CA-Clipper®

For DOS

Version 5.3

Getting Started Guide

June 1995

COMPUTER ASSOCIATES®
Software superior by design.

© Copyright 1995 Computer Associates International, Inc.
One Computer Associates Plaza, Islandia, NY 11788. 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

Welcome to CA-Clipper! 1–1
What's New in CA-Clipper? 1–2
    Version 5.3 Features 1–2
In This Guide 1–6
    Installation 1–6
    The Benefits of Using CA-Clipper 1–6
    Learning the Basics 1–7
    Debugging a Simple Program 1–7
    Creating Data Structures 1–7
    DOS Online Documentation: The Guide To CA-Clipper 1–7
    Where to Go from Here 1–8

Chapter 2 : Installation

In This Chapter 2–1
System Requirements 2–2
Installation Procedures 2–3
    Installing CA-Clipper from Windows 2–3
        The Windows CA-Installer 2–4
        Full Installation 2–5
        Partial Installation 2–5
    Installing CA-Clipper from DOS 2–8
    The DOS CA-Installer 2–8
        Full Installation or Partial Installation 2–9
The CA-Clipper Development Environment 2–11
    The Directory Structure 2–11
    Environment Variables 2–13

Before Moving On 2–14
Starting the CA-Clipper Workbench 2–17
Summary 2–17

Chapter 3 : Making the Transition to CA-Clipper

In This Chapter 3–1
Requirements for This Chapter 3–2
Compile, Link, and Go 3–2
  Runtime Issues 3–2
  Compiling 3–3
  Linking 3–4
  Using RMAKE 3–4
Immediate Benefits 3–5
  LIM 4.0 Compliance 3–5
  RSIS Compliance 3–5
  Virtual Memory Management 3–6
  Graphic Mode 3–6
Source Code Changes to Consider 3–7
  Use Manifest Constants 3–7
  Use Pseudofunctions 3–10
  Use Header Files 3–12
  Use Lexically Scoped Variables 3–13
    Replace Privates with Locals 3–15
    Use Statics Instead of SAVE and RESTORE 3–17
    Replace Publics with Filewide Statics 3–18
  Declare Everything 3–19
    Use FIELD and MEMVAR Declarations 3–19
    Use New Compiler Options 3–20
  Use Lexically Scoped Functions and Procedures 3–20
  Replace Macros with Code Blocks 3–24
  Update Array Usage 3–26
An Overview of the Workbench 3–29
The Repository 3–29

The Workbench Tools 3–31
The Browsers 3–33
Application Browser 3–34
Module Browser 3–35
Entity Browser 3–36
Error Browser 3–39
The Editors 3–40
Source Code Editor 3–41
Data Server Editors 3–42
Form Editor 3–43
Menu Editor 3–49
The Debugger 3–51
Summary 3–52

Chapter 4 : Learning the Basics

In This Chapter 4–1
LESSON 1: Creating an Application 4–2
LESSON 2: Adding a Menu Structure 4–4
Creating a Menu Entity 4–5
Naming the Menu Structure 4–7
Creating Menus and Defining Menu Items 4–9
Adding Menu Items to the File Menu 4–13
Adding Menu Items to the Edit Menu 4–15
Alternate Method of Adding Menu Items 4–19
Previewing Your Menus 4–21
LESSON 3: Creating a Data Server and Field Specs 4–23
Defining a Data Server 4–23
Defining Field Properties 4–26

LESSON 4: Using the Form Editor 4–29
Creating a Form 4–29
Specifying Form Properties 4–31
Using Auto Layout to Place Controls 4–32
Specifying Control Properties 4–34
Changing Control Types 4–35
Creating a List Box 4–35
Customizing a List Box 4–37
Creating a Radio Button Group 4–40
Customizing a Radio Button Group 4–41
Adding Push Buttons to a Form 4–43
Specifying Control Order 4–46
LESSON 5: Importing an External File 4–49
Importing DC_UTILS.PRG 4–49
Using an Entity Browser to View DC_UTILS 4–51
LESSON 6: Using the Source Code Editor 4–53
Modifying the DC_MENUTest() Function 4–54
Modifying the DiskDispTest() Function 4–56
Modifying the DiskDispAppend() Function 4–57
LESSON 7: Compiler and Linker Options 4–60
Setting Compiler Options at the Application Level 4–60
Setting Linker Options at the Application Level 4–62
LESSON 8: Building and Executing DISKCAT 4–64
Building an Application 4–64
Executing an Application 4–65
Summary 4–67

Chapter 5 : Debugging a Simple Program

In This Chapter 5–1
Requirements for This Chapter 5–2
Creating a Program with Errors 5–3
Preparing Your Program for Execution 5–4
Executing Your Program Using the Debugger 5–5
  Isolating the First Error 5–7
  Correcting the First Error 5–8
Starting the Next Debugger Session 5–9
  Isolating the Next Error 5–9
  Setting Watchpoints 5–11
  Using Single Step Mode 5–12
  Implementing a Temporary Fix 5–13
  Continuing Program Execution 5–13
  Viewing Your Program Results 5–14
  Correcting the Final Error 5–14
Running the Final Test 5–15
Summary 5–16

Chapter 6 : Creating Data Structures

In This Chapter 6–1
Requirements for This Chapter 6–2
Executing DBU 6–2
Navigation 6–3
Creating a Database File 6–5
Creating an Index File 6–9
Adding Data 6–12
Editing Data 6–15
Leaving DBU 6–16
Summary 6–16

Chapter 7 : DOS Online Documentation:The Guide To CA-Clipper

In This Chapter 7–1
Requirements for This Chapter 7–2
What Is The Guide To CA-Clipper? 7–2
Loading The Guide To CA-Clipper 7–3
Navigation 7–4
Finding Information 7–5
Selecting a New Documentation Database 7–8
Leaving The Guide To CA-Clipper 7–9
Summary 7–9

Chapter 8 : Where to Go from Here

Manual Organization 8–1
Development Tools 8–4

Chapter 1
Introduction

Welcome to CA-Clipper!

Welcome to CA-Clipper 5.3, the development system for database applications. This is a vastly enhanced release of the popular CA-Clipper compiler, offering new opportunities and technology to programmers of all levels and backgrounds.

Building on the foundation of an open architecture and a powerful language, CA-Clipper furthers these frontiers with exciting, new technology:

- New predefined classes, as well as new Get and TBrowse class enhancements
- A Microsoft Windows-based Workbench
- Graphic mