cross reference of the library contents, and is used by Blinker to speed up the processing of libraries. If use of the extended dictionary is disabled using this command, Blinker will process the library slightly lower. Some librarians have been known to create invalid extended dictionaries, which can lead to erratic results at link time—this command will ensure that such libraries will be processed correctly. This command applies globally to all libraries processed during the link and is disabled by default.

See Also EXTDICTIONARY

NOTABLEOFCONTENTS

Purpose Disable processing of library table of contents

Syntax NOTABLEOFCONTENTS

Description By default, NOTABLEOFCONTENTS is disabled. This command forces Blinker to SEARCH all libraries, i.e., ignore all the library table of contents. This is necessary for some libraries built with certain versions of Borland TLIB such as the Borland C++ 4.0 libraries. Alternatively you may SEARCH just the libraries containing the unresolved externals. This command applies globally to all libraries processed during the link.

See Also NOEXTDICTIONARY

OUTPUT

Purpose Specify the name of the output executable file

Syntax OUTPUT [drive:][path]<*filename*>

Arguments <*filename*> is the name of the output executable file.

Description The OUTPUT command is used to specify to the linker the drive, path and filename of the output executable file to be created by the linker. If no OUTPUT command is used then the linker will use the name and path of the first object file appearing on a FILE command in the script file (or on the command line). The OUTPUT file name extension does not have to be .EXE, but if no file extension is specified, a default of .EXE is used.

Examples This example specifies the name of an output executable file:

```
# Create a linked file called OUT.EXE
OUTPUT OUT.EXE
```

READONLY

Purpose

Specify that the executable file will be set to read-only status.

Syntax

READONLY

Description

By default, READONLY is disabled. The READONLY command causes Blinker to create the .EXE file with a READONLY file attribute. This command will be of primary interest to developers linking and testing a program over a network, as it prevents sharing violations when executing the program from several workstations.

Note: The READONLY attribute is lost if the file is copied.

SEARCH

Purpose

Prioritize all symbols in the specified libraries.

Syntax

SEARCH [d:][path]<*libname*> [, [d:][path]<*libname*> ...]

Arguments

<*libname*> is the name of the specified library.

Description

The SEARCH command is an alternative form of the LIBRARY command, but operates slightly differently. During the normal linking process, Blinker will make multiple passes over a given library until it is unable to resolve any further external references, at which point processing of the next library begins. If, after all libraries have been processed, there are still unresolved externals, Blinker will make another pass over the libraries and so on.

When Blinker encounters a SEARCH command, it loads all the symbol names from the named library into a table in memory. This has the effect of causing any previous unresolved external references, and any future external references encountered to be resolved from the SEARCHed library (if they are defined there) in preference to a subsequent library. Use of the SEARCH command should be limited to those cases where it is required, as it slows the linking process, and uses large amounts of memory at link time. By default, this command is disabled.

Examples

This example illustrates the use of SEARCH:

```
SEARCH C:\LIBS\MYLIB1, D:\ABC\LIB2
```

See Also

LIBRARY

SECTION INTO

Purpose

Specify files to be placed in an external overlay

Syntax

SECTION INTO <*ovlname*> [FILE [d:][path]<*filename*>

[, [d:][path]<*filename*>...]]

Arguments

<*ovlname*> is the name of the external overlay in which the specified files are to be placed.

<*filename*> is the name of the specified external overlay file.

Description

By default, this command is disabled. The SECTION INTO command is used to specify the name of an external overlay file for the overlays. The file may have any extension except .EXE, and if no extension is specified, then .OVL is assumed. The first SECTION INTO command must precede the first overlaid FILE command, otherwise an internal overlay within the OUTPUT file will be created and all SECTION INTO commands will be ignored.

Note: This command does NOT cause overlays to be created, it merely causes overlays (if any) to be stored external to the EXE file (in OVL files). The BEGINAREA/ENDAREA commands are used to create overlays.

Once an overlay file has been specified using the SECTION INTO command, all overlaid code will be placed in the same file until a SECTION INTO command with a different file name is found. At this time a new overlay file will be created. Output may be spread between the overlay files by the use of multiple SECTION INTO commands. Once an internal overlay has been created, all SECTION INTOs will be ignored and all overlaid procedures will be stored internally.

Once an external overlay has been created it is not possible to create an internal overlay. Blinker has a limit of 15 external overlays in this version.

Note: At program runtime, one file handle is allocated to each overlay file, so it may be necessary to increase the number of file handles available to the program.

Examples

This example illustrates the use of SECTION INTO:

```
FILE MAIN
BEGINAREA
    SECTION INTO OVL1
    FILE ABC, DEF, GHI      # Placed in OVL1
    SECTION INTO OVL2
    FILE JKL, MNO, PQR      # Placed in OVL2
ENDAREA
```

See Also

BEGINAREA, ENDAREA

STACK

Purpose

Specify stack size of .EXE file

Syntax

STACK <*nuSize*>

Arguments

<*nuSize*> is the stack size of .EXE file.

Description

The STACK command is used to specify to Blinker in hexadecimal the stack size in bytes required by the program .EXE file.

Note: A preferable way of increasing the stack size, and therefore the maximum depth of procedure calls is to use the BLINKER PROCEDURE DEPTH command. Blinker's default stack size when linking CA-Clipper programs is 1400h, which corresponds to a PROCEDURE DEPTH of approximately 50.

Examples

This example sets the stack size to 1800h:

```
# Set stack size to 1800h bytes
STACK 1800
```

See Also

BLINKER PROCEDURE DEPTH

UPPERCASE

Purpose Convert all symbols to uppercase

Syntax UPPERCASE

Description The UPPERCASE command is used to specify to the linker that all symbols should be converted to uppercase before being added to the symbol table. Blinker defaults to uppercase.

See Also MIXCASE

VERBOSE

Purpose Display status information during linking

Syntax VERBOSE [0 | 1 | 2]

Description The VERBOSE command is used to specify to the linker that it is to display information regarding the current operation being performed. This command will slow down the linking process significantly, but provides the developer with a means of isolating the cause of a link-time error. This command is disabled by default.

When VERBOSE is specified, Blinker displays each object module and the file containing it, as it is processed. This output may be redirected to a file using standard DOS redirection if required. VERBOSE's numerical parameter indicates the desired level of output detail:

0 disables VERBOSE output
1 outputs filenames while processing
2 outputs filenames and module names
(default if no number is specified)

Examples

This example illustrates the use of VERBOSE:

```
# Creates VERBOSE output, redirects to TEST.MSG
BLINKER @TEST VERBOSE >TEST.MSG
```

See Also

BLINKER MESSAGE DUPLICATES, MAP

WORKFILE

Purpose

Specify drive, path, and file name of the temporary WORKFILE

Syntax

WORKFILE [d:][path]<*filename*>

Arguments

<*filename*> is the name of the temporary WORKFILE.

Description

The WORKFILE command is used to specify to the linker the drive, path and file name of the temporary work file to use at link time if one is required. By default, Blinker will use either XMS (2.0 or higher) or EMS (3.2 or higher) memory in preference to a work file, when available. If no WORKFILE command is specified and Blinker needs to swap to disk during a link, it will create a unique file in the directory specified by the TEMP environment variable if one exists, or the current directory if it does not exist.

Examples

This example specifies drive, path, and file name of the temporary work file "Temp":

```
WORKFILE C:\TEMP\Temp
```

Blinker Function Reference

BLIVERNUM

Purpose

Return the Blinker version number as an integer

Syntax

nuValue := BLIVERNUM()

Return

Blinker version used to link the program.

Description

This function returns an integer containing the version number of Blinker used to link the program. The number represents major version * 100 + minor version, e.g., 100 represents Blinker for CA-Clipper 5.3 version 1.00.

Examples

This example illustrates the use of BLIVERNUM:

```
* BLIVERNUM()*
? "The Blinker version number is "
?? STR (BLIVERNUM()/100, 4, 2)
```

Chapter 13
CA-Clipper Debugger—CLD.LIB

In This Chapter

Once you have finished writing a program, you must test it to make sure there are no errors. The compiler and linker will spot some errors that you can correct without having to actually execute the program, but it is not at all unusual for a program to contain errors which you can detect only while the program is running. Examples of this type of error are runtime and execution errors, cosmetic and other display errors, and logic errors. It is with this type of error that a debugger comes in handy.

CA-Clipper offers two methods for debugging an application, the Workbench debugger, and the DOS-level debugger, CLD.LIB.



The easiest and more user-friendly method for debugging an application is to use the Workbench debugger, which places all the debugging tools within an easy-to-use GUI environment. At the touch of a button, the Workbench debugger allows you to:

- Control the execution of your application
- Evaluate and trace expressions
- Set, reset, and clear breakpoints
- View and modify variables
- Create watch expressions, and much more

This chapter documents the DOS-level debugger only, with an occasional reference to the equivalent Workbench debugger function so that you can compare the two debugging methods. For complete details on how to use the Workbench debugger, see the *Workbench User Guide*.

As its name suggests, a debugger tool helps you track down and remove errors from your source code. CA-Clipper's DOS-level debugger (referred to more simply as the debugger throughout this chapter) is a source code debugger which means that it allows you to view your source code while your program is running. You use the debugger to execute your application so that you can switch back and forth between running the program and viewing the source code to spot potential problem areas.

The debugger provides you with certain features that make it easier to isolate and identify errors. Among these features are the ability to:

- Trace source code line-by-line (single step mode)
- Watch a variable as it is updated (watchpoint)
- Monitor variables by storage class
- Inspect work areas and set values
- Change the value of a variable
- Create new variables and change the values of existing ones
- Execute procedures and user-defined functions linked into your application
- Stop program execution when a variable changes value (tracepoint)
- Stop program execution at a line of code or when a function is called (breakpoint)

When you execute your application using the debugger, all of these features and more are available to you so that you can tightly control how your application runs and keep an eye on things as it does. In this manner, you can quickly and easily identify errors in your source code.

Whenever you notice an error, simply edit the code, recompile, and relink, although you must perform these steps outside the context of the debugger. You may need to repeat this debug/edit/compile/link cycle several times if your program has a lot of errors.

This chapter documents the debugger command line syntax and its various modes of operation, describes the debugger display including all windows and menus that you will encounter, explains how to debug a program using the debugger, and offers a complete menu command reference. These major topics are covered:

- Starting the debugger
- The debugger display
- Debugging a program
- Menu command reference

Starting the Debugger

In order to debug an application, you must first create an executable file (.EXE) using the compiler option for debugging. You can then invoke the debugger and begin the debugging session. Some parts of your application may require source code changes in order to be properly debugged.

Preparing Your Programs for Debugging

To take full advantage of the features offered by the debugger, there are certain things that you should consider. Some programming practices inhibit the debugging process while others require the knowledge of special debugger features. Also, there are a couple of compiler options that you need to know about before you can debug a program. This section discusses special programming, debugging, and compiling issues that you need to consider before attempting to debug your application. See the "Debugging Your Application" chapter of the *Workbench User Guide* for the equivalent information concerning the Workbench debugger.

Programming Considerations

Certain parts of your code present special problems when debugging. This section explains the techniques for debugging multistatement command lines, header files, code blocks, and macros.

Multistatement Lines

CA-Clipper allows you to place more than one program statement on a single line. For example:

```
nNewPage := (nLineNo > 55); ReportPage(nNewPage)
```

When you use the Up arrow or Down arrow in the debugger Code window, the cursor moves up and down one complete line at a time. This is true regardless of how many statements are on the line in question. In the above example, the debugger does not allow you to step through the first and second statements independently nor does it allow you to set a breakpoint at either statement. The entire line is treated as a single entity.

The code in the above example should be broken up into two lines, as follows:

```
nNewPage := (nLineNo > 55)
ReportPage(nNewPage)
```

This makes debugging easier and also makes the code more readable.

Header Files

Header files are files that are referenced using the #include preprocessor directive. They typically contain manifest constant and pseudofunction definitions. The debugger does not automatically display header files in the Code window, but you can view them with the File Open command.

Code Blocks

A code block contains executable program code that is treated by CA-Clipper as data and can be passed as a parameter to other programs. Inspecting a code block in the debugger reveals only its name, not the actual code.

However, code blocks can be executed from within the debugger. The following command, when entered in the Command window, causes the code block *bMyBlock* to be evaluated. Any code within *bMyBlock* is executed and the results are displayed accordingly:

```
? EVAL(bMyBlock)
```

In addition to allowing you to execute code blocks interactively, the debugger also traces code blocks back to their definition as long as Options Codeblock is on. When using single step mode to execute a piece of code containing a code block, the debugger moves the execution bar to the line of code where the block was created each time the code block is evaluated. This allows you to see the contents of the block (which are unavailable during normal inspection) and occurs regardless of whether the block was declared in the current routine.

Macro Substitution

The debugger treats macros as ordinary character strings. As such, they can be inspected or viewed using the ? | ?? command. For example:

```
? &macroVar
```

displays the contents of the expanded macro variable *macroVar*.

Compiling Your Source Code

To debug an application, you must compile with the /B option or, from the Workbench, check Debug Information in the Compiler Options dialog box. This instructs the compiler to include debugging information in the object file (.OBJ). For example:

```
CLIPPER MainProg /B
```

In this example, all programs called by Mainprog.prg are also compiled with debugging information.

If, however, you use a (.clp) file or compile each program in your application separately, you must specify that debugging information be included for each file every time you invoke the compiler. During the debugging session, any subsidiary programs which have been compiled without this option are ignored.

In addition, you must compile *without* the /L option or, from the Workbench, uncheck Suppress Line Number in Obj in the Compiler Options dialog box. Effective debugging is not possible without line numbers in the .OBJ file.

Finally, if you use any of the preprocessor directives, and are interested in viewing the preprocessor output while debugging, you should use the /P option, or check Generate PPO File in the Compiler Options dialog box from the Workbench.

As their name suggests, preprocessor directives are instructions to the preprocessor that do not exist at runtime. This means, for example, that you cannot inspect #define constants and pseudofunctions in the same way as other expressions when you are debugging.

If you compile a program using the /P (or Generate PPO File) compiler option, a file with a (.ppo) extension is produced. This file shows the preprocessor output which can be viewed using the Options Preprocessed command. With this command, the debugger displays each line of source code in the Code window with the output from the preprocessor shown underneath.

Important! *In order to use the debugger, your application must be compiled with CA-Clipper with /B options.*

Invoking the Debugger

There are several ways to invoke the debugger, each of which is discussed below. All of the methods require that the application be compiled with /B and *without* /L (or, from the Workbench, with Debug Information checked and Suppress Line Numbers in Obj unchecked in the Compiler Options dialog box)—debugging is not possible otherwise.

From DOS

To invoke the debugger from the DOS prompt for real mode applications, use the following syntax:

```
CLDR [[/43 | /50 | /S /D] [@<scriptFile>]
   <exeFile> [<argument list>]]
```

CLDR.LIB is the real mode debugger library. If you installed the CA-Clipper development system using the default configuration, CLD.LIB is located in the \CLIP53\BIN directory and your DOS PATH should be altered accordingly.

/43 | */50* | */S* specifies the screen mode used by the debugger. These three options are mutually exclusive and, if more than one is specified, CLDR.LIB uses the last one that occurs on the command line. The default screen mode is 25 lines, with the debugger using the entire screen.

/43 specifies 43-line mode and is available on EGA monitors only.

/50 specifies 50-line mode and is available on EGA and VGA monitors only.

/D specifies that the debugger screen will have focus and the application will be suspended, ready to execute the first line of code.

/S is available on EGA and VGA monitors only. This option splits the screen between your application and the debugger, allowing you to view the application and the debugger simultaneously.

On a VGA monitor /S uses 50-line mode, and on an EGA monitor it uses 43-line mode. In split screen mode, the top 25 lines of the screen are used by your application, and the remaining lines are used for the debugger display.

If the /D option is not used, execution of the application will begin immediately. At this point, to invoke the debugger, you may use the Alt+D keystroke. If the /D option is used, the source code associated with the *<exeFile>* appears in the code window, and you can begin running and debugging your application.

<scriptFile> is the name of a script file with a default extension of (.cld). CLDR searches for the specified *<scriptFile>* in the current directory then searches the DOS PATH. A script file is simply an ASCII text file containing one or more debugger commands, with each command appearing on a separate line. When the debugger is invoked with a script file, each command in the file is executed automatically after the *<exeFile>* file is loaded. See Using a Script File later in this chapter for more information on this subject.

In addition to any script file called for on the CLDR command line, the debugger automatically searches for a script file with the name Init.cld. If a file by this name is located in the current directory or anywhere in the DOS PATH, the debugger executes it as a script file. If both Init.cld and a command line script file are present, Init.cld is executed first, followed by the command line script file.

<exeFile> is the name of the executable (.EXE) file you want to debug. CLDR searches for the *<exeFile>* in the current directory only—the DOS PATH is not searched. As stated earlier, the file must include debugging information and line numbers.

<argument list> is the argument list for *<exeFile>*. There must be a space between *<exeFile>* and *<argument list>*.

Note that with the exception of the argument list, all other CLDR arguments must come before the executable filename on the command line.

If no command line arguments are specified, CLD displays a brief help screen.

Linking the Debugger

For protected mode applications, you *must* link the debugger into your application. Doing this would allow you, for example, to debug a program at a customer's site in the event that an unforeseen problem occurred after the application was already in the field.

To include the debugger as part of your application, the debugger library, CLD.LIB, must be linked into your program as an object file (not as a library file) as in the following example which uses the protected-mode linker:

```
EXOSPACE FI MainProg, CLD.LIB
```

Note: Real mode applications can also use the linked debugger. Instead of using CLD.LIB, the protected mode debug library, use CLDR.LIB for real mode debugging.

When debugging an application with an integrated debugger, real or protected mode, the environment variable, CLIPPERDEBUG, may be used to specify "command line" argument switches. The switches available are discussed here in some detail.

Controlling Execution of Application—/D or /d

Instead of the application beginning execution immediately, as is normal, the debugger screen will have focus and the application will be suspended, ready to execute the first line of code. The /D or /d switch can be used in conjunction with the screen mode switches, following.

Permitting Concurrent Viewing of Application and Debug Screens—/S or /s

The /S or /s option, available with EGA and VGA adapters, permits the application and debug screens to be viewed concurrently by partitioning the maximum lines supported between the application and the debugger.

Setting screen lines—/4(3)

If supported by the adapter, /4[3] sets screen lines to 43.

Setting screen lines—/5(0)

If supported by the adapter, /5[0] sets screen lines to 50.

Note: The screen mode switches are mutually exclusive, the last of one or more will predominate.

Again, your program must be compiled with debugging information and line numbers present. When the debugger is linked into a program in this manner, it does not come up automatically when you run your application. Instead, it must be invoked with Alt+D.

Note: If you deliver your application with the debugger linked as an object file, you should also include the help file, CLD.HLP, located in \CLIP53\BIN in the default installation. When you ask for help, the debugger searches the current directory followed by all directories in the DOS PATH. If CLD.HLP is not found in any of these locations, the debugger will prompt you for a directory name. If CLD.HLP is still not found, you will not have access to help text within the debugger.

From Windows

You can also invoke CLD from within the Workbench. To do so, simply compile your application with debugging information by checking the Debug Information option on the Compiler Options dialog box. You must also include CLD.LIB in the Additional Object Files edit control of the Linker Options dialog box. Once your application is successfully built, simply run the application. This creates a DOS window with CLD active. You can then activate the debugger using Alt+D or ALTD(), as discussed below.



In order to use the Workbench debugger, you must choose the Debug menu's Run option. For more information, please see the "Debugging Your Applications" chapter in the *Workbench User Guide*.

Invoking with Alt+D

When an application is linked with CLD.LIB as one of the object files, the debugger is enabled and can be invoked by pressing Alt+D while the application is running. Doing this invokes the debugger with the execution bar positioned on the current line of executing code.

You can also use Alt+D when the debugger is invoked with the CLD command line. When executing the application in run mode, simply press Alt+D at any time to terminate the application and return control to the debugger. Run mode is one of the many execution modes that is available to you when you debug an application using the debugger. For more information on modes of execution, see Debugging a Program later in this chapter.

Using ALTD()

The ALTD() function serves two purposes: it allows you to control whether the debugger is enabled or disabled, and it allows you to invoke the debugger if it is enabled. ALTD() can be used regardless of whether the debugger is invoked using the CLD command line or using Alt+D.

When it appears in a program compiled with debugging information, ALTD() used without arguments acts like a breakpoint by stopping the application and giving control to the debugger.

ALTD(0) temporarily disables the debugger so that any code following the function call is executed as if the debugger was not present.

When the debugger is disabled, all breakpoints and tracepoints are ignored and even calls to ALTD() and pressing Alt+D will not invoke the debugger.

ALTD(1) is used to enable the debugger once it has been disabled. Note that the default state is enabled for programs compiled with debugging information.

For applications that are linked with CLD.LIB, you can use ALTD() to prevent the user from inadvertently invoking the debugger. By including code similar to the following at the beginning of the main program, the ability to invoke the debugger is controlled by the presence of a command line parameter:

```
PARAMETERS dBug

IF dBug != NIL
   ALTD(1)    // Invoke debugger with ALTD() or Alt+D
ELSE
   ALTD(0)    // Debugger cannot be invoked
ENDIF
```

Using the previous code, the debugger is disabled unless the application is invoked with an argument. Even though the program may contain calls to ALTD() and the user may press Alt+D, the debugger cannot be invoked unless the application command line argument is specified. If your application accepts other arguments, use *dBug* as the last one in the list.

How the Debugger Searches for Files

When you make a file open request to the debugger, you can specify the drive, directory, and file extension explicitly as part of the filename. Otherwise, the debugger makes certain assumptions about the filename and location.

Depending on the context of the file open request, the default file extension supplied by the debugger varies. For example, when the request is for a script file the debugger assumes a (.cld) extension and for a preprocessed output file the default extension is (.ppo). Default file extensions are supplied only when you do not specify an explicit file extension.

If no explicit file location is specified or if the file location cannot be obtained from the application, the debugger looks for the file in the current directory. Then, with the exception of the .EXE file, the debugger searches the DOS PATH. Only after these locations are exhausted will an error message be displayed indicating that the file could not be found.

Important! *The debugger searches for the executable file in the current directory only.*

Using a Script File

The debugger allows you to record commands in a script file and execute the commands directly from the file. A script file is simply an ASCII text file. The file consists of one or more debugger commands, with each command on a separate line, and has a default extension of (.cld).

Some programs may require several debug/edit/compile/link cycles in order to trace persistent errors. Recording repetitive commands in a script file eliminates the amount of typing necessary to reach the same position in your application each time.

For example, the following script file sets a breakpoint at the first call to the function ViewData(), designates the variables *lMadeChanges* and *lSaveChanges* as watchpoints, and specifies *nFieldNum* and *nFileArea* as tracepoints:

```
BP ViewData
Point Watchpoint lMadeChanges
Point Watchpoint lSaveChanges
Point Tracepoint nFieldNum
Point Tracepoint nFileArea
```

Another advantage to using script files is to record preferred debugger settings so that you do not have to set them each time you debug.

The easiest way to record option settings in a script file is to set them using the debugger menus. Then, use the Options Save command as in the following example:

```
Options Save MyScript
```

The script file, Myscript.cld, is saved to disk and can be used or modified at any time. The settings that you can save in this way include most Options menu settings, the Monitor menu settings, and case sensitivity and call stack status.

As an alternative to setting the options and saving them to a script file using the Options menu, you can type the appropriate menu commands directly into a script file. Menu commands are formed using the menu name followed by the first word of the menu option you want to set. Any necessary arguments are placed at the end of the command.

An example of some menu commands recorded in a script file follows:

```
View Callstack
Monitor Local
Options Codeblock
```

When the debugger is active, you can execute a script file with the Options Restore command as in the following example:

```
Options Restore MyScript
```

You can also execute a script file from the debugger command line as in the following example:

```
CLD @MyScript MainProg
```

Each time the debugger is invoked, it automatically searches for a script file with the name Init.cld. If a file by this name is located in the current directory or anywhere in the DOS PATH, the debugger executes it as a script file. If both Init.cld and a command line script file are present, Init.cld is executed first, followed by the command line script file.

Getting Help

The debugger offers online help in the form of a Help window that is divided into two panes: the left pane contains a list of topics for which help is available, and the right pane contains the help text for the currently highlighted topic.

You can activate the Help window three ways. The first is by using the Help command. The second is by clicking F1-Help on the function key line at the bottom of the screen. The third is by simply pressing F1.

There are several main topics of discussion in the Help window including About Help, Keys, Windows, Menus, and Commands. When the Help window is first activated, one of these topics is highlighted on the left, and the text on the right discusses the topic.

To get help on a particular topic, highlight it using the Up arrow or Down arrow or by clicking on it. As you move the highlight, the associated help text on the right changes to reflect the current topic.

If the indicator at the bottom right of the window shows more than one page of information, use PgUp and PgDn to scroll the help text.

To remove the Help window and continue debugging, press Esc.

For more information on the Help window, see The Debugger Display later in this chapter.

Leaving the Debugger

When you have finished your debugging session, enter the File Exit command or simply press Alt+X. The debugger automatically closes all files and returns.

The Debugger Display

Before you can make full use of the debugger, you must be familiar with the various windows, menus, and keys on which the debugging environment is based.



See "Debugging Your Application" in the *Workbench User Guide* for details on its windows, menus, and keys. You can also access the Workbench's online Help system for more information.

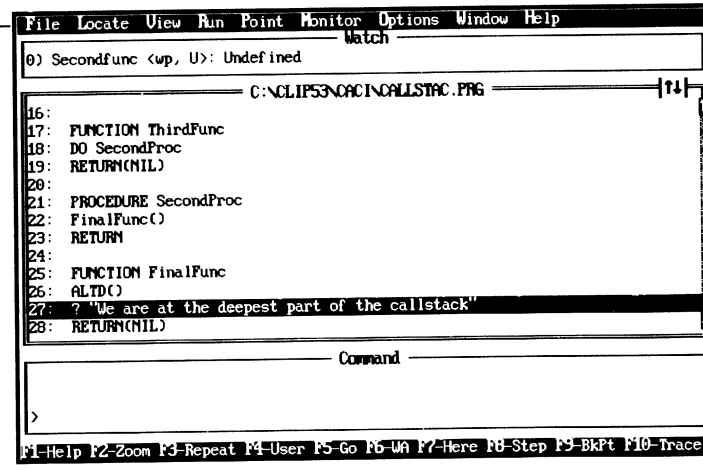
The Debugger Menus

The following section contains a description of the menus available in the debugger and how to access and execute options in those menus. Specific information on the operation of individual menu options, however, is not discussed here. Refer to the Menu Command Reference at the end of this chapter for more detailed information on how menu options work.

The Menu Bar

Menu selections appear on the menu bar at the top of the screen. Each menu contains a group of similar, commonly used options. For example, the Find, Next, Previous, Goto Line, and Case Sensitive options are found on the Locate menu:

Menu bar



Although each of the options has an associated command that performs the same function (in the above example: Locate Find, Locate Next, Locate Previous, Locate Goto, and Locate Case), the menu system provides a quick way for the beginner to learn how to use the debugger.

The File Menu

The File menu lets you view other files and access DOS without leaving the current program.

The Locate Menu

The Locate menu lets you search for a string in a program and move the cursor in the Code window to a particular line number.

Note: If the Command window is active when a Locate menu option is selected, you will not see the cursor move to its new location in the Code window. You must select the Code window in order to see the new cursor position in the file.

The View Menu

The View menu lets you view certain information that is not normally displayed as part of the debugger screen.

The Run Menu

The Run menu lets you run the current application using one of several debugger execution modes. Additionally, it lets you control how much of the application to run using the Code window cursor and restart the current application from the beginning.

The Point Menu

The Point menu allows you to set and delete breakpoints, watchpoints, and tracepoints.

The Monitor Menu

The Monitor menu lets you control the display of public, private, static, and local variables in the Monitor window.

The Options Menu

The Options menu lets you control the debugger display options and create and run debugger script files.

The Window Menu

The Window menu lets you perform certain window operations. including sizing and moving the active window.

The Help Menu

The Help menu lets you activate the Help window. The menu options select the help topic that is initially highlighted when the window is opened.

Menu Operations

Accessing a Menu

To access a menu, click on it, or hold down the Alt key and press the first letter of the menu name. For example, to access the View menu, press Alt+V.

Whenever a menu is displayed on the screen, any menu on the menu bar can be accessed.

- To access the menu to the right of the current one, press Right arrow. If you are positioned on the last menu, the cursor wraps around to the first menu on the menu bar.
- To access the menu to the left of the current one, press Left arrow. If you are positioned on the first menu, the cursor wraps around to the last menu on the menu bar.
- To close the current menu, press Esc or click anywhere outside of the menu.

The following table summarizes the keys discussed in this section:

Key	Action
Alt+<First letter>	Activate designated menu
Left arrow/Ctrl+S	Activate menu to left; wrap if on first
Right arrow/Ctrl+D	Activate menu to right; wrap if on last
Esc	Close menu

Selecting a Menu Option

To select an option from an open menu, either click on it, or:

1. Press Up or Down arrow to highlight the option.
2. Press Enter to select the option and close the menu.

When selecting a menu option, pressing Up arrow on the first option causes the highlight bar to wrap around to the last option; the reverse is true if you press Down arrow on the last option. The following table lists the active keys within a menu:

Key	Action
<Option letter>	Select designated option
Enter	Select highlighted option
Up arrow/Ctrl+E	Move up one option; wrap if on first
Down arrow/Ctrl+X	Move down one option; wrap if on last

Accelerator Keys

Once a menu is open, all options within the menu have an associated accelerator key that you can press to select the option. The accelerator key is a single letter which is usually the first letter of the option name.

Within a menu, the accelerator key for each option is highlighted in the option name. Typing the indicated key is equivalent to highlighting that option and pressing Enter. For example, if the File menu is open, pressing X selects the Exit option.

Shortcut Keys

In addition to accelerator keys, many options also have an associated shortcut key that allows you to select the option without first opening its associated menu. These keynames appear to the right of the option name in the menu.

For example, to select File Exit using its shortcut, press Alt+X.

All shortcut and accelerator key equivalents are documented in the Menu Command Reference at the end this chapter.

Menu Commands

Any menu option can be turned into a command that can be executed from the Command window or from a script file. These commands, called menu commands, are formed using the menu name followed by the first word of the option name.

For example, to execute the Monitor Public menu option type:

```
Monitor Public
```

Arguments can be specified following a menu command if the menu option requires further input. For example, selecting Run Speed prompts you for the step delay using a dialog box.

To specify a delay speed of .5 seconds using a menu command:

```
Run Speed 5
```

Menu commands can usually be abbreviated down to one letter per word, but in some cases a second or third letter is required in the option keyword to distinguish it from another option that begins with the same letter(s).

For instance, instead of Run Speed 5 you can type:

```
R Sp 5
```

The abbreviation "Sp" distinguishes Run Speed from the Run Step command.

All of the debugger commands are documented in detail in the Menu Command Reference at the end of this chapter.

The Function Keys

Many of the function keys used in the debugger are shortcuts to using a menu selection or command. For example, using F9 to set a breakpoint is slightly easier than selecting the Point Breakpoint menu option and much easier than entering the appropriate BP or Delete command. The following table shows a list of all the available function keys and their actions:

Key	Action
F1	Help
F2	Full-screen/window toggle
F3	Retype last command
F4	View application screen
F5	Execute application
Ctrl+F5	Execute until next activation
F6	View Workareas window
F7	Execute application to cursor
F8	Execute application in single step mode
F9	Set/delete breakpoint
F10	Trace

The function key assignments are displayed on the function key line at the bottom of the screen. They can be selected by simply clicking on them.



For details on all of the Workbench function keys, access the Workbench online Help system.

The Debugger Windows

The debugger display is based on a series of windows, each with a unique purpose. The following table gives the name and a brief description of each of these windows:

Window Name	Purpose
Callstack	Display the list of pending activations
Code	Display program code
Command	Display commands and their results
Help	Display help information
Monitor	Display monitored variables
Set Colors	Display and modify color settings
View Sets	Display and modify system settings
View Workareas	Display work area and (.dbf) information
Watch	Display watchpoints and tracepoints



The Workbench also provides the same functionality in its point and click environment. For details on setting colors see the "Working in the Workbench" chapter of the *Workbench User Guide*. The "Debugging Your Applications" chapter has details on the other topics.

Window Operations

This section describes the general behavior of windows that appear on the main debugger screen (that is, Code, Command, Monitor, Watch, and Callstack), including sizing and moving between windows. The instructions regarding selecting menu options are intentionally generic—use the method that best suit your needs, be it clicking the option with the mouse, pressing the appropriate accelerator, shortcut, or function key, or entering the equivalent command in the Command window.

Note: The remaining windows, Help, View Sets, View Workareas, and Set Colors, are not part of the main debugger display. When you open one of these windows, you enter a mode in which the window takes control of the screen and normal debugger operation is temporarily suspended. When you are finished with the window, you must close it before you can continue debugging. The operations in this section do not apply to these modal windows.



For details on the Workbench window operations, see Workbench Basics in the "Working in the Workbench" chapter of the *Workbench User Guide*.

Navigating Between Windows

Of all the windows on the main screen, one is said to be the active window and the rest are inactive. When you select a window, you make it the active window. The active window is indicated by a highlighted border; inactive windows have a single-line border.

- To select a window on the screen, click anywhere on it with the mouse.
- To select the next window on the screen, press Tab.
- To select the previous window, press Shift+Tab.

When a window is active, keystrokes affect only that window. For example, when the Monitor window is active, pressing Down arrow moves the highlight bar to the next variable in the list of monitored variables but does not affect the cursor or highlight bar in any other window.

The exception to this rule is that a command can be typed and executed from any active window. For example if you type "List BP" while the Code window is active, the command will appear in the Command window as you are typing. As soon as you press Enter, the command will be executed and the result displayed in the Command window. Furthermore, Enter always executes the command pending in the Command window (if any), taking precedence over the normal operation of the Enter key for the active window.

Opening and Closing

Not all debugger windows must be open on the debugger screen at all times. The debugger automatically controls the opening and closing of certain windows, while leaving the control of others up to you. This section describes how to open and close each window.

The Code and Command windows are always open and cannot be closed by you or the debugger.

The debugger automatically opens the Watch window when there are tracepoints and/or watchpoints defined. When all tracepoint and watchpoint definitions are deleted, the debugger closes this window.

The debugger automatically opens the Monitor window when one or more variable classes is being monitored. When there are no monitored variables, the debugger closes this window.

You control opening and closing of the Callstack window using the View Callstack command.

Iconizing

Any window on the screen can be effectively put out of sight without actually closing the window. For example, if the Callstack window is open and you are not interested in its contents for the moment, you can shrink it down so small that only the window name is visible.

To iconize a window, click the Down arrow at the top right-hand corner of the window, or:

1. Select the window.
2. Choose the Window Iconize menu option.

   The active window is replaced by an icon (its name) so that you can no longer see its contents.

When a window is iconized, certain window operations, such as sizing, are not available.

To return an iconized window to view, double-click the icon, or:

1. Select the iconized window.
2. Choose the Window Iconize menu option.

   The window is restored.

Zooming

Windows can be zoomed to full-screen in order to view more information at one time.

To zoom a window to full-screen, click on the up arrow at the top right-hand corner of the window, or:

1. Select the window.
2. Choose the Window Zoom menu option.

To return a zoomed window to its original window display, click the restore arrow at the top right-hand corner of the window, or:

1. Select the zoomed window.
2. Choose the Window Zoom menu option.

When a window is zoomed, other window operations such as moving and sizing are not allowed.

Sizing

The height and width of all debugger windows is determined by the display mode and those windows currently open. The size of a window, however, can be changed to suit your particular needs.

To size a window, press and hold the left mouse button on the window's lower right-hand corner while dragging the mouse, or:

1. Select the window you want to size.
2. Choose the Window Size menu option.

   The border of the window changes to a different pattern to indicate that size mode is active.
3. Use the Direction keys to size the window.
4. Press Enter to complete the sizing.

The following table lists some shortcut keys for changing the height of a window:

Key	Action
Alt+G	Grow active window by one line
Alt+S	Shrink active window by one line
Alt+D	Shrink Command window by one line
Alt+U	Grow Command window by one line

Note: You cannot size a window that is zoomed to full-screen using these Alt+key combinations.

Moving

The location of all debugger windows is determined by the display mode and which windows are open at any given time and can be changed to suit your particular needs.

To move a window, press and hold the left mouse button anywhere on the window's border, except a scroll bar or the sizing area, while dragging the mouse, or:

1. Select the window you want to move.
2. Choose the Window Move menu option.

   The border of the window changes to a different pattern to indicate that move mode is active.
3. Use the Direction keys to move the window to the desired screen location.
4. Press Enter to complete the move process.

Note: You cannot move a window that is zoomed to full-screen.

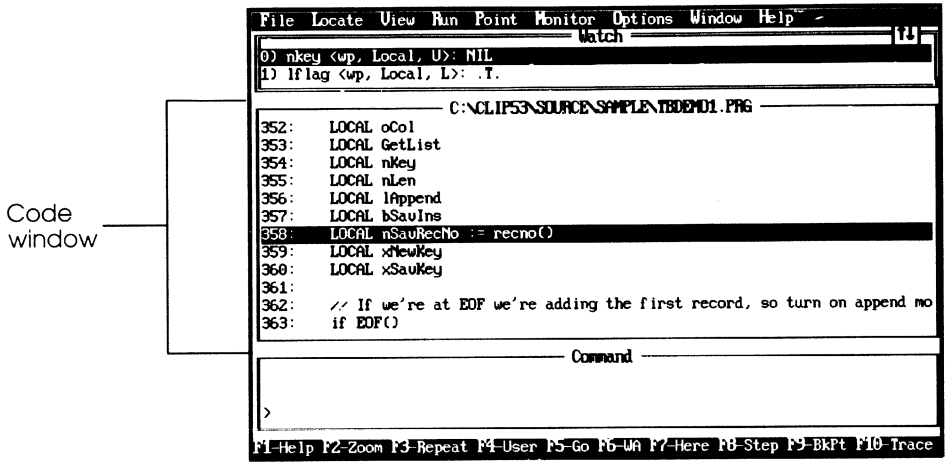
Tiling

The Window Tile command is a quick way to clean up the screen. This command restores each window on the screen to its default location and size. Any windows that have been zoomed or iconized are also restored to the original window display mode.

To tile the debugger windows, choose the Window Tile menu option.

The Code Window

The Code window is initially located underneath the menu bar and is used to display program code and header (.ch) files. Preprocessed output can be displayed beneath program code on a line-by-line basis using the Options Preprocessed command. The name of the file currently being displayed is shown at the top of the Code window.



The Code window is initially set to display a certain number of lines but can be sized to display fewer or more lines (see Window Operations earlier in this chapter for more information). The minimum number of display lines is zero, and the initial and maximum number of lines depend on the display mode.

A scroll bar appears on the right side of the Code window when there is too much code to see at once. Using the mouse, you can scroll:

- Line-by-line by clicking on the Up and Down arrows
- Screen-by-screen by clicking the area above or below the scroll box
- Through a percentage of the code by clicking and dragging the scroll box to the location you want to view

Inside the Code window is a highlight bar called the execution bar which is positioned on the line of code about to be executed. The execution bar moves as execution continues.

When the Code window is active, the cursor appears in the window to indicate your current position in the file being viewed. Initially the cursor and execution bar appear on the same line, but the cursor can be moved up and down using the Direction keys. The cursor is used to show the result of Locate commands, to mark (or delete) a line of code as a breakpoint, and to tell the Run To command where to stop.

To navigate within the Code window, click on a source line or the scroll bar, or use the keys shown in the following table:

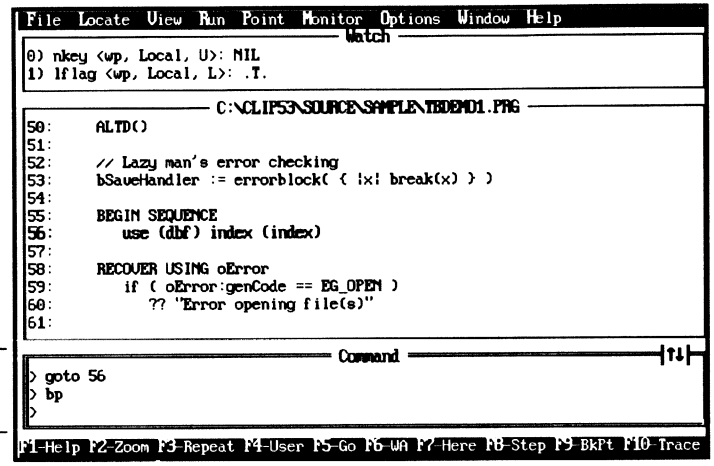
Key	Action
Up arrow/Ctrl+E	Move cursor up one line
Down arrow/Ctrl+X	Move cursor down one line
Left arrow/Ctrl+S	Pan left one character
Right arrow/Ctrl+D	Pan right one character
Home/Ctrl+A	Pan left one screen
End/Ctrl+F	Pan right one screen
PgUp/Ctrl+R	Scroll window contents up
PgDn/Ctrl+C	Scroll window contents down
Ctrl+PgUp	Move cursor to first line
Ctrl+PgDn	Move cursor to last line
Enter	Execute pending command
Tab	Activate next window
Shift+Tab	Activate previous window
F2	Toggle full-screen/window display
F3	Retype last command in command history



For details on viewing the application code using the Workbench, see The Module Browser in the "Browsing Applications, Modules, and Entities" chapter of the *Workbench User Guide*.

The Command Window

The Command window, initially displayed at the bottom of the screen, shows the debugger commands that you enter. The output of debugger commands (if any) is also displayed in this window directly underneath the command.



The Command window is like most other debugger windows in that it can be sized, moved, and zoomed to full-screen when it is the active window (see Window Operations earlier in this chapter for more information).

Note, however, that the Command window does not have to be active in order to enter commands. Commands are entered in the same manner no matter what window happens to be active at the time. To execute a command, type the command and press Enter or make the equivalent menu selection.

The only difference between these two methods of command execution is that commands you type appear in the Command window and are, therefore, available in the command history buffer discussed below—not so with menu selections.

See the Menu Command Reference at the end of this chapter for an alphabetical list of debugger commands. See The Debugger Menus later in this chapter for a brief description of each of the debugger menus and information regarding its use.

The following table gives a summary of keys that are available when the Command window is active:

Key	Action
Up arrow/Ctrl+E	Move cursor to previous line in command history
Down arrow/Ctrl+X	Move cursor to next line in command history
Left arrow/Ctrl+S	Move cursor one character to the left
Right arrow/Ctrl+D	Move cursor one character to the right
Home/Ctrl+A	Move cursor to the beginning of line
End/Ctrl+F	Move cursor to the end of line
Insert/Ctrl+V	Toggle the insert mode on or off
Delete/Ctrl+G	Delete character under cursor
Backspace/Ctrl+H	Delete character to the left of cursor
Esc	Clear command line
Enter	Execute pending command
Tab	Activate next window
Shift+Tab	Activate previous window
F2	Toggle full-screen/window display
F3	Retype last command in command history

History

Commands that you type are saved in a history buffer where they can be accessed when the Command window is active.

To execute a command in the history buffer:

1. Select the Command window.
2. Press Up arrow until the desired command appears.
3. Press Enter to execute the command.

To change a command in the history buffer before executing it:

1. Select the Command window.
2. Press Up or Down arrow until the desired command appears.
3. Edit the command to suit your needs.
4. Press Enter to execute the edited command.

Overwrite and Insert Modes

When the Command window is active, there are two data entry modes: overwrite and insert.

To toggle between overwrite and insert modes:

1. Select the Command window.
2. Press Insert.

   The cursor changes to indicate the current mode.

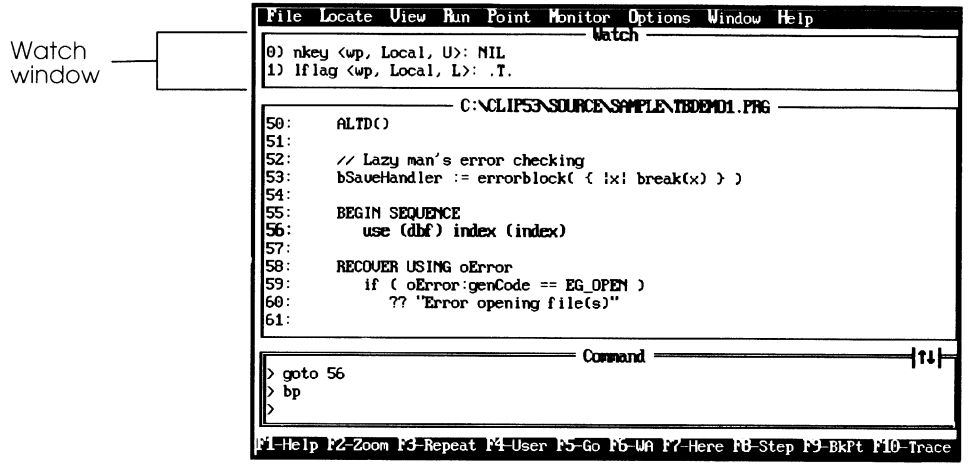
Overwrite mode (the default) is indicated by the underscore cursor. In this mode, new characters that you type overwrite existing characters, causing one character to be deleted for each character typed.

Insert mode is indicated by the block cursor. In this mode, new characters that you type are inserted to the right of the cursor, pushing existing characters to the right.

The Watch Window

The Watch window is initially displayed at the top of the screen, underneath the menu bar and above the Code window. It appears whenever a watchpoint or tracepoint is created.

In this window, the number and name of each watchpoint and tracepoint is displayed along with its data type, value, and storage class (for example, local). Watchpoints and tracepoints are distinguished using the abbreviations "wp" and "tp."



Tracepoints and watchpoints are created using the Point Tracepoint and Point Watchpoint commands, respectively. To remove a point definition, use the Point Delete command or press Delete when the point is highlighted in the Watch window.

When the Watch window is active, you can inspect the value of almost any variable or expression that appears in the window. Code blocks cannot be inspected, and inspecting arrays and objects is slightly more complicated than other data types.

To inspect an array or object:

1. Select the Watch window.
2. Click on the array or object you want to inspect, or press Up or Down arrow to highlight it.
3. Press Enter to view the item in a dialog box.

   The dialog box shows an iconic representation of the array or object.

To inspect array elements or object instance variables:

1. Press Enter again.

   Another dialog box opens in which the individual array or object components can be viewed and changed.
2. Click on the element you want to change, or press Up or Down arrow to highlight it.
3. Press Enter to enter edit mode, or simply type a new value.
4. Press Esc twice to close both dialog boxes.

   All changes that you have made are saved and control returns to the Watch window.

To inspect other data types:

1. Select the Watch window.
2. Click on the variable you are interested in, or press Up or Down arrow to highlight it.
3. Press Enter to view the value in a dialog box.
4. Press Enter to enter edit mode, or simply type a new value.
5. Press Enter to accept the changes or Esc to abandon them.

The dialog box closes and control returns to the Watch window.

The following table summarizes the keys that are available when the Watch window is active:

Key	Action
Up arrow/Ctrl+E	Move highlight bar up one line
Down arrow/Ctrl+X	Move highlight bar down one line
PgUp/Ctrl+R	Scroll window contents up
PgDn/Ctrl+C	Scroll window contents down
Ctrl+PgUp	Move highlight bar to first line
Ctrl+PgDn	Move highlight bar to last line
Enter	Execute pending command or change selected item
Tab	Activate next window
Shift+Tab	Activate previous window
Delete	Delete currently highlighted watchpoint or tracepoint
F2	Toggle full-screen/window display



For specifics on using watchpoints with the Workbench debugger, see Using Watch Expressions in the "Debugging Your Applications" chapter of the *Workbench User Guide*.

The Monitor Window

The Monitor window is similar to the Watch window except that it is used to monitor variables of a particular storage class rather than variables that are set as watchpoints and tracepoints. The Monitor window appears on the screen only when one or more of the storage classes in the Monitor menu is turned on (indicated by a checkmark next to the storage class menu option).

Monitored variables relate to the program currently displayed in the Code window. This is true regardless of whether the program is the current activation or a pending activation that is being viewed using the Callstack window. The values displayed for monitored variables are the values they held when the routine in the Code window was active. Furthermore, the only variables displayed in the Monitor window are those that are visible to the routine in the Code window.

Note: Inspecting monitored variables with the ?|?? command or specifying them as watchpoints or tracepoints yields the *current* value, which may be different from the value displayed in the Monitor window.

To monitor a particular storage class (or stop monitoring one), use the associated Monitor command (for example, Monitor Local for local variables). Each variable name that is being monitored is displayed along with its storage class, data type, and value.

By default, the variables are grouped by storage class. If Monitor Sort is on (indicated by a checkmark beside the Monitor Sort menu option), variables are listed in the Monitor window in alphabetical order by variable name.

When the Monitor window is active, you can change the value of almost any variable in the window by highlighting it and pressing Enter. Editing variables in this window is identical to the method described above for the Watch window and with the same exceptions—code blocks cannot be edited, and editing array values and objects is slightly more complicated than other data types.

The keys for this window are summarized in the table below:

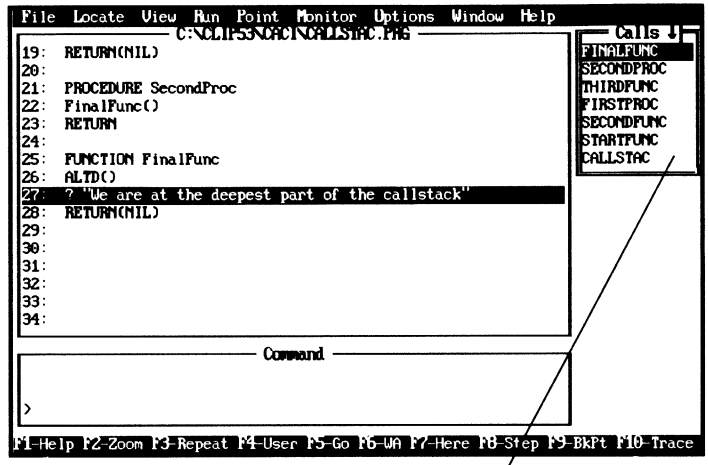
Key	Action
Up arrow/Ctrl+E	Move highlight bar up one line
Down arrow/Ctrl+X	Move highlight bar down one line
PgUp/Ctrl+R	Scroll window contents up
PgDn/Ctrl+C	Scroll window contents down
Ctrl+PgUp	Move highlight bar to first line
Ctrl+PgDn	Move highlight bar to last line
Enter	Execute pending command or change selected item
Tab	Activate next window
Shift+Tab	Activate previous window
F2	Toggle full-screen/window display



You can monitor variables easily using the Workbench debugger. For details, see Evaluating Expressions and Viewing Local and Private Variables in the "Debugging Your Applications" chapter of the *Workbench User Guide*.

The Callstack Window

The Callstack window initially appears on the right side of the screen and contains the names of all pending activations. This list is called the call stack. The current activation is always at the top of the call stack.



Callstack window

To view a pending activation in the call stack:

1. Choose the View Callstack menu option.
2. Select the Callstack window.
3. Click the activation you are interested in, or press Up or Down arrow to highlight it.

   The Code, Monitor, and Watch windows are updated to reflect the highlighted activation.

To return to the current activation, either highlight it or close the Callstack window using the View Callstack command.

The following is a list of keys available when the Callstack window is active:

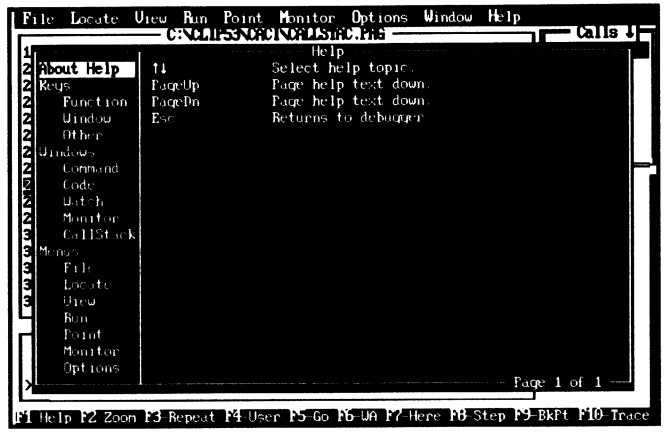
Key	Action
Up arrow/Ctrl+E	Move highlight bar up one line
Down arrow/Ctrl+X	Move highlight bar down one line
PgUp/Ctrl+R	Scroll window contents up
PgDn/Ctrl+C	Scroll window contents down
Ctrl+PgUp	Move highlight bar to first line
Ctrl+PgDn	Move highlight bar to last line
Enter	Execute pending command
Tab	Activate next window
Shift+Tab	Activate previous window
F2	Toggle full-screen/window display



You can view the call stack information easily when using the Workbench debugger. For details, see Viewing the Call Stack in the "Debugging Your Applications" chapter of the *Workbench User Guide*.

The Help Window

The debugger offers online help in the form of a Help window, which is divided into two panes: the left pane contains a list of topics for which help is available, and the right pane contains the help text for the currently highlighted topic. You can activate the Help window by selecting a specific choice on the Help menu using the keyboard or mouse, pressing F1, or entering the Help command.



There are several main topics of discussion in the Help window, and when the window is first activated, one of these topics is highlighted on the left with its associated help text displayed on the right. The following table summarizes the keys used to navigate within the Help window:

Key	Action
Up arrow/Ctrl+E	Move highlight bar up one line
Down arrow/Ctrl+X	Move highlight bar down one line
PgUp/Ctrl+R	Scroll window contents up
PgDn/Ctrl+C	Scroll window contents down
Esc	Leave Help window



In addition to an index of help topics and details on how to use the Help system, the Workbench provides context-sensitive help for all of its windows. To access Help, press F1 or click on the Help menu while working anywhere in the Workbench.

The View Sets Window

When the View Sets window is active, you can view and change the status of the CA-Clipper system settings.

To change a system setting using the View Sets window:

1. Choose the View Sets menu option.
2. Click on the setting you want to change, or press Up or Down arrow to highlight it.
3. Press Enter to enter edit mode, or simply type a new value next to the setting.
4. Press Enter to complete the editing process and move on to the next setting.
5. Double-click on the window close icon in the upper left corner of the window, or press Esc to close the View Sets window and continue debugging.

When you close the View Sets window, the new settings are saved and take effect immediately in your program.



To set system options using the Workbench, see Setting System Options in the "Working in the Workbench" chapter of the *Workbench User Guide*.

The View Workareas Window

The View Workareas window allows you to view database (.dbf) and other work area information. The window is divided into three panes called Area, Status, and Structure.

To activate the View Workareas window, click on View Workareas, or press F6 to execute the View Workareas command. Once the window is open:

- To move to the next window pane, click on it, or press Tab.
- To move to the previous window pane, click on it, or press Shift+Tab.
- To move the highlight up or down within the current window pane, press Up or Down arrow.

The Area pane displays the alias name for each open database file with the active file highlighted. Information regarding the currently highlighted file is shown in the other two window panes.

The Status pane shows the current database driver name and the status of most work area flag settings with information regarding the selected database file underneath. The database information is in the form of an outline that can be expanded and collapsed.

To expand or collapse an entry in the Status pane:

1. Click on the entry, or press Up or Down arrow to highlight it.
2. Press Enter.

Enter acts as a toggle, expanding the entry if it is collapsed and collapsing it if it is expanded. For instance, highlight Current Record and press Enter. The field values usually displayed underneath disappear. Press Enter again, and the field values reappear.

The Structure pane lists the structure of the selected database file.

To close the View Workareas window and continue debugging, double-click on the window close icon in the upper left-hand corner of the window, or press Esc.



The Workbench debugger provides you with the ability to view work areas too. Viewing Work Areas in the "Debugging Your Applications" chapter of the *Workbench User Guide* explains how to view work areas in detail.

The Set Colors Window

When the Set Colors window is active, you can view and change the status of the debugger color settings.

To change a color setting using the Set Colors window:

1. Choose the Options Colors menu option.
2. Click on the color setting you want to change, or press Up or Down arrow to highlight it.
3. Press Enter to enter edit mode, or simply type a new value next to the setting.

   Each setting is a foreground/background color string enclosed in double quotes (see SETCOLOR() in the *Reference Guide* for a list of colors).
4. Press Enter to complete the editing process and move on to the next setting.
5. Double-click on the window close icon in the upper left-hand corner of the window, or press Esc to close the Set Colors window and continue debugging.

When you close the Set Colors window, the new colors take effect immediately but will be lost as soon as you exit the debugger. To save the new colors in a script file for future use, execute the Options Save command.



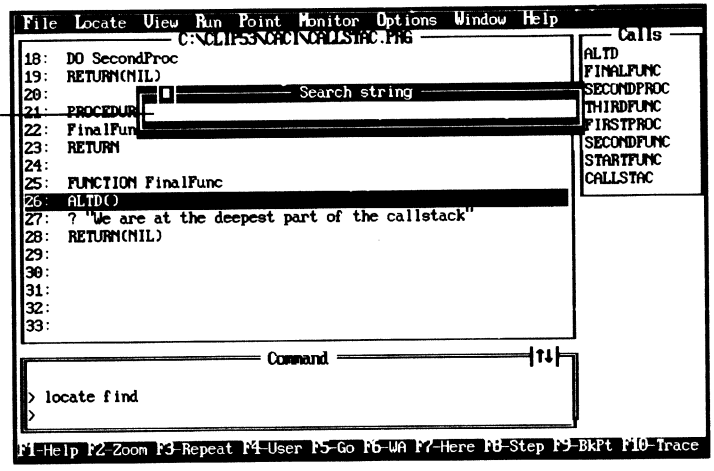
To customize your Source Code Editor window when using the Workbench, see Selecting Source Code Editor Colors in the "Working in the Workbench" chapter of the *Workbench User Guide*.

Dialog Boxes

Several menu options and window selections require that you enter further information into a dialog box before continuing. Menu options that require more input are always indicated by an ellipsis (...) to the right of the option name. Some examples of window selections that require additional information are items in the Monitor and Watch windows.

Dialog boxes are also used if you enter an incomplete command. For example, Locate Find with no search string opens a dialog box to ask for the search string.

Dialog box



The following table summarizes the keys that are available when a dialog box is open:

Key	Action
Left arrow/Ctrl+S	Move cursor one character to the left
Right arrow/Ctrl+D	Move cursor one character to the right
Home/Ctrl+A	Move cursor to the beginning of line
End/Ctrl+F	Move cursor to the end of line
Insert/Ctrl+V	Toggle the insert mode on or off
Delete/Ctrl+G	Delete character under cursor
Backspace/Ctrl+H	Delete character to the left of cursor
Esc	Close dialog box without executing
Enter	Execute and close dialog box

Debugging a Program

In order to make full use of the information presented so far in this chapter, this section provides a more detailed study of the most powerful features, together with suggestions on the most appropriate places to use them.

When you are told how to accomplish a particular task, this section often refers to the debugger command without mentioning any of its shortcuts, such as a function key equivalent. Keep in mind, however, that there are almost always menu and key equivalents that you can use to make debugger command execution easier. The Debugger Display section earlier in this chapter describes how commands are formed from menu options and discusses key equivalents. You can also use the Menu Command Reference at the end of this chapter to look up any command and quickly see its key equivalents.

Executing Program Code

The most basic function of a debugger is to execute an application and display the results. This section describes the various ways to run a program and explains how to control the speed and view the output.

Modes of Execution

The debugger provides several different ways to execute a program, called execution modes. The following table lists each of these modes together with the command and/or function key that is used to access the mode:

Execution Mode	Command/Function Key
Animate	Run Animate
Run	Run Go, F5
Run Next	Run Next, Ctrl+F5
Run to Cursor	Run To, F7
Single Step	Run Step, F8
Trace	Run Trace, F10



For the execution commands available with the Workbench debugger, see Execution Commands in the "Debugging Your Applications" chapter of the *Workbench User Guide*.

Animate Mode

To run a program in animate mode, use the Run Animate command (there is no function key equivalent). This execution mode allows you to execute large portions of an application, stopping when the value of a suspect variable or expression changes.

In animate mode, the debugger executes the line of code currently highlighted by the execution bar, moves the execution bar to the next line of code, and continues executing sequentially in this manner until it reaches a breakpoint or a tracepoint.

When a program is executing in animate mode, you can voluntarily terminate the program at any time by pressing Alt+D, assuming this key has not been disabled with ALTD(0). Doing this returns control to the debugger where the execution bar will be positioned on the next line of code to be executed.

The Options Exchange command controls whether the debugger displays program output. By default Options Exchange is on, causing the debugger to display the output of each line of code after it is executed.

The Run Speed command controls the speed at which animate mode executes.

Run Mode

Run mode is the fastest mode of execution offered by the debugger. In this mode, the debugger runs your application in much the same manner as it would in animate mode. The difference between these two execution modes is that run mode does not return to the debugger display after each line of code is executed.

If you have not set any breakpoints or tracepoints, when you execute an application in run mode the application runs just as if you had executed it without using the debugger. The only difference is that when the application terminates, it returns control to the debugger rather than the operating system.

If you have set breakpoints or tracepoints, the application runs until it reaches one then returns control to the debugger.

When a program is executing in run mode, you can voluntarily terminate the program at any time by pressing Alt+D, assuming this key has not been disabled with ALTD(0). Doing this returns control to the debugger where the execution bar will be positioned on the next line of code to be executed.

There are three versions of run mode. The first is the one described above which you enter by pressing F5 or selecting the Run Go command.

The second is identical to the first except that the debugger runs the application only until the start of the next activation. This mode is accessed by pressing Ctrl+F5 or selecting the Run Next command and is equivalent to setting a tracepoint of PROCLINE() = 0.

The third is identical to the first except that the debugger runs the application only to the current cursor position in the Code window rather than to the application's end. This mode is accessed by pressing F7 or selecting the Run To command.

Single Step Mode

To run a program in single step mode, press F8 or select the Run Step command. This execution mode allows you to step through each line of code, viewing the output and examining variables as you go. Ideally, you will use single step mode in situations where the error you are looking for has been narrowed down to a few lines of code.

In single step mode, the debugger executes the line of program code highlighted by the execution bar, moves the execution bar to the next line of code to be executed, and stops. As functions are called by the current program, the debugger displays the function code in the Code window.

If a code block is evaluated when executing a program in single step mode, the debugger moves the execution bar to the line of code where the block was created. This allows you to see the contents of the block (which are unavailable during normal inspection), and occurs regardless of whether the block was declared in the current routine. Press F8 again to move past the code block definition to the next executable line.

You can control this behavior by selecting the Options Codeblock command. By default, Options Codeblock is on and code block tracing is enabled. Turn Options Codeblock off if you prefer not to view the code block definition.

Trace Mode

To run a program in trace mode, press F10 or select the Run Trace command. This execution mode is similar to single step mode in that it allows you to execute one line of program code at a time. However, trace mode does not display the code for functions and procedures called by the current program nor does it trace code blocks back to their definition.

Finding Program Errors

As you have already seen, the debugger provides several ways to execute a program. The method you use depends on how far you have progressed in your debugging effort. The following sections explain three common debugging stages and suggest an appropriate method for each stage.

Starting Out

The easiest way to debug a program for the first time is to execute the application in run mode to show the program output in the context of the application. You will use the Run Go command (or simply press F5) to invoke run mode.

Simple runtime errors can be identified using run mode. When a runtime error is encountered, control is automatically returned to the debugger where you can see the offending line of code. A runtime error normally causes the application to terminate prematurely because continued execution is prohibited by the error. Thus, you usually have to correct the error, recompile, and relink before you can continue debugging.

Cosmetic errors, such as mistakes in screen design, can also be identified in run mode, but it usually is not necessary to correct them before continuing since they do not affect the actual execution of the program. As soon as you notice a cosmetic error, press Alt+D to invoke the debugger and make note of the code that needs to be changed.

After you have identified and noted all cosmetic errors, you will probably want to go ahead and edit your source code to make the necessary corrections. Then you must compile and link the application again so that the changes will take effect the next time you execute.

Note that all source code changes must take place outside of the context of the debugger (that is, you must leave the debugger to make them) since the debugger has no facility for correcting errors on-the-fly.

For more complex problems such as mistakes in program logic or other runtime errors that do not have an obvious solution, continue to the next stage of debugging.

Taking a Closer Look

When the cause of a problem is not immediately obvious, the next step is to examine the contents of suspected fields, variables, and expressions. Often, you can use the ? | ?? command to reveal an unexpected value for one of these items. If this is the case, specify the item as a watchpoint or tracepoint. This allows you to pinpoint exactly where in the application the erroneous value was stored.

If you suspect that several variables in a particular storage class may be at fault, use the options on the Monitor menu to display these variables in the Monitor window. For example, to display private variables, select the Monitor Private command. As you continue your debugging session, the private variables visible to the routine displayed in the Code window are updated in the Monitor window.

After setting up the necessary watchpoints and tracepoints, use the Run Animate command to run the program slowly, displaying the code for each line executed (animate mode). Use the Run Speed command to control the speed of the display.

Continue this process until the problem has been confined to a few lines of code. Now it is time to move on to the final stage.

Final Stage

When the cause of an error has been narrowed down to a few lines of code, move the cursor in the Code window to the first of these lines and use the F9 key or the Point Breakpoint command to set a breakpoint there. Then press F5 to execute the application in run mode up to that breakpoint.

When you have reached the suspect area of code, press F8 to step through the code one line at a time (single step mode). If you are certain that the error is in the current program and not in any subsidiary procedures or functions, use F10 instead (trace mode). Remember that in trace mode the debugger does not display the code for called routines.

Executing your application in single step or trace mode while watching the values of the items in the Watch window is usually sufficient to uncover the cause of a problem. Having done that, simply alter your code, recompile, relink, and begin searching for the next error.



You can easily set breakpoints then run your application from the debugger when using the Workbench. For details, see Resolving Runtime Errors in the "Debugging Your Applications" chapter of the *Workbench User Guide*.

Viewing Program Output

There may be times when you want to look at the output of a program rather than the code. To do this, press F4, the function key equivalent for View App. When you do, the debugger erases the screen and displays the current output exactly as the person running the application will see it. Press any key to return to the debugger display.

Alternatively, you can view program output following execution of each individual line of code by running the application in animate mode (using the Run Animate command) with Options Exchange turned on.

Inspecting Data and Expressions

One of the most common uses for a debugger is examining (inspecting) and altering the values of variables and expressions. The debugger provides several ways to do this, each of which is explained in the following sections.



Using the Workbench, you can inspect and modify the values of expressions quickly and easily. For details, see Resolving Runtime Errors in the "Debugging Your Applications" chapter of the *Workbench User Guide*.

Using Watchpoints and Tracepoints

Watchpoints and tracepoints are two of the most valuable features of the debugger. They allow you to examine fields, variables, and more complex expressions and to pinpoint exactly where in your application their values change. This section gives a definition of watchpoints and tracepoints and explains how they are set, deleted, and inspected.

Definition of Watchpoints and Tracepoints

A watchpoint is an expression whose current value is displayed in the Watch window. The value or each watchpoint is updated as your application executes.

A tracepoint is similar to a watchpoint; however, whenever its value changes, program execution pauses and control passes to the debugger. In this respect, a tracepoint is similar to a breakpoint.

Setting Watchpoints and Tracepoints

To set watchpoints and tracepoints use the Point Watchpoint and Point Tracepoint menu commands. If you choose the menu options, the debugger opens a dialog box where you are prompted to enter an expression.

If you use the Point Watchpoint or Point Tracepoint commands, you can specify the expression on the command line as in the following examples:

```
Point Watchpoint nPageNum
Point Tracepoint nInvNum = 1000
```

In either case, the debugger creates the watchpoint or tracepoint and displays it in the Watch window, together with its type, value, and the abbreviation "wp" or "tp."

Deleting Watchpoints and Tracepoints

To delete an individual watchpoint or tracepoint, use the Point Delete menu option. When the dialog box appears, enter the number that appears on the far left-hand side of the Watch window.

Alternatively, you can use the Point Delete command and specify the watchpoint or tracepoint number on the command line or simply highlight the point in the Watch window and press Delete. No matter which method you use, the specified entry is removed from the Watch window and the numbers of all the remaining entries are updated.

To delete all watchpoints and tracepoints, use the Delete command as follows:

```
Delete All WP
```

Note that since Delete does not distinguish between watchpoints and tracepoints, you could have used Delete All TP just as effectively.

Warning! *Be careful not to use Delete All on its own unless you want to delete all watchpoints, tracepoints, and breakpoints.*

Listing Watchpoints and Tracepoints

You can always see watchpoints and tracepoints in the Watch window as long as it is open. You can also display them in the Command window using the List command:

```
List WP
```

Note that since List does not distinguish between watchpoints and tracepoints, you could have used List TP just as effectively.

Inspecting Watchpoints and Tracepoints

To inspect a watchpoint or tracepoint and change its value, select the Watch window. Then, move the highlight bar to the item you wish to inspect and press Enter. The current value of the expression is displayed in a dialog box. Within the dialog box, you can either enter a new value and press Enter or press Esc to leave the variable as it is.

Note that although you can inspect the value of almost any expression that appears in the Watch window, code blocks cannot be inspected, and inspecting arrays and objects is slightly more complicated than other data types.

When you are finished inspecting items in the Watch window, you can select another window. For more information on the keys used to navigate the Watch window and for specific instructions on inspecting arrays and objects, see The Debugger Display earlier in this chapter.

Creating New Variables

When you enter "?" followed by an expression, the expression is evaluated and the return value displayed in the Command window. This is a useful feature that allows you to execute procedures and functions linked into your application. It also allows you to make inline assignments to change existing variables or create new ones, which can be handy in certain situations.

Sometimes, for example, an error may be caused by something as simple as a misspelled variable name. In this case, you may want to create the variable from within the debugger. Ultimately, you will have to address this type of error in your source code, but it is very convenient to make a temporary fix from within the debugger so that you can continue the debugging process without having to recompile and relink right away.

Using the ?|?? command, you can create a variable and assign a value to it. For example:

```
? myVar := 100
```

This creates a variable called *myVar* and assigns to it a value of 100. This variable is private to the routine being executed when it was created.

Note: When attempting to create a variable in this manner, you must use the inline assignment operator (:=). If you use the = operator, the debugger will perform a comparison and most likely return an error message.

Inspecting Program Code

As you debug an application, you may need to examine the code of programs other than the one currently being executed. This section describes how to view other programs and header files, how to use the call stack to access programs that have already been executed, and how to use breakpoints.

Using Breakpoints

Breakpoints are similar to (and just as powerful as) watchpoints and tracepoints. However, breakpoints refer to actual lines of code rather than variables and expressions.

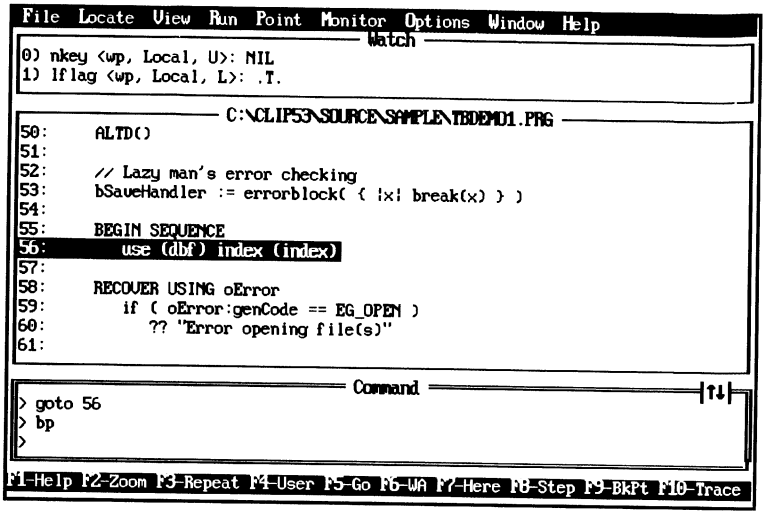
Breakpoints allow you to return control to the debugger and execute problem areas of code in single step mode (for an explanation of the modes of execution, see Executing Program Code earlier in this chapter). This section explains what breakpoints are and how to set, delete, and list them.

Definition of a Breakpoint

A breakpoint is a line of program code or a function or procedure call that, when encountered, causes the debugger to halt execution of the program. In other words, a breakpoint defines a physical breaking point in a program.

Setting Breakpoints

To set a breakpoint in the current program, move the cursor to the appropriate line in the Code window and execute the Point Breakpoint command (or simply press F9). The selected line is highlighted:



To define a function or procedure as a breakpoint, use the BP command instead of Point Breakpoint. These commands are similar in that they both define breakpoints, but BP is more flexible and powerful. The following command defines the routine SayData() as a breakpoint. Whenever a call is made to this routine, the debugger halts execution:

```
BP SayData
```

You can also set breakpoints in programs other than the one currently displayed in the Code window using the BP command. This example sets a breakpoint at line 15 in the program Opendbfs.prg:

```
BP 15 In OpenDbfs.prg
```

Deleting a Breakpoint

There are several ways to delete an individual breakpoint. The first is to move the cursor to the appropriate line in the Code window and execute the Point Breakpoint command (or simply press F9 or issue BP command with no parameters). When a breakpoint is deleted, the line returns to its normal color.

Another way to delete a breakpoint is to use the Delete BP command followed by the breakpoint number on the command line. Note that breakpoint numbering begins with zero. If you do not know the number, use the List command.

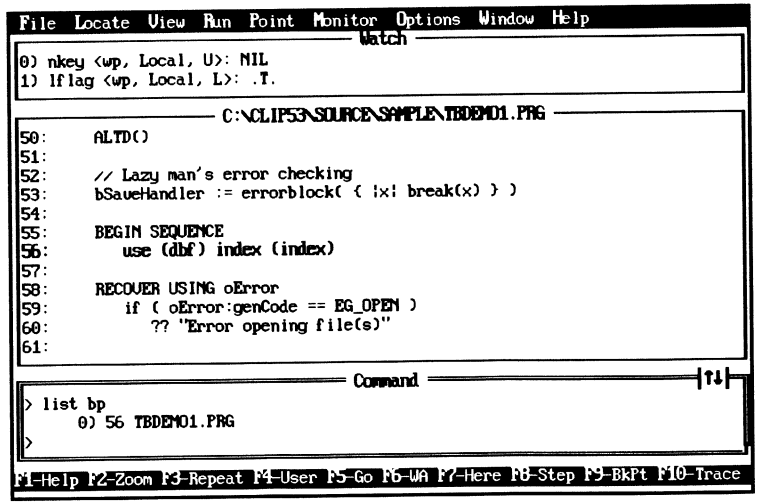
To delete all breakpoints, use the Delete command as follows:

```
Delete All BP
```

Warning! *Delete All without any other arguments also deletes all watchpoints and tracepoints.*

Listing Breakpoints

To display all breakpoints, use the List BP command. This lists and numbers breakpoints in the order they were entered, as shown in the Command window below. Note that numbering begins with zero:



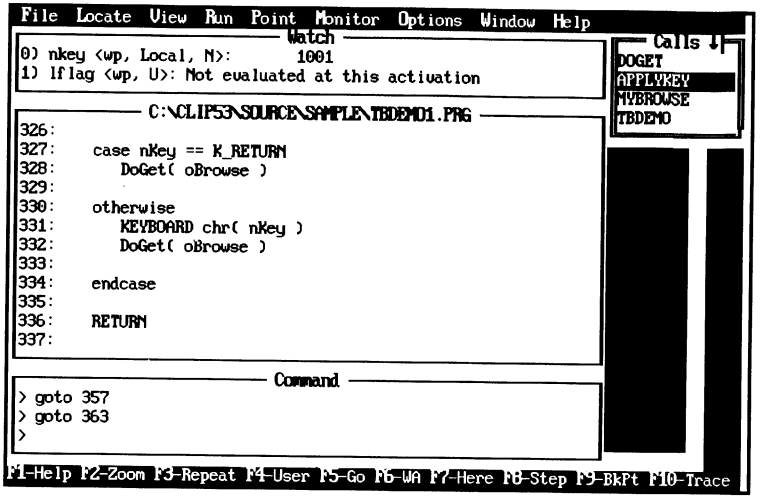


Setting and viewing breakpoints is easy using the Workbench. See Resolving Runtime Errors in the "Debugging Your Applications" chapter of the *Workbench User Guide.*

Navigating the Call Stack

The call stack is a list of pending activations, including procedures, functions, code blocks, and message sends. It is displayed in the Callstack window on the right-hand side of the screen, which is controlled using the View Callstack command.

When it is active, the Callstack window allows you to highlight a program in the call stack and view it in the Code window.



When the Callstack window is active, the highlight bar appears on the first item in the window. You can use Up and Down arrow to move to the activation whose code you wish to view. As you move the highlight within the window, the Code, Monitor, and Watch windows are immediately updated to reflect the highlighted activation.

If the activation you have chosen is a code block (denoted by a lowercase "b"), the name in the call stack is the name of the routine in which the block was created. The code for this routine appears in the Code window.

To return to the program originally being debugged, use the File Resume command. For a list of the call stack navigation keys, see The Debugger Display earlier in this chapter.

The call stack is concerned only with executable code. If you want to view the contents of a header file you must use the procedure outlined in the next section.

Viewing Files

Although inspecting the call stack provides an easy way to select and view program code, it does not display header files. If you already know the name of the program you want to view or you want to examine a header file, use the File Open command.

The following example displays the code for the header file Inkey.ch in the Code window. Notice that this example gives a file extension; if none is supplied, a (.prg) extension is assumed:

```
File Open \CLIP53\INCLUDE\Inkey.ch
```

You can also use the File Open command to view other program files in your application. This feature is convenient if you want to set a breakpoint in another program but do not know the line number and cannot, therefore, set it using the BP command.

To return to the original program, use the File Resume command. The debugger saves any breakpoints that you have set and respects them the next time you run the application.

For a list of navigation keys used within the Code window, see The Debugger Display earlier in this chapter.



For information on viewing and editing various types of files using the Workbench, see the "Browsing Applications, Modules, and Entities" and "Using the Source Code Editor" chapters of the *Workbench User Guide*.

Accessing DOS

There may be times when you wish to execute a DOS command without leaving the debugger. For example, you may want to see a directory listing of all program (.prg) files. To do this, use the File DOS command. The DOS prompt will be displayed, and you can execute any DOS command you like.

Warning! *Accessing DOS invokes a temporary copy of COMMAND.COM. Make sure that you have sufficient memory available to load this file and any others you may wish to execute. Remember that you are using a temporary copy of DOS, so any environment variables you set will be lost upon returning to the debugger.*

When you have finished and wish to continue executing your application, type "Exit" at the DOS prompt.

Menu Command Reference

This section is a complete reference to the syntax and operation of all the debugger commands. You can execute debugger commands by selecting options from the menu bar or by entering commands directly into the Command window. You have seen the former method described in The Debugger Display section and the latter method used in the examples in Debugging a Program earlier in this chapter.

The important thing to remember about the debugger is that any menu option can be expressed in the form of a command using the menu name followed by the first word of the option name. If required, arguments are placed at the end of the menu command line. Selecting a menu option and entering the equivalent menu command have the same effect.

All of the menu command equivalents and several other commands that have no menu equivalents are explained in this section. To execute a command, either type the command and press Enter to execute, or make the equivalent menu selection if one is available. The only difference between these two methods of command execution is that commands you type appear in the Command window and are, therefore, available in the command history buffer, which is not so with menu selections.



When using the Workbench, the available menu selections varies depending on the object being viewed. For details on the menu commands available for each object, access the Workbench online Help system.

? | ??

Purpose

Display the value of an expression

Syntax

? | ?? <*exp*>

Arguments

<*exp*> The expression to be displayed.

Description

? causes the value of the specified expression to be displayed in the Command window. ?? displays the value in a pop-up window which is closed by pressing Esc. The expression used as the ? | ?? command line argument can be any valid CA-Clipper expression, including user-defined functions, arrays, and inline assignments.

When the display expression is a code block, the value cannot actually be displayed. For this data type, an icon is displayed instead.

If the expression is an array or an object, an icon is displayed if the value is displayed with either ? or ??. ??, however, will also display the array or object contents if you press Enter in the first pop-up window. To close both pop-ups, you must press Esc twice.

Both ? and ?? can be used to query individual array elements by subscripting the array name. Similarly, object instance variables can be queried using the send operator (:), as in <*objName*>:<*idInstance*>.

The ability to display the return value of a user-defined function allows you to extend the abilities of the debugger. For example, you might want to modify the structure of a database file while debugging an application. To do this, simply write a function to perform the modification, compile and link it with your application, and execute it using the ? | ?? command. Any feature that does not already exist can be written as a user-defined function and executed in this way.

Using the inline assignment operator (:=) with the ? | ?? command allows you to create new variables.

Examples

The following examples query various values (a field, a logical expression, and a CA-Clipper function). In each case, the result is shown below the ? command as it would be in the Command window:

```
? InvNum
    465
? X = Y
    .T.
? EOF()
    .F.
```

In this example, the EVAL() function is used to execute a code block:

```
? EVAL(bSortBlock)
```

This example creates a new private variable called *nPageNo*, assigns it the value 2, and displays the variable in a pop-up window. To close the window, press Esc:

```
?? nPageNo := 2
```

BP

Purpose

Set or remove a breakpoint

Syntax

BP [[At] [<*lineNum*>] [[In] <*idProgramFile*>]]
BP [<*idFunction*> | <*idProcedure*>]

Arguments

At <*lineNum*>

The line number where you want program execution to stop.

In <*idProgramFile*>

The filename of the program in which to set the breakpoint. If no extension is specified, (.prg) is assumed.

<*idFunction*> | <*idProcedure*>

The name of a routine specified without parentheses. When you set a breakpoint in this manner, execution stops each time the routine is called.

If no options are specified, BP refers to the line indicated by the cursor in the Code window.

Description

BP designates a line of program code or a call to a routine as a breakpoint. When you execute this command, the line of code is indicated in a new color to distinguish it as a breakpoint and execution will pause as soon as the line is encountered or the routine is called.

BP acts as a toggle so that if the specified line of code is already a breakpoint, the breakpoint is deleted and the color of the line returns to normal.

To delete a breakpoint, use the Delete command. To see all breakpoint settings in the Command window, use the List BP command. For more information on the use of breakpoints, see Debugging a Program earlier in this chapter.

Examples

This example toggles the breakpoint setting at line 12 of the program displayed in the Code window:

```
BP 12
```

This example toggles the breakpoint setting at the current line in the Code window:

```
BP
```

This example toggles the breakpoint setting at line 15 of File2.prg:

```
BP At 15 In File2
```

Here, the breakpoint is designated using a function name:

```
BP TestUdf
```

See Also

Delete, List, Point Breakpoint, Run Go, Run Step

Callstack

Purpose

Control the Callstack window

Syntax

Callstack [on | Off]

Arguments

On

Opens the Callstack window on the right-hand side of the screen.

Off

Closes the Callstack window.

If no options are specified, Callstack acts as a toggle, changing the current status of the Callstack window. Executed without an option in this manner, Callstack is functionally equivalent to the View Callstack command.

Description

Callstack toggles the display of the Callstack window. This window contains the call stack, a list of pending activations with the current activation at the top. For more information on the call stack, see Debugging a Program earlier in this chapter.

Examples

To open the Callstack window:

```
Callstack On
```

To close the window:

```
Callstack Off
```

See Also

View Callstack

Delete

Purpose

Delete debugger settings

Syntax

Delete All [BP | TP | WP]
Delete BP | TP | WP <*number*>

Arguments

All — Deletes all settings of a specified type or all settings if no type is specified.

BP — Specifies breakpoints.

TP — Specifies tracepoints and watchpoints.

WP — Specifies watchpoints and tracepoints.

<*number*> — The number of the individual watchpoint, tracepoint, or breakpoint to delete. If not specified, Delete opens a dialog box to prompt you for the item number.

Description

Delete removes watchpoints, tracepoints, and breakpoints, either individually or as a whole.

To delete all items of a particular type, use the Delete All form of the command.

To perform a single deletion, specify the number and type of the item to be deleted. For watchpoints and tracepoints, this is the number that appears on the left-hand side of the Watch window. For breakpoints the List command must be used to determine the number.

Note: Tracepoints and watchpoints are not distinguished by the Delete command. Thus, Delete TP and Delete WP are synonymous as are Delete All TP and Delete All WP.

For a detailed explanation of breakpoints, tracepoints, and watchpoints, see Debugging a Program earlier in this chapter.

Examples

To delete a breakpoint, you must first find out where it falls in the list of breakpoints:

```
List BP
```

If the breakpoint you want to delete is number three on the list:

```
Delete BP 3
```

This example deletes all watchpoints (and tracepoints):

```
Delete All WP
```

Here, all breakpoints, tracepoints, and watchpoints are deleted:

```
Delete All
```

See Also

BP, List, Point Breakpoint, Point Delete, Point Tracepoint, Point Watchpoint

File DOS

Purpose

Access DOS without leaving the current application

Syntax

File DOS

Shortcut	Equivalent
Abbreviation	F D
Accelerator key	Alt+F, D
Synonym	DOS

Description

File DOS loads a temporary copy of COMMAND.COM, allowing you to enter DOS commands without leaving the current application. To return to the debugger, type "Exit" at the DOS prompt.

When using this command, you must make sure sufficient memory is available to load COMMAND.COM and any additional programs you want to execute. If the amount of memory is insufficient, an error message will be displayed.

File Exit

Purpose

Terminate the debugger, close all files, and return to DOS

Syntax

File Exit

Shortcut	Equivalent
Abbreviation	F E
Accelerator key	Alt+F, X
Shortcut key	Alt+X
Synonym	Quit

File Open

Purpose

View a file during the current debugging session

Syntax

File Open <*idFileName*>

Shortcut	Equivalent
Abbreviation	F O
Accelerator key	Alt+F, O

Arguments

<*idFileName*>

The name of the file you want to view. If no extension is specified, (.prg) is assumed. If <*idFileName*> is not specified on the command line, File Open opens a dialog box to prompt you for a filename.

Description

File Open allows you to look at other files without leaving the current debugging session, including header files specified with the #include preprocessor directive. Breakpoints can be set and are saved when you return to the original program.

Unless you have defined a source file search path with the Options Path command, the filename that you specify is searched for in the current directory only. If, however, you have defined a search path, the directories in the path are searched in order until the file is found.

When you view a file with File Open, the current program is cleared from the Code window and the new file is displayed in its place. To continue with the original program, use File Resume.

Note: File Open is almost identical to the View command. The only difference between these two commands is that File Open assumes a (.prg) extension whereas View assumes no file extension.

Examples

To view ListDbfs.prg:

```
File Open ListDbfs
```

To view another file, PrintDbf.prg, before returning to the original program:

```
File Open PrintDbf
```

See Also

File Resume, Options Path, View

File Resume

Purpose

Return from viewing a file

Syntax

File Resume

Shortcut	Equivalent
Abbreviation	F R
Accelerator key	Alt+F, R
Synonym	Resume

Description

File Resume clears the file being viewed with the File Open command (or using the Callstack window) and redisplays the file originally shown in the Code window. Any breakpoints which have been set are saved.

Examples

Suppose the program currently being debugged is called PrtData.prg. The following line temporarily removes PrtData.prg from the Code window and displays Rpthead.prg in its place:

```
File Open RptHead
```

The following line returns PrtData.prg to the Code window:

```
File Resume
```

See Also

File Open

Help

Purpose

Activate the Help window

Syntax

Help [Keys | Windows | Menus | Commands]

Shortcut	Equivalent
Abbreviations	H K; H W; H M; H C
Accelerator keys	Alt+H, K; Alt+H, W; Alt+H, M; Alt+H, C
Shortcut key	F1

Arguments

Keys — Highlights the Keys topic when the Help window is activated.

Windows — Highlights the Windows topic when the Help window is activated.

Menus — Highlights the Menus topic when the Help window is activated.

Commands — Highlights the Commands topic when the Help window is activated.

If issued without an argument, Help highlights the About Help topic when the Help window is activated.

Description

The debugger Help window is divided into two panes: the left pane contains a list of topics for which help is available, and the right pane contains the help text for the currently highlighted topic. When the Help window is activated with the Help command, one of the topics in the left pane is highlighted, and a general discussion of the topic is displayed in the right pane of Help window.

The text that the debugger displays in the Help window is contained in a separate file, CLD.HLP, which must be present on your disk. When you ask for help, the debugger searches the current directory followed by all directories in the DOS PATH. If CLD.HLP is not found in any of these locations, the debugger will prompt you for a directory name. If CLD.HLP is still not found, you will not have access to help text within the debugger.

For information on using the Help window, see The Debugger Display earlier in this chapter.

List

Purpose

List watchpoints, tracepoints, and breakpoints

Syntax

List BP | TP | WP

Arguments

BP — Specifies breakpoints.

TP — Specifies tracepoints and watchpoints.

WP — Specifies watchpoints and tracepoints.

Description

List displays all watchpoints, tracepoints, or breakpoints in the Command window. This command is useful when you want to Delete settings by number.

Note: Tracepoints and watchpoints are not distinguished by the List command. Thus, List TP and List WP are synonymous. The Watch window can also be used to determine the number of these items.

For more information on the setting and deleting of these items, see Debugging a Program earlier in this chapter.

Examples

If there are breakpoints at lines 10 and 15 of AddData.prg and lines 2 and 6 of ViewData.prg, List BP displays the following:

```
0) 10 ADDDATA.PRG
1) 15 ADDDATA.PRG
2) 2 VIEWDATA.PRG
3) 6 VIEWDATA.PRG
```

To list all watchpoints (and tracepoints):

```
List WP
```

See Also

BP, Delete, Point Breakpoint, Point Delete, Point Tracepoint, Point Watchpoint

Locate Case

Purpose

Toggle search case-sensitivity setting

Syntax

Locate Case

Shortcut	Equivalent
Abbreviation	L C
Accelerator key	Alt+L, C

Description

Locate Case changes the current case-sensitivity setting. This setting is used by all the Locate search commands (that is, Locate Find, Locate Next, and Locate Previous) to determine whether to ignore or respect the case of the letters in the search string when testing for a match.

If case-sensitivity is on, a string must match the contents and case of the specified search string. If off (the default), only the contents need be the same for a successful match.

The Locate Case Sensitive menu option can be used to check the status of the case-sensitivity setting. An on setting is indicated by a checkmark to the left of the menu option. An off setting is indicated by the absence of the checkmark in the menu.

Examples

Suppose you want to locate all occurrences of the USE command in your program, but are not always consistent in using uppercase letters for commands. If case-sensitivity is currently on, Locate Find USE will not find all occurrences of the command but only those in which all uppercase letters are used. To turn the case-sensitivity setting off and locate the USE command in all its possible forms:

```
Locate Case
Locate Find USE
```

See Also

Locate Find, Locate Next, Locate Previous

Locate Find

Purpose

Locate a string

Syntax

Locate Find *<searchString>*

Shortcut	Equivalent
Abbreviation	L F
Accelerator key	Alt+L, F
Synonym	Find

Arguments

<searchString>

The literal string you want to locate. If *<searchString>* is not specified on the command line, a dialog box opens on the screen to prompt you for a search string.

Description

Locate Find searches the file displayed in the Code window for a specified search string.

The search always begins at the first line of code and moves down through the file, regardless of the current cursor position in the Code window. If the search string is found, the debugger moves the cursor to the line containing the first occurrence of the string in the Code window; otherwise, the cursor remains at its current location.

Note: If the Command window is active when Locate Find is executed, you will not see the cursor move to its new location in the Code window. You must select the Code window in order to see the new cursor position in the file.

This command obeys the Locate Case Sensitive menu setting. If this menu setting is on (indicated by a checkmark), Locate Find looks for a string that matches the contents and case of the specified search string. If off, only the contents need be the same for a successful match. The Locate Case command is used to toggle this menu setting on and off.

Examples

If you suspect that the error you are trying to eliminate relates to a database (.dbf) file that is not open, search for the keyword USE:

```
Locate Find USE
```

See Also

Locate Case, Locate Next, Locate Previous

Locate Goto

Purpose

Move the cursor to a specified line in the Code window

Syntax

Locate Goto <*lineNum*>

Shortcut	Equivalent
Abbreviation	L G
Accelerator key	Alt+L, G
Synonym	Goto

Arguments

<*lineNum*>

The number of the line where you want to move the cursor. If <*lineNum*> is not specified on the command line, Locate Goto opens a dialog box to prompt you for a line number.

Description

Locate Goto moves the cursor from its current position in the Code window to the specified line number.

This command works regardless of whether line numbers are currently displayed in the Code window. However, for the sake of readability we recommend that line numbers be displayed. For more information refer to the Options Line command in this section.

Note: If the Command window is active when Locate Goto is executed, you will not see the cursor move to its new location in the Code window. You must select the Code window in order to see the new cursor position in the file.

Examples

This example moves the cursor to line 30 of the program currently being debugged:

```
Locate Goto 30
```

See Also

Num, Options Line

Locate Next

Purpose

Locate the next occurrence of a string

Syntax

Locate Next

Shortcut	Equivalent
Abbreviation	L N
Accelerator key	Alt+L, N
Synonym	Next

Description

Locate Next locates the next occurrence of the most recently defined search string. A search string is defined using Locate Find. If no search string has been defined, Locate Next opens a dialog box to prompt you for one.

The search begins at the current cursor position in the Code window and moves down through the file. If a match is found, the debugger moves the cursor to the line containing the next occurrence of the string; otherwise, the cursor remains at its current location.

Note: If the Command window is active when Locate Next is executed, you will not see the cursor move to its new location in the Code window. You must select the Code window in order to see the new cursor position in the file.

Like Locate Find, Locate Next obeys the current Locate Case Sensitive menu setting. The Locate Case command is used to toggle this menu setting on and off.

See Also

Locate Case, Locate Find, Locate Previous

Locate Previous

Purpose

Locate the previous occurrence of a string

Syntax

Locate Previous

Shortcut	Equivalent
Abbreviation	L P
Accelerator key	Alt+L, P
Synonym	Prev

Description

Locate Previous searches for the previous occurrence of the most recently defined search string. A search string is defined using the Locate Find command or the Locate Find menu option. If no search string has been defined, Locate Previous opens a dialog box to prompt you for one.

The search begins at the current cursor position in the Code window and moves up through the file.

If a match is found, the debugger moves the cursor to the line containing the previous occurrence of the string; otherwise, the cursor remains at its current location.

Note: If the Command window is active when Locate Previous is executed, you will not see the cursor move to its new location in the Code window. You must select the Code window in order to see the new cursor position in the file.

Like Locate Find, Locate Previous obeys the current Locate Case Sensitive menu setting. The Locate Case command is used to toggle this menu setting on and off.

See Also

Locate Case, Locate Find, Locate Next

Monitor All

Purpose

Toggle the display of variables in all storage classes in the Monitor window

Syntax

Monitor All

Shortcut	Equivalent
Abbreviation	M A
Accelerator key	Alt+M, A

Description

Monitor All acts as a toggle by changing the current display status of all storage classes (that is, local, private, public, and static) in the Monitor window. The default setting for Monitor All is off.

Monitored variables relate to the routine currently displayed in the Code window. When a pending activation is viewed via the Callstack window, the values displayed for the monitored variables are the values that they held when the pending routine was active.

Inspecting the variables with the ? | ?? command or specifying them as watchpoints or tracepoints yields the *current* value, which may be different from the value displayed in the Monitor window. Any existing variables that are not visible to the activation in the Code window do not appear in the Monitor window.

Examples

To view all variables in all storage classes in the Monitor window:

```
Monitor All
```

See Also

Monitor Local, Monitor Private, Monitor Public, Monitor Sort, Monitor Static

Monitor Local

Purpose

Toggle the display of local variables in the Monitor window

Syntax

Monitor Local

Shortcut	Equivalent
Abbreviation	M L
Accelerator key	Alt+M, L

Description

Monitor Local acts as a toggle by changing the current display status of local variables in the Monitor window (for example, if local variables are not being monitored, Monitor Local begins monitoring them). The default setting for Monitor Local is off, meaning that local variables are not monitored.

The Monitor window indicates that local variables are being monitored using the word "Local" in its window title.

Monitored variables relate to the routine currently displayed in the Code window. When a pending activation is viewed via the Callstack window, the values displayed for the monitored variables are the values that they held when the pending routine was active.

Inspecting the variables with the ? | ?? command or specifying them as watchpoints or tracepoints yields the *current* value, which may be different from the value displayed in the Monitor window. Any existing variables that are not visible to the activation in the Code window do not appear in the Monitor window.

Examples

To change the display status of local variables in the Monitor window:

```
Monitor Local
```

See Also

Monitor All, Monitor Private, Monitor Public, Monitor Sort, Monitor Static

Monitor Private

Purpose

Toggle the display of private variables in the Monitor window

Syntax

Monitor Private

Shortcut	Equivalent
Abbreviation	M Pr
Accelerator key	Alt+M, V

Description

Monitor Private acts as a toggle by changing the current display status of private variables in the Monitor window (for example, if private variables are not being monitored, Monitor Private begins monitoring them). The default setting for Monitor Private is off, meaning that private variables are not monitored.

The Monitor window indicates that private variables are being monitored using the word "Private" in its window title.

Monitored variables relate to the routine currently displayed in the Code window. When a pending activation is viewed via the Callstack window, the values displayed for the monitored variables are the values that they held when the pending routine was active.

Inspecting the variables with the ?|?? command or specifying them as watchpoints or tracepoints yields the *current* value, which may be different from the value displayed in the Monitor window. Any existing variables that are not visible to the activation in the Code window do not appear in the Monitor window.

Examples

To change the display status of private variables in the Monitor window:

```
Monitor Private
```

See Also

Monitor All, Monitor Local, Monitor Public, Monitor Sort, Monitor Static

Monitor Public

Purpose

Toggle the display of public variables in the Monitor window

Syntax

Monitor Public

Shortcut	Equivalent
Abbreviation	M P
Accelerator key	Alt+M, P

Description

Monitor Public acts as a toggle by changing the current display status of public variables in the Monitor window (for example, if public variables are not being monitored, Monitor Public begins monitoring them). The default setting for Monitor Public is off, meaning that public variables are not monitored.

The Monitor window indicates that public variables are being monitored using the word "Public" in its window title.

Monitored variables relate to the routine currently displayed in the Code window. When a pending activation is viewed via the Callstack window, the values displayed for the monitored variables are the values that they held when the pending routine was active.

Inspecting the variables with the ? | ?? command or specifying them as watchpoints or tracepoints yields the *current* value, which may be different from the value displayed in the Monitor window. Any existing variables that are not visible to the activation in the Code window do not appear in the Monitor window.

Examples

To change the display status of public variables in the Monitor window:

```
Monitor Public
```

See Also

Monitor All, Monitor Local, Monitor Private, Monitor Sort, Monitor Static

Monitor Sort

Purpose

Control the order in which variables are displayed in the Monitor window

Syntax

Monitor Sort

Shortcut	Equivalent
Abbreviation	M So
Accelerator key	Alt+M, O

Description

When Monitor Sort is on (indicated by a checkmark to the left of the Monitor Sort menu option), items in the Monitor window are displayed in alphabetical order according to variable name.

When Monitor Sort is off (the default), the monitored variables are grouped by storage class.

Examples

To sort monitored variables by name if they are currently grouped by storage class:

```
Monitor Sort
```

See Also

Monitor All, Monitor Local, Monitor Private, Monitor Public, Monitor Static

Monitor Static

Purpose

Toggle the display of static variables in the Monitor window

Syntax

Monitor Static

Shortcut	Equivalent
Abbreviation	M S
Accelerator key	Alt+M, S

Description

Monitor Static acts as a toggle by changing the current display status of static variables in the Monitor window (for example, if static variables are not being monitored, Monitor Static begins monitoring them). The default setting for Monitor Static is off, meaning that static variables are not monitored.

The Monitor window indicates that static variables are being monitored using the word "Static" in its window title.

Monitored variables relate to the routine currently displayed in the Code window. When a pending activation is viewed via the Callstack window, the values displayed for the monitored variables are the values that they held when the pending routine was active.

Inspecting the variables with the ? | ?? command or specifying them as watchpoints or tracepoints yields the *current* value, which may be different from the value displayed in the Monitor window. Any existing variables that are not visible to the activation in the Code window do not appear in the Monitor window.

Examples

To change the display status of static variables in the Monitor window:

```
Monitor Static
```

See Also

Monitor All, Monitor Local, Monitor Private, Monitor Public, Monitor Sort

Num

Purpose

Toggle the display of line numbers

Syntax

Num [On | off]

Arguments

On — Displays line numbers at the beginning of each line of code.

Off — Removes the line numbers.

If no options are specified, Num acts as a toggle by changing the line number status. Executed without an option in this manner, Num is functionally equivalent to the Options Line command.

Description

Num toggles the display of line numbers at the beginning of each line of code in the Code window. When you first start the debugger, Num is On and line numbers are displayed. This is particularly useful when using the Locate Goto command to move the cursor to a certain line in the Code window.

See Also

Locate Goto, Options Line

Options Codeblock

Purpose

Control the tracing of code blocks during single step mode

Syntax

Options Codeblock

Shortcut	Equivalent
Abbreviation	O C
Accelerator key	Alt+O, B

Description

Options Codeblock acts as a toggle to control whether the debugger traces code blocks in single step mode.

By default, Options Codeblock is on (indicated by a checkmark next to the Options Codeblock Trace menu option), causing single step mode to trace a code block back to its definition each time the code block is evaluated. It does this by moving the execution bar to the line where the code block was defined, allowing you to see the code block definition.

Tracing code blocks involves an extra step each time a code block is evaluated because you also have to step over the line of code defining the code block. If you do not want to trace code block definitions, turn Options Codeblock off.

For more information on modes of execution, see Debugging a Program earlier in this chapter.

Examples

To change the current code block trace status:

```
Options Codeblock
```

See Also

Run Step

Options Color

Purpose

Open the Set Colors window

Syntax

Options Color

Shortcut	Equivalent
Abbreviation	O Col
Accelerator key	Alt+O, C

Description

Options Color activates the Set Colors window where you can inspect and change the debugger color settings. For information on using the Set Colors window, see The Debugger Display earlier in this chapter.

Note that although Options Color expects additional input, the color settings cannot be specified on the command line as with other menu commands.

Note: When the display mode is monochrome (see Options Mono), the Options Color command is not operational.

Examples

To inspect the debugger color settings:

```
Options Color
```

See Also

Options Mono, View Sets, View Workareas

Options Exchange

Purpose

Control the display of program output while in animate mode

Syntax

Options Exchange

Shortcut	Equivalent
Abbreviation	O E
Accelerator key	Alt+O, E

Description

Options Exchange acts as a toggle to control the display of program output while in animate mode.

By default, this option is on (indicated by a checkmark next to the Options Exchange Screens menu option), causing animate mode to display the application output for each line of code executed.

If Options Exchange is off, the debugger displays the application screen only when input is required. For more information on modes of execution, see Debugging a Program earlier in this chapter.

Examples

To change the current program output display status:

```
Options Exchange
```

See Also

Options Swap, Run Animate, View App

Options Line

Purpose

Toggle the display of line numbers in the Code window

Syntax

Options Line

Shortcut	Equivalent
Abbreviation	O L
Accelerator key	Alt+O, L

Description

Options Line toggles the display of line numbers at the beginning of each line of code in the Code window (for example, if line numbers are being displayed, Options Line turns their display off). The default setting for Options Line is on, meaning that line numbers are displayed.

Having line numbers in the code listing is particularly useful when using the Locate Goto command to move the cursor to a certain line in the Code window.

Options Line is functionally equivalent to the Num command specified without an option.

Examples

To change the display status of line numbers in the Code window:

```
Options Line
```

See Also

Locate Goto, Num

Options Menu

Purpose

Toggle the debugger menu bar display

Syntax

Options Menu

Shortcut	Equivalent
Abbreviation	O M
Accelerator key	Alt+O, M

Description

Options Menu toggles the display of the debugger menu bar. The default setting for Options Menu is on, meaning that the menu bar is visible.

Note: Menu options can be executed using the appropriate accelerator key even when the menu bar is hidden from view. The menu bar will be displayed while the menu selection is being made but will disappear after the option is executed.

Examples

To change the display status of the menu bar:

```
Options Menu
```

Options Mono

Purpose

Toggle the debugger display mode between color and monochrome

Syntax

Options Mono

Shortcut	Equivalent
Abbreviation	O Mo
Accelerator key	Alt+O, D

Description

Options Mono toggles the debugger display mode between color and monochrome. This command is effective for color monitors only.

The default setting for Options Mono depends on the kind of monitor you are using (that is, for color monitors the default is off, whereas for monochrome monitors it is on).

Note: When the display mode is set to monochrome, the Options Color command is not available.

Examples

To change the debugger display mode:

```
Options Mono
```

See Also

Options Color

Options Path

Purpose

Define the search path for source files

Syntax

Options Path <*idPathList*>

Shortcut	Equivalent
Abbreviation	O Pa
Accelerator key	Alt+O, F

Arguments

<*idPathList*>

A list of one or more directory names separated by semicolons. This option specifies the path used by the debugger to search for source files if they cannot be found in the default directory. If <*idPathList*> is not specified on the command line, Options Path opens a dialog box to prompt you for a search path.

Description

Options Path allows you to specify one or more alternative directories to be searched if a particular source file cannot be found in the default directory.

The search path that you specify pertains to source files only (that is, the View and File Open commands). Other file searches are not affected.

Examples

To specify the DBU and RL source code directories:

```
Options Path \CLIP53\SOURCE\DBU; \CLIP53\SOURCE\RL
```

See Also

File Open, View

Options Preprocessed

Purpose

Toggle the display of preprocessed code in the Code window

Syntax

Options Preprocessed

Shortcut	Equivalent
Abbreviation	O P
Accelerator key	Alt+O, P

Description

Options Preprocessed acts as a toggle by changing the current display status of preprocessed code in the Code window. By default, Options Preprocessed is off, meaning that preprocessed code is not displayed.

When Options Preprocessed is on (indicated by a checkmark to the left of the Options Preprocessed Code menu option), preprocessed code for the current program appears underneath each line of source code. Since the preprocessed code is taken from the corresponding (.ppo) file, the program in the Code window must be compiled with the /P option or, from the Workbench, with Generate PPO File checked in the Compiler Options dialog box.

The Code window indicates the display status of preprocessed code in its window title as well as in the window itself. The presence of the corresponding (.ppo) filename in the window title indicates that preprocessed code is being displayed.

Examples

To change the display status of preprocessed code in the Code window:

```
Options Preprocessed
```

Options Restore

Purpose

Read commands from a script file

Syntax

Options Restore <*idScriptFile*>

Shortcut	Equivalent
Abbreviation	O R
Accelerator key	Alt+O, R
Synonym	Input

Arguments

<*idScriptFile*>

The name of the script file from which to read commands. If no extension is specified, (.cld) is assumed. If <*idScriptFile*> is not specified on the command line, Options Restore opens a dialog box to prompt you for a filename.

Description

Options Restore causes the debugger to read and execute all commands in the specified script file then resume accepting input from the keyboard.

Script files can be created automatically using the Options Save command and can also be written using a text editor or word processor. For a full explanation of script files and their uses, see Starting the Debugger earlier in this chapter.

Examples

To execute the commands in the script file Viewbug.cld:

```
Options Restore ViewBug
```

See Also

Options Save

Options Save

Purpose

Save the current debugger settings to a script file

Syntax

Options Save *<idScriptFile>*

Shortcut	Equivalent
Abbreviation	O Sa
Accelerator key	Alt+O, S

Arguments

<idScriptFile>

The name of the script file to which you want to save the current debugger settings. If no extension is specified, (.cld) is assumed. If *<idScriptFile>* is not specified on the command line, Options Save opens a dialog box to prompt you for a filename.

Description

Options Save writes the current debugger settings to a script file. The settings are written to the script file using standard menu commands.

The script file can be executed using the Options Restore command or from the CLD command line the next time you execute the debugger. For more information on script files, see Starting the Debugger earlier in this chapter.

Examples

To save the current debugger settings to a file named Viewbug.cld:

```
Options Save ViewBug
```

See Also

Options Restore

Options Swap

Purpose

Control the display of the application screen when input is required

Syntax

Options Swap

Shortcut	Equivalent
Abbreviation	O S
Accelerator key	Alt+O, I

Description

If Options Exchange is on, Options Swap has no effect. If, however, Options Exchange is off, Options Swap acts as a toggle to control whether the application screen is displayed when input is required.

By default, Options Swap is on (indicated by a checkmark next to the Options Swap on Input menu option), which causes Run Animate to swap to the application screen when input is required. If Options Swap is off, the application screen is not displayed during animate mode.

For more information on modes of execution, see Debugging a Program earlier in this chapter.

Examples

To change the current application screen display status:

```
Options Swap
```

See Also

Options Exchange, Run Animate, View App

Options Tab

Purpose

Set the tab size for the Code window

Syntax

Options Tab *<tabSize>*

Shortcut	Equivalent
Accelerator key	Alt+O, T
Abbreviation	O T

Arguments

<tabSize>

A numeric value indicating the size to which the tabs are expanded when displayed in the Code window. The default tab size is 4. If *<tabSize>* is not specified on the command line, Options Tab opens a dialog box to prompt you for the tab size.

Description

Options Tab allows you to set the tab size for the Code window. This command is effective only if the file you are viewing contains tabs. Lines that are indented with spaces are not affected.

Examples

To change the tab size to ten:

```
Options Tab 10
```

See Also

Options Color, Options Mono

Point Breakpoint

Purpose

Set or remove a breakpoint at the current cursor position

Syntax

Point Breakpoint

Shortcut	Equivalent
Abbreviation	P B
Accelerator key	Alt+P, B
Shortcut key	F9

Description

Point Breakpoint designates the line of code indicated by the cursor in the Code window as a breakpoint. When you execute this command, the line of code is indicated in a new color to distinguish it as a breakpoint.

Point Breakpoint acts as a toggle so that if the current line of code is already a breakpoint, the breakpoint is deleted and the color of the line returns to normal.

To see all breakpoint settings in the Command window, use the List BP command. For more information on the use of breakpoints, see Debugging a Program earlier in this chapter.

Examples

To toggle the breakpoint setting for the current line of code:

```
Point Breakpoint
```

See Also

BP, Delete, List, Point Tracepoint, Point Watchpoint

Point Delete

Purpose

Delete a tracepoint or watchpoint setting

Syntax

Point Delete <*number*>

Shortcut	Equivalent
Abbreviation	P D
Accelerator key	Alt+P, D
Synonyms	Delete TP, Delete WP

Arguments

<*number*>

The number of the individual watchpoint or tracepoint to delete. If <*number*> is not specified on the command line, Point Delete opens a dialog box to prompt you for a number.

Description

Point Delete allows you to delete a single watchpoint or tracepoint by specifying its associated number. The number that you specify appears to the left of the watchpoint or tracepoint definition in the Watch window. Alternatively, you could use List TP or List WP to show the numbers and point definitions in the Command window.

For a detailed explanation of tracepoints and watchpoints, see Debugging a Program earlier in this chapter.

Examples

This example deletes the first watchpoint or tracepoint defined in the Watch window:

```
Point Delete 0
```

See Also

Delete, List, Point Tracepoint, Point Watchpoint

Point Tracepoint

Purpose

Specify an expression as a tracepoint

Syntax

Point Tracepoint <*exp*>

Shortcut	Equivalent
Abbreviation	P T
Accelerator key	Alt+P, T
Synonym	TP

Arguments

<*exp*>

An expression of any data type to be traced. The expression can be as simple as a field or variable name, or it can be more complex, including function calls and operators. If <*exp*> is not specified on the command line, Point Tracepoint opens a dialog box to prompt you for an expression.

Description

Point Tracepoint designates the specified expression as a tracepoint. The tracepoint is then added to the Watch window, along with its type, value, and the abbreviation "tp."

A tracepoint can be thought of as a conditional breakpoint. The two are very similar, except that a tracepoint halts program execution as soon as its value changes. For more information on tracepoints, see Debugging a Program earlier in this chapter.

Examples

This example halts execution as soon as the value of the variable *nInvNum* changes:

```
Point Tracepoint nInvNum
```

This pauses when end of file is reached:

```
Point Tracepoint EOF()
```

This example halts as soon as the value of *i* exceeds 10 and is very useful in looping operations:

```
Point Tracepoint i > 10
```

See Also

Delete, Point Breakpoint, Point Delete, Point Watchpoint

Point Watchpoint

Purpose

Specify an expression as a watchpoint

Syntax

Point Watchpoint <*exp*>

Shortcut	Equivalent
Abbreviation	P W
Accelerator key	Alt+P, W
Synonym	WP

Arguments

<*exp*>

An expression of any data type to be watched. The expression can be as simple as a field or variable name, or it can be more complex, including function calls and operators. If <*exp*> is not specified on the command line, Point Watchpoint opens a dialog box to prompt you for an expression.

Description

Point Watchpoint designates the specified expression as a watchpoint. The watchpoint is then added to the list in the Watch window, along with its type, value, and the abbreviation "wp." As each line of the current program is executed, the value of the watchpoint is updated in the Watch window.

A watchpoint is identical to a tracepoint, except that it does not cause a break in program execution every time its value changes. For more information on watchpoints, see Debugging a Program earlier in this chapter.

Examples

This example specifies the field *CustNum* as a watchpoint:

```
Point Watchpoint CustNum
```

In this example, the current record number is displayed as each line of code is executed:

```
Point Watchpoint RECNO()
```

See Also

Delete, Point Breakpoint, Point Delete, Point Tracepoint

Run Animate

Purpose

Run application in animate mode

Syntax

Run Animate

Shortcut	Equivalent
Abbreviation	R A
Accelerator key	Alt+R, A
Synonym	Animate

Description

In animate mode, the debugger executes a single line, moves the execution bar to the next line, executes it, and so on. If Options Exchange Screens is on (indicated by a checkmark), the output of each line is displayed after the line has been executed.

Execution continues in this manner until a breakpoint or tracepoint is reached. To control the speed of this process, use the Run Speed command. Press any key to stop execution.

For more information on execution modes, see Debugging a Program earlier in this chapter.

See Also

Run Speed

Run Go

Purpose

Execute the application in run mode

Syntax

Run Go

Shortcut	Equivalent
Abbreviation	R G
Accelerator key	Alt+R, G
Shortcut key	F5
Synonym	Go

Description

Run Go displays your application screen and executes the application until a breakpoint or tracepoint is reached, or until the debugger is deliberately invoked (that is, by pressing Alt+D or using ALTD()). At that point, the debugger screen is redisplayed and program execution halts.

This is known as run mode. Using Run Go again causes the application to be executed from the current position to the next breakpoint or tracepoint. To reload the application so that it can be executed from the beginning, use Run Restart.

For more information on modes of execution and the use of breakpoints and tracepoints, see Debugging a Program earlier in this chapter.

Examples

Assuming breakpoints have been set at lines 15 and 30 of the current program, the following command executes the application as far as line 15:

```
Run Go
```

Entering Run Go again executes up to line 30, and so on.

See Also

Run Next, Run Restart, Run To

Run Next

Purpose

Execute the application in run mode up to the start of the next activation

Syntax

Run Next

Shortcut	Equivalent
Abbreviation	R N
Accelerator key	Alt+R, N
Shortcut key	Ctrl+F5

Description

Run Next displays your application screen and executes the application until it reaches line zero of the next activation (that is, function, procedure, code block, or message send). The application is executed in run mode until that point is reached. This is equivalent to setting a tracepoint of "PROCLINE() = 0."

Examples

Assuming the current program contains its first function call on line 15, the following command executes the application as far as line 15:

```
Run Next
```

Entering Run Next again executes up to the next activation.

See Also

Run Go, Run Restart, Run To

Run Restart

Purpose

Reload the current application

Syntax

Run Restart

Shortcut	Equivalent
Abbreviation	R R
Accelerator key	Alt+R, R
Synonym	Restart

Description

Run Restart reloads the current application in preparation to be re-executed, keeping intact all debugger settings. This command is the only way to execute an application which has already been run.

Examples

In this example, the current application is executed:

```
Run Go
```

If errors are discovered, you may want to make changes and run the application again:

```
Run Restart
```

See Also

Run Next, Run Go

Run Speed

Purpose

Set step delay for animate mode

Syntax

Run Speed <*delay*>

Shortcut	Equivalent
Abbreviation	R Sp
Accelerator key	Alt+R, P
Synonym	Speed

Arguments

<*delay*>

The increment of delay for animation in tenths of seconds. If <*delay*> is not specified, Run Speed opens a dialog box which displays the current setting and allows you to enter a new one.

Description

Run Speed controls the speed of display while in animate mode (for an explanation of this mode, see Debugging a Program earlier in this chapter). Remember that this setting is expressed in tenths of seconds so that smaller settings are faster than larger ones.

See Also

Run Animate

Run Step

Purpose

Execute the current program in single step mode

Syntax

Run Step

Shortcut	Equivalent
Abbreviation	R S
Accelerator key	Alt+R, S
Shortcut key	F8
Synonym	Step

Description

In single step mode, the debugger executes the line of code at the execution bar, moves the execution bar to the next line, and stops. As functions are called by the current program, their code is displayed in the Code window.

Examples

This example executes the next line of the application being debugged. The debugger stops with the execution bar on the next line to be executed:

```
Run Step
```

See Also

Options Codeblock, Run Trace

Run To

Purpose

Execute the current program up to the current cursor position

Syntax

Run To

Shortcut	Equivalent
Abbreviation	R To
Accelerator key	Alt+R, C
Shortcut key	F7

Description

Run To executes only those lines of code up to the line indicated by the current cursor position in the Code window. The application is executed in run mode until that line is reached.

If the line indicated by the current cursor position is never executed, the debugger continues to the end of the application.

See Also

Run Go, Run Next

Run Trace

Purpose

Execute the current program in trace mode

Syntax

Run Trace

Shortcut	Equivalent
Abbreviation	R T
Accelerator key	Alt+R, T
Shortcut key	F10

Description

The Run Trace command is similar to Run Step in that it executes one line of program code at a time. However, Run Trace does not display the code for functions called by the current program.

See Also

Run Step

View

Purpose

Examine a file in the Code window during the current debugging session

Syntax

View *<idFileName>*

Arguments

<idFileName>

The name of the file you want to examine. Unless one is explicitly specified as part of *<idFileName>*, no extension is assumed. If *<idFileName>* is not specified on the command line, View opens a dialog box to prompt you for a filename.

Description

View allows you to look at other files without leaving the current debugging session. It can also be used to view header files specified with the #include preprocessor directive. Breakpoints can be set and are saved when you return to the original program.

Unless you have defined a source file search path with the Options Path command, View searches for *<idFileName>* in the current directory only. If, however, you have defined a search path, the directories in the path are searched in order until the file is found.

When you view a file with View, the current program is cleared from the Code window and the new file is displayed in its place. To continue with the original program, use File Resume.

Note: View is almost identical to the File Open command. The only difference between these two commands is that File Open assumes a (.prg) extension, whereas View assumes no file extension.

Examples

To view ListDbfs.prg:

```
View ListDbfs.prg
```

To view another file, PrintDbf.prg, before returning to the original program:

```
View PrintDbf.prg
```

See Also

File Open, File Resume, Options Path

View App

Purpose

Display program output

Syntax

View App

Shortcut	Equivalent
Abbreviation	V A
Accelerator key	Alt+V, A
Shortcut key	F4
Synonym	Output

Description

View App temporarily clears the debugger screen and displays your application screen in its place. This allows you to see the output of the application itself. To return to the debugger, press any key.

View Callstack

Purpose

Control the Callstack window

Syntax

View Callstack

Shortcut	Equivalent
Abbreviation	V C
Accelerator key	Alt+V, C

Description

View Callstack acts as a toggle by changing the current status of the Callstack window. By default, View Callstack is off, meaning that the Callstack window is closed.

The Callstack window contains a list of pending activations with the current activation at the top. For more information on the call stack, see Debugging a Program earlier in this chapter.

View Callstack is functionally equivalent to the Callstack command specified without an option.

Examples

To change the current status of the Callstack window:

```
View Callstack
```

See Also

Callstack

View Sets

Purpose

Display the View Sets window

Syntax

View Sets

Shortcut	Equivalent
Abbreviation	V S
Accelerator key	Alt+V, S

Description

View Sets activates the View Sets window. When this window is active, you can view and change the status of the CA-Clipper system settings.

For information on using the View Sets window, see The Debugger Display earlier in this chapter.

See Also

Options Color, View Workareas

View Workareas

Purpose

Display the View Workareas window

Syntax

View Workareas

Shortcut	Equivalent
Abbreviation	V W
Accelerator key	Alt+V, W
Shortcut key	F6

Description

View Workareas activates the View Workareas window. This window allows you to view information regarding all database files that are currently in use.

For information on using the View Workareas window, see The Debugger Display earlier in this chapter.

See Also

Options Color, View Sets

Window Iconize

Purpose

Toggle active debugger window between icon and window display

Syntax

Window Iconize

Shortcut	Equivalent
Abbreviation	W I
Accelerator key	Alt+W, I

Description

Window Iconize reduces the active window to an icon (its name). This command acts as a toggle between the icon and window display modes so that when the active window is iconized, executing Window Iconize resumes the original window display.

A window that is iconized remains open, but you cannot see the window contents. Certain window operations such as moving, however, are possible.

Examples

To iconize the active window:

```
Window Iconize
```

See Also

Window Move, Window Size, Window Tile, Window Zoom

Window Move

Purpose Move the active debugger window

Syntax Window Move

Shortcut	Equivalent
Accelerator key	Alt+W, M
Abbreviation	W M

Description Window Move allows you to move the active window around on the screen. When you execute this command, the border of the active window changes to a different pattern and the cursor keys are used to move the window. Enter completes the moving process.

Note: You cannot move a window that is zoomed to full-screen.

Examples To move the active window:

```
Window Move
```

See Also Window Iconize, Window Size, Window Tile, Window Zoom

Window Next

Purpose Select the next debugger window

Syntax Window Next

Shortcut	Equivalent
Abbreviation	W N
Accelerator key	Alt+W, N
Shortcut key	Tab

See Also Window Prev

Window Prev

Purpose

Select the previous debugger window

Syntax

Window Prev

Shortcut	Equivalent
Abbreviation	W P
Accelerator key	Alt+W, P
Shortcut key	Shift+Tab

See Also

Window Next

Window Size

Purpose

Change the size of the active debugger window

Syntax

Window Size

Shortcut	Equivalent
Abbreviation	W S
Accelerator key	Alt+W, S

Description

Window Size allows you to change both the height and the width of the active window. When you execute this command, the border of the active window changes to a different pattern and the cursor keys are used to change the size of the window. Enter completes the sizing process.

Note: You cannot size a window that is zoomed to full-screen.

Examples

To size the active window:

```
Window Size
```

See Also

Window Iconize, Window Move, Window Tile, Window Zoom

Window Tile

Purpose

Restore debugger windows to default size and location

Syntax

Window Tile

Shortcut	Equivalent
Abbreviation	W T
Accelerator key	Alt+W, T

Description

Window Tile provides a quick way to clean up the debugger screen by restoring each window to its default location and size. Any windows that have been zoomed or iconized are also restored to the original window display mode.

Examples

To restore debugger windows:

```
Window Tile
```

See Also

Window Iconize, Window Move, Window Size, Window Zoom

Window Zoom

Purpose

Toggle active debugger window between window and full-screen display

Syntax

Window Zoom

Shortcut	Equivalent
Abbreviation	W Z
Accelerator key	Alt+W, Z
Shortcut key	F2

Description

Window Zoom allows you to zoom the active window to full-screen. This command acts as a toggle between the full-screen and window display modes so that when the active window is zoomed to full-screen, executing Window Zoom resumes the original window display.

Note: When a window is zoomed, some window operations, such as moving and sizing, are not allowed.

Examples

To zoom the active window to full-screen:

```
Window Zoom
```

See Also

Window Iconize, Window Move, Window Size, Window Tile

Chapter 14
Program Maintenance—RMAKE.EXE

In This Chapter

CA-Clipper provides a program maintenance facility with RMAKE.EXE. RMAKE is a powerful tool for keeping programs involving several source, header, object, and library files up-to-date. It does this by comparing the date and time stamps of files related to one another and performing a series of actions if they do not match.

A make facility is generally used to speed the process of compiling a program system composed of several source files by compiling only those files that have changed since the last program build.

This chapter describes the operation of RMAKE. The following general subjects are covered:

- Invoking RMAKE
- The RMAKE environment variable
- RMAKE options
- How RMAKE works
- How RMAKE searches for files
- The make file

Invoking RMAKE

If you installed the default configuration of the CA-Clipper development system, RMAKE.EXE is located in the \CLIP53\BIN directory. To execute RMAKE from DOS, use the following syntax:

```
RMAKE [<makeFile list>] [<macroDef list>]
    [<option list>]
```

<makeFile list> is a list (separated by spaces) of one or more make files to process. If no extension is specified, (.RMK) is assumed. Each filename can optionally include a drive designator and a path reference. You can specify a maximum of 16 files in the *<makeFile list>*.

<macroDef list> is a list of one or more macro definitions of the form *<macroName>*=[*<value>*]. If you specify more than one macro definition, separate them with a space. If you define a macro on the command line, you cannot redefine it in the make file without first undefining it. In this way, command line macros take precedence over make file macros of the same name.

<option list> is a list of one or more options to control the course of the make. All available options are discussed under RMAKE Options later in this chapter. To get a list of options, specify the RMAKE command line with no arguments.

Like all other CA-Clipper utilities, you can execute RMAKE from any drive or directory since the appropriate directory was placed in the PATH list during installation.

The RMAKE Environment Variable

In addition to specifying options on the RMAKE command line, you can specify them using the RMAKE environment variable. To define RMAKE, use the DOS SET command as follows:

```
SET RMAKE=[<option list>]
```

<option list> is a list of one or more options that is read and processed each time RMAKE is invoked. You specify these options just as you would on the RMAKE command line.

To save yourself from having to enter this SET command repeatedly, you can place it in your AUTOEXEC.BAT file where it will be processed automatically each time you reset your computer.

The following example causes RMAKE to display warning messages and debugging information:

```
SET RMAKE=/W /B
```

Note: In addition to command line options, you can use the RMAKE environment variable to specify one or more make files, just as you can on the RMAKE command line. You cannot, however, define macros like you can on the command line because the DOS SET syntax does not permit using an equal sign—use the /D option instead.

RMAKE Options

The options described in this section control the behavior of RMAKE. As already discussed, you can specify options on the RMAKE command line and in the RMAKE environment variable. Command line options take precedence if there is a conflict.

All options are shown in uppercase preceded by a slash (/). Note, however, that options are not case-sensitive and that you can replace the slash with a dash (-) if you prefer.

Some options have arguments. If an option has arguments, specify them after the option, with no space between the option and its first argument.

/B

Displays debugging information.

/D

/D<*macroName*>[:<*value*>]

Defines a macro and an optional value. If you do not supply the value, the macro is defined with a null value.

As with command line macros, if you define a macro using the /D option, you cannot redefine it in the make file without first undefining it. In this way, /D macros take precedence over make file macros of the same name.

/F

Forces RMAKE to perform all actions defined in the make file(s) without regard to the date and time stamps of the files in the dependency and inference rules (see How RMAKE Works later in this chapter for more information). In other words, the application will be rebuilt, regardless of whether you have updated any files since the last build.

/I — Causes RMAKE to continue if a command executed from the make file causes an error (that is, a DOS return code greater than zero). By default, errors of this nature halt the make process. Note that fatal errors produced by RMAKE terminate the make process, regardless of /I.

/N — Displays the commands to be executed without actually executing them.

/Q — Prevents the RMAKE copyright message from displaying.

/U — Enables the # character as a comment indicator. If this option is specified, you must precede make file directives (explained in the Directives section later in this chapter) by the exclamation (!) symbol instead of the hash (#) symbol.

/W — Displays warning messages while processing.

/XS — /XS<*numSymbols*>

Sets the size of the internal symbol table. The default size is 500 symbols.

/XW — /XW<*numBytes*>

Sets the size of the internal workspace. The default size is 2048 bytes.

How RMAKE Works

If you are familiar with UNIX Make, you will probably find RMAKE easy to understand and use. Generally, it is upwardly compatible with UNIX Make and will, in most cases, successfully process UNIX make files; however, RMAKE provides several additional features.

In general, you describe your program in terms of dependency rules using a text file called a make file. Then, instead of determining the compiling and linking steps to build the program each time you make a change to a source file, you use RMAKE with one or more make files designed for that particular program.

RMAKE performs only the compiling and linking steps necessary to bring the executable version of the program up-to-date.

The basic element of a make system is the *dependency rule* consisting of a *dependency statement* that establishes a relationship between a *target file* and a series of *dependent files*. The dependency statement is then followed by a series of *actions* to update the target file if any of its dependent files are more recent.

RMAKE is a two-pass system. The first pass, called the *parsing phase*, parses the make files. The second pass, called the *make phase*, examines the dependency rules and performs the necessary actions .

Parsing Phase

When invoked, RMAKE first performs the parsing phase by processing each make file sequentially in the order encountered. Files specified on the RMAKE command line are processed from left to right. All file searching is performed during the parsing phase, and all inference rules are applied. If macros or rules are redefined, the new definitions apply to the dependencies specified after them.

Make Phase

After the parsing phase, RMAKE performs the make phase during which all of the rules defined in all of the files are applied in order to produce the target files. For each rule in the make file, the date of the target file is compared to the date of each dependent file. If any dependent file has a more recent date and time stamp than its associated target file, the accompanying actions are executed. If none of the dependent files have been changed since the target file was last updated, the rule is bypassed and the process is repeated for the next rule.

Note: If you specify the /F option, all actions are performed regardless of the date and time stamps of the files specified in the dependency and inference rules.

RMAKE correctly handles the case where two input files define dependencies (and possibly actions) for the same target. It always builds all target files to ensure that multiple-specified targets (for example, .LIB files) are not time stamped before all make files get a chance to specify dependencies. Although no particular order is guaranteed during the make phase, RMAKE does guarantee that no target will be built until all of its dependencies have been built.

Error Handling

When RMAKE terminates, it sets the DOS return code to indicate the state of the make process at termination. A return code of 0 means that the make was successful (that is, there were no errors), and a return code of 1 means that errors were encountered during processing.

You can test the RMAKE return code within a batch file using the DOS ERRORLEVEL condition. For example:

```
RMAKE RL
IF ERRORLEVEL 1 GOTO ERROR
.
.<commands>
.
GOTO END
:ERROR
.
.<commands>
.
:END
```

How RMAKE Searches for Files

There are only a few types of files that serve as input to RMAKE. This section describes each input file type and any special rules governing how RMAKE searches for files of that type.

Make Files

When RMAKE is first executed, it attempts to locate the make files specified on the command line and in the RMAKE environment variable. By default, these files are assumed to have an extension of .RMK, which you can override by specifying another extension as part of the make filename. To indicate a file with no extension, include a period at the end of the filename.

You can specify a full or partial path as part of any make filename to force RMAKE to look for the file in a particular drive and/or directory. If you specify no path, RMAKE searches only the current directory.

Target and Dependency Files

In addition to finding make files, RMAKE must find the target and dependency files referenced in dependency statements in the make file. By default, RMAKE searches only the current directory for these files. However, there are special macros that you can define to specify where target and dependency files are to be found and/or created. The form of these special macro definitions is:

```
makepath[.<extension>] = <pathSpec>
```

<extension> specifies the file type and *<pathSpec>* specifies one or more paths to be searched when trying to find or create files of the given type. To avoid problems with the backslash (\) character in a make file, enclose the path list in quotes as in the following examples.

If you define a makepath macro for a particular file type, the current directory is not searched unless it is explicitly included in the <*pathSpec*>. Only the paths specified in the <*pathSpec*> are searched, and they are searched in the order specified during the parsing phase.

If a file is not found along any of those paths, RMAKE attaches the path at the front of the list to the filename. For a target file, this means that the first path in the list for its file type determines where the file is created when the make actions are performed. For dependent files, it means that RMAKE will not be able to find the file and will result in an error.

You can specify makepath macros on the RMAKE command line, in the RMAKE environment variable (using the /D option), and in make files. Some examples follow:

```
makepath[.h] = "C:\CLIP53\SOURCE;C:\CLIP53\INCLUDE"
makepath[.obj] = "C:\CLIP53\OBJ"
makepath[.lib] = "C:\CLIP53\LIB;C:\MSC\LIB"
makepath[.exe] = "C:\CLIP53\BIN"
```

To access a makepath macro in a make file, enclose the macro name in parentheses preceded by a dollar sign. This is the standard make file syntax for accessing macro values. For example, to augment a path list:

```
makepath[.h] := "C:\MSC\INCLUDE;$(makepath[.h])"
```

Note that here you use the immediate assignment operator (:=) rather than the assignment operator (=) that you used to originally create the macro. This is to prevent the normal delay that would occur in expanding the referenced makepath[.h] macro. (See the Macros section later in this chapter for more information.)

If you specify target and dependency files with a path as part of the filename, RMAKE searches only the explicitly named directory regardless of the existence of an appropriate makepath macro.

Note: Actions are executed during the make phase and thus have no effect on file searches. This means, for example, that you cannot have an action line that places a file in a subdirectory specified with a path name with the intent of triggering a subsequent dependency rule.

The Make File

The fundamental element of a make system is the make file, an ASCII text file created with a standard text editor. This file defines which files update other files and the update actions to perform when files are out-of-date. With a typical CA-Clipper program, a make file lists the program and header files required to create each object file and all the object files necessary to create the executable file. The make file also lists the commands required to build each file.

A make file consists of the following basic components:

- *Dependency rules* that define how each file is built
- *Inference rules* that define what to do if a dependency rule has no accompanying actions
- *Macros* that allow variable information to be used when constructing rules
- *Directives* that allow you to further control the manner in which the make file is processed

Using Quotation Marks

There are instances when quote marks are necessary to force RMAKE *not* to interpret certain symbols as commands. You can use either single or double quotes.

For instance, both the colon and the backslash have special meaning to RMAKE; therefore, when you specify a filename with a drive and/or path specification in a dependency or inference rule, you must enclose the filename in quotes. Another instance is a macro name definition containing these same or other special symbols. As a general principle, use quotes whenever you are in doubt, since they are never interpreted literally.

Note: Using quotation marks does not affect the manner in which RMAKE interprets the macro character ($). Macros are expanded both inside and outside of quotes.

Line Continuation

You can continue any comment, rule, or action in a make file to the next line with a backslash (\) character at the end of a line. If there is a comment on a continued line, you must place it after the continuation symbol. For example:

```
TEST.OBJ: Test1.prg Test2.prg \    // line continued
Test3.prg Test4.prg
   CLIPPER @Test.clp
```

Comments

You can specify comments in a make file using C-style inline comments (/* ... */) or C-style line comments (//) or, if you use the /U option, UNIX make-style comments (#). You can place a comment on a line by itself or embedded at the end of a command line in a dependency rule. For example:

```
/* This is a comment line */
TEST.OBJ: Test.prg
   CLIPPER Test.prg   // This is an embedded comment
```

You cannot nest comments.

Dependency Rules

The key component of the make file is the *dependency rule* which consists of a dependency statement followed by one or more actions. The dependency statement establishes a relationship between a target file and one or more dependent files. If any of the dependent files has a date and time stamp newer than the target file, the actions that follow the dependency statement are performed to bring the target file up-to-date.

Note: If you specify the /F option, all actions are performed regardless of the date and time stamps of the files specified in the dependency statement.

A dependency rule has the following basic form:

```
<targetFile>: <dependentFile list>
   [<action>]
   [<action>]...
```

You can specify up to 128 files in the *<dependentFile list>*. You can specify up to 32 *<action>* lines per dependency rule, provided that the maximum number of bytes in all *<action>* lines does not exceed 1024.

Makepath Examples

You can specify any filename that is part of a dependency rule with a path and drive reference. If you specify no path or drive, RMAKE assumes the current directory or the path list defined by the appropriate makepath macro (see How RMAKE Searches for Files earlier in this chapter for more information). Use quotes to delimit a filename that contains special characters—since the colon has special meaning in a dependency rule, you must enclose any filename with a drive letter in quotes. You must specify extensions for all filenames in a dependency rule.

The *<targetFile>* is the name of the file to build followed by a colon and at least one space. For example, this target file is built in the current directory:

```
MYAPP.EXE: MYOBJ.OBJ BIOS.OBJ
```

This target file is built in the specified directory:

```
"C:\CLIP53\APPS\MYAPP.EXE": MYOBJ.OBJ BIOS.OBJ
```

The *<dependentFile list>* is a list of one or more filenames separated by spaces. The following dependency statement defines two dependent files:

```
MYAPP.EXE: MYOBJ.OBJ "\CLIP53\OBJ\IBM\BIOS.OBJ"
```

RMAKE searches for MYOBJ.OBJ in the current directory and for BIOS.OBJ in the \CLIP53\OBJ\IBM directory of the current drive.

The *<action>* statements are valid DOS command lines with arguments. You must place each *<action>* on a line by itself and must indent it using spaces or tab characters. The action lines are assumed to end at the first line that is not indented, but you can also terminate them using a blank line.

Note: If you specify no *<actions>*, you should create a corresponding inference rule that defines the actions to take. See Inference Rules below for more information.

Tip: Because RMAKE generates DOS executable commands from them, action lines cannot exceed the DOS command line limit of 128 characters. This is particularly important if the action line contains macros because the limit applies after macro expansion (see the Macros section later in this chapter for more information). For example, if an executable file depends on a large number of object files, the action line to link the object files may be too long. In this case, you can create a linker script file (.LNK) containing the list of dependent object files to link and use the script file in place of the object file list on the linker action line. A good example of linking with a script file can be found in the DBU.RMK make file located in the \CLIP53\SOURCE\DBU directory.

The following dependency rule recompiles Test.prg only if someone has changed it since it was last compiled. RMAKE accomplishes this by comparing Test.prg to its corresponding object file, TEST.OBJ, and performing the compilation only if the date and time stamp for the source file is more recent than that of the object file:

```
TEST.OBJ: Test.prg
   CLIPPER Test
```

Likewise, you can create the target .EXE file with a similar dependency rule. This example assumes that WINDOW.OBJ must also be linked to create the executable file. The rule to create TEST.EXE when either TEST.OBJ or WINDOW.OBJ is more recent is as follows:

```
TEST.EXE: TEST.OBJ WINDOW.OBJ
   EXOSPACE FILE TEST, WINDOW
```

Error Handling

It is not necessary for each *<action>* to result in the creation of a target file. In fact, after the actions are executed, RMAKE does not verify that the target file was successfully created. However, RMAKE does check the DOS ERRORLEVEL variable upon completion of each item in the *<action>* list.

If it contains a nonzero value, RMAKE will halt processing and produce an error message. The CA-Clipper compiler and most linkers properly set ERRORLEVEL so that RMAKE will halt in the event that any errors are encountered.

Note: You can disable this default error processing using the /I option. With this option, RMAKE does not stop when DOS's ERRORLEVEL is set. Instead, it continues processing as if nothing had happened.

Inference Rules

An *inference rule* specifies a series of actions for a dependency statement that is not followed by an action list. You can use it as a shortcut to perform the same set of actions for several different dependency statements. The basic form of an inference rule is as follows:

```
.<dependentExtension>.<targetExtension>:
    [<action>]
    [<action>]...
```

In comparison with a dependency statement, you specify an inference rule with the order of the target and dependent file reversed. You can specify up to 32 *<action>* lines per inference rule, provided that the maximum number of bytes in all *<action>* lines does not exceed 1024.

When RMAKE encounters a dependency statement without an action list, it searches for an inference rule that matches the extensions of the target and dependent files in the dependency statement. If an inference rule is found, RMAKE performs the accompanying actions, including expanding any macros.

When searching for an inference rule, the actual filenames in the dependency statement are insignificant—only the extensions are compared. If a dependency statement refers to more than one dependent file, the extension of the first file in the list is used. If the extensions match, the rule applies.

As in a dependency rule, the *<action>* statements are valid DOS command lines with arguments. You must place each *<action>* on a line by itself and must indent it using spaces or tab characters. The action lines are assumed to end at the first line that is not indented, but you can also terminate them using a blank line.

For example, the following inference rule compiles a single (.prg) file defined in a dependency statement to an .OBJ file:

```
.prg.obj:
   CLIPPER $< /M /N
```

Note: The $< symbol on the compiler command line is a predefined RMAKE macro described in the Macros section later in this chapter. Basically, it causes the dependent filename to be substituted on the command line. It is especially useful in inference rules because you usually have no way of knowing which dependency statement triggered the rule.

Using an inference rule such as this one, you could specify each individual object/source in your application as a separate dependency statement:

```
RLFRONT.OBJ: Rlfront.prg
RLBACK.OBJ: Rlback.prg
RLDIALG.OBJ: Rldialg.prg
```

The resulting action would be to compile only .prg files with outdated .OBJ counterparts.

You can see that this is easier than specifying equivalent dependency rules, as in:

```
RLFRONT.OBJ: Rlfront.prg
    CLIPPER Rlfront /M /N
RLBACK.OBJ: Rlback.prg
    CLIPPER Rlback /M /N
RLDIALG.OBJ: Rldialg.prg
    CLIPPER Rldialg /M /N
```

Using an inference rule, you not only save on the number of lines in the make file, but you also make it easier to maintain. For example, if you add a source file to your application, you add only one line (the dependency statement) to the make file instead of two.

Setting Environment Variables

If an action line in either a dependency or an inference rule contains a DOS SET command, it is interpreted directly by RMAKE during the make phase rather than being passed to the DOS command processor like other action lines. This allows you to set environment variables as part of an action. For example:

```
.prg.obj:
    SET CLIPPERCMD=/M /N
    CLIPPER $<
```

Each program that runs under DOS receives a local copy of the DOS environment variables. RMAKE allows you to make changes to its copy so that programs in action lines can inherit a modified environment. Any changes made are discarded once RMAKE terminates. (Like most programs, RMAKE is unable to change its parent's environment.)

If RMAKE sees a DOS command in an action list, it simply shells to execute the command. However, the DOS shell does not inherit the modified environment, so the lone SET command below displays the INCLUDE setting prior to running RMAKE:

```
.prg.obj:
    SET INCLUDE=$(makepath[.ch])
    SET                          // <<--- THIS COMMAND
    CLIPPER $<
```

The SET command is internal to DOS (there is no SET.COM or SET.EXE file). Therefore, SET can access only the initial (DOS) copy of the environment. However, CA-Clipper will see the new setting of INCLUDE because it is an application in its own right and will inherit a copy of RMAKE's modified environment.

Therefore, if you are distributing a make file, you do not have to make any assumptions about the environment. Simply set up everything you need in the make file and trust that the user's environment will be undisturbed when the make process is complete.

You can access any environment variable in a make file by surrounding it with parentheses preceded by a dollar sign. For example, the following code shows how to temporarily append the makepath[.ch] list to the current DOS INCLUDE variable:

```
.prg.obj:
   SET INCLUDE=$(include);$(makepath[.ch])
   CLIPPER $<
```

Note: In the sense that you can expand them using the syntax *$(<envVariable>)*, environment variables are treated like make file macros. However, this is where the similarity ends. In no other way are environment variables treated like macros. For example, you cannot test for their existence with #if directives, and you cannot undefine them with the #undef directive. See the Macros section later in this chapter for more information on macros.

Directives

Several directives are provided for use in make files. You must specify each directive on a separate line in the make file.

Directives are handled during the parsing phase of the make. Therefore, you cannot use directives in dependency or inference rule action lines, and macros encountered in directive arguments are always expanded immediately (see the Macros section later in this chapter for more information about macro expansion).

The syntax for all directives in this section includes an initial hash (#) symbol that you must specify. Note, however, that you can use the exclamation (!) symbol interchangeably with the hash (#) symbol unless you specify the /U option. In this instance, you must use the ! symbol or the directives will be treated as comments.

Execute Action

#!<*action*>

#! directly executes the <*action*> during the parsing phase. Like other actions specified in make files, this one can be any valid DOS command with arguments. For example:

```
#!DIR *.OBJ
```

Process if Macro Exists

#ifdef <*macroName*>
  <*statements*>...
[#else]
  <*statements*>...
#end[if]

#ifdef processes subsequent lines if the named macro exists. Do not use a dollar sign in front of the macro name unless you are trying to test for the existence of the macro whose name will result from the expansion of the macro you are specifying. For example:

```
#ifdef prgpath
   makepath[.prg] := "$(prgpath);$(makepath[.prg])"
#endif
```

Process If Two Words are Identical

#ifeq <*word1*> <*word2*>
  <*statements*>...
[#else]
  <*statements*>...
#end[if]

#ifeq processes subsequent lines if the two words are identical. For example:

```
#ifeq $(os) windows
.obj.exe:
    EXOSPACE FILE $(objs) OUTPUT RL.EXE
#endif
```

The words must be enclosed in quotes if they contain any spaces since a space is used to separate them. Note that the comparison is not case-sensitive.

Process If Specified File Exists

#iffile <*fileSpec*>
  <*statements*>...
[#else]
  <*statements*>...
#end[if]

#iffile processes subsequent lines if the specified file exists. Wildcards are allowed in the <*fileSpec*> so that you can test to see, for example, if there are any files in a particular directory. Specifying the directory name by itself, however, will fail. For example:

```
#iffile "C:\MYDIR\*.*"
    prgpath="C:\MYDIR"
#endif
```

Remember that directives are handled during the parsing phase; therefore, you cannot use #iffile to check for the presence of a file as part of an action. Also, makepath macros do not apply when searching for the #iffile.

Process If Macro Does Not Exist

#ifndef <*macroName*>
  <*statements*>...
[#else]
  <*statements*>...
#end[if]

#ifndef is the same as #ifdef except that #ifndef processes subsequent lines if the named macro *does not* exist. For example:

```
#ifndef "makepath[.prg]"
   makepath[.prg] := "$(src)"
#endif
```

Include File

#include "<*fileName*>"

#include inserts and processes the contents of the specified file in place of the #include directive as part of the make file. The filename can include a path and/or extension, but no defaults are assumed. For example:

```
#include "RMAKE.INI"
```

The #include directive does not use the INCLUDE environment variable nor will it use a makepath macro when searching for the specified file. Unless you explicitly specify a path as part of the filename, only the current directory is searched.

Display Error Message

#stderr "<*text*>"

#stderr writes <*text*> to the standard error file or device. For example:

```
#stderr "Error in compile step."
```

Display Message

#stdout "<*text*>"

#stdout writes <*text*> to the standard output file or device. For example:

```
#stdout "Making new version..."
```

Undefine Macro

#undef <*macroName*>

#undef removes any previous definition of the macro, including one supplied on the command line or in the RMAKE environment variable.

Note: Except for the fact that you can expand them, environment variables are not treated like make file macros. It is possible, therefore, to define a macro with the same name as an environment variable and effectively hide an environment setting within the context of the make file. As soon as you #undef the macro, the environment variable will once again be visible. You cannot, however, #undef an environment variable setting.

Macros

RMAKE provides a macro facility that allows you to associate a string of characters with a macro name. Macros are used in a number of situations, including the following:

- To define comma-separated file lists as required by most linkers
- For maintenance purposes, to define any value that is used more than once in the make file
- To conditionally control the course of a make
- To allow user-defined variables in the make file

With RMAKE, there are three types of macros: makepath, user-defined, and predefined. Makepath macros were discussed earlier in this chapter under How RMAKE Searches for Files. User-defined and predefined macros are discussed in this section.

Note: Except for the fact that you can expand them, environment variables are not treated like make file macros. For example, you cannot test for their existence with #if directives, and you cannot undefine them with the #undef directive.

User-Defined Macros

In a make file, a macro definition occurs on a line by itself and takes the following form:

```
<macroName>=[<value>]
```

The macro <*value*> can be any string of characters, including embedded spaces. If you do not supply the value, the macro is defined with a null value. You can nest macro definitions (that is, you can reference another macro name within the macro <*value*>).

Warning! *Do not use the following characters in a macro name: dollar sign, parentheses, colon, period, and the percent sign.*

To subsequently use the macro, refer to it as follows:

```
$(<macroName>)
```

When RMAKE encounters a $(<*macroName*>), it replaces the name with the associated string. If the macro is not defined, it is replaced with a null string. A macro remains in effect until it is redefined or undefined, or until RMAKE terminates.

As an example, the following statement creates a macro definition for a dependent file list:

```
files=RLFRONT.OBJ RLBACK.OBJ RLDIALG.OBJ
```

Later in the make file, you can use the macro as a substitute for the file list in a rule:

```
RL.EXE: $(files)
```

Macro Expansion

There are two cases in which macros are not immediately expanded when they are encountered. First, macros in inference rule actions are not expanded until those actions are attached to a dependency rule. Second, macros within macro definitions are not expanded until the macro which contains them is expanded. For example:

```
mac1 = $(cfile)          // cfile is not defined yet
cfile = Xfile.prg
XFILE.OBJ: $(mac1)       // XFILE.OBJ depends cfile
```

Delayed expansion makes it possible for a command line macro to refer to a make file macro, as long as the make file macro has been defined by the time the command line macro is used. For example, mac1 could have been defined on the RMAKE command line (or in the RMAKE environment variable).

However, since this behavior causes difficulty in some situations, you can cause macros to be expanded without delay using the immediate assignment operator (:=). For example:

```
mac1 := $(cfile)  // Won't work unless cfile exists
```

Tip: When making assignments to makepath macros in which you reference a macro, you will always use the immediate assignment operator.

Note: You can define macros on the RMAKE command line using the same syntax as in a make file or using the /D option (which you can also specify in the RMAKE environment variable). If you define a macro like this, you cannot redefine it in the make file without first undefining it. In this way, command line and environment variable macros take precedence over make file macros of the same name.

Predefined Macros

In addition to user-defined macros, there are several predefined macros that allow you to access the target and dependent file information from the last dependency statement encountered. You use predefined macros when specifying commands associated with a dependency statement or inference rule. The predefined macros are listed in the following table:

Macro	Meaning
$*	Expand to target filename without path or extension
$@	Expand to target filename including path and extension
$**	Expand to complete list of full dependency filenames
$<	Expand to full name of first file in dependent file list
$?	Expand to list of dependencies that have a more recent date and time stamp than target file

As an example, the following inference rule uses the $< macro to compile a single program file (.prg) dependent on an object file (.OBJ) in a corresponding dependency statement:

```
.prg.obj:
   CLIPPER $< /M /N
```

This example does not make any assumptions about your dependency statements and will compile successfully even if you specify more than one dependent file, because the $< macro extracts only the first file from the dependency list. However, when using this technique to compile a single file at a time, you would normally specify a separate dependency statement for each .OBJ file.

To compile more than one source file into the same object file, use a compiler script file to list all source files to compile into the target object file. To use this technique, give the script file the same name as the object file but with a (.clp) extension. Then, when you create your make file, define the inference and dependency rules for the compile operation as follows:

```
// Standard inference rule for compiling
// with a script file
.prg.obj:
   CLIPPER @$*

// Typical dependency rule to maintain an
// object file
RL.OBJ: Rlfront.prg Rlback.prg Rldialg.prg
```

The file, Rl.clp, simply lists the three dependent (.prg) files to compile into RL.OBJ.

Specifying the compiler command using the $* macro is the key because this macro substitutes the root portion of the target filename. This results in the command CLIPPER @RL when the inference rule is evaluated, which tells CA-Clipper to read the source filenames from Rl.clp.

A Complete Make File

So far in this chapter, you have seen only bits and pieces of the various components that comprise a make file. The following example is a complete make file (RL.RMK) designed to maintain RL.EXE, the CA-Clipper report and label utility. This file, along with the source code files necessary to build RL.EXE, are located in \CLIP53\SOURCE\RL:

```
//
//  RL.RMK
//
//  Make file for RL, CA-Clipper report and
//  label utility

// Define comma-separated obj list as a macro
objs=RLFRONT, RLBACK, RLDIALG

// Inference rule for compiling (.prg) to .OBJ files
.prg.obj:
   CLIPPER $< /M /N

// Dependency statements for .OBJ files
RLFRONT.OBJ: Rlfront.prg
RLBACK.OBJ: Rlback.prg
RLDIALG.OBJ: Rldialg.prg

// Dependency rule for linking .OBJ files to .EXE
file
RL.EXE: RLFRONT.OBJ RLBACK.OBJ RLDIALG.OBJ
   EXOSPACE FILE $(objs) OUTPUT $@
```

The following source files are compiled individually to make up this application:

- Rlfront.prg
- Rlback.prg
- Rldialg.prg

Thus, RL.RMK defines dependency statements for each object file, making it dependent on a corresponding source file. It then sets up an inference rule that will be triggered if any of the dependent object files is out-of-date. The inference rule calls CLIPPER.EXE to compile a single file at a time.

For the link phase, RL.RMK sets up a dependency rule making the executable file, RL.EXE, dependent on all three object files. If any one of these files is out-of-date, the rule triggers its action line which calls EXOSPACE.EXE. The FILE list on the EXOSPACE command line is defined with a user-defined macro. Since the name of the executable file is not the same as the first object file in this list, the OUTPUT file is designated using the predefined macro, $@, which gets the full path for the executable filename from the dependency statement.

Assuming your current DOS directory is \CLIP53\SOURCE\RL, you can easily rebuild RL.EXE after you make source code changes with the following command:

```
RMAKE RL
```

Chapter 15
Program Editor—PE.EXE

In This Chapter

CA-Clipper gives you the ability to create and modify source code and header files by supplying a simple program editor, PE.EXE. PE is a stand-alone application written in CA-Clipper that provides basic editing features for text and source files.

This chapter provides a brief overview of the CA-Clipper program editor and general usage information. The following topics are covered:

- Invoking the program editor
- Navigation and editing
- Leaving the program editor
- The PE system architecture



The CA-Clipper Workbench provides similar functionality through its Source Code Editor. For more information, refer to "Using the Source Code Editor" in the Workbench User Guide.

Invoking the Program Editor

If you installed the default configuration of the CA-Clipper development system, PE.EXE is located in \CLIP53\BIN. To execute PE from DOS, use the following syntax:

```
PE [<filename>]
```

<filename> is the name of the text file to edit and can include a drive, path, and/or extension. If you specify no extension, a (.prg) extension is assumed. If the specified filename exists, it is loaded for editing; otherwise, a new file is created and you are presented with an empty edit window. If you do not specify a filename, you are queried for the name of a file to edit.

Like all other CA-Clipper utilities, you can execute PE from any drive or directory since the appropriate directory was placed in the PATH list during installation.

Navigation and Editing

When you load PE, the file specified on the command line is either loaded or created, and you are presented with the edit window. Once in the edit window, the contents of the file are displayed. You can now add new or modify existing text by moving the cursor to the location of the change and typing just as you would with another text editor or word processor. PE, however, is fairly simple and does not implement any block operations such as select, cut, copy, paste, or replace. While you are editing, you can save changes with Alt+W.

The following table lists the editing keys for PE. Navigation keys are the same as those used by MEMOEDIT(). See the MEMOEDIT() entry in the "Language Reference" chapter of the *Reference Guide* for a complete list of these keys.

Key	Action
Up arrow, Ctrl+E	Line up
Down arrow, Ctrl+X	Line down
Left arrow, Ctrl+S	Character left
Right arrow, Ctrl+D	Character right
Ctrl+Left arrow, Ctrl+A	Word left
Ctrl+Right arrow, Ctrl+F	Word right
Home	Beginning of line
End	End of line
Ctrl+Home	Top of window
Ctrl+End	End of window
PgUp	Previous window
PgDn	Next window

Continued

Continued

Key	Action
Ctrl+PgUp	Top of file
Ctrl+PgDn	End of file
Return	Begin next line
Delete	Delete character
Backspace	Delete character left
Tab	Insert tab/spaces
Ctrl+Y	Delete line
Ctrl+T	Delete word right
Alt+H, F1	Display help screen
Ctrl+W	Save and exit
Alt+W	Save and continue
Alt+O	New output filename
Alt+X, Esc	Exit
Alt+F	Display filename
Alt+S	Search
Alt+A	Search again
Alt+I, Insert	Toggle insert mode

Leaving the Program Editor

To exit the edit window and save your most recent changes, press Ctrl+W. To exit the edit window without saving changes, press Esc. If you have made changes to the file without saving them, you are prompted to save changes before the edit window is closed.

In either case, the edit window closes and you are returned to the initialization screen. On the initialization screen, you can either specify another file to edit or return to DOS. To edit another file, enter its name at the File To Edit prompt. Otherwise, press Esc to return to DOS.

The PE System Architecture

PE is provided in source code form. Having access to the source code allows you to modify the program to suit your own needs and serves as a model for programming the MEMOEDIT() function. If you installed CA-Clipper in the default configuration, the source files are located in the \CLIP53\SOURCE\PE directory.

To facilitate maintenance of the PE program files, the make file, PE.RMK, is also installed with the PE source files. Assuming your current DOS directory is \CLIP53\SOURCE\PE, you can easily rebuild PE.EXE after you make source code changes with the following command:

```
RMAKE PE
```

If your changes involve additional source modules or changes in the existing file dependencies, you will need to update PE.RMK to reflect those changes. See the "Program Maintenance—RMAKE.EXE" chapter in this guide for more information about make files.

Chapter 16
Database Utility—DBU.EXE

In This Chapter

In order to provide you with an interactive database design environment, CA-Clipper includes the DBU system. The system is a stand-alone application written in CA-Clipper that allows you to build database files, add data to the files, browse existing data, create and attach index files, and construct views using a completely menu-driven system. This chapter explains how to use all of the features provided in DBU.

In this chapter, the following topics are discussed:

- Invoking the database utility
- The main DBU screen
- Leaving DBU
- The DBU menus
- The DBU system architecture

Invoking the Database Utility

If you installed the default configuration of the CA-Clipper development system, DBU.EXE and its help file, DBU.HLP, are located in \CLIP53\BIN. To execute DBU from DOS, use the following syntax:

```
DBU [/<colorString>][<filename>] /E
```

You can specify the DBU command line arguments in any order.

<colorString> determines whether the DBU screen display is color or monochrome. Color is specified using /C and monochrome using /M. If a color option is not specified, the default color mode is the result of ISCOLOR().

<filename> is the name of a view file (.vew) previously created in DBU or a database file (.dbf). If a view and database file have the same name, the view file is assumed unless you explicitly specify the database file with an extension.

Specifying *<filename>* on the DBU command line opens the named file in the Browse window after DBU is loaded (see F5 Browse in The DBU Menus section later in this chapter for information on how to use the Browse window). Otherwise, the main screen is active where you can open files and access the DBU menu bar.

/E opens the file EXCLUSIVE. The parameter is not case-sensitive.

Like all other CA-Clipper utilities, you can execute DBU from any drive or directory since the appropriate directory was placed in the PATH list during installation.



The DBU Utility can also be accessed from the CA-Clipper Workbench by double-clicking on the DBU Utility icon in the CA-Clipper program group in the Windows Program Manager.

The Main DBU Screen

If you invoke DBU without a *<filename>* argument, the first thing you see is the main screen which gives you a visual image of the current view. As you open and close files, the main screen is updated to show the current view definition.

A *view* consists of one or more database files open in separate work areas, their associated index files, the relationships between the files, the active fields, and a logical record filter for each work area. In fact, it only takes a single open database file to comprise a view—all other parts of the view are optional.

While on the main screen you can add, change, or delete items in one of three ways:

- Typing the name of a file or a field
- Pressing a command key (Enter, Insert, or Delete)
- Pressing a menu key to activate one of the pull-down menus represented on the menu bar

This section describes the various components of the DBU screen and how each one operates.

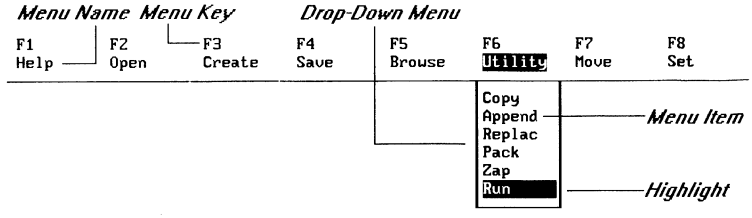
The Menu Bar

The *menu bar* at the top of the screen consists of several menu names with menu keys that you press to activate associated pull-down menus. The menu bar is always available except when there is a pending prompt.

The *menu names* on the menu bar are designed to give you a brief description of what operations you can perform with the associated pull-down menu. The *menu key* is a function key whose name is displayed directly above the menu name. Pressing it activates the pull-down menu.

A *pull-down menu* is a vertical menu that is displayed directly below the menu name. When one is active, you can see all of the *menu items* that it has to offer. Available items are displayed in high intensity, while items that are disabled appear dimmer.

The *highlight* marks the current item and does not appear in the menu if all of the items are disabled. The figure below shows the DBU menu bar and all of its components:



When a menu is active, the Up and Down arrows move the highlight within the menu and Enter selects, or executes, the currently highlighted item. When using the Direction keys to navigate within a menu, navigation is not circular (for example, pressing the Down arrow on the last menu item does not wrap around to the first item).

A shortcut for navigating to a particular menu item is to type the first letter of its name. When using the first letter shortcut, menu navigation is circular, which means that the highlight moves past the last item by wrapping around to the top of the menu if necessary.

The Left and Right arrows activate the previous and next menu, respectively, on the menu bar. Pressing Esc terminates the menu and returns control to the previous operation.

Note: Pressing the same menu key twice activates the specified menu and selects the currently highlighted menu item, allowing you to perform the same operation repetitively.

The Message and Prompt Area

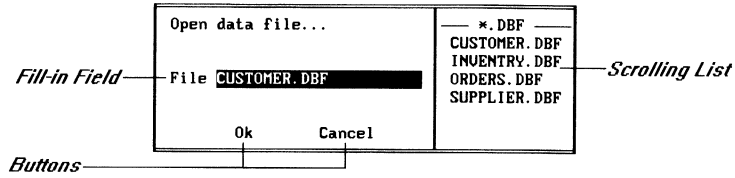
The message and prompt area is located in the upper left-hand area of the main screen, just below the menu bar. This area is used to display status messages while DBU performs certain operations and error messages when an error condition is encountered. Error messages remain on the screen until you press a key.

In addition to messages, this area of the screen is used for prompts, which are messages requiring a user response. Generally, these are warnings requiring you to verify that you want to perform a potentially destructive action. Pressing Esc or N terminates a prompt without performing the action. Pressing Y executes the selected action.

Dialog Boxes

Most selections that you make from the menu bar cause a dialog box to open on the main screen, prompting you for additional information that is needed to complete a selection.

Dialog boxes are similar to pull-down menus in that you use the Direction keys to move the highlight from one item to another and use Enter to make a selection. They are different in that selecting an item does not necessarily close the dialog box, since it is likely that you will want to make several selections. The items in a dialog box are called *controls*. The figure below illustrates the components of a typical DBU dialog box:



Note: When a dialog box is open on the screen, you can perform only the actions available in the box. An attempt to activate a menu (with the exception of F1 Help, which is always available) will reveal that all options are disabled.

All dialog boxes in DBU are similar in their behavior, although slightly different in appearance depending on the information needed. This section describes each possible type of control. Not all dialog boxes have each type of control described but, when a particular control is present, it always behaves in the manner described.

Buttons

Buttons are controls in a dialog box that are usually grouped together and displayed side-by-side. In DBU, there are *push buttons* to perform some sort of action and *radio buttons* to make one of several mutually exclusive selections.

Push Buttons

OK and Cancel are examples of push buttons. Selecting OK confirms the choices you have made within the dialog box and performs the associated menu action. Selecting Cancel cancels the dialog box, as well as the associated menu action.

Radio Buttons

SDF and DELIMITED are examples of radio buttons which do not perform an action, but rather set some parameter. Selecting a radio button surrounds it with a box as an indicator and deselects any other one that might have been selected. You can deselect a button in the same manner as you selected it, thereby leaving all buttons in the group unselected.

Fill-in Fields

A *fill-in field* is a control in which you type text from the keyboard into a data entry area. Data is entered in the same manner as all data entry is performed in CA-Clipper. For example, the Direction keys are used to move the cursor around within the entry area, Insert is used to toggle insert/overwrite mode, Delete is to delete the current character, and Esc undoes any changes that were made. Fill-in fields are used, for example, to enter file names and expressions.

Unlike other controls, you do not have to press Enter to select a fill-in field—you can just start typing once it is highlighted. If the fill-in field already has contents, it is overwritten. To edit a fill-in field, however, you must press Enter before you begin to type, move the cursor to the correct location, and make the change.

In many cases, the information you enter into a fill-in field will be a logical condition or some other type of expression. In these cases, you must enter a valid expression according to the rules established by the CA-Clipper language. For more information on expressions, see the "Basic Concepts" chapter in this guide.

Scrolling Lists

A *scrolling list* is a control that displays a list of items. Selecting a scrolling list item places the item in the currently highlighted fill-in field. The list is called a scrolling list because if there are more items than will fit in the dialog box (there are small arrows to indicate this situation), the list scrolls to bring those items into view.

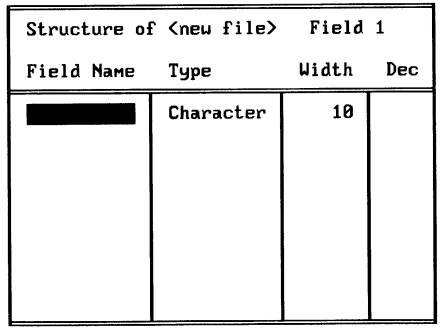
There are several examples of scrolling lists in the DBU interface, including file and field lists. Navigation and selection in a scrolling list are the same as in a pull-down menu.

Closing a Dialog Box

To close a dialog box, you can select either the OK or the Cancel button at the bottom of the box. Selecting OK closes the dialog box and performs the action which activated it using all of the controls that you have set. Selecting Cancel (or pressing Esc) closes the box without performing the action. Both selections return you to the main screen.

Windows

Other menu selections open a window on the main screen. A window is similar in appearance to a dialog box but has a special purpose. Several windows are available in the DBU interface (for example, F3 Create Database and F5 Browse Database), and each behaves more or less independently to accomplish its specific task. Unlike a dialog box, certain menu items other than F1 Help are available when a window is active. The figure below shows a typical DBU window:

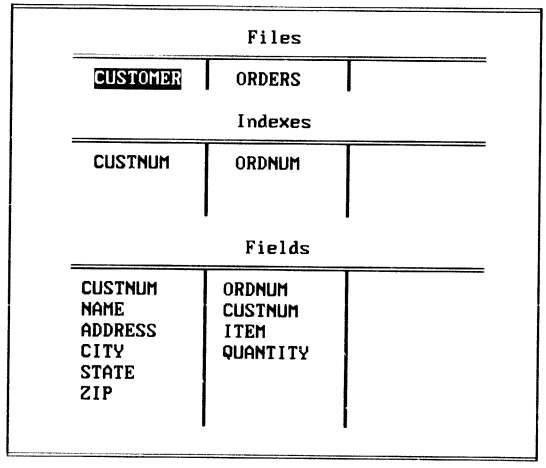


How you navigate and what you can do inside a window depends on the window. Each window is described individually in The DBU Menus section later in this chapter.

Work Areas

The screen is divided into columns with each column representing a database *work area*. Up to six work areas are available in DBU. Each work area is divided into three separate sections on the screen which identify its attributes and the database file, an index file list, and a field list. The files from all work areas, taken together with relations and filters comprise the current view.

The figure below shows the DBU work areas with two open database files:



Files

The Files area is for opening and closing database files (.dbf). Pressing Enter when the Files area is highlighted allows you to open an existing database file (.dbf) either by selecting a file name from a scrolling list or by typing a file name in a dialog box. You can open up to six database files at one time in DBU.

When you open a database file, an unused Files column is also opened to represent the next available work area (unless all six work areas are already occupied). When the highlight is located in the unused work area, you cannot move it to the Indexes or Fields areas, nor can you select any menu item that requires an open database file. Other operations, such as opening or creating a database file, however, are allowed.

Indexes

Highlighting the Indexes area allows you to open up to seven index files per work area. Opening an index file is accomplished in the same manner as opening a database file. To open more than one index file, move the highlight below the current index file name and press Enter.

Opening database and index files in this manner (that is, with Enter) closes any other open files that happened to occupy that work area. Insert, on the other hand, allows you to open a file by pushing the other files across one column—if necessary, closing the file occupying the last work area to make room for the new one. Opening database and index files using Insert is equivalent to using the F2 Open menu. Delete closes the file in the currently highlighted work area.

Fields

Highlighting the Fields area allows you to delete, insert, and overwrite fields in the field list for the active database file. Operations on the Fields area do not affect the database file structure—only the field list that is used when you edit with the F5 Browse menu.

To delete a field from the list, highlight the field name and press Delete.

To insert a field, highlight the proper location for the field and press Insert. The Set Fields dialog box opens allowing you to select the field to insert from a scrolling list. This action is equivalent to selecting F8 Set Fields from the menu bar.

You can combine the actions of inserting a new field and deleting the current one by highlighting the field you want to delete and pressing Enter. Similar to pressing Insert, this action opens the Set Fields dialog box; however, pressing Enter causes the field that you select to overwrite the current field—thereby deleting the current field and inserting a new one. Using a combination of these actions, you can construct a field list for the current database file that is reflected when you select F5 Browse Database.

Note: Besides using Insert, Delete, and Enter to manipulate files and fields, you can type file and field names directly on the main screen in the appropriate work area sections. Typing names in this manner is equivalent to selecting them from a dialog box.

Navigation on the Main Screen

The following table summarizes the keys that are available for use when operating on the main screen in DBU. These keys apply only when the main screen is active and you are dealing directly with the various work area attributes. In short, they do not apply when a menu, dialog box, or window is active.

Key	Action	Attribute
Left arrow	Previous column	All
Right arrow	Next column	All
Up arrow	Previous row	All
Down arrow	Next row	All
PgUp	First row or previous section	All
PgDn	Last row or next section	All
Ctrl+PgUp	First row in current column/section	All
Ctrl+PgDn	Last row in current column/section	All
Home	First column in current section	All
End	Last column in current section	All
Enter	Open file and close current; select field and delete current	Files; indexes; fields
Insert	Open file without closing current; select field without deleting current	Files; indexes; fields
Delete	Close file; delete field	Files; indexes; fields
Esc	Quit DBU	All
Menu Key	Activate pull-down menu	All

Leaving DBU

When you are finished with DBU, you can exit the utility by pressing Esc and responding with Y at the prompt. All files that you created and any data that you added or changed are automatically saved. You must, on the other hand, save a view to disk if you plan to use it again without having to set up all of the components in the next DBU session.

Once a view is set up on the main screen with all the necessary files and fields represented, you can save the image to a view file (.vew). To do this, select F4 Save View and enter the requested information into the dialog box. Later on, you can select F2 Open View and enter the view file name as it was saved. All files in the view are opened and all field lists established. For more information on view files, see F4 Save in The DBU Menus section later in this chapter.

The DBU Menus

The DBU menu bar has a menu name and function key for each menu in the system. The menus are always available and are activated by pressing the corresponding menu key to activate a pull-down menu.

A highlight indicates the current menu item. Items that appear in bold are available for selection, while dimmer items are disabled. If all items are disabled, no highlight appears and you cannot make a selection from that menu.

The following table summarizes the keys used to navigate and make selections when a menu is active:

Key	Action
Enter	Select current item
Esc	Abandon menu without selecting
Up arrow	Highlight previous item
Down arrow	Highlight next item
Left arrow	Activate previous menu
Right arrow	Activate next menu
Home	Highlight first item
End	Highlight last item

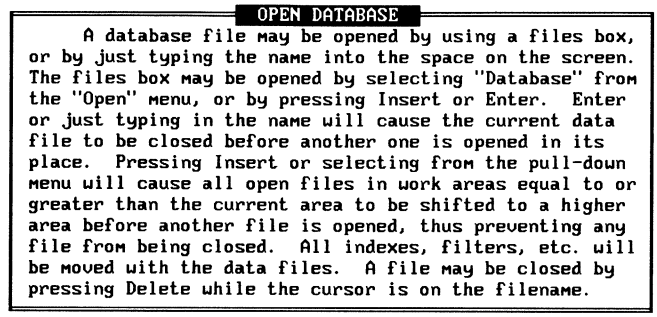
In addition to these keys, you can navigate to a particular menu item by typing the first letter of its name. If there are several items in the menu beginning with the same first letter, this method navigates to the next one, wrapping around to the top of the menu if necessary.

DBU uses the highlight on the main screen to make decisions about how certain menu selections operate. For example if you select F6 Utility Copy, DBU assumes you want to copy from the currently highlighted database file.

Note: You can press a menu key when a menu is active to quickly open another menu. Pressing the same menu key twice in succession activates the menu and selects the currently highlighted item, allowing you to perform the same operation repetitively.

F1 Help

The F1 Help menu contains only a single menu item, also named Help. Selecting it gives you context-sensitive help, depending on what you are trying to do when you make the menu selection. For instance, if you are trying to open a database file and the Open dialog box is active on the screen, requesting help gives you information on opening a database file:



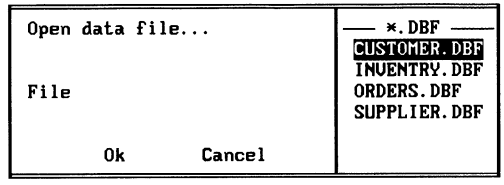
The help text is displayed in a window. If there is more information than will fit in the window, use Up arrow, Down arrow, PgUp, and PgDn to scroll through the text. Press Esc when you have finished and want to close the window.

F2 Open

F2 Open allows you to open files. The menu contains the following items:

- Database
- Index
- View

All menu items activate an Open dialog box similar to the one shown below. The scrolling list contains a different file type depending on the item you select.



If the highlight on the main screen is located in the Indexes area, Index is the current item when you activate the Open menu. Otherwise, the current item is Database.

For all Open menu items, selecting an already open file or entering the name of a nonexistent file causes an error.

Database

Database opens a database file (.dbf) in the currently highlighted work area on the main screen. Selecting this menu item activates an Open dialog box where you can either select a database file name from a scrolling list or type one directly into a fill-in field.

When you select OK, the file is opened in the currently highlighted work area, pushing all other open files to the right by one work area. Any file occupying work area six is closed.

You can activate the Open Database dialog box by pressing Enter or Insert instead of using the menu. If you know the name of the file that you want to open, you can also type it directly on the main screen.

Index

Index opens an index file (.ntx) in the currently highlighted work area. This menu item is available only if there is a corresponding open database file. Selecting Index activates an Open dialog box where you can either select an index file name from a scrolling list or type one directly into a fill-in field.

When you select OK, the file is opened in the currently highlighted slot, pushing all other open files across one slot. Any file occupying slot seven is closed.

You can activate the Open Index dialog box by pressing Enter or Insert instead of using the menu. If you know the name of the file that you want to open, you can also type it directly on the main screen. The index file at the top of the column is the controlling index.

View

View opens a view file (.vew) by opening all its associated database and index files, establishing the field lists for each database file, establishing the relations between the files, and activating filters for each work area. Selecting this menu item activates an Open dialog box where you can either select a view file name from a scrolling list or type one directly into a fill-in field.

Note: You can design, save, and open a view file only in DBU—you cannot use it subsequently in a CA-Clipper application without further programming.

F3 Create

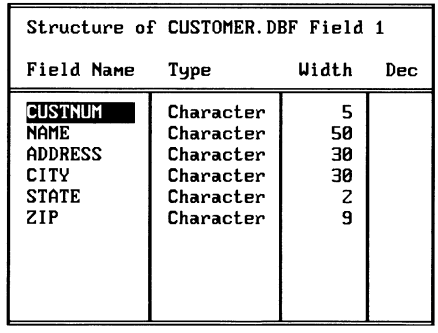
F3 Create allows you to create and modify database and index files. The menu contains the following items:

- Database
- Index

If the highlight on the main screen is located in the Indexes area, Index is the current item when you activate the Create menu. Otherwise, the current item is Database.

Database

Selecting Database opens the Structure window in which you enter field definitions to define a database file structure. If a database file is open in the current work area on the main screen, its structure is displayed in the window for you to modify, as shown below. Otherwise, the window is empty, under the assumption that you want to create a new file.



The following table summarizes the keys that are available when the Structure window is active:

Key	Action
Enter	Enter input mode; go to next column
Esc	Abandon input mode without saving; close Structure window without saving
F4	Close Structure window and save
Space bar	Select next value in Type column
Up arrow	Go to previous field definition
Down arrow	Go to next field definition
Left arrow	Go to previous column
Right arrow	Go to next column
Insert	Insert new field definition
Delete	Delete current field definition
Home	Go to first column
End	Go to last column
Ctrl+Home	Go to first column
Ctrl+End	Go to last column
PgUp	Scroll field definitions back
PgDn	Scroll field definitions forward
Ctrl+PgUp	Go to first field definition
Ctrl+PgDn	Go to last field definition

For each field in the database file, you must define a field name and data type. Then, depending on the data type, you may also have to define the field width and number of decimal places.

A field definition is checked for validity when you attempt to move the highlight to another field (for example, by pressing Down arrow to move to the next field). If you have made any mistakes an error message is displayed, and the highlight is moved to the erroneous column.

Field Name

To create a new field in the Structure window, move the highlight to an empty row in the Field Name column and type the field name. Field names can be up to ten characters in length, must begin with a letter, and can contain letters, numbers, and the underscore character only.

To edit an existing field name, highlight it and press Enter before typing. This places you in input mode where data is entered in the same manner as all data entry is performed in CA-Clipper. For example, the Direction keys are used to move the cursor around within the entry area, Insert is used to toggle insert/overwrite mode, and Esc undoes changes. Enter is used to terminate input mode. Field Name is essentially the same as a fill-in field in a dialog box.

Type

The Type column defines the data type of the field. In CA-Clipper, the valid field types are as follows:

- Character
- Date
- Logical
- Memo
- Numeric

To select a data type for a field, highlight the Type column of the field and type the first letter of the type you want. As an alternative, you can press the Space bar to move through the available data types until the one that you want is displayed.

Width

The Width column defines the length of the field. For date, logical, and memo fields, DBU assigns the column width automatically as 8, 1, and 10, respectively.

To enter a field width for a character or numeric field, move the highlight to the Width column and type a number. For character fields, the number must be between one and 1024, and the default is ten. For numeric fields, the width must be between one and 19, and the default is also ten.

To edit an existing field width, highlight it and press Enter before typing. This places you in input mode where data is entered in the same manner as all data entry is performed in CA-Clipper. Width is essentially the same as a fill-in field in a dialog box.

Decimals

The Dec column is for numeric fields only. It defines the number of decimal places allowed in the field.

The number that you enter in this column must be at least two less than the field width. Thus, it can range between one and 17, but the actual range depends on the current field width. Like Field Name and Width, Dec behaves like a fill-in field in a dialog box—just start typing for a new value or press Enter first to edit the current value.

Inserting and Deleting Fields

To insert a new field when you have one or more field definitions already in the Structure window, highlight the field before which you want to insert the new field and press Insert. An empty row opens up, and all existing fields move down one space.

To delete a field definition, highlight it and press Delete. The field is deleted, and all fields following it move up to close the space.

Saving the File Structure

When you have made all of the changes that you want to the database file structure, you can choose to either save the changes or abandon them altogether.

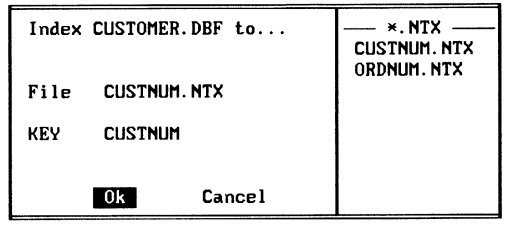
To abandon all changes and return to the main screen, press Esc. A prompt is displayed for you to confirm your choice.

To save the database file structure, select F4 Save Struct. A Save dialog box opens on the screen where you can enter the name of a new database file or accept the current file name. If you select OK, the file is created and opened in the currently highlighted work area. This dialog box is described under F4 Save later in this section.

Warning! *When you are modifying a database file structure, certain changes cause data to be lost when you save the file. In all cases, DBU attempts to save as much data as possible and prompts you if it has a decision to make regarding whether or not to preserve data. To make sure that you do not lose anything important, make backup copies of the database file before modifying its structure significantly.*

Index

Index allows you to create a new index file or modify the key expression of an existing one for the database file in the current work area. To create a new file, move the highlight to an unused slot in the Indexes column; to modify an existing index key expression, make sure the index file is open in one of the Indexes columns and highlight its file name. Selecting this menu item activates a dialog box where you can enter an index file name and key expression:



You can select the file name from a scrolling list or type it directly into a fill-in field. If an open index file is currently highlighted on the main screen, its name (and key expression) automatically appear in the dialog box. If you change the file name, a new index file is created with that name; otherwise, the current index file is recreated. If there is no current index file when you make the menu selection, no file name appears, and you must provide one.

You type the index key expression into a fill-in field. If there is already a key expression present, you can edit it by pressing Enter before you begin typing.

When you select OK in the Create Index dialog box, the file is created and opened in the currently highlighted slot.

Note: You can use Create Index to reindex a database file. To do this, go through the steps as if you were going to modify the key expression without changing anything. Simply select OK, and the index is recreated.

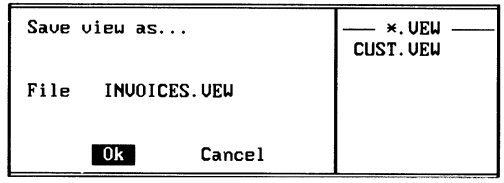
F4 Save

F4 Save allows you save a newly created or modified database file structure to a (.dbf) file or to save the view defined on the main screen to a (.vew) file. The menu contains the following items:

- View
- Struct

View

View saves the database environment defined on the main screen, including open database and index files, relations, filters, and field lists. Selecting this menu item opens a dialog box as shown below. You can select a file name from a scrolling list or type one directly into a fill-in field. Using an existing file name overwrites the file, while typing a new file name creates a new file:



When you select OK, a view file (.vew) is created which you can open and modify in DBU. You cannot, however, use a view file outside of DBU in a CA-Clipper application without further programming.

Struct

Struct saves the database file structure that you are actively creating or modifying in the Structure window. Selecting this menu item opens a dialog box in which you can select a file name from a scrolling list or type one directly into a fill-in field.

Selecting OK saves the named database file (.dbf) and appends the original data back into the file, if necessary.

Struct is not available unless you have an open Structure window on the screen (see F3 Create earlier in this section).

F5 Browse

F5 Browse allows you to view and edit data in a database file or a view. The menu contains the following items:

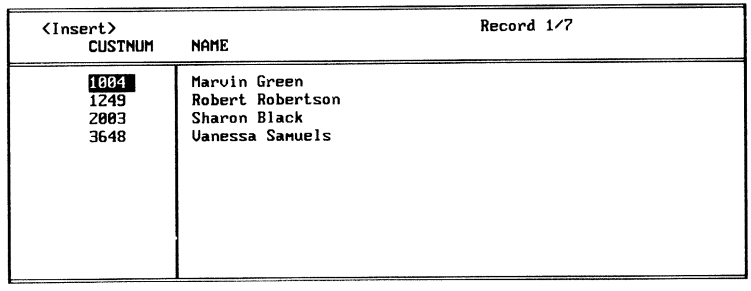
- Database
- View

Each of these menu items opens a Browse window on the screen, and this section describes how to operate within that window. To close the window, press Esc. Changes that you make are automatically saved.

Note: Search functions are available for quick navigation within the Browse window. See F7 Move later in this section for more information.

Database

Database opens a Browse window for the database file in the currently highlighted work area:



Only those fields shown in the Fields area on the main screen are available for editing. You can make changes to the existing record simply by typing over the old information, and you can add new records by moving the highlight below the last record in the file. Pressing Delete marks the current record for deletion or reinstates the record if it is already marked. (See Pack in the F6 Utility section later in this chapter for information on how to permanently remove records marked for deletion.)

View

View opens a Browse window for the entire view defined on the main screen. All fields in the Fields area for all files in the Files area are available for editing. The window behaves as if you were editing a database file, except that you can neither edit data from multiple files nor append new records.

Navigation and Editing

The following table summarizes the keys that you can use when a Browse window is active:

Key	Action
Enter	Enter input mode; go to next field
Esc	Abandon input mode without saving; close Browse window
Up arrow	Go to previous record
Down arrow	Go to next record
Left arrow	Go to previous field
Right arrow	Go to next field
Ctrl+Left arrow	Pan screen to the right
Ctrl+Right arrow	Pan screen to the left
Insert	Toggle insert mode
Delete	Toggle record delete status
Home	Go to first field on the screen
End	Go to last field on the screen
Ctrl+Home	Go to first field
Ctrl+End	Go to last field
PgUp	Scroll records back
PgDn	Scroll records forward
Ctrl+PgUp	Go to first record
Ctrl+PgDn	Go to last record

Editing Memo Fields

You edit memo fields in a pop-up window. To open the window, highlight the memo field. Then, press Enter or just start typing.

The following table summarizes all of the editing and navigation keys available when a memo editing window is active:

Key	Action
Enter	Go to next line
Esc	Close pop-up window without saving
Up arrow	Go to previous line
Down arrow	Go to next line
Left arrow	Go to previous character
Right arrow	Go to next character
Ctrl+Left arrow	Go to first character of previous word
Ctrl+Right arrow	Go to first character of next word
Insert	Toggle insert mode
Delete	Delete current character
Backspace	Delete previous character
Home	Go to beginning of line
End	Go to end of line
Ctrl+Home	Go to beginning of file
Ctrl+End	Go to end of file
PgUp	Scroll screen back
PgDn	Scroll screen forward
Ctrl+PgUp	Go to beginning of screen
Ctrl+PgDn	Go to end of screen
Ctrl+B	Reformat current paragraph
Ctrl+N	Insert blank line
Ctrl+T	Delete next word
Ctrl+W	Close pop-up window and save
Ctrl+Y	Delete current line

F6 Utility

F6 Utility provides you with several batch database operations. Each of the menu items listed below directly corresponds to a CA-Clipper command of the same name:

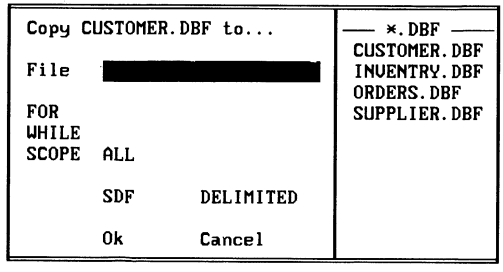
- Copy
- Append
- Replace
- Pack
- Zap
- Run

Copy

Copy allows you to copy the current database file to another database file or a text file. Any filter condition that is in effect is respected, but the field list is ignored as are any file relations. Only the active database file is copied, and it is copied in order according to the first index file in the Indexes column, if any.

To copy a file, move the highlight to the database file work area that you want to copy—you cannot select Copy if the unused work area is highlighted.

Selecting this menu item activates a dialog box where you can enter a file name, FOR and WHILE conditions, a record scope, and specify a text file by selecting SDF or DELIMITED:



You can select the file name from a scrolling list or type it directly into a fill-in field.

Text Files

If SDF or DELIMITED is selected, the scrolling list shows text files (.txt); otherwise, it shows database files (.dbf). SDF and DELIMITED are radio buttons—you can select only one at a time. If one is selected, it is surrounded by a box as an indicator. To select (or deselect) one of these buttons, highlight it and press Enter.

SDF indicates a text file that is undelimited and without separators between fields. DELIMITED indicates a text file in which fields are separated by commas and character fields are enclosed in quotation marks.

The remaining controls in the dialog box are used to designate a subset to operate on in lieu of all records in the current database file.

Scope

The SCOPE can be either ALL records or a specific number of records. This control acts like a kind of specialized fill-in field. If you select it, the ALL changes to NEXT with a 0 to the right of it, and you are expected to enter a positive number. Pressing Enter with the number set at zero returns the scope to the default, ALL records. Any other number changes the scope to NEXT <*n*> (where <*n*> is the number you entered), causing the operation to begin with the current record and continue until <*n*> records have been processed or until the end of file is encountered.

FOR and WHILE are both fill-in fields where you enter expressions to extract a logical subset of records from the file. You must enter a valid logical expression.

For Condition

A FOR condition is evaluated as each new record is considered, and those records that evaluate to false (.F.) are not processed. This process continues until the end of file is reached.

While Condition

A WHILE condition causes processing to begin with the current record and continue only while the condition evaluates to true (.T.). If both FOR and WHILE conditions are entered, WHILE takes precedence over FOR.

After all controls are set, selecting OK creates the new file but does not open it on the main screen.

Append

Append allows you to add records to the active database file from either a database file or a text file.

To add records from one file to another, move the highlight to the work area containing the file that you want to append to—you cannot select Append if the unused work area is highlighted.

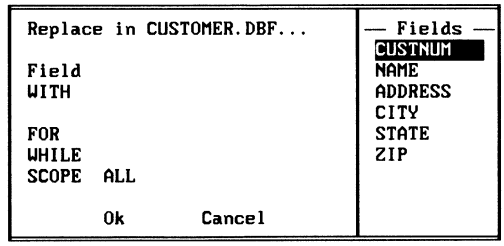
Selecting this menu item activates a dialog box that is almost identical to the Copy dialog box, which you can refer to for details on all controls. Selecting OK appends the designated records to the end of the current file and updates all open index files in the same work area.

Replace

Replace performs conditional or global field replacements in the active database file. In DBU, you can replace only one field at a time.

To replace a field in a database file, move the highlight to the appropriate work area—you cannot select Replace if the Utility menu is activated when the unused work area is highlighted.

Selecting this menu item activates the dialog box shown below. To perform the replace, you must select a field name from the scrolling list and enter a WITH expression in the fill-in field. You can optionally designate FOR and WHILE conditions and a record scope.



The Field area in this dialog box is not a fill-in field; you must select the field from the scrolling list. This is done as a precaution to make sure that only valid field names are used.

WITH is a fill-in field, and you must enter a valid expression of the same data type as the field you select.

The remaining controls were described earlier in the discussion of the Copy dialog box, which you can refer to for more information. Selecting OK replaces the indicated field in the specified records and updates all index files in the same work area.

Pack

Pack permanently removes records that are marked for deletion in the active database files. In DBU, records are marked for deletion in the Browse window using the Delete key.

To pack a database file, move the highlight to the appropriate work area—you cannot select Pack if the unused work area is highlighted.

Selecting this menu item displays a prompt which you must answer before the operation can proceed. Typing Y packs the database file and updates all open index files. Typing N or pressing Esc abandons the operation without packing the file.

Zap

Zap permanently removes all records from the active database file, regardless of their delete status. This item is useful for getting rid of test data in a file.

To zap a database file, move the highlight to the appropriate work area—you cannot select Zap if the unused work area is highlighted.

Selecting this menu item displays a prompt which you must answer before the operation can proceed. Typing Y zaps the database file and updates all open index files. Typing N or pressing Esc abandons the operation.

Run

Run allows you to execute another program from within DBU. Selecting Run displays a prompt at the bottom of the screen. To the right of the prompt is a large fill-in field where you can enter any DOS command, just as if the DOS prompt were on the screen. Press Enter to execute the command.

After the command that you enter has completed execution, you are returned to DBU with a new prompt. At this point, you can execute another command or press Esc to go back to the main screen.

Warning! *In order to use this menu item, you must have enough available memory to load the other program.*

F7 Move

F7 Move gives you a quick way to move around in a Browse window. The menu items are as follows:

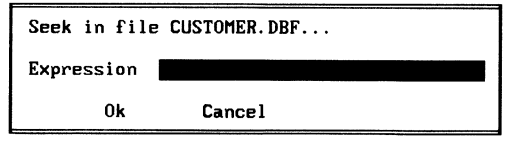
- Seek
- Goto
- Locate
- Skip

None of the items in this menu are available unless you have an open Browse window on the screen (see F5 Browse earlier in this section). All record pointer movement is performed in the current work area or the first work area if you are browsing a view.

Seek

Seek locates a particular index key value in the Browse window. This menu item is available only if there is an open index file.

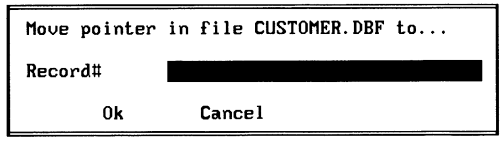
When you select Seek, a dialog box opens and allows you to enter an expression into a fill-in field. You must enter a valid expression that matches the data type of the controlling index key expression. Do not use quotes when seeking a character expression:



Selecting OK locates the first record in the file with a matching index key and highlights it for editing in the window. If not found, an error message is displayed, and the dialog box remains open on the screen. You can either Cancel or enter a new key value.

Goto

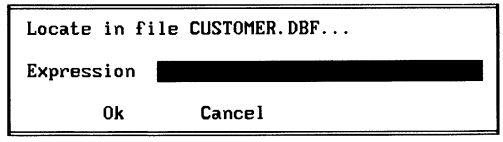
Goto jumps to a particular record number in the Browse window. When you select this menu item, a dialog box opens in which you enter a number:



Selecting OK locates the record with the indicated record number and highlights it for editing in the window. If there is no such record number, an error message is displayed, and the dialog box remains open on the screen. You can either Cancel or enter a new number.

Locate

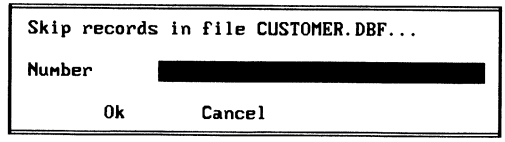
Locate searches forward from the current record position for the first record matching a logical search condition. When you select this menu item, a dialog box opens and allows you to enter a logical expression into a fill-in field:



Selecting OK locates the next record in the file that meets the search condition and highlights it for editing in the window. If not found, an error message is displayed, and the dialog box remains open on the screen. You can either Cancel or enter a new search condition.

Skip

Skip moves the record pointer forward or backward in the current work area. When you select this menu item, a dialog box opens in which you enter the number of records to skip:



Enter a positive number to move the record pointer forward and a negative number to move it backward.

Selecting OK updates the record pointer by skipping the specified number of records and highlights it for editing in the window. Specifying a number which would advance the record pointer beyond the beginning or end of file selects the first or last record—no error message is displayed.

F8 Set

F8 Set allows you to establish relationships between database files and to set up a field list and filter condition to define a view. The menu items are as follows:

- Relation
- Filter
- Fields

All of these menu items require at least one open database file. Relation requires at least two, in addition to the possible requirement of an index file.

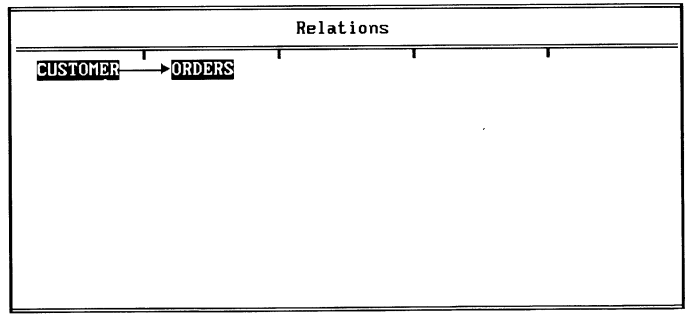
To save the Set menu settings, you must select F4 Save View. Otherwise, the settings will be lost when you exit DBU.

Relation

Relation allows you to model relationships between the open database files in the Files area, just as you would with the SET RELATION command but from an interactive window. To use this item, make sure that all the necessary database and index files are open in an order that reflects the hierarchy of the relationship you want to establish—this is important since you can only establish relations from left to right.

Selecting Relation opens the Set Relation window in which you select files and enter expressions to relate them. If there are already relations established between the files on the main screen, they appear in the Set Relation window for you to edit. Otherwise, the window is empty and you have to start from scratch.

The figure below shows a window in which a relation is being established:



To establish a relation between a parent and a child:

1. Move the highlight in the window to the work area column containing the parent file and press Enter.

   The file name appears in the window.

2. To select a child file, press Enter again.

   An arrow appears to visually indicate that a connection is being made between the two files.

3. Move the highlight to the work area column containing the child file and press Enter.

   A fill-in field opens up just beneath the file names.

4. Type the relation expression, and press Enter when you are finished.

   Depending on whether or not the child file is indexed, the value that you enter here takes on a different significance. If the file is indexed, the two files are related based on the controlling index key of the child; if not, they are related based on record number. In the former (indexed) case, enter the field (or expression) in the parent that you want compared to the index key in the child. In the latter (unindexed) case, enter a numeric value (usually the RECNO() function) that you want compared to the record number in the child.

At this point, you can press Down arrow and establish another relation by repeating the steps above. If a particular database file is the parent of more than one relation, a multiple parent-child relation is established. Insert inserts a relation definition and pushes existing ones down in the window, and Delete deletes the current relation. You can establish up to 15 relations per view in DBU.

The following table summarizes the keys that are available for use when a Set Relation window is active.

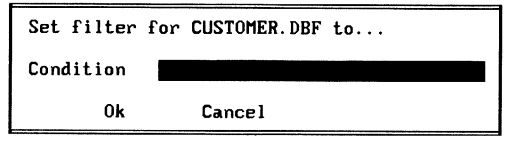
Key	Action
Enter	Enter select mode; select parent or child; exit select mode
Esc	Exit select mode without saving; close Set Relation window
Up arrow	Go to previous relation
Down arrow	Go to next relation
Left arrow	In input mode, go to previous file
Right arrow	In input mode, go to next file
Insert	Insert new relation
Delete	Delete current relation
PgUp	Scroll relations back
PgDn	Scroll relations forward

After a relation is established, selecting F5 Browse View reflects the relation by showing how the records in the various database files are connected. For example, if two files are related using an index key, records with matching key values in the parent and child file are on the same line, appearing as one record in the Browse window.

Filter

Filter allows you to impose a logical condition that each record in the database file must meet, just as you would with the SET FILTER command. This menu item operates on the currently highlighted database file, allowing you to set a filter condition for each open file.

When you select Filter, a dialog box opens and allows you to enter a logical expression into a fill-in field:



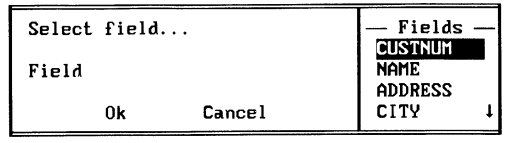
Selecting OK activates the filter for the current file so that only those records meeting the condition are available for processing. If there is an error in the condition, an error message is displayed, and the dialog box remains open on the screen. You can either Cancel or enter a new condition.

After a filter is established, selecting F5 Browse Database reflects the filter condition by showing only those records that meet the criteria. The filter condition is also reflected in the operation of certain F6 Utility menu items such as Copy and Replace. The filter remains in effect until you remove it by selecting F8 Set Filter and deleting the condition.

Fields

Fields allows you to establish a field list for the current database file. This menu item operates on the currently highlighted database file, allowing you to set a field list for each open file.

Selecting Fields activates the Set Fields dialog box where you can select a single field name to add to the current field list:



When you select OK, the field is inserted in the Fields section for the current work area, pushing all other fields down one slot.

You can activate the Set Fields dialog box by pressing Enter or Insert instead of using the menu. If you know the name of the field that you want to include, you can also type it directly on the main screen.

The fields list that you see on the main screen and that you establish with the F8 Set Fields dialog box is used by F5 Browse to determine which fields to show in the Browse window and in what order. All other operations in DBU use all fields defined in the database file structure.

The DBU System Architecture

DBU is provided in source code form. If you installed CA-Clipper in the default configuration, the following source files are located in the \CLIP53\SOURCE\DBU directory:

- Dbu.prg
- Dbu.hlp
- Dbucopy.prg
- Dbuedit.prg
- Dbuhelp.prg
- Dbuindx.prg
- Dbunet.prg
- Dbustru.prg
- Dbuutil.prg
- Dbuview.prg

The system is provided as source code so you can modify it to suit your specific needs. You may want to make enhancements specific to your development environment or simply add the DBU system to your application, thereby giving your users the power to create and modify their own database file structures.

To facilitate maintenance of the DBU program files, the make file, DBU.RMK, is also installed with the DBU source files. Assuming your current DOS directory is \CLIP53\SOURCE\DBU, you can easily rebuild DBU.EXE after you make source code changes with the following command:

```
RMAKE DBU
```

If your changes involve additional source modules or changes in the existing file dependencies, you will need to update DBU.RMK to reflect those changes. See the "Program Maintenance—RMAKE.EXE" chapter in this guide for more information about make files.

Chapter 17
Report and Label Utility—RL.EXE

In This Chapter

CA-Clipper gives you the ability to create and modify standard dBASE III PLUS report and label definitions by supplying a report and label utility, RL. RL is a stand-alone application written in CA-Clipper that emulates the dBASE CREATE | MODIFY REPORT and LABEL commands.

RL creates a binary report (.frm) or label (.lbl) file, which you can later use with the REPORT FORM or LABEL FORM commands to format and print data on a record by record basis from one or more open database files.

This chapter explains how to use RL to create and modify report and label definitions and also provides information on how to use the resulting report and label definitions in your CA-Clipper applications.

The following general topics are covered in this chapter:

- Loading the report and label utility
- Creating and modifying reports
- Creating and modifying labels
- Leaving RL
- The RL system architecture

Loading the Report and Label Utility

If you installed the default configuration of the CA-Clipper development system, RL.EXE is located in \CLIP53\BIN. To execute RL from DOS, use the following syntax:

```
RL
```

Like all other CA-Clipper utilities, you can execute RL from any drive or directory since the appropriate directory was placed in the PATH list during installation.

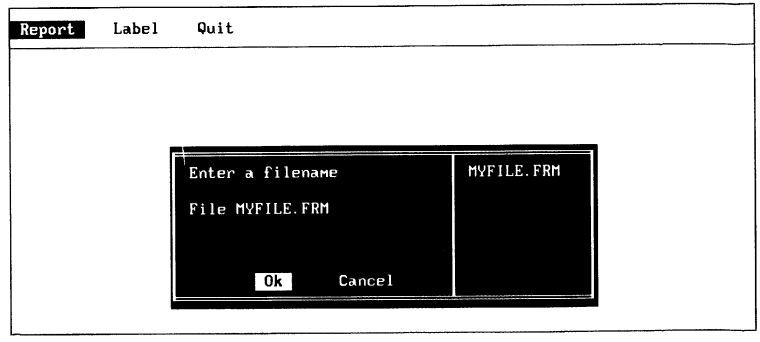
Creating and Modifying Reports

The main menu contains a Report item that activates the report editor. The report definitions you can design are standard columnar reports with report and column headings. When totaling numeric columns you can have up to two levels of subtotals and generate a grand total for each column.

From there, you can create and modify report form (.frm) files that are executable with the REPORT FORM command. For more information on the REPORT FORM command refer to the "Language Reference" chapter of the *Reference Guide*.

Creating or Modifying a Report

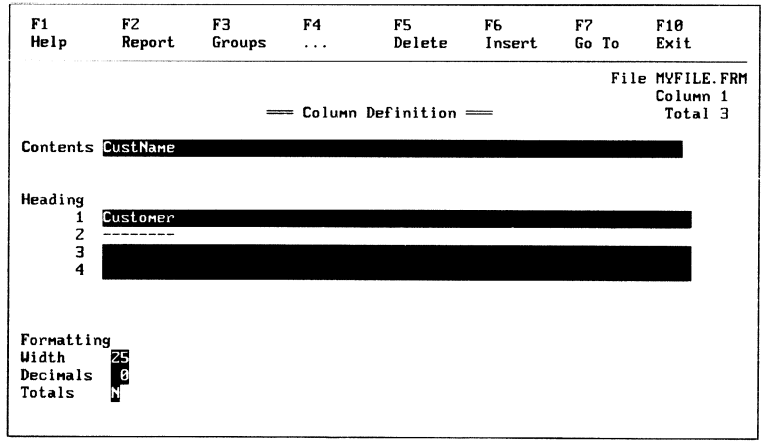
To create or modify a report form, select Report from the main menu. A File dialog box opens as shown below:



You can either create a file by entering a new filename or select an existing file to modify from a scrolling list displayed on the right side of the dialog box. To create or open the (.frm) file you specified or highlighted, select OK. To cancel and return to the main RL menu, select Cancel.

Defining Report Columns

Having selected a report to edit, you are presented with the Column Definition screen, where you can design a new report or make changes to an existing one. The Column Definition screen corresponds to the F4 Columns menu item. It is here that you define the columns of your report by adding, changing, or deleting columns. The maximum number of columns per report is 24.



Columns appear on the resulting report in the order you define them. A detail line is displayed for each record in the corresponding database file.

A report column is defined by entering each of its attributes on the Column Definition screen. To do this, move the cursor to the setting that you want to define and type in the corresponding information. To add subsequent columns, press PgDn to navigate to a new Column Definition screen.

Once you have defined several column definitions in this way, you can use PgUp and PgDn to navigate to the next or previous column definition for editing.

Contents

Each column has a contents expression. It is here that you enter the name of a field or a more complicated expression—you can specify most data types, including memo fields. Values of each data type display in the default format and alignment settings, unless explicitly formatted. As a general rule, you can use the TRANSFORM() function to apply picture formatting to each data type. Refer to the "Language Reference" chapter of the *Reference Guide* for the syntax and arguments. Specific formatting considerations for each data type are discussed below:

Character strings: Display left-justified and can be formatted using TRANSFORM(). Character strings also wrap within the column boundaries if longer than the specified Width setting. To force a line break and wrap the rest of the column to a new line, embed a semicolon (;) using an expression like *<cString1>* + ";" + *<cString2>*.

Memo fields: Treated in the same way as character strings, but there are some special considerations since they can contain embedded soft or hard carriage returns which adversely affect word wrapping. For a memo field to display correctly, you must replace these characters using either HARDCR() or MEMOTRAN().

Date fields: Display left-justified and are formatted according SET DATE. To format the date differently, create a user-defined function.

Logical values: Display left-justified as either true (.T.) or false (.F.). To define a new format, you can use either a user-defined function or the IF() function.

Logical values: Display right-justified according to the current DECIMALS and FIXED settings. To format numeric values to business or other common numeric formats, use TRANSFORM(). Note, however, that TRANSFORM() converts a numeric value to a character value, which prevents totaling of the column. Unfortunately, this is a limitation of the REPORT FORM architecture.

Note: If you specify functions located in libraries other than CLIPPER.LIB in a report definition and do not explicitly reference these functions elsewhere in your program, you must declare them to the linker using the REQUEST statement. The REQUEST statement forces the linker to bring the code for specified routines into your program even though there are no references to the routines in any of the compiled object modules. You can declare these functions external in any source file using the REQUEST statement as long it is compiled and linked into the current executable file. Refer to the "Language Reference" chapter of the *Reference Guide* for more information on declaring external routines.

Heading

Heading is a literal string of up to four lines that displays above the current column. The heading displays left-justified regardless of the data type of the current column.

Formatting

You specify how a column is formatted using this group of options.

Width: Defines the display width of the current column. If the column type is numeric and the Decimals setting is greater than zero, the Width includes the decimal point and the decimal digits. The default width is 10. When the report is run, if the contents exceed the specified Width and the type is character, the column is wrapped to the next line. If the type is numeric, a numeric overflow occurs and the column is filled with a row of asterisk (*) characters.

For numeric columns, you can define these additional attributes:

Decimals: Defines the number of decimal digits to display. The default is zero.

Totals: Determines whether or not a numeric column is totaled. A value of Y causes the column to be totaled. The default is N, indicating that no totaling takes place. Three levels of totaling are available when the Totals setting is Y.

A grand total prints after all other report lines print and, if there are Groups or Subgroups defined, a subtotal prints for each level of grouping as well.

Deleting a Column

After you have added one or more column definitions, you can delete any column you do not want in the report definition. To delete the current column definition, delete the Contents expression by pressing Ctrl+Y then use F5 Delete.

Note: F5 Delete will not delete a column definition unless the Contents area is empty. This is implemented as a precautionary measure to prevent you from deleting a column by accident.

Inserting a New Column

Since the format of the report directly relates to the order of the column definitions, you may need to insert columns between existing column definitions. To do this, position the cursor at the desired location and use F6 Insert to insert a new blank column definition. This new column becomes the current column. You can then enter the column definition attributes.

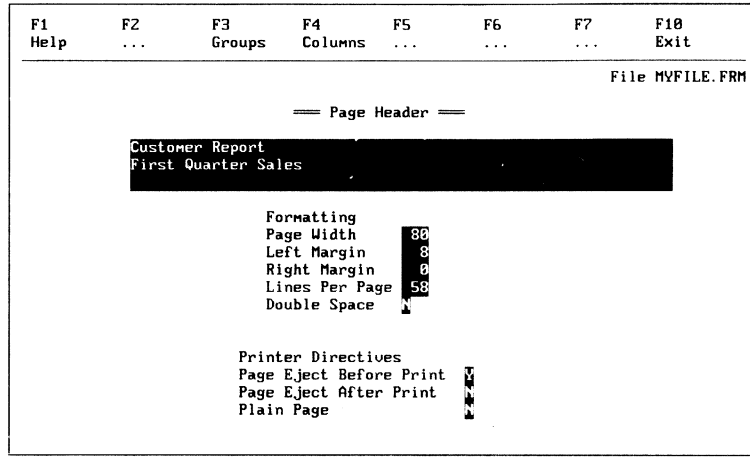
Note: F6 Insert will not insert a column before the first column definition in the report.

Locating a Column

To navigate to an existing column definition, use F7 Goto. A message appears prompting you to enter the number of the desired column. Pressing Enter confirms your entry.

Defining Report Options

Columns are only a part, although a major one, of the entire report definition you can create. There are also options you can specify which apply at the report level. For example, headings and margins apply to the entire report rather than an individual column. To change the default report layout, use F2 Report. The following screen displays:



The report options screen allows you to define the following report attributes:

Page Header

Prints a four line literal string on the top and center of each page if the Plain Page option is N (the default setting). You can achieve the same effect using the REPORT FORM command with the HEADING clause.

Page Width

Determines the number of characters allowed for the combined widths of all report columns. The default is 80.

Left Margin

Determines the number of characters the report is indented. If the report is directed to the printer, this value is added to the SET MARGIN value. The default is eight.

Right Margin	Performs no useful function and is supplied for dBASE compatibility purposes only. The default is zero.
Lines Per Page	Determines the total number of lines to print on each page of the report. This number includes the page heading, column titles, detail lines, and total lines. The default is 58.
Double Space	Determines whether detail lines are double spaced when printed. The default is N.
Page Eject Before Print	Determines whether CHR(12), the formfeed character, is sent to the printer prior to the first report line. You can override this setting by specifying the NOEJECT clause on the REPORT FORM command line. The default is Y.
Page Eject After Print	Determines whether a formfeed is sent to the printer after the last report line is printed. You cannot override this setting from the REPORT FORM command line. The default setting is N.
Plain Page	Corresponds to the PLAIN clause of the REPORT FORM command. If set to Y, the page heading, page number, and report date are not printed and the REPORT FORM HEADING is printed only on the first page. In addition, there are no page breaks. The default is N.

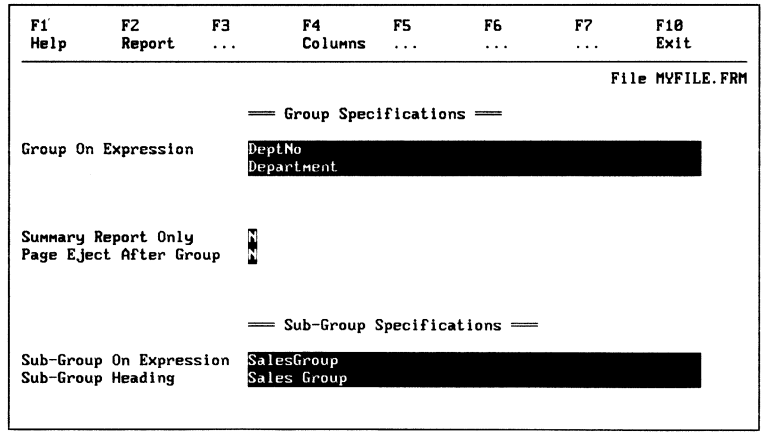
Defining Groups

RL allows you to define two levels of grouping in order to summarize information from one or more related database files.

A group defines a set of consecutive records having the same key value. Each time there is a new key value, a new group begins and a group header line prints.

Each time the current group ends, a summary line prints. If grouping criteria is defined and the Totals attribute on any Column Definition screen is set to Y, subtotals print for numeric report columns. Within a group, subgroups behave in the same manner.

To create groups for the current report, use F3 Groups and the following Group Definition screen displays:



This screen allows you to define both group and subgroup definitions for the current database file. If you do not define grouping criteria, only a grand total is displayed for numeric columns with the Totals attribute on the Column Definition screen set to Y.

The options that you can define on this screen are described below:

Group on Expression — Specifies the primary key expression to group on. This can be a single field or any valid expression involving at least one field. As an example, the group expression might be *DeptNo* so you can generate column totals for each department.

Group Heading — Specifies the title of the group as a literal string. This heading prints before the first report line for each group and just to the left of the group expression.

Summary Report Only — Determines whether detail lines are printed in a grouped report. If you specify Y, detail lines are suppressed and only group header and summary lines print. The default is N. Specifying Y is equivalent to specifying the SUMMARY clause on the REPORT FORM command line.

Page Eject After Group — Determines whether there is a page eject at the end of each group of records. The default value is N. Specifying Y causes each group to begin on a new page.

Subgroup on Expression — Specifies a secondary key expression defining a second level of grouping. As an example, if the group expression is *DeptNo* the subgroup expression might be *SalesGroup* to obtain a breakdown of each sales group within a department.

Subgroup Heading — Specifies a literal string to print at the beginning of each subgroup.

For grouping to work properly in a report, the corresponding database file must be in order according to the group and subgroup expressions. The most convenient method for ordering a database file is with an index. In keeping with the example given above, the index key would be *DeptNo* + *SalesGroup*. See INDEX in the "Language Reference" chapter of the *Reference Guide* for more information.

Saving the Report Definition

From any screen, you can save the Report Definition using F10 Exit. If you have not made any changes, F10 returns you directly to the main menu. Otherwise, you are presented with a dialog box in which you have three choices:

OK — Saves all changes you have made to the report form (.frm) file and returns you to the main menu.

No — Discards all changes, leaving the original report form file intact, and returns you to the main menu. In the case of a new file, the file is simply not created.

Cancel — Returns you to the current screen where you can continue editing the report definition.

Printing a Report

To print a report form within an application, you must have a corresponding database file as well as an open index file if grouping was specified. Substitute the appropriate filenames in the following CA-Clipper commands and include the commands in a program to print the report:

```
USE <xcDatabase> INDEX <xcIndex list> NEW
REPORT FORM <xcReport> TO PRINTER
```

There are other options available for the REPORT FORM command. See the REPORT FORM entry in the "Language Reference" chapter of the *Reference Guide* for more information on printing reports that you create with RL.

Reporting from Related Work Areas

Using a report form, you can print field information from more than one work area by relating work areas using the SET RELATION command. There are, however, several requirements to make this work.

In RL, you must specify *all* field references using an alias:

<idAlias>-><idField>

This guarantees that a field will always be available when the report is printed.

Before you execute the REPORT FORM command, you must have all the required database files in use and relations set in the proper order. In one-to-one relations, report from the main work area and not from the look up.

For example, suppose you have two databases files, Customer and ZipCode. Customer is the main and ZipCode is the lookup linked by a common field, Zip. The code to execute a report form that prints a Customer list would look something like:

```
USE ZipCode INDEX ZipCode NEW     // Open the lookup
USE Customer NEW                  // Report from main
SET RELATION TO Zip INTO ZipCode  // Establish link
REPORT FORM CustList TO PRINTER
```

If the relationship between the work areas is one-to-many, the order is reversed. Report from the lookup work area instead of the main work area. In addition, the main work area must have an index for the linking key. For example, this time you are reporting Invoice amounts for each Customer. The code to execute this kind of a report would look something like:

```
USE Customer INDEX InvNum NEW  // Open main first
USE Invoices INDEX CustNum NEW // Report from lookup
// Establish the link
SET RELATION TO CustNum INTO Customer
REPORT FORM CustInv TO PRINTER
```

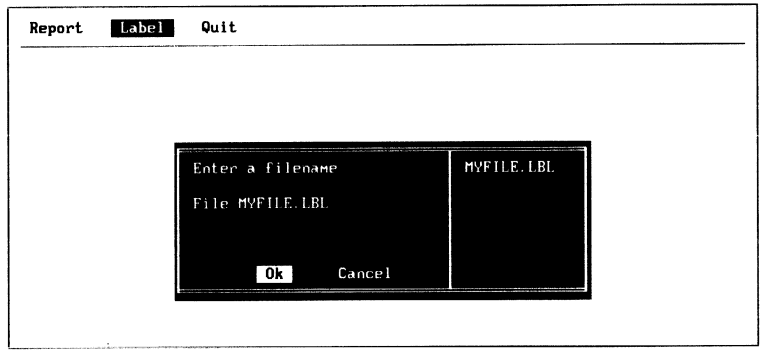
Creating and Modifying Labels

The main menu contains a Label item that activates the label editor. The label definitions you can design are quite versatile, allowing you to define label attributes such as lines per label and number of labels across.

From there, you can create and modify label (.lbl) files executable with the LABEL FORM command. For more information on LABEL FORM refer to the "Language Reference" chapter of the *Reference Guide*.

Creating or Modifying a Label

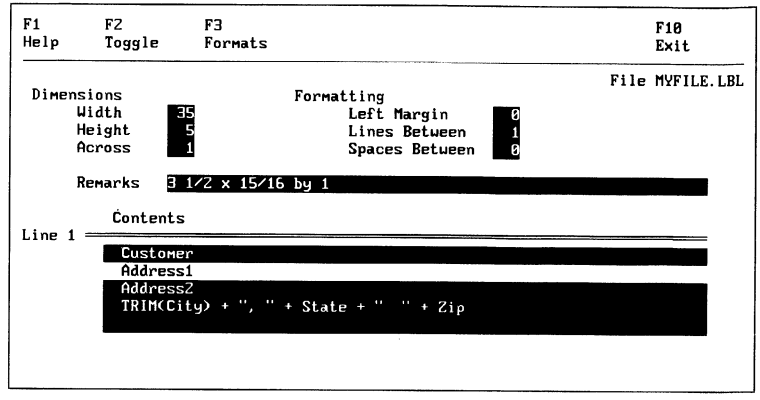
To create or modify a label form, select Label from the main menu. A file dialog box opens as shown below:



You can either create a file by entering a new filename or select an existing file to modify from a scrolling list displayed on the right side of the dialog box. To create or open the (.lbl) file you specified or highlighted, select OK. To cancel and return to the main RL menu, select Cancel.

The Label Editor Screen

After you create a new or select an existing (.lbl) file to edit, the Label Editor screen displays. This screen is divided into two sections as shown below:



Within the top section you define the Dimensions and Formatting attributes of the label. In the bottom section, you define the Contents expressions for each label row. When you first access the editor, the Dimensions and Formatting section is active. F2 Toggle moves the cursor between sections.

Defining the Label Dimensions and Formatting

You can define the following label attributes in the Dimensions and Formatting section of the Label Editor screen:

Width	Determines the horizontal width of an individual label. This value can range from one to 255. The default is 35 characters.
Height	Determines the vertical number of lines in an individual label. This value can range from one to 16 lines. The default is five lines.
Across	Determines the number of labels printed across the page. For multiple across labels, the last label contents printed on a line is trimmed automatically. This value can range from one to 255. The default is one label across.
Left Margin	Specifies the left margin and determines the first print position of the leftmost label. When the labels are printed, this value is added to the SET MARGIN value. This value can range from zero to 255. The default is zero.
Lines Between	Determines the number of blank horizontal lines printed between labels. This value can range from zero to 255. The default is one blank line at the bottom of each label.
Spaces Between	Determines the amount of vertical space printed between labels if the number of labels across is greater than one. You can also use this setting to set the left margin for labels after the first label. Its value can range from zero to 255, and the default is zero.
Comments	Is a comment field that contains standard label format size definitions. Selecting one of the label formats described in the next section fills this area with the appropriate format definition. The default label definition is 3 1/2 x 15/16 by 1. You can also customize the Remarks comment field using your own label format size definitions.

Standard Label Formats

Instead of specifying the dimension and formatting criteria directly, you can select a standard label format from the F3 Formats menu. The table below gives the available standard formats and the resulting label dimensions and formatting:

Remarks	Width	Height	Across	Margin	Lines	Spaces
3 1/2 x 15/16 by 1	35	5	1	0	1	0
3 1/2 x 15/16 by 2	35	5	2	0	1	2
3 1/2 x 15/16 by 3	35	5	3	0	1	2
4 x 17/16 by 1	40	8	1	0	1	0
3 2/10 x 11/12 by 3 (Cheshire)	32	5	3	0	1	2

To select one of these standard label formats, use F3 Formats. When the menu appears on the screen, move the highlight to the desired label format and press Enter. The label dimension and formatting information are automatically updated using your selection.

Defining the Label Contents

After you have defined the label dimension and formatting attributes, F2 Toggle moves the cursor to the Contents section in the bottom half of the Label Editor screen. It is here that you define the actual contents of each label row.

Depending on the Height you have chosen for your label definition, the number of available lines in the Contents area is defined. On each line, you can enter the name of a field or a more complicated expression. You can specify most data types, including memo fields, and the expression itself can be up to 60 characters in length. The result of a contents expression can be any length, but a result longer than the specified label width is truncated when printed by LABEL FORM.

For each label line, enter the expression whose result you want to print on that line. For example, a familiar label content might be:

```
TRIM(FirstName) + " " + LastName
Address1
Address2
TRIM(City) + ", " + State + "  " + Zip
```

To enter the contents for a line, move the highlight to that line and begin typing the expression. Pressing Enter moves the highlight to the next line.

Note: If you specify functions located in libraries other than CLIPPER.LIB in a label definition and do not explicitly reference these functions elsewhere in your program, you must declare them to the linker using the REQUEST statement. The REQUEST statement forces the linker to bring the code for specified routines into your program even though there are no references to the routines in any of the compiled object modules. You can declare these functions external in any source file using the REQUEST statement as long it is compiled and linked into the current executable file. Refer to the "Language Reference" chapter of the *Reference Guide* for more information on declaring external routines.

Important! *The CA-Clipper LABEL FORM command does not support a list of expressions for the label content as in dBASE III PLUS. If more than one expression is specified, a runtime error occurs.*

Blank Lines

In CA-Clipper, a blank contents expression prints a blank line when the label is printed. For example, if you want a blank line between the name and address, the label contents might look like this:

```
TRIM(FirstName) + " " + LastName

Address1
Address2
TRIM(City) + ", " + State + "  " + Zip
```

If, however, a label contents expression that returns a null string ("") is specified, the resulting blank line is suppressed when the label is printed. A good example of this is the second address line in a mailing list which is often empty.

To display a blank line when the contents value is a null ("") string, the contents expression must return a nonprintable character. You can generally enforce this by specifying the following IF() expression on the label contents line:

```
IF(!EMPTY(<cString>), <cString>, CHR(255))
```

You can also create a user-defined function that performs the same action and use it as the label content (e.g., NoSkip(Address2)):

```
FUNCTION NoSkip(exp)
   RETURN IF(!EMPTY(exp), exp, CHR(255))
```

Saving the Label Design

When you have finished specifying the label, you can save the definition using F10 Exit. If you have not made any changes to the current label definition, F10 returns you directly to the main menu. Otherwise, you are presented with a dialog box offering the following three choices:

OK — Saves all changes you have made to the label form (.lbl) file and returns you to the main menu.

No — Discards all changes, leaving the original label file intact, and returns you to the main menu. In the case of a new file, the file is simply not created.

Cancel — Returns you to the current screen where you can continue editing the label definition.

Printing the Labels

To print the label using the label definition you have designed, you must have a corresponding database file in USE then invoke the LABEL FORM command specifying the appropriate label definition:

```
USE <xcDatabase> INDEX <xcIndex list>
LABEL FORM <xcLabel> TO PRINTER
```

The LABEL FORM command has a number of options to determine which records to print and whether to print a sample set of labels in place of the actual labels. Refer to the LABEL FORM entry in the "Language Reference "chapter of the *Reference Guide* for more information on printing labels.

Leaving RL

When you have finished designing report and label definitions, you can exit RL by selecting Quit or by pressing Esc from the main menu.

The RL System Architecture

RL is provided in source code form. If you installed CA-Clipper in the default configuration, the following source files are located in the \CLIP53\SOURCE\RL directory:

- Rlfront.prg
- Rlback.prg
- Rldialg.prg

The system is provided as source code so you can modify it to suit your specific needs. You may want to make enhancements specific to your development environment or simply add the RL system to your application, thereby giving your users the power to create their own reports and labels.

The RL system is broken up into two primary subsystems: front and back. The RL front subsystem (found in Rlfront.prg) is the user interface of the entire RL system. The back subsystem (found in Rlback.prg) contains all of the routines for reading and writing (.frm) and (.lbl) files. The system is organized in this way so that you can create your own user interface and easily access the lower-level routines to read and write the form files.

To facilitate maintenance of the RL program files, the make file, RL.RMK, is also installed with the RL source files. Assuming your current DOS directory is \CLIP53\SOURCE\RL, you can easily rebuild RL.EXE after you make source code changes with the following command:

```
RMAKE RL
```

If your changes involve additional source modules or changes in the existing file dependencies, you will need to update RL.RMK to reflect those changes. See the "Program Maintenance—RMAKE.EXE" chapter in this guide for more information about make files.

Chapter 18
Online Documentation—NG.EXE

In This Chapter

The CA-Clipper DOS-reference documentation is provided in the form of several databases referred to collectively as *The Guide To CA-Clipper*. It contains the most timely reference information on CA-Clipper commands, functions, and utilities.

The Guide To CA-Clipper is accessible via the Norton Instant Access Engine™ (NG.EXE), a memory-resident program also included with CA-Clipper. In the default configuration, all of the documentation database files and the Instant Access Engine are installed in the \NG directory, and the PATH is updated to include this directory.



Note: The Workbench provides a separate online help system for all menu commands, dialog boxes, and procedural steps. From the Workbench, press F1 or click on the Help pull-down menu to access this online help system.

In this chapter, the following general topics are discussed:

- Loading the Instant Access Engine
- How the Instant Access Engine searches for files
- Using the Access window
- Viewing a documentation database
- Instant Access Engine navigation keys
- Configuring the Instant Access Engine
- Leaving the Instant Access Engine

Loading the Instant Access Engine

The Instant Access Engine can be loaded in two ways:

- *Memory-resident mode* where the Instant Access Engine loads as a TSR (terminate and stay resident) program and remains in memory until explicitly removed.
- *Pass through mode* where the Instant Access Engine loads and then runs a specified application program and remains in memory until the application program terminates.

In either mode, the Instant Access Engine occupies approximately 65 KB of RAM, no matter how large the documentation database you access.

To use *The Guide To CA-Clipper* or any other documentation database, you must first load NG.EXE, the Instant Access Engine. The general syntax is as follows:

```
NG [<command line>]
```

<command line> is any valid DOS command, including arguments. If *<command line>* is specified, NG is loaded in pass through mode. Otherwise, it is loaded in memory-resident mode.

Like all other CA-Clipper utilities, you can execute the Instant Access Engine from any drive or directory since the appropriate directory was placed in the PATH list during installation.

Using Memory-Resident Mode

In memory-resident mode, the Instant Access Engine remains in memory until you explicitly remove it from memory using Options Uninstall or until you reboot your computer. This means you have access to the current documentation database from the DOS prompt or from within any program, including your editor.

Using Pass Through Mode

In pass through mode, the Instant Access Engine is loaded into memory, then the command line program is executed. Terminating the command line program automatically removes the Instant Access Engine from memory. This means that you have access to a documentation database only as long as the command line program is running. For example, the following command line loads the Instant Access Engine then the CA-Clipper program editor:

```
NG PE Sample.prg
```

While operating in the editor, you can activate the Instant Access Engine at any time to view the current documentation database. When you save the program file you are editing and leave the editor, the Instant Access Engine is automatically removed from memory.

Accessing the Instant Access Engine

Once the Instant Access Engine is loaded, you can access the current documentation database by pressing the activation *hot key*. The default hot key is Shift+F1. If this conflicts with a key used by another program, you can change it to a new value using the Options Hot key menu item described later in this chapter.

Note: If you are using DOS 5.0 or later, you must press Esc immediately after Shift+F1 to activate the Instant Access Engine. If you place the statement SWITCHES=/k in your CONFIG.SYS file, you will not have to press Esc.

Important! *The Instant Access Engine will not display if your screen is currently in graphics mode. If the screen is in either EGA 43-line or VGA 50-line mode, the Instant Access Engine will display using 24 lines only.*

How the Instant Access Engine Searches for Files

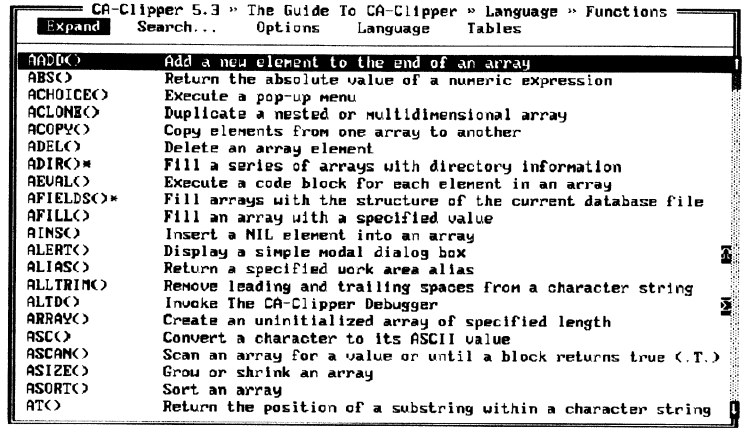
The Instant Access Engine can display documentation databases from any drive or directory, as long as they are in the current directory or the same directory as NG.EXE. When you access the Options Database to select a documentation database to view, the Instant Access Engine shows a scrolling list of documentation databases from the NG.EXE directory followed by documentation databases found in the current directory.

All documentation databases that are part of the CA-Clipper development system were placed in the same directory as NG.EXE during installation; therefore, *The Guide To CA-Clipper* is accessible regardless of the drive and directory that is current when you load the Instant Access Engine.

Tip: If you purchase a third-party product that includes a compatible documentation database, copy the database into the \NG directory. This makes the database accessible from the Options Database list independent of your current directory location.

Using the Access Window

When the Instant Access Engine is first activated, a window appears similar to the one shown below:



At the top of the screen is the name of the current documentation database followed by the major and minor category names. Just below this is the menu bar.

The Menu Bar

The menu bar provides you with options to select different documentation databases to view, search for keywords in the current list of short entries, and change the default configuration of the Instant Access Engine. The first three menus displayed (that is, Expand, Search, and Options) are Instant Access Engine system menus and are always displayed. The remaining menus are placed on the menu bar by the current documentation database.

A menu can execute a function directly, prompt you for input, or pull down a menu. Of the system menus, Expand executes a function directly, Search displays an input field, and Options pulls down a menu of configuration items.

Selecting Menus and Menu Items

When the menu bar is active, you can select a menu either by pressing the first letter of the menu name or by moving the highlight using the Right or Left arrow. If you use the first letter method to select either the Expand or the Search menu, pressing the key executes the menu. If you use the Direction keys to navigate to a menu, you must press Enter to execute the menu.

When you access a pull-down menu, a list of menu items appears. You can then select a menu item the same way you select a menu: either by using the first letter method or by moving the highlight to the desired item with Up or Down arrow and pressing Enter.

To cancel a pull-down menu, press Esc and the highlight moves back to the Expand menu.

Sizing and Moving the Access Window

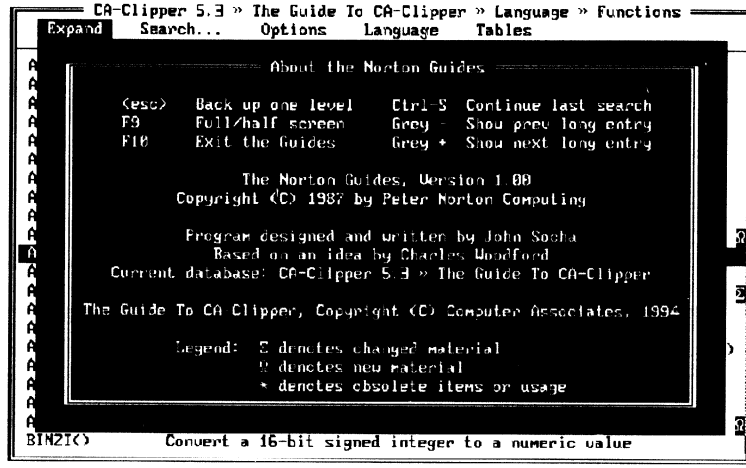
By default, the Access window takes up the entire screen. If you need to see the application program screen without exiting the Instant Access Engine, the Options Full screen menu item, or F9, toggles the size of the Access window between full-screen and half-screen modes. A check mark indicates the current mode is full-screen.

When the Access window is activated in half-screen mode, it pops up either at the top or the bottom of the screen, whichever is furthest away from the current cursor position. If the Access window obscures important information on the application program screen, you can move the window up or down on the screen.

To do this, press Scroll lock then use Up arrow, Down arrow, PgUp, or PgDn to move the window to a new location. When you have finished, press Scroll lock again to freeze the window position. Note, however, that the Instant Access Engine does not remember the new window location. Each time you exit and reactivate in half-screen mode, the Access window always pops up away from the cursor.

Getting Help

While viewing in the Access window, you can pop up a help window like the one shown below by pressing F1:



This window gives you information about the Instant Access Engine and the current documentation database.

Viewing a Documentation Database

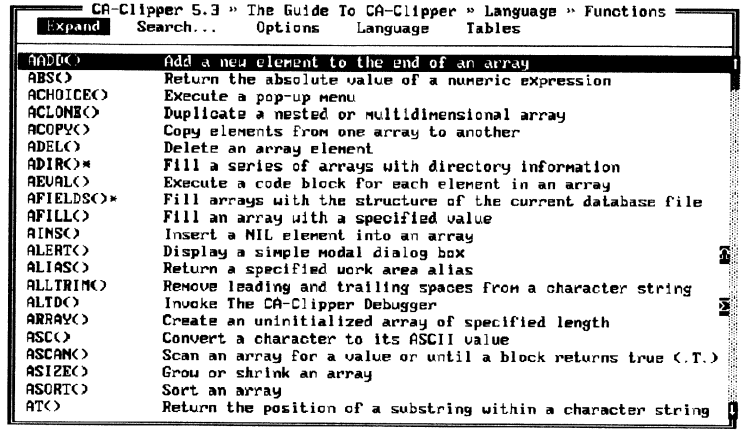
A documentation database is a hierarchical structure consisting of the following items:

- Menus
- Menu items
- Short entries
- Long entries
- See also references

In a typical documentation database, a menu lists a series of categories as its items. Each menu item, in turn, refers to a list of short entries or a single long entry. Each short entry can also refer to another list of short entries or a long entry. If you consider a documentation database as a tree structure, long entries are nodes of the tree and contain the actual topical information. Since long entries are the nodes of the information tree, they can only refer to other long entries by way of the See Also list.

The Short Entry List

Having selected a subject area of the current documentation database using the menu system, you are generally presented with a list of short entries. Short entries usually consist of a keyword, such as a command or function name, followed by a short description. For example, the following shows the short entry list of CA-Clipper functions:



To navigate a short entry list, you can use the typical cursor navigation keys (refer to the Instant Access Engine Navigation Keys section later in this chapter) or the Search menu.

Searching a Short Entry List

Instead of scrolling or paging through a list of short entries, you can search for a short entry containing a specified text string using the Search menu. When you select Search, a dialog box pops up into which you can enter a string to locate.

If you have already performed a search, the previous search string reappears. To enter a new search string, begin typing and the previous entry is automatically erased. If you wish to edit the existing text, press Home (or any other cursor movement key) before typing any new text.

Once you have entered the search string, press Enter to perform the search operation. Searching begins with the next entry and continues to the end of the list. If there is no match, the search continues from the top of the list.

The search operation is not case-sensitive, allowing you to use any combination of uppercase and lowercase letters. In addition, each search scans short entries for a matching substring, allowing you to specify a partial search string. For example, if Functions is the current short entry list, searching for "ST" locates the short entry for STR().

To continue a search for the next occurrence of the search string, re-execute the Search menu or press Ctrl+S.

Expanding an Entry—Moving Down a Level

Once you have located a short entry of interest, you can move to a lower-level in the database hierarchy by executing the Expand menu. The referenced entry can be another short entry list or a long entry.

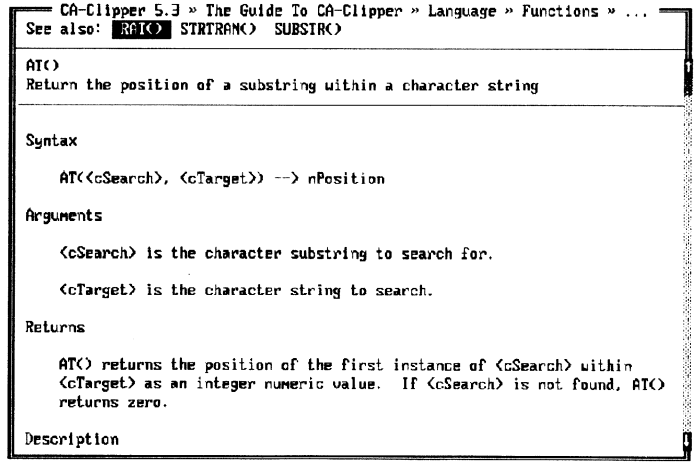
Note that Expand is always the default menu unless you explicitly access another menu. Even then, after you make a selection the highlight automatically moves back to Expand. This makes moving down a level very easy since pressing Enter almost invariably executes Expand.

If the referenced entry is a long entry, the Access window display area is replaced with the long entry text and the menu bar with the associated See Also list. Once you are located within a long entry, you can navigate using the cursor keys. For example, Up arrow, Down arrow, PgUp, and PgDn move within the window, while Left and Right arrow navigate the See Also references (if there are any).

See Also References

Within a long entry, the menu bar is replaced with a list of See Also references. A See Also reference allows you to navigate directly to another long entry without navigating through the menus and short entry lists.

To select a See Also reference, move the highlight using Left or Right arrow and press Enter. You can also select a See Also reference using the first letter method and pressing Enter. The following shows a typical long entry and See Also list:



If you navigated to the current long entry by way of a See Also reference, the Instant Access Engine provides two ways to return to the referring long entry:

- If the current long entry has the referring long entry name as a See Also reference, that reference is highlighted. To return to the referring long entry, press Enter. As an example, if you begin with the SUBSTR() function long entry and execute the AT() See Also reference, the SUBSTR() See Also reference in the AT() long entry is highlighted. By pressing Enter you return to the SUBSTR() long entry, and the AT() See Also reference is again highlighted. In this way, you can move quickly between two related long entries by pressing Enter consecutively.
- If there is no See Also reference to the referring long entry in the current See Also list, the Instant Access Engine puts a special See Also reference, Previous, at the end of the current list. As before, you can press Enter to return to the referring long entry. Note, however, the Instant Access Engine remembers only one level of reference with Previous.

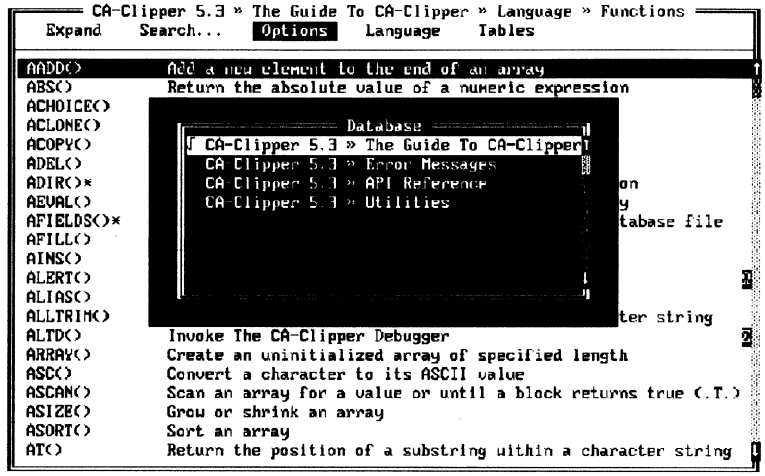
Moving Up a Level

From any level in the database hierarchy, you can move up to the previous level by pressing Esc. If you are currently located in a long entry, this moves you up to the referring short entry list. If you are at the top level of the database hierarchy, Esc causes you to exit the Instant Access Engine altogether.

Note: If you accidentally exit the Instant Access Engine when you press Esc, simply reactivate it using the hot key.

Selecting a New Documentation Database

To select another documentation database to view, use the Options Database menu item. This allows you to select a documentation database from a scrolling list as shown in the following screen:



Documentation databases found in the NG.EXE directory are listed first, followed by those found in the current directory. Documentation databases are arranged in the order the actual database (.NG) files are encountered. To change the order, you must physically sort the .NG files with a disk utility.

To select a new documentation database to view, navigate the list using Up arrow, Down arrow, Home, or End, and press Enter to select the currently highlighted documentation database. Esc cancels the selection.

Instant Access Engine Navigation Keys

The following is a list of all the navigation keys available while you are operating within the Instant Access Engine:

Key	Function	Mode
F1	Help	All
F9	Full/half-screen toggle	All
F10, Shift+F1	Exit	All
Esc	Up a level Exit	Short, Long Top-level Short
Enter	Down a level Select item Select See Also	Short Menu bar, Menu Long
Gray -	Previous long item	Long
Gray +	Next long item	Long
Ctrl+S	Continue last search	Long
Scroll lock	Enable/disable moving	All
Up arrow	Previous item Previous line Window up a line	Menu, Short Long Move
Down arrow	Next item Next line Window down a line	Menu, Short Long Move
Left arrow	Previous item	Menu bar, See also
Right arrow	Next item	Menu bar, See also

Continued

Continued

Key	Function	Mode
PgUp	Last item previous screen	Short
	Last line previous screen	Long
	Window to top	Move
PgDn	First item next screen	Short
	First line next screen	Long
	Window to bottom	Move
Home	First short item	Short
	First line	Long
	Move window top	Move
End	Last short item	Short
	Last line	Long
	Window to bottom	Move

Configuring the Instant Access Engine

The Options menu allows you to change certain parameters affecting the display and operation of the Instant Access Engine. The menu contains the following items:

- Database
- Color
- Full screen F9
- Auto lookup
- Hot key
- Uninstall
- Save options

The Database and Full screen items have already been discussed. The remaining options are discussed in this section.

Toggling Color

The Options Color menu item toggles color and is useful only if you have a color monitor. A checkmark indicates that color is on.

Toggling Auto Lookup

The Options Auto lookup menu item toggles the automatic search mode. A checkmark indicates the mode is turned on.

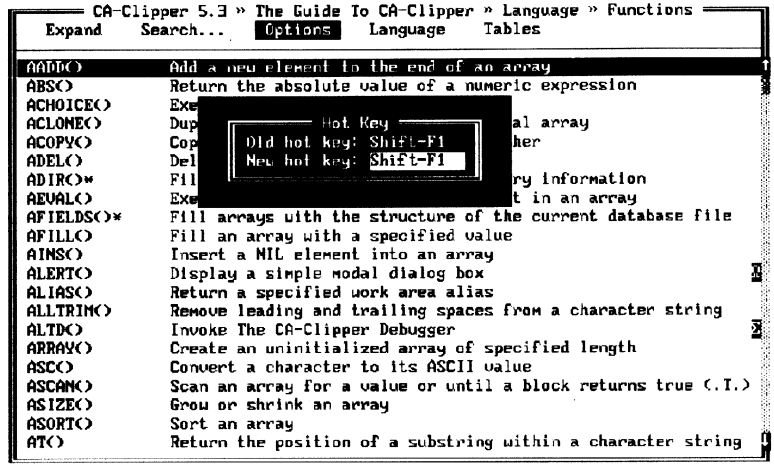
When this option is on, activating the Instant Access Engine automatically searches the current short entry list for the word at the application program cursor position. If found, the matching entry is highlighted in the short entry list.

If Auto lookup is off, the entry highlighted when you last left the Instant Access Engine is still highlighted when you reactivate it.

This menu item is very useful if you are accessing the Instant Access Engine from your editor while writing or editing a program. For instance, if you do not know the correct syntax for a function you can type the function name and press Shift+F1. The function will be highlighted automatically and pressing Enter zooms into the long entry for that function, allowing you to view the function's syntax.

Changing the Hot Key

The Options Hot key menu item allows you to change the key pressed to activate or exit the Instant Access Engine. The default hot key is Shift+F1, but you can change this to any valid key including the Shift, Alt, and Ctrl key combinations. When you select Options Hot key, the dialog box shown below displays:



To change the hot key, press the new hot key or key combination that you want to use. Pressing Enter confirms your choice, and pressing Esc cancels any changes you have made.

Note: Since the Instant Access Engine hot key takes precedence over your application program key definition for the same key, be sure to choose a hot key that does not conflict with other software that you are using.

Saving the New Configuration

While you are using the Instant Access Engine, items that you change in the Options menu remain in effect until you remove the Instant Access Engine from memory. If you want to save your changes as the new default settings, execute the Options Save options menu item. Then, the next time you load the Instant Access Engine, these settings will be the default settings.

When you execute the Options Save options menu item, the Instant Access Engine saves the current options in NG.INI, located in the same directory as NG.EXE.

Leaving the Instant Access Engine

To leave the Instant Access Engine, you can either return to where you were in your application program or remove the Instant Access Engine from memory.

Exiting the Instant Access Engine

When you have finished viewing in the Instant Access Engine, you can return to the original position in your application program by pressing the hot key or F10 from any level or view. You can also press Esc from the top-level short entry list.

Uninstalling the Instant Access Engine

If you loaded the Instant Access Engine in memory-resident mode, you can remove the Instant Access Engine from memory using the Options Uninstall menu item. If you loaded the Instant Access Engine in pass through mode, the Engine is automatically unloaded from memory when the launched application is terminated, and the Options Uninstall is not operational.

Note: In some cases, you may not be able to remove the Instant Access Engine from memory. This may occur if you load another memory-resident program after the Instant Access Engine, and it is still memory-resident. If this happens, uninstall the memory-resident program loaded after the Instant Access Engine, then uninstall the Instant Access Engine. Failing this, you may have to reboot your computer using Ctrl+Alt+Delete.

Appendix A
CLIPPER.BML

This appendix provides information about the resource files that are included in the CLIPPER.BML bitmap library file. This file is installed in the \CLIP53\LIB directory. A listing of the files in CLIPPER.BML can be obtained by using the BMLDIR.COM utility, which is also installed in the \CLIP53\BIN directory.

The following table describes the available bitmap files:

File	Graphic	Description
ARROW_D.BMU		Down arrow used by vertical scroll bars
ARROW_E.BMU		Thumb wheel used in scroll bars
ARROW_L.BMU		Left arrow used by horizontal scroll bars
ARROW_R.BMU		Right arrow used by horizontal scroll bars
ARROW_U.BMU		Up arrow used by vertical scroll bars
CHECK_E.BMU		Unselected check box
CHECK_F.BMU		Selected check box
DROPBOX.BMU		Drop-down list box
RADIO_E.BMU		Unselected radio button
RADIO_F.BMU		Selected radio button

The following tables list the available font (.FNT/.FND) files:

.FNT Files	.FNT Files
ARIAL04.FNT	C_NEW14.FNT
ARIAL05.FNT	C_NEW16.FNT
ARIAL06.FNT	C_NEW20.FNT
ARIAL08.FNT	C_NEW24.FNT
ARIAL10.FNT	C_NEW30.FNT
ARIAL12.FNT	C_NEW36.FNT
ARIAL14.FNT	C_NEW40.FNT
ARIAL16.FNT	SMALL06.FNT
ARIAL20.FNT	TNR04.FNT
ARIAL24.FNT	TNR05.FNT
ARIAL30.FNT	TNR06.FNT
ARIAL36.FNT	TNR08.FNT
ARIAL42.FNT	TNR10.FNT
ARIAL43.FNT	TNR12.FNT
C_NEW04.FNT	TNR14.FNT
C_NEW05.FNT	TNR16.FNT
C_NEW06.FNT	TNR20.FNT
C_NEW08.FNT	TNR24.FNT
C_NEW10.FNT	TNR36.FNT
C_NEW12.FNT	TNR43.FNT

.FND Files	.FND Files
ANTIQUE.FND	FRESN.FND
BIGSF.FND	ITALIC.FND
BLOCK.FND	KIDS.FND
BOLD.FND	NORMS.FND
BOXROUND.FND	OCR.FND
BWAY.FND	ROM8PIX.FND
CNTDOWN.FND	ROMAN.FND
COURRIER.FND	THIN1.FND
CURFONT.FND	THIN2.FND

Index

!

! (negate), 2-53

!= (not equal), 2-52

#

(not equal), 2-52

#include, 2-8, *See also* Header files, 10-14

$

$ (substring), 2-52

%

% (modulus), 2-51

%= (modulus and assign), 2-57

&

& (compile and run), 2-60, 2-66, 2-80, 2-95

&& (comment), 2-8

(

() (group), 2-49, 2-60, 2-65

*

* (comment), 2-8

* (multiplication), 2-51

** (exponentiation), 2-51

**= (exponentiation and assign), 2-57

*/ (comment), 2-8

*= (multiplication and assign), 2-57

+

+ (addition), 2-51

+ (concatenation), 2-50

++ (increment), 2-58

+= (addition and assign), 2-57

+= (concatenate and assign), 2-57

-

- (concatenation), 2-50

- (subtraction), 2-51

-- (decrement), 2-58

-= (concatenate and assign), 2-57

-= (subtraction and assign), 2-57

-> (alias), 2-33, 2-60

.

.AND., 2-53

.AND., truth table, 2-53

.dbf files, 2-105, *See also* Database files

.dbt files, *See also* Memo fields, Memo files, 2-42, 2-105

.EXE
- ignoring linking errors, 11-20
- including specified packages, 11-21
- loading into memory, 11-2
- overriding environmental settings, 11-17
- specifying default options, 11-18

.ndx files, 2-109, *See also* Index files

.NOT., 2-53

.NOT., truth table, 2-53

.ntx files, 2-109, *See also* Index files

.OR., 2-53

.OR., truth table, 2-53

.ppo files, 10-7, 10-10, 10-16

.RMK file, See Make file

.RTLink, using, 11-5

.vew file, 16-13, 16-17, 16-24

/

/ (division), 2-51

/* (comment), 2-8

// (comment), 2-8

/= (division and assign), 2-57

/B compiler option, *See* Compiler options, *See* Compiler options

:

: (send), 2-97, 2-98, 2-99

:= (inline assign), 2-56, 2-57, 2-77, 2-99

;

; (continuation), 2-9

<

< (less than), 2-52

<= (less than or equal), 2-52

<> (not equal), 2-52

=

= (assign), 2-55, 2-57, 2-77

= (equal), 2-52, 2-55

== (array equivalence), 2-84

== (equal), 2-52

== (exactly equal), 2-52

>

> (greater than), 2-52

>= (greater than or equal), 2-52

?

?, 13-5, 13-51, 13-53, 13-55, 13-56, 13-64, 13-82, 13-83, 13-84, 13-85, 13-87

@

@ (pass by reference), 2-17

@ (pass-by-reference), 2-60

@...GET, 4-7

@...SAY, 2-116

[

[] (array element), 2-60

^

^ (exponentiation), 2-51

^= (exponentiation and assign), 2-57

{

{} (array delimiter), 2-60, 2-80

{} (code block delimiter), 2-60, 2-89

|

| | (code block argument delimiter), 2-60, 2-89

A

A20 problems, 11-37

AADD(), 2-83, 2-85

Accelerator keys, 8-2, 8-7, 8-12

Accessing DOS, 13-62, 13-71

ACLONE(), 2-86

ACOPY(), 2-86

AddRec(), 4-10

ADEL(), 2-86

AEVAL(), 2-90
    example, 2-82
    general discussion, 2-57

AFILL(), 2-77

AINS(), 2-86

Alias
    expressions, 2-60
    field, 2-55
    operator, 2-60, 2-101

ALTD(), 2-122, 13-12

APPEND BLANK, 4-10

APPEND FROM, 4-4

application
    controlling execution of, 13-9
    viewing screen, 13-10

Application failures, 11-31

ARRAY(), 2-75, 2-77

Arrays
adding new elements, 2-85
addressing elements, 2-76
and the macro operator, 2-71
as return values, 2-20, 2-81
assigning values to, 2-77
changing the size of, 2-83, 2-85
comparison, 2-84
constant, 2-60, 2-80
copying elements, 2-86
creating, 2-75
definition, 2-75
deleting elements, 2-85, 2-86
determining number of elements in, 2-84
determining size of, 2-84
duplicating, 2-86
subarrays, 2-86
dynamic, 2-75
empty, 2-83, 2-84
equivalence, 2-84
general discussion, 2-75
growing, 2-83, 2-85
initializing, 2-77
inserting elements, 2-86
lexically scoped, 2-75
literal, 2-80
local, 2-75
multidimensional, 2-75, 2-79, 2-86
passing as parameters, 2-16, 2-18, 2-81
private, 2-75
public, 2-75
scanning for a value, 2-88
shrinking, 2-85
sorting, 2-87
static, 2-75
subscript numbering, 2-76
syntax for addressing, 2-76
syntax for creating, 2-75
traversing, 2-82
used with macro operator, 2-80

ASCAN(), 2-88

ASCII files, *See also* Output to text file

ASIZE(), 2-83, 2-85

ASORT(), 2-87

Assignment
array, 2-77
operators, 2-55, 2-57, 2-77
precedence of operators, 2-62, 2-65
table of compound operators, 2-57

AT&T 6300 Plus, running CA-Clipper/Exospace on, 11-35

AUTOEXEC.BAT, 3-2

Automatically overlaying code, 11-2

B

Bad memory, 11-39

BADCACHE, 3-10, 3-12

Beginning of file, 2-126

BIN directory, 10-13, 11-7, 15-2, 16-2

Binary file
closing a, 2-126
creating a, 2-124
moving pointer in a, 2-126
opening a, 2-124
reading a, 2-125, 2-126
writing to a, 2-125, 2-126

BIOS
incompatibility with CA-Clipper/Exospace, 11-31, 11-32
Interrupt 15h, function 88h, used by CA-Clipper/Exospace, 11-46

Blinker commands
(comment), 12-4
// (comment), 12-5
@ (nested script), 12-5
BEGINAREA, 12-6
BLINKER CACHE EMS, 12-7
BLINKER CACHE XMS, 12-8
BLINKER CLIPPER PAGE, 12-9

BLINKER ENVIRONMENT CLIPPER, 12-9
BLINKER ENVIRONMENT NAME, 12-10
BLINKER ENVIRONMENT OVERRIDE, 12-11
BLINKER EXECUTABLE CLIPPER, 12-12
BLINKER EXECUTABLE NODELETE, 12-14
BLINKER LINK EMS, 12-14
BLINKER LINK PAGEFRAME, 12-15
BLINKER LINK XMS, 12-15
BLINKER MESSAGE DUPLICATES, 12-16
BLINKER MESSAGE NOBLINK, 12-17
BLINKER MESSAGE NOWARNING, 12-18
BLINKER MESSAGE WINK, 12-18
BLINKER OVERLAY OPSIZE, 12-19
BLINKER OVERLAY PAGEFRAME, 12-20
BLINKER OVERLAY THRESHOLD, 12-21
BLINKER OVERLAY UMB, 12-22
BLINKER PROCEDURE DEPTH, 12-23
DEFINE, 12-24
DEFLIB, 12-25
ECHO, 12-25
ENDAREA, 12-26
EXTDICTIONARY, 12-26
FILE, 12-27
LIBRARY, 12-28
MAP, 12-29
MIXCASE, 12-29
MODULE, 12-30
MURPHY, 12-32
NOBELL, 12-32
NODEFLIB, 12-33
NOEXTDICTIONARY, 12-33
NOTABLEOFCONTENTS, 12-34
OUTPUT, 12-34
READONLY, 12-35
SEARCH, 12-35
SECTION INTO, 12-36
STACK, 12-37
UPPERCASE, 12-38
VERBOSE, 12-38
WORKFILE, 12-39

Blinker functions, BLIVERNUM, 12-40

Blinker, using, 11-5

BLINKER.TXT, 12-3

BML2RES.COM, 5-12

BMLDIR.COM, 5-12

Boolean algebra, 2-46, 2-53

BP, 13-14, 13-22, 13-58, 13-66, *See also* Point Breakpoint

Branching, conditional, 2-25, 2-26

Breakpoints, 13-4, 13-47, 13-57, 13-61, 13-72, 13-73, 13-106, 13-112
definition, 13-57, 13-66, 13-101
deleting, 13-22, 13-59, 13-66, 13-69, 13-101
listing, 13-59, 13-76
setting, 13-14, 13-22, 13-51, 13-58, 13-66, 13-101

Buffer, management in a network environment, 4-8

C

CA-Clipper
5.01 or 5.01a
linking, 11-15
specifying, 11-15
bitmap library, 1-1
CLIPPER.BML, 5-13, 1-1
debugger, 11-33, 11-34
procedure stack, setting maximum depth, 11-23
UNIVESA.EXE, 5-17

CA-Clipper/Exospace
application failures, 11-31
BIOS Interrupt 15h, function 88h, 11-46
compatibility, 11-41
compatibility with
DOS, 11-47
DOS 5.0 and 6.0, 11-47
DOS 5.0 Task Switcher, 11-48
OS/2 Virtual DOS Machines, 11-49
third-party libraries, 11-41
device drivers, 11-46
DOS return code, 11-7, 11-10
DPMI extended memory protocol, 11-44
error handling, 11-7
extended memory, 11-43
incompatibility with BIOS, 11-31, 11-32
input files, 11-10
invoking, 11-4
linking with
.RTLink, 11-5
Blinker, 11-5
loading, 11-2
memory
requirements, 11-43
shortage, 11-32
output files, 11-8
overview, 11-2
path searching, 11-10
problems with applications, 11-31
running
an application, 11-41
on AT&T 6300 Plus, 11-35
script file, 11-4, 11-11
searching for memory managers, 11-45
software
compatibility with, 11-35
incompatibility, 11-32
syntax for, 11-4
temporary file directory, 11-9
VCPI extended memory protocol, 11-44
working in
Carousel environment, 11-47
DESQview environment, 11-47
DoubleDOS environment, 11-47
Windows environment, 11-47
XMS extended memory protocol, 11-44

CA-Clipper/Exospace commands
abbreviations, 11-12
EXOSPACE CLIPPER 501, 11-15
EXOSPACE ENVIRONMENT CLIPPER, 11-16, 11-17, 11-19
EXOSPACE ENVIRONMENT OVERRIDE, 11-17, 11-19
EXOSPACE EXECUTABLE CLIPPER, 11-17, 11-18
EXOSPACE EXECUTABLE NODELETE, 11-7, 11-20
EXOSPACE PACKAGE, 11-21, 11-22
EXOSPACE PROCEDURE DEPTH, 11-23, 11-30
FILE, 11-8, 11-24
ignored .RTLink commands, 11-13
ignored Blinker commands, 11-14
LIBRARY, 11-25
list of, 11-12
MAP, 11-9, 11-26
MODULE...FROM, 11-13, 11-27
NODEFLIB, 11-28
OUTPUT, 11-8, 11-29
specifying, 11-5
STACK, 11-30
supported .RTLink commands, 11-13
supported Blinker commands, 11-14

Call Stack, 13-60

Calling conventions
command, 2-5, 2-17
function, 2-5, 2-17, 2-18

Callstack, 13-39, 13-68, *See also* View Callstack, 13-115

Callstack window, *See* Windows

Carousel environment, 11-47

Cascading menus, 8-2, 8-15, 8-17

CASE, *See also* DO CASE

CGACURS, 3-10

Changing name of
BLINKER environment variable, 12-10
CLIPPER environment variable, 11-16, 12-9

Changing the DOS Environment Size, 3-4

Character string
comparison, 2-42, 2-43
constants, 2-41
delimiters, 2-41
fixed length, 2-40
literals, 2-41
maximum length of, 2-40
operators, 2-50, 2-57
precedence of operators, 2-62
table of operators, 2-50
variable length, 2-42

CHR(), 2-121

CL.BAT, 10-13, 11-7

Class, *See also* Objects
definition, 2-96

CLD.LIB, 13-11, 13-12
command line syntax, 13-7

CLEAR TYPEAHEAD, 2-123

CLIPPER
BADCACHE setting, 3-10, 3-12
CGACURS setting, 3-10
DYNF setting, 3-10
F setting, 3-12
INFO setting, 3-12
NOIDLE setting, 3-13
SWAPK setting, 3-14
SWAPPATH setting, 3-14
TEMPPATH setting, 3-14
X setting, 3-15

CLIPPER.BML, 5-13

CLIPPER.EXE
command line syntax, 10-2

CLIPPER.LIB, 10-7, 11-25, 11-28

CLIPPERCMD, 10-3, 10-4

Closing files
database, 2-112
index, 2-112
low-level, 2-126

Code blocks, 13-5, 13-60
and the macro operator, 2-72
as return values, 2-20
compared to macro expressions, 2-73, 2-89
compiling with macro operator, 2-68, 2-95
creating, 2-60, 2-89
evaluating, 2-90
evaluating for arrays, 2-82
evaluating for database records, 2-109
example using, 2-92, 2-93
general discussion, 2-89
passing as parameters, 2-16
syntax for defining, 2-89
used to sort arrays, 2-87
variable scoping within a, 2-30, 2-90

Code paging
disabling, 12-9
enabling, 12-9

Code window, *See* Windows

Command window, *See* Windows

Commands,user-defined, 2-89, 2-109

Comments, 2-8

COMMIT, 2-112

Compatibility, *See also* dBASE III PLUS compatibility
notes, 11-35
previous releases, 2-123, 3-5

Compilation, conditional, 10-5

Compile and link batch file, 10-13, 11-7

Compiler
checking syntax with, 10-7
configuration, 10-3
DOS return code, 10-9, 10-13
error handling, 10-9, 10-13
include file directory, 10-5, 10-14
options, 10-4
output files, 10-15
script file, 10-2, 10-9, 10-12
switches, 10-4
syntax for, 10-2
temporary file directory, 10-7, 10-15

Compiler assumptions, 2-101

Compiler options
/B, 13-6, 13-7
/P, 13-6

Compiler switches
/A, 10-4
/B, 10-4, 10-6
/D, 10-5
/ES, 10-5
/I, 10-5, 10-14
/L, 10-3, 10-4, 10-6, 10-7
/M, 10-6, 10-7, 10-9, 10-12
/N, 2-4, 2-20, 2-30, 2-35, 10-6
/O, 10-3, 10-7, 10-15
/P, 10-7, 10-10, 10-16
/Q, 10-7
/R, 10-7
/S, 10-7
/T, 10-7, 10-15
/U, 10-3, 10-8, 10-14
/V, 2-32, 10-8
/W, 2-32, 10-8
/Z, 10-8

Compiling program code, 13-6

CONFIG.SYS, 11-44

Console commands
directing output to a file, 2-119
general discussion, 2-115
printing, 2-117

Control structures
decision-making, 2-25
DO CASE, 2-26
DO WHILE, 2-23
FOR, 2-23, 2-82
general discussion, 2-22
IF, 2-25
looping, 2-22, 2-82
nesting, 2-22

Controlling Allocation of Overlay Pool—/OPc, 3-17

Controlling Allocation of Overlay Pool—/OUc, 3-17

Controlling Amount of EMS Memory—/CEnnn,nn, 3-16

Controlling Amount of XMS Memory—/CXnnn,nn, 3-16

Controlling Operating Size of Blinker Overlay Pool—/OOnnn, 3-16

Conventional memory available, 11-44

Conversion functions
BIN2I(), 2-125
BIN2L(), 2-125
BIN2W(), 2-125
I2BIN(), 2-125
L2BIN(), 2-125

COPY FILE, 4-4

COPY STRUCTURE, 4-4

COPY STRUCTURE EXTENDED, 4-4

COPY TO, 4-4

COUNT, 4-8

CREATE
full-screen, 16-18

CREATE FROM, 4-4

Creating
an .EXE, 11-20, 12-14
files, in DBU, 16-18

CTOD(), 2-44

Cursor, controlling use of, 3-10

D

Data structures, 2-40

Data type
- array, 2-40, 2-75
- character, 2-40, 2-50
- code block, 2-40, 2-89
- date, 2-43, 2-51
- general discussion, 2-40
- logical, 2-46, 2-53
- memo, 2-42, 2-50
- NIL, 2-52
- numeric, 2-45, 2-51
- object, 2-96

Database commands
- syntax for specifying scope, 2-107
- table, 2-106

Database files
- attributes of, 2-106
- closing, 2-112
- creating in DBU, 16-18
- definition, 2-105
- general discussion, 2-100
- maximum number of open in DBU, 16-10
- modifying records, 6-13
- opening, 2-101
- opening in DBU, 16-16
- operations, 2-106
- processing with DBEVAL(), 2-109
- saving structure in DBU, 16-24
- table of attributes, 2-106

Database functions,table, 2-106

Database utility, 2-100, *See* DBU

Dates
- blank, 2-44
- comparison, 2-44
- constants, 2-43, 2-44
- literal, 2-43, 2-44
- operators, 2-51, 2-57
- precedence of operators, 2-62
- range of, 2-44
- separators, 2-43
- validation, 2-44

DBAPPEND(), 4-10

dBASE III PLUS
- database driver, 2-109

dBASE III PLUS compatibility
- compiling dBASE source code, 10-12
- CREATE LABEL, 17-1
- CREATE REPORT, 17-1
- database file format, 2-105
- indexing, 2-109
- LABEL FORM, 17-16

DBEVAL(), 2-90, 2-109

DBFNTX.LIB, 10-7, 11-25, 11-28

DBRLOCK(), 4-9

DBRUNLOCK, 4-11

DBSTRUCT(), 2-75, 2-77

DBU
- appending to a file in, 16-30
- Browse menu, 16-2, 16-8, 16-11, 16-25, 16-33, 16-36, 16-40
- Browse window, 16-25, 16-33
- browsing
  - a database file, 16-25
  - a view, 16-26
- color settings, 16-2
- copying a file in, 16-28
- Create menu, 16-8, 16-18, 16-24
- creating
  - database files in, 16-18
  - files, 16-18
  - index files in, 16-23
- data types, 16-20
- decimal restrictions, 16-21
- dialog
  - boxes, 16-5
  - controls, 16-5
- editing a memo field in, 16-27
- error messages, 16-5
- executing DOS commands from, 16-32

field
naming conventions, 16-20
width restrictions, 16-21
Help
menu, 16-5, 16-8, 16-15
window, 16-15
leaving, 16-13
main screen
navigation keys, 16-12
usages, 16-9
make file, 16-41
marking deleted records in, 16-25
menu
bar, 16-3, 16-13
bar, navigation keys, 16-13
navigation, 16-3
Move menu, 16-25, 16-33
Open menu, 16-10, 16-13, 16-16
opening
database files in, 16-10, 16-16
files, 16-16
index files in, 16-10, 16-17
view files in, 16-17
performing logical searches in, 16-34
prompts, 16-5
removing deleted records in, 16-31
replacing fields in, 16-30
Save menu, 16-13, 16-22, 16-24, 16-35
saving
a database file structure, 16-22, 16-24
a view file, 16-13, 16-24
searching indexes in, 16-33
Set menu, 16-11, 16-35
Set Relation window, 16-36
setting
a field list in, 16-11, 16-40
relations in, 16-36
skipping records in, 16-35
Structure window, 16-8, 16-18
Utility menu, 16-13, 16-28
windows, 16-8

DBU.EXE, 2-100, 16-1, *See also* DBU, 16-41
command line syntax, 16-2
source code, 16-41

DBU.HLP, 16-2

DBUNLOCK, 4-11

Debugger
definition, 13-1
invoking, 2-122, 13-7
menus
accelerator keys, 13-20
accessing, 13-19
File menu, 13-18
Help menu, 13-18, 13-41
Locate menu, 13-18
menu commands, 13-21
Monitor menu, 13-18, 13-37, 13-51
Options menu, 13-18, 13-44
Point menu, 13-18, 13-34
Run menu, 13-18
selecting an option, 13-19
shortcut keys, 13-20
the menu bar, 13-17, 13-19, 13-93
View menu, 13-18, 13-42, 13-43
Window menu, 13-18
quitting, 13-16, 13-71

Debugging, 10-4, 10-6, 14-4
error, 11-33

Declarations
compile time, 2-6, 2-27
general discussion of statements, 2-33
runtime, 2-27

Decrement operators, 2-58

Default libraries, ignoring in a search, 11-28

DELETE, 4-7

Delete, 13-22, 13-54, 13-59, 13-66, 13-69, *See also* Point Breakpoint, Point Delete

Dependency file, 14-6, 14-12
searching for, 14-8

Dependency rule, 14-6, 14-10
syntax for, 14-12

Dependency statement, 14-12, 14-15

DESQview environment, 11-47

Device drivers, 11-46
RAMDRIVE.SYS, 11-47
SMARTDRIVE.SYS, 11-47
VDISK.SYS, 11-47

DEVICE= statement, 11-32

Dialog boxes, 13-45, 13-54, 13-55

Directing output to a file, 2-119, *See also* Output to text file

DIRECTORY(), 2-75, 2-77

Disabling
blinking eyes, 12-17
menu items, 8-15
processing of table of contents, 12-34
use of EMS pageframe, 3-15
VM package, 11-22
warning messages, 12-18
winking during linking, 12-18

DISPLAY, 4-4

Displaying
script on screen, 12-25
status information, 12-38
text on screen, 12-25
variables and expressions, 13-64

DO CASE, 2-26

DO WHILE, 2-23

DO, compiler implications, 10-12

Documentation, online, 18-1

DOS
access from a make file, 14-19
BUFFERS setting, 3-3
command line length limitation, 11-4, 14-14
COMMAND.COM, 3-5
compatibility with
CA-Clipper/Exospace
3.x to 4.x, 11-47
5.0 and 6.0, 11-47
5.0 Task Switcher, 11-48
COMSPEC setting, 3-5
CONFIG.SYS, 3-2, 3-12
default directory, 2-124
error number, 2-127
ERRORLEVEL, 10-5, 10-9, 10-13, 11-7, 11-10, 14-5, 14-6, 14-7
executing commands from DBU, 16-32
file
attributes, 2-124
pointer, 2-125
file pointer, 2-126
FILES setting, 3-3, 3-12
flushing buffers, 2-126
MEM command, 11-44
open mode, 2-124, 4-4
PATH, 11-8, 15-2, 15-5, 16-2, 18-1, 18-2
redirection, 10-9
SHELL directive, 3-4
used with a memory manager, 11-47, 11-48
version requirements for networking, 4-3
version specific information, 3-3

DOS extender, 12-3

DOS16M environmental variable, 11-36

DoubleDOS environment, 11-47

DPMI
extended memory protocol, 11-44
host, problems with, 11-40

Duplicate
modules, enabling warnings for, 12-16
symbols, enabling warnings for, 12-16

Dynamic overlays
caching of, 12-2
managing of, 12-2
specifying file handles at runtime, 3-10
specifying start of area, 12-6

DYNF, 3-10

E

ELSE, *See also* IF

ELSEIF, *See also* IF

EMM, 3-10, 3-12

EMPTY(), 2-83

EMS
- controlling use of pageframe, 12-15
- enabling use of pageframe, 12-20
- maximum amount of, 12-7
- minimum amount of, 12-7
- specifying memory use, 12-14

Enabling
- threshold size, 12-21
- use of EMS pageframe, 12-20
- use of extended library dictionaries, 12-33
- use of library extended dictionaries, 12-26
- use of XMS, 12-22

Enabling menu items, 8-15

End-of-file, 2-126

ENDCASE, *See also* DO CASE

ENDDO, *See also* DO WHILE

ENDIF, *See also* IF

ENDTEXT, *See also* TEXT

Environment
- settings
  - overriding .EXE defaults, 11-17
  - overriding defaults, 12-11
  - specifying, 12-11
  - specifying default, 12-12
- variables
  - changing the name of, 11-16
  - DOS16M, 11-36

Environment variables, 14-22
- BLINKER, 3-15
- changing name of, 12-9, 12-10
- changing settings, 3-15
- CLIPPER, 3-3, 3-9, 3-10, 3-12, 3-13, 3-14, 3-15, 11-16, 11-17, 11-19
- CLIPPERCMD, 10-3, 10-4
- defining in make files, 14-15
- INCLUDE, 10-5, 10-14, 14-21
- LIB, 3-4, 11-10
- OBJ, 11-11
- PATH, 11-8, 16-2
- RMAKE, 14-3, 14-4, 14-8, 14-23
- TMP, 3-5, 10-7, 10-15, 11-9

Error block
- BREAK, 9-11, 9-16, 9-19
- ERRORBLOCK(), 9-16, 9-19
- ERRORNEW(), 9-11, 9-16, 9-19
- EVAL(), 9-11, 9-16, 9-19
- RECOVER, 9-11, 9-16, 9-19
- SEQUENCE, 9-11

Error class
- canDefault, 9-25
- canRetry, 9-25
- canSubstitute, 9-25
- DefError(), 9-25, 9-28
- example, 9-25

Error handling
- ALERT(), 9-28
- compiler, 10-9, 10-13
- default handling, 9-8
- error block vs SEQUENCE, 9-19
- error objects, 9-21
- error processing, 9-4
- error scoping, 9-4
- example, 9-23, 9-30, 9-32, 9-37
- exception
  - handling concepts, 9-2
  - mechanisms, 9-10
- general, 9-1
- handling errors, 9-8
- in a network environment, 4-3, 4-12
- linker, 11-7
- localization, 9-4
- low-level, 2-127

modular programming, 9-2
NETERR(), 9-37
on a network environment, 9-37
posted error block, 9-16
raising errors, 9-4
RMAKE, 14-5, 14-7
runtime, 2-50, 9-1, 10-6
SEQUENCE contstruct, 9-10
strategies, 9-1, 9-28
substitution, 9-8
VAL(), 9-23, 9-25, 9-28

Error messages
not a DOS/16M executable, 11-33
not enough memory, 11-38

Error object
DefError(), 9-37
Error.ch, 9-24
ERRORNEW(), 9-21, 9-23
RECOVER, 9-32

ERRORBLOCK(), 9-28

EVAL(), 2-90

Exclusive mode, 4-1, 4-3, 4-4
commands that require, 4-4

Executable files, *See also* runtime , 10-12
default
extension, 11-8, 11-29
name, 11-4, 11-8, 11-29
discussion of runtime environment, 3-1
generating, 11-4, 11-8
requesting a segment map of, 12-29
searching for, 11-8
setting to read-only status, 12-35
specifying stack size of, 12-37

Executing
code blocks, 13-5
individual functions and procedures, 13-53, 13-64
program code, 13-47, 13-48, 13-49, 13-50, 13-51, 13-105, 13-106, 13-107, 13-108

Execution modes, 13-47
animate mode, 13-47, 13-51, 13-52, 13-91, 13-99, 13-105, 13-109
run mode, 13-48, 13-50, 13-51, 13-106, 13-107, 13-111
single step mode, 13-49, 13-51, 13-89, 13-110
trace mode, 13-49, 13-51, 13-112

EXIT, *See also* DO WHILE, FOR

Exospace commands
EXOSPACE EXECUTABLE CLIPPER, 3-8
STACK, 11-23

EXOSPACE PACKAGE NOVM command, 11-22

EXOSPACE.EXE, 11-43
command line syntax, 11-4

EXOSPACE.LIB, 11-41

Expanded
memory manager, 3-10, 3-12, 11-38
memory usage, 3-10, 3-12

Expressions
displaying, 13-64
evaluation of, 2-48
general discussion, 2-39
inspecting, 13-51, 13-55
passing as parameters, 2-16
precedence used to evaluate, 2-63

EXTEND.LIB, 10-7, 11-25, 11-28

Extended library dictionaries
disabling processing of, 12-33

Extended memory
available, 11-44
in CA-Clipper/Exospace, 11-38
linking an application, 11-43
protocols
DPMI, 11-44
VCPI, 11-44
XMS, 11-44

running an application, 11-43
sharing, 11-35
solving lack of, 11-36
used in CA-Clipper/Exospace, 11-43
using the memory manager, 11-36

EXTERNAL
and macro expressions, 2-74
in LABEL FORMs, 17-5, 17-17
in REPORT FORMs, 17-5, 17-17

External overlays
name of, 12-36
specifing files to be in, 12-36

F

F, 3-12

FIELD alias, 2-33

Fields
declaring, 2-38
passing as parameters, 2-16

FILE command, 11-24

File DOS, 13-62, 13-71

File Exit, 13-16, 13-71

File handle
accessing a, 2-125
obtaining a, 2-124
specifying number at runtime, 3-12

File menu, 13-18
DOS Access option, 13-71
Exit option, 13-71
Open option, 13-72
Resume option, 13-73

File Open, 13-5, 13-61, 13-72, *See also* View, *See also* View, 13-73

File Resume, 13-60, 13-72, 13-73, 13-112

FilLock(), 4-9

FLOCK(), 4-9

FOR, 2-23

FOR condition, 2-107

Format files, compiling, 10-12

FOUND(), 2-111

FROM option, 12-30

Full-screen commands
directing output to a file, 2-119
printing, 2-117

Full-screen operations, general discussion, 2-116

FUNCTION
compiler implications, 10-8

Function keys, 13-22
Ctrl+F5 Run Next, 13-48
F1 Help, 13-16, 13-41
F10 Trace, 13-49, 13-51
F4 Application Screen, 13-52
F5 Execute Application, 13-48, 13-50, 13-51
F7 Run to Cursor, 13-48
F8 Step, 13-49, 13-51
F9 Set/Delete Breakpoint, 13-22, 13-51, 13-58, 13-59
programming, 2-123

Functions, *See also* Library functions, User-defined functions
calling conventions, 2-5, 2-17, 2-18
defining, 2-7
executing, 13-53, 13-64

G

General Protection Fault interrupt, 11-33

Generating
- a map file, 11-26
- an executable file, 11-29

Get Class
- data dictionary, 7-5

Get class
- assign(), 7-6
- block, 6-13, 7-2, 7-3, 7-4
- buffer, 7-6
- cargo, 7-5, 7-27
- clear, 7-6, 7-32
- col, 7-2
- colorDisp(), 7-6
- colorSpec, 7-6
- decPos, 7-6
- display(), 7-6
- exported instance variables, 7-2
- extending Read layer, 7-9
- insert(), 7-6
- messages, 7-22
- name, 7-2
- original, 7-6
- overStrike(), 7-6
- pos, 7-6
- row, 7-2
- setFocus(), 7-6
- setting and retrieving values, 7-4
- undo(), 7-6
- unTransform method, 7-32
- varGet(), 7-4
- varPut method, 7-32
- varPut(), 7-4

Get function layer, 7-31
- creating new, 7-32

Get system
- code blocks, 7-14
- SET KEY, 7-14
- using code blocks for WHEN/VALID, 7-14

Get system commands
- @ GET, 7-7
- @...GET, 7-9, 7-14, 7-18, 7-27
- CLEAR GETS, 7-18
- READ, 7-7

Get system functions
- GETACTIVE(), 7-8, 7-10
- GETAPPLYKEY(), 7-7, 7-25, 7-27, 7-31, 7-32
- GETDOSETKEY(), 7-7
- GetNew(), 7-2
- GETPOSTVALIDATE(), 7-7
- GETPREVALIDATE(), 7-7, 7-25
- GETREADER(), 7-8, 7-25, 7-27, 7-32
- READVAR(), 7-10

Graphic mode
- bitmaps/icons, 5-9
- BML2RES.COM, 5-12
- BMLDIR.COM, 5-12
- CLIPPER.BML, 5-13, 1-1
- common problems, 5-14
- compiling and linking, 5-10
- hardware compatibility, 5-17
- input/output system, 2-113, 5-2
- invoking, 5-5
- memory allocation, 5-18
- overview, 5-2
- pixel coordinates, 5-2
- programming techniques, 5-4, 5-8, 5-9
- push buttons, 5-9
- RES2BML.COM, 5-11
- row/column coordinates, 5-9
- shadowing restrictions, 5-7
- software compatibility, 5-16
- troubleshooting, 5-14
- UNIVESA.EXE, 5-17
- using, 5-6
- utilities, 5-11
- VESA drivers, 5-17
- VGA drivers, 5-17
- x/y coordinates, 5-9

Guide To CA-Clipper, 18-1

H

HARDCR(),in REPORT FORMs, 17-4

Hardware
- compatibility, 11-42
- problems, 11-34
- requirements, 11-42

Header files, 13-5, 13-61, 13-72
- changing the standard, 10-8
- editing, 15-1
- general discussion, 2-8
- RMAKE, 14-21
- searching for, 10-5, 10-8, 10-14
- specifying a directory for, 10-14

Help, 2-123, 13-16, 13-41, 13-74
- in DBU, 16-15
- online, 1-1, 18-1

Help menu, 13-18
- Commands option, 13-74
- Keys option, 13-74
- Menus option, 13-74
- Windows option, 13-74

Help window, *See* Windows

High Memory Area, 11-38

High memory area, putting DOS code in, 11-47

High speed linking, 12-2

I

IF, 2-25

IF()
- in LABEL FORMs, 17-16
- in REPORT FORMs, 17-4

INCLUDE directory, 10-5, 10-14

Increment operators, 2-58

INDEX, 2-110, 4-4

Index files
- closing, 2-112
- controlling index, 2-111, 2-112
- creating, 2-110
  - in DBU, 16-23
- general discussion, 2-109
- maximum number of open, 2-110
- maximum number of open in DBU, 16-10
- opening, 2-110
- opening in DBU, 16-17
- order of, 2-111
- relative searching, 2-111
- searching, 2-111
- updating, 2-112

Index key
- maximum length, 2-110
- unique values, 2-110

Inference rule, 14-10
- syntax for, 14-15

INFO, 3-12

Init.cld, 13-7, 13-14

Input/output system
- graphic mode, 2-113, 5-2
- overview, 2-113
- text mode, 2-113

Insert mode, 13-33

Inspecting
- code blocks, 13-5
- macros, 13-5
- variables and expressions, 13-51, 13-55

Installation
- default
  - directory structure, 11-11
- default directory structure, 10-13, 10-14, 11-7, 11-10, 15-2, 15-5, 16-2, 18-1, 18-2

Instance variables, 2-98
- accessing, 2-99
- assignable, 2-99
- exported, 2-98
- syntax for assigning, 2-99
- syntax for retrieving, 2-99

Instances
creating, 2-97
definition, 2-97

ISCOLOR(), 16-2

J

JOIN, 4-4

K

KEYBOARD, 2-121, 2-123
controlling the, 2-120, 2-122
number of programmable keys available, 2-123
polling the, 2-121
programming, 2-123

Keyboard buffer
and function keys, 2-121
changing the size of, 2-120
clearing the, 2-120, 2-121, 2-123
disabling the, 2-120
extracting a key from, 2-121
general discussion, 2-120
maximum number of characters in, 2-120
minimum size of, 2-120
reading pending key, 2-121
returning last key extracted, 2-121
stuffing the, 2-121

KEYBOARD command, 7-10

Keyboard functions
INKEY(), 2-120, 2-121
LASTKEY(), 2-121
NEXTKEY(), 2-121

L

LABEL FORM, 4-4
creating, 17-1
printing, 17-19

LAN, 4-1
requirements for using CA-Clipper with, 4-3
workstation requirements, 4-3

LEN(), 2-42, 2-84

LIB directory, 3-4, 11-10

Libraries
compared with packages, 11-21
disabling processing of table of contents, 12-34
Disabling use of extended library dictionaries, 12-33
Enabling use of Library extended dictionaries, 12-26
ignoring default library search records, 12-33
locating and linking, 12-25
prioritizing symbols in, 12-35
third-party compatibility, 11-41

Library files
embedded name in object file, 10-7
search paths, 11-10
searching for, 11-10

Library modules
overriding linking sequence, 11-27
specifying placement of, 12-30

Linker
resolving external references, 10-7

Linking
CA-Clipper 5.01 or 5.01a, with, 11-15
object
files, 11-24
modules and libraries with script files, 11-5
required library files, 11-25
sequence, overriding, 11-27

LIST, 4-4, 4-8

List, 13-34, 13-55, 13-59, 13-69, 13-76, 13-101, 13-102

Load size, 11-2

Loading
.EXE into memory, 11-2
CA-Clipper/Exospace, 11-2

Local Area Network, *See* LAN

Locate Case, 13-77, 13-78, 13-80, 13-81

Locate Find, 13-77, 13-78, 13-80, 13-81

Locate Goto, 13-79, 13-88, 13-92

Locate menu, 13-18
Case sensitive option, 13-77, 13-78, 13-80, 13-81
Find option, 13-78, 13-80, 13-81
Goto Line option, 13-79
Next option, 13-80
Previous option, 13-81

Locate Next, 13-77, 13-80

Locate Previous, 13-77, 13-81

Locking
automatic, 4-10
commands that require, 4-7
file locking, 4-1, 4-7, 4-9, 4-12
programming strategies, 4-12
record locking, 4-1, 4-7, 4-9, 4-10, 4-12
releasing locks, 4-11

Locks.prg
AddRec(), 4-10
NetUse(), 4-6

Logical
comparison, 2-47, 2-53
constants, 2-46
delimiters, 2-46
literals, 2-46
operators, 2-53
precedence of operators, 2-62, 2-64
table of operators, 2-53
truth tables, 2-53

LOOP, *See also* DO WHILE, FOR

Looping structures
conditional, 2-23
counter, 2-23, 2-82

Low-level file functions
error table, 2-127
FCLOSE(), 2-126
FCREATE(), 2-124
FERROR(), 2-127
FOPEN(), 2-124
FREAD(), 2-125
FREADSTR(), 2-125
FSEEK(), 2-126
FWRITE(), 2-125
general discussion, 2-124

M

Macro expressions
and external references, 2-74
compared to code blocks, 2-89
creating, 2-60
general discussion, 2-66

Macro operator
and arrays, 2-71
and code blocks, 2-72
compared to extended expressions, 2-69
general discussion, 2-66
limitations, 2-69, 2-71, 2-72, 2-73
nesting, 2-74
used for compiling on the fly, 2-68
used for text substitution, 2-67

Macro variable
definition, 2-66
terminating, 2-66

Macros, 13-5

Main procedure, *See also* Startup procedure

Make, *See also* RMAKE

Make file, 14-2, 14-6, 14-10
comments in, 14-12
default extension, 14-8
defining environment variables in, 14-12
DOS access from, 14-19
example, 14-27
line continuation in, 14-11
searching for, 14-8
using quotation marks in, 14-8, 14-11, 14-12, 14-20, 14-23

Makepath,RMAKE macro, 14-8, 14-12, 14-21

Manifest constants, 13-6
defining, 2-8, 10-5

MAP command, 11-33

MAP file
finding module and routine errors, 11-33
generating, 11-26

Map files
default extension, 11-9
generating, 11-9

Mathematical operators, 2-51

MEM DOS command, 11-44

Memo fields
and character strings, 2-42
comparison, 2-42, 2-43
maximum length of, 2-43
operators, 2-50
relationship to memo file, 2-105

Memo files, *See also* .dbt files
general information, 2-105

MEMOREAD(), 4-4

Memory
requirements, 11-43
shortage, 11-32, 11-38

Memory manager
providing extended memory, 11-36
searched in the CA-Clipper/Exospace environment, 11-45
used with DOS, 11-47, 11-48

MEMOTRAN()
in REPORT FORMs, 17-4

MEMVAR, 10-4

MEMVAR alias, 2-33

MenuItem class
caption, 8-7, 8-9, 8-13, 8-16
checked, 8-13, 8-14, 8-17
data, 8-3, 8-4, 8-8, 8-9
defining
menu commands, 8-2
properties, 8-2, 8-9, 8-10
separators, 8-13
toggles, 8-13, 8-14, 8-17
disabling items, 8-15
enabled, 8-4, 8-15
enabling items, 8-15
id, 8-9, 8-18
isPopUp(), 8-4
MenuItem() function, 8-3, 8-5, 8-7, 8-8, 8-9, 8-18
message, 8-9, 8-17
overview, 8-1
shortcut, 8-8, 8-9, 8-16
style, 8-16, 8-17

MenuItem(), 8-3, 8-5, 8-7, 8-8, 8-9, 8-18

MENUMODAL(), 8-17, 8-18

Menus
accelerator keys, 8-2, 8-7, 8-12
actions for, 8-8
activating, 8-17, 8-18
cascading, 8-2, 8-15, 8-17
colors for, 8-12
creating menu objects, 8-1
disabling items, 8-15
displaying, 8-17, 8-18
enabling items, 8-15
making selections, 8-2

MenuItem class, 8-1
parts of a, 8-2
PopUpMenu class, 8-1
separators, 8-13
shortcut keys, 8-2, 8-8, 8-9, 8-16
submenus, 8-2, 8-15, 8-17
toggles, 8-13, 8-14, 8-17
TopBarMenu class, 8-1
visually effective, 8-12, 8-13, 8-15, 8-16

Messages
arguments, 2-98
sending, 2-97, 2-98
syntax for sending, 2-98
use of parentheses, 2-98

Methods,definition, 2-98

Modes, *See also* Execution modes
Insert, 13-33
Overwrite, 13-33

MODIFY STRUCTURE
full-screen, 16-18

Module
errors,finding with MAP file, 11-33
names,referring to, 12-30

MODULE...FROM command, 11-14

Monitor Local, 13-82, 13-83

Monitor menu, 13-18, 13-51
Local option, 13-82, 13-83
Private option, 13-84
Public option, 13-85
Sort option, 13-37, 13-86
Static option, 13-87

Monitor Private, 13-84

Monitor Public, 13-85

Monitor Sort, 13-37, 13-86

Monitor Static, 13-87

Monitor window, *See* Windows

Multistatement command lines, 13-4

N

Nested READs, 7-9

NETERR(), 4-3

NetUse(), 4-3, 4-6

Network environment
@...GET, 4-7
APPEND BLANK, 4-10
APPEND FROM, 4-4
COPY STRUCTURE, 4-4
COPY STRUCTURE EXTENDED, 4-4
COPY TO, 4-4
COUNT, 4-8
CREATE, 4-4
CREATE FROM, 4-4
DBAPPEND(), 4-10
DELETE, 4-7
DISPLAY, 4-4
FOPEN(), 4-4
INDEX, 4-4
JOIN, 4-4
LABEL FORM, 4-4
LIST, 4-4, 4-8
MEMOREAD(), 4-4
PACK, 4-4
RECALL, 4-7
REINDEX, 4-4
REPLACE, 4-7
REPORT FORM, 4-4, 4-8
RESTORE, 4-4
SAVE, 4-4
SET ALTERNATE, 4-4
SET INDEX, 4-3
SET PRINTER, 4-4
SORT, 4-4
TEXT, 4-4
TO FILE, 4-4
TOTAL, 4-4
TYPE, 4-4
UPDATE, 4-4, 4-7
USE, 4-3, 4-4
ZAP, 4-4

Network programming, 4-3

Networking, 4-1
assigning rights, 3-18
hardware requirements, 3-17
runtime considerations, 3-17

NEXT, *See also* FOR

NG directory, 18-1, 18-2

NG.EXE, *See also* Norton Guides, 18-1
command line syntax, 18-2

NG.INI, 18-17

NIL
as a return value, 2-5, 2-7, 2-20
as an initializer, 2-6, 2-47, 2-77
assignment, 2-47
assignment to omitted parameters, 2-14, 2-19
comparison, 2-47, 2-52, 2-83
operations, 2-52

NOIDLE, 3-13

Norton Guides, *See also* Guide To CA-Clipper
Access window, 18-5, 18-6
accessing, 18-3
configuration, 18-15
Expand menu, 18-5, 18-6
getting help, 18-7
hot key, 18-3, 18-17, 18-18
Instant Access Engine, 18-1
leaving, 18-18
loading, 18-2
memory used by, 18-2
memory-resident mode, 18-2
menu bar, 18-5
menu bar navigation keys, 18-6
menu navigation, 18-6
navigation, 18-14
NG.INI, 18-17
operation, 18-4
Options menu, 18-5, 18-6
pass through mode, 18-2, 18-3
Search menu, 18-5, 18-9
syntax, 18-2
uninstalling, 18-18

Norton guides
navigation, 18-10

Not a DOS/16M executable error message, 11-33

Not enough memory error message, 11-38

Notes on compatibility, 11-35

Null
date, 2-44
string, 2-41
terminator, 2-40

Num, 13-88, *See also* Options Line

Numeric
comparison, 2-46
constants, 2-45
limitations, 2-45
literals, 2-45
operators, 2-51, 2-57
overflow in REPORT FORMs, 17-5
precedence of operators, 2-62, 2-64
precision, 2-45
range of values, 2-45
storage format, 2-45
table of operators, 2-51

O

OBJ directory, 11-11

Object files
default name, 10-2, 10-7, 10-9, 10-15
embedding library name in, 10-7
generating, 10-2, 10-12
reducing size of, 10-6
search paths, 11-11
searching for, 11-11
specifying a directory for, 10-7
specifying a name for, 10-7, 10-15
specifying name of, 12-27

Objects
as return values, 2-20, 2-97
creating new objects, 2-97
general discussion, 2-96
passing as parameters, 2-16, 2-18, 2-97
sending messages, 2-97, 2-98

Opening files
database, 2-101
in a network environment, 4-4, 4-12
in DBU, 16-16
in shared mode, 4-3
index, 2-110

Operating systems, 11-47

Operations
array, 2-75
code block, 2-89
database file, 2-106
index, 2-109

Operators
addition, 2-51
alias, 2-33, 2-60
array element, 2-60
array equivalence, 2-84
assign, 2-55, 2-77
binary, 2-49
character, 2-50, 2-57
code block, 2-60
code block arguments, 2-89
compile and run, 2-60, 2-66, 2-68, 2-80, 2-95
compound assign, 2-57
concatenation, 2-50
constant array, 2-60, 2-80
date, 2-51, 2-57, 2-58
decrement, 2-58
division, 2-51
equal, 2-52
exactly equal, 2-52
exponentiation, 2-51
general discussion, 2-40, 2-48
greater than, 2-52
greater than or equal, 2-52
grouping, 2-60
increment, 2-58
infix, 2-49
inline addition and assign, 2-57
inline assign, 2-56, 2-57, 2-77, 2-99
inline concatenation and assign, 2-57
inline division and assign, 2-57
inline exponentiation and assign, 2-57
inline modulus and assign, 2-57
inline multiplication and assign, 2-57
inline subtraction and assign, 2-57
less than, 2-52
less than or equal, 2-52
logical, 2-53
macro, 2-66
mathematical, 2-58
memo, 2-50
modulus, 2-51
multiple inline assign, 2-56
multiplication, 2-51
not equal, 2-52
numeric, 2-51, 2-57
overloading, 2-48, 2-55
pass by reference, 2-17
pass-by-reference, 2-60
postdecrement, 2-58
postfix, 2-49, 2-58
postincrement, 2-58
precedence of, 2-49, 2-51, 2-61, 2-62
predecrement, 2-58
prefix, 2-49, 2-58
preincrement, 2-58
relational, 2-52, 2-84
send, 2-97, 2-98, 2-99
special, 2-60
substring, 2-52
subtraction, 2-51
table of decrement, 2-58
table of increment, 2-58
table of special, 2-60
unary, 2-49

Options Codeblock, 13-89

Options Color, 13-90

Options Colors, 13-44

Options Exchange, 13-47, 13-52, 13-89, 13-91, 13-99

Options Line, 13-92, *See also* Num

Options Menu, 13-93

Options menu, 13-18
- Codeblock Trace option, 13-89
- Color option, 13-90
- Colors option, 13-44
- Exchange Screens option, 13-91, 13-105
- Line Numbers option, 13-92
- Menu Bar option, 13-93
- Mono Display option, 13-94
- Path for Files option, 13-95
- Preprocessed Code option, 13-96
- Restore Settings option, 13-97
- Save Settings option, 13-98
- Swap on Input option, 13-99
- Tab Width option, 13-100

Options Mono, 13-94

Options Path, 13-95

Options Preprocessed, 13-6, 13-96

Options Restore, 13-14, 13-97

Options Save, 13-14, 13-98

Options Swap, 13-99

Options Tab, 13-100

Organization,manual, 1-1

OS/2 Virtual DOS Machines, compatibility with CA-Clipper/Exospace, 11-49

OTHERWISE, *See also* DO CASE

Output executable file
- specifying name of, 12-34

Output to text file
- @...SAY, 2-119
- console commands, 2-119

Overlay
- area,specifying end of, 12-26
- manager,forcing display of errors, 12-32
- pool
  - adjusting opsize of, 12-19
  - default size of, 12-19
  - request size for, 12-19

Overlay pool,size of, 3-15

Overlaying code,automatically, 11-2

Overriding default linking sequence, 11-27

Overview of Blinker, 12-2

Overview of CA-Clipper/Exospace, 11-2

Overwrite mode, 13-33

P

PACK, 4-4

Packages
- compared with libraries, 11-21
- including in .EXE, 11-21
- protected-mode support, 11-21
- third-party libraries, 11-22

Pageframe
- for overlays, 12-20
- use of, 12-20

Paging system, support for, 12-2

Parameter
- declared, 2-14
- definition of actual, 2-14
- definition of formal, 2-14
- methods of specifying, 2-14
- number passed, 2-14, 2-19
- omitting, 2-14, 2-19
- optional, 2-14, 2-19
- passing, 2-14, 2-19
- passing arrays as, 2-16, 2-18, 2-81
- passing by reference, 2-17, 2-18
- passing by value, 2-16, 2-18
- passing code blocks as, 2-16
- passing fields as, 2-16
- passing from DOS command line, 2-20
- passing objects as, 2-16, 2-18, 2-97
- passing with DO...WITH, 2-17

PARAMETERS, 10-4

Parentheses, and precedence, 2-49, 2-65

Pass-by-reference, 2-17, 2-60

PCOL(), 2-117

PCOUNT()
effect of omitting parameters on, 2-19

PCOUNT(),effect of omitting parameters on, 2-14

PE.EXE, 15-1, *See also* Program editor

PMINFO command, 11-31, 11-32

PMINFO, defined, 11-34

Point Breakpoint, 13-51, 13-58, 13-59, 13-101, *See also* BP

Point Delete, 13-34, 13-54, 13-102, *See also* Delete

Point menu, 13-18
Breakpoint option, 13-101
Delete option, 13-54, 13-102
Tracepoint option, 13-54, 13-103
Watchpoint option, 13-54, 13-104

Point Tracepoint, 13-14, 13-34, 13-54, 13-103

Point Watchpoint, 13-14, 13-34, 13-54, 13-104

PopUp(), 8-3, 8-6

PopUpMenu class
addItem(), 8-4
colorSpec, 8-4, 8-10, 8-12
creating pull-down menu, 8-2
defining properties, 8-10
overview, 8-1
PopUp() function, 8-3, 8-6

Postdecrement
operators, 2-58
precedence of operators, 2-62, 2-65

Postincrement
operators, 2-58
precedence of operators, 2-62, 2-65

Precedence, 2-49, 2-61
and algebraic expression evaluation, 2-63

Predecrement
operators, 2-58
precedence of operators, 2-62, 2-63

Preincrement
operators, 2-58
precedence of operators, 2-62, 2-63

Preprocessed output,viewing, 13-6, 13-96

Preprocessor, 10-10
defining an identifier, 10-5
source listing file, 10-7, 10-16
switches, 10-4

Preprocessor directives, 10-8
#define, 2-8
#include, 2-8, 10-14
summary of, 10-10

Printing
@...SAY, 2-117
general discussion, 2-117
redirecting output to a file, 2-119
with console commands, 2-117

PRIVATE, 10-4

Problems
running CA-Clipper/Exospace applications, 11-31
when running under DPMI, 11-40

Procedure stack, setting maximum depth, 11-23

PROCEDURE, compiler implications, 10-8

Procedures
calling conventions, 2-5, 2-17
declaring, 2-7
defining, 2-7
executing, 2-5, 13-53, 13-64
files, compiling, 10-12
nesting,specifying maximum depth of, 12-23
passing parameters to, 2-14
startup, 2-4

PROCLINE(), 2-123

PROCNAME(), 2-123

Program
- comments, 2-8
- multistatement line, 2-9
- termination, 2-122

Program editor, 15-1
- architecture, 15-5
- make file, 15-5
- navigation, 15-3
- operation, 15-3
- source code, 15-5
- syntax, 15-2

Program files
- compiling, 10-2, 10-12
- linking, 11-4

Program libraries
- specifying names of, 12-28
- specifying names to, 12-28

Programming
- general discussion, 2-2
- techniques,text vs. graphic mode, 5-2

Protected mode
- programming restrictions
  - problems in third-party libraries, 11-41
- running in, 11-2
- support packages, 11-21
- switching to real mode, 11-42
- virtual memory engine, 11-2

Providing extended memory, 11-36

PROW(), 2-117

Pseudofunctions, 13-6

PUBLIC, 10-4

R

RAM, 11-2, 11-22

RAMDRIVE.SYS device driver, 11-47

Read layer, 7-17, 7-25, 7-27
- @...GET, 7-8
- creating, 7-19, 7-27
- implementation rules, 7-18, 7-27
- implementation steps, 7-19
- READMODAL, 7-8
- SET MESSAGE TO, 7-22
- STD.CH, 7-8

READ, exiting from a, 2-122

READEXIT(), 2-122

Reading DEFAULTS.LNK, 12-4

Reading OTHER.LNK, 12-4

READVAR(), 2-123

Real mode
- memory availability, 12-2
- memory optimizing usage, 12-2
- memory overcoming limit, 12-2
- switching to protected mode, 11-42

RecLock(), 4-10

RECNO(), 16-36

Record
- processing with DBEVAL(), 2-109
- scoping, 2-107, 2-109

Record pointer,and index updating, 2-112

Recursion, 2-34

REINDEX, 2-112, 4-4

Relational
- operators, 2-52, 2-84
- precedence of operators, 2-62, 2-64
- table of operators, 2-52

REPLACE, 2-55, 4-7
- and inline assignment, 2-57

Report and label utility, *See* RL

REPORT FORM, 4-4, 4-8
creating, 17-1
for related files, 17-12
printing, 17-11

Required library files for linking, 11-25

Requirements
available memory, 11-6
hardware, 11-42
memory, 11-43

RES2BML.COM, 5-11

Reserved words, 2-10, 2-47

RESTORE, 4-4

Return values, from a user-defined function, 2-20

RETURN, from a user-defined function, 2-20

RL
abandoning changes, 17-11, 17-18
creating a new label, 17-13
creating a new report, 17-2
inserting line breaks in reports, 17-4
Label Exit menu, 17-18
Label Formats menu, 17-16
Label Toggle menu, 17-14, 17-16
leaving, 17-19
make file, 14-27, 17-20
maximum number of report columns, 17-3
modifying an existing label, 17-13
modifying an existing report, 17-2
Report Columns menu, 17-3
Report Delete menu, 17-6
Report Exit menu, 17-11
Report Goto menu, 17-6
Report Groups menu, 17-9
Report Insert menu, 17-6
Report menu, 17-7
saving a label definition, 17-18
saving a report definition, 17-11
specifying a standard label format, 17-16
specifying blank label lines, 17-16
specifying fields from multiple files, 17-12
specifying label contents, 17-16
specifying report level options, 17-7

RL.EXE, 17-1, *See also* RL, 17-20
command line syntax, 17-2
source code, 17-20

RLOCK(), 4-9

RMAKE, See also Make file, 14-1
commands, 14-10
comments in, 14-5, 14-12, 14-19
conditional make, 14-19, 14-20, 14-21
configuration, 14-3
DBU make file, 16-41
debugging, 14-4
dependency rule, 14-6, 14-10, 14-12
dependency statement, 14-6
directives, 14-19
display warning messages, 14-5
DOS access from, 14-19
DOS return code, 14-5, 14-6, 14-7
DOS SETs, 14-17
error handling, 14-5, 14-7
examples, 14-12, 14-15, 14-17, 14-23, 14-25, 14-27
header files, 14-21
inference rule, 14-10, 14-15
input files, 14-8
internal workspace, 14-5
macro definition, 14-23
macro expansion, 14-23
macros, 14-2, 14-3, 14-4, 14-8, 14-10, 14-11, 14-22
make phase, 14-6, 14-8, 14-15
parsing phase, 14-6, 14-8, 14-19
PE make file, 15-5
predefined macros, 14-15, 14-25
RL make file, 14-27, 17-20
special macros, 14-8, 14-12, 14-21, 14-25
switches, 14-2, 14-4
symbol table, 14-5
user-defined macros, 14-23

RMAKE directives

#!, 14-19
#else, 14-19, 14-20, 14-21
#endif, 14-19, 14-20, 14-21
#ifdef, 14-19
#ifeq, 14-20
#iffile, 14-20
#ifndef, 14-21
#include, 14-21
#stderr, 14-21
#stdout, 14-21
#undef, 14-22

RMAKE options, See RMAKE switches

RMAKE switches
/B, 14-4
/D, 14-3, 14-4, 14-8, 14-23
/F, 14-4, 14-7, 14-12
/I, 14-5, 14-15
/N, 14-5
/U, 14-5, 14-12, 14-19
/W, 14-5
/XS, 14-5
/XW, 14-5

RMAKE.EXE, See also RMAKE
command line syntax, 14-2

Routine errors, finding with MAP file, 11-33

Run Animate, 13-47, 13-51, 13-52, 13-105

Run Go, 13-48, 13-50, 13-51, 13-106

Run menu, 13-18
Animate option, 13-105
Go option, 13-106
Next option, 13-107
Restart option, 13-108
Speed option, 13-109
Step option, 13-110
To Cursor option, 13-111
Trace option, 13-112

Run Next, 13-48, 13-107

Run Restart, 13-108

Run Speed, 13-47, 13-51, 13-105, 13-109

Run Step, 13-110, 13-112

Run To, 13-48, 13-111

Run Trace, 13-112

RUN, runtime implications, 3-5

Running
an application using CA-Clipper/Exospace, 11-41
in protected mode, 11-2

Runtime
configuration, 3-7, 3-8, 11-16, 11-17, 11-19
configuring number of files at, 3-12
controlling extended cursor, 3-10
displaying memory configuration at, 3-12
environment settings, 3-7, 3-8, 11-16, 11-17, 11-19
excluding available memory at, 3-15
general discussion of environment, 3-1
list of parameters, 12-12
preventing idle time detection at, 3-13
restoring EMM page frame, 3-10
specifying dynamic overlay file handles, 3-10
specifying swap file location at, 3-14
specifying swap file size at, 3-14
specifying temporary file location at, 3-14

S

SAVE, 4-4

Scope, syntax for specifying, 2-107

Script files, 13-7, 13-13, 13-14, 13-21, 13-44, 13-97, 13-98
- comments, 11-4
- compiling with, 10-2, 10-9, 10-12
- default extension, 11-4
- defaulting to a .LNK extension, 12-4
- DEFAULTS.LNK, 12-4
- indicating text as comment, 12-4, 12-5
- linking with, 11-4
- nesting, 11-4
- OTHER.LNK, 12-4
- searching for, 11-11
- specifying name of, 12-5
- specifying on the command line, 12-4

Search path, 13-13

SEEK, 2-111

SELECT, 2-101

Selector
- definition, 2-98
- instance variable, 2-99
- message, 2-98

Separators, 8-13

SET ALTERNATE, 2-115, 2-119, 4-4

SET CENTURY, 2-44

Set Colors window, *See* Windows

SET CONSOLE, 2-115, 2-117

SET DATE, 2-43

SET DEVICE, 2-116, 2-117

SET EPOCH, 2-43

SET ESCAPE, 2-122

SET EXACT, 2-42

SET EXCLUSIVE, 4-3, 4-4

SET FORMAT, compiler implications, 10-12

SET FUNCTION, 2-123

SET INDEX, 2-110, 2-111, 4-3

SET KEY, 2-122, 2-123, 7-10

SET MARGIN, 2-117

SET ORDER, 2-111

SET PRINTER, 2-115, 2-117, 2-119, 4-4

SET RELATION, 16-36
- and REPORT FORMs, 17-12

SET SOFTSEEK, 2-111

SET TYPEAHEAD, 2-120

SETCANCEL(), 2-122

SETCOLOR(), 2-117, 8-12

SETPRC(), 2-117

Setting
- maximum depth of CA-Clipper procedure stack, 11-23
- parameters for Virtual DOS Machine, 11-49
- read-only status, 12-35

Shared mode, 4-1, 4-3, 4-4, 4-7, 4-9

Sharing extended memory, 11-35

Shortcut keys, 8-2, 8-8, 8-9, 8-16

SMARTDRIVE.SYS device driver, 11-47

Software
- compatibility, 11-35, 11-42
- incompatibility, 11-32

Solutions to CA-Clipper/Exospace application problems, 11-31

SORT, 4-4

Sorting arrays, 2-87

SOURCE directory, 15-5

Source files
compiling, 10-12
editing, 15-1
line numbers, 10-6, 10-7

Specifying
.LIB, 11-2
.OBJ, 11-2
CA-Clipper, 5.01 or 5.01a, 11-15
case sensitivity, 12-29
conversion of symbols to uppercase, 12-38
default
.EXE options, 11-18
drive of temporary workfile, 12-39
end of overlay area, 12-26
executable file to be generated, 11-29
file name of temporary workfile, 12-39
files to be in external overlays, 12-36
name of object files, 12-27
name of output executable file, 12-34
object files to be linked, 11-24
path of temporary workfile, 12-39
required library files for linking, 11-25
setting to read-only status, 12-35
stack size of application, 11-30
stack size of executable files, 12-37
suppressing of beep, 12-32
symbols to be excluded, 12-24
XMS memory use, 12-15

Specifying the Location of Executable Files, 3-4

Stack size
of application, 11-30
of executable files, 12-37

Standard header file,changing, 10-8

Startup procedure, 10-6

Status information
displaying, 12-38
during linking, 12-38

STD.CH, 10-8

Storage classes,defined, 2-27

STORE, 2-55, 2-56

Submenus, 8-2, 8-15, 8-17

Substring operator, 2-52

SUM, 4-8

SWAPK, 3-14

SWAPPATH, 3-14

Symbol table, compression of, 12-2

Symbols
converting to uppercase, 12-38
prioritizing of, 12-35
specifying symbols to be excluded, 12-24

System
hanging, 11-34
viruses, 11-33

T

Target file, 14-6, 14-12
creating, 14-8
searching for, 14-8

TBColumn class
block, 6-13, 6-20
cargo, 6-12
colorBlock, 6-20, 6-23
defColor, 6-19, 6-20, 6-23

TBrowse
adding Color, 6-18
browsing search results, 6-30
browsing with Get, 6-13
Controlling highlight, 6-22
controlling scope, 6-24
Creating TBrowse Objects, 6-3
DBEVAL(), 6-30
multi-user applications, 6-9
obtaining quicker response time, 6-8
Optimization, 6-7
repositioning record pointer, 6-10

SET COLOR, 6-18
SET FILTER, 6-30
using TBColumn
cargo, 6-12
using TBrowse
cargo, 6-12
using TRANSFORM(), 6-7
viewing a key value, 6-26

TBrowse class, 6-3
applyKey() method, 6-6
autoLite, 6-22
cargo, 6-12
colorRect, 6-23
colorSpec, 6-18
deHilite, 6-22
forceStable() method, 6-5
goBottomBlock, 6-24, 6-26, 6-30
goTopBlock, 6-24, 6-26, 6-30
hilite, 6-22
refreshAll() method, 6-10
refreshCurrent() method, 6-9
setKey() method, 6-6
skipBlock, 6-24, 6-26, 6-27, 6-30
stabilization, 6-5
stabilize() method, 6-5, 6-8
up() method, 6-10

TBrowseDB(), 6-3

TBrowseNEW(), 6-3

Temporary files
specifying a directory for, 3-5, 10-7, 10-15, 11-9

TEMPPATH, 3-14

TERMINAL.LIB, 11-25, 11-28

TEXT, 4-4

Text
mode
programming techniques, 5-3

Text files
editing, 15-1

Text mode
input/output system, 2-113

Third-party
compatibility, 11-41
library
compatibility, 11-41
packages, 11-22
violations and solutions, 11-41
programming restrictions in protected mode, 11-41

Threshold size,enabling, 12-21

TMP directory, 3-5, 10-7, 10-15, 11-9

TO FILE clause, 2-115, 2-119

TO PRINTER clause, 2-115, 2-117

Toggles, 8-13, 8-14, 8-17

TopBar(), 8-3, 8-5

TopBarMenu class
activating objects, 8-17, 8-18
addItem(), 8-4, 8-5, 8-6
colorSpec, 8-4, 8-10, 8-12
creating menu bar, 8-2
defining properties, 8-10
display(), 8-4
displaying objects, 8-17, 8-18
left, 8-3
overview, 8-1
right, 8-3
row, 8-3
TopBar() function, 8-3, 8-5

TOTAL, 4-4

Tracepoints, 13-34, 13-47, 13-51, 13-53, 13-106
and monitored variables, 13-82, 13-83, 13-84, 13-85, 13-87
definition, 13-53, 13-103
deleting, 13-54, 13-69, 13-102
listing, 13-55, 13-76
setting, 13-14, 13-54, 13-103

TRANSFORM(), 6-7
in REPORT FORMs, 17-4

TSR programs, 3-10

TYPE, 4-4

TYPE(),and arrays, 2-75

U

UNIVESA.EXE, 5-17

UNLOCK, 4-11

UPDATE, 4-4, 4-7
- and inline assignment, 2-57

Upgrade offer, 12-3

USE, 2-101, 2-110, 2-111, 4-3, 4-4

User-defined commands, 2-89, 2-109

User-defined functions
- calling conventions, 2-5
- declaring, 2-7
- executing, 2-5
- passing arguments to, 2-14, 2-81

Using high memory area, 11-38

V

VALTYPE()
- and arrays, 2-75
- used to test for omitted parameters, 2-19

Variable names
- changing of, 3-15
- resolving references, 2-32

Variables
- ambiguous references to, 2-32
- changing values of, 13-34, 13-37, 13-55
- creating new, 13-56, 13-64
- creation, 2-29
- declaration, 2-6, 2-27, 2-33, 2-47
- definition of, 2-27
- displaying, 13-64
- dynamically scoped, 2-27
- field, 2-38
- initialization, 2-33, 2-47
- inspecting, 13-51, 13-55
- lexically scoped, 2-6, 2-27
- lifetime of, 2-29
- local, 2-34
- monitored, 13-37, 13-51, 13-82, 13-83, 13-84, 13-85, 13-86, 13-87
- naming, 2-31
- passing as parameters, 2-16
- private, 2-36
- public, 2-37
- scoping in code blocks, 2-30, 2-90
- scoping of, 2-6, 2-27, 2-30
- static, 2-35
- undefined, 2-47
- visibility of, 2-29

VCPI extended memory protocol, 11-44

VDISK.SYS device driver, 11-47

Version number
- representing, 12-40
- return as integer, 12-40

View, 13-112, *See also* File Open, 16-13, 16-17, 16-24, 16-35
- definition, 16-3, 16-9

View App, 13-52, 13-114

View Callstack, 13-39, 13-60, 13-115, *See also* Callstack

View menu, 13-18
- App Screen option, 13-114
- Callstack option, 13-115
- Sets option, 13-42, 13-116
- Workareas option, 13-43, 13-117

View Sets, 13-42, 13-116

View Sets window, *See* Windows

View Workareas, 13-43, 13-117

View Workareas window, *See* Windows

Viewing
- header files, 13-5, 13-61, 13-72, 13-112
- preprocessed output, 13-6, 13-96
- program code, 13-61, 13-72, 13-73, 13-105, 13-112
- program output, 13-52, 13-91, 13-114

Virtual
DOS Machine, setting parameters for, 11-49
memory manager, 3-10, 3-12, 3-14

Virtual Memory Manager, 11-2

Virus, in system, 11-33

VM package
disabling, 11-22
excluding, 11-22

VMM, 3-10, 3-12, 3-14

W

Wait states, keyboard buffer, 2-120

Watch window, *See* Windows

Watchpoints, 13-34, 13-51, 13-53
and monitored variables, 13-82, 13-83, 13-84, 13-85, 13-87
definition, 13-53, 13-104
deleting, 13-54, 13-69, 13-102
listing, 13-55, 13-76
setting, 13-14, 13-54, 13-104

WHILE condition, 2-107

Window Iconize, 13-118

Window menu, 13-18
Iconize option, 13-118
Move option, 13-119
Next option, 13-119
Prev option, 13-120
Size option, 13-120
Tile option, 13-121
Zoom option, 13-122

Window Move, 13-119

Window Next, 13-119

Window Prev, 13-120

Window Size, 13-120

Window Tile, 13-121

Window Zoom, 13-122

Windows
activating, 13-24
closing, 13-25
environment, 11-47
iconizing, 13-26, 13-118
moving, 13-28, 13-119
navigating between, 13-24, 13-119, 13-120
opening, 13-25
sizing, 13-27, 13-120
the Callstack window, 13-24, 13-37, 13-39, 13-60, 13-68, 13-115
the Code window, 13-6, 13-24, 13-29, 13-37, 13-61, 13-68, 13-82, 13-83, 13-84, 13-85, 13-87, 13-88, 13-91, 13-92, 13-94, 13-96, 13-100
the Command window, 13-24, 13-31, 13-53, 13-63
the Help window, 13-16, 13-41, 13-74
the Monitor window, 13-24, 13-37, 13-45, 13-82, 13-83, 13-84, 13-85, 13-86, 13-87
the Set Colors window, 13-44, 13-90
the View Sets window, 13-42, 13-116
the View Workareas window, 13-43, 13-117
the Watch window, 13-24, 13-34, 13-45, 13-51, 13-54, 13-55, 13-69, 13-76, 13-103, 13-104
tiling, 13-28, 13-121
zooming, 13-26, 13-122

Work area
attributes in DBU, 16-9
attributes of, 2-104
changing, 2-101
default aliases, 2-101
in DBU, 16-9
next available, 2-101
number available, 2-101
number available in DBU, 16-9
table of attributes, 2-104

Workstation requirements, 4-3

X

X, 3-15

XMS
- enabling use of, 12-22
- extended memory protocol, 11-44
- maximum amount of, 12-8
- minimum amount of, 12-8
- specifying memory use, 12-15

Z

ZAP, 4-4

Printed in the U.S.A.
US0595EN

COMPUTER ASSOCIATES®
Software superior by design.

00-55-110-3501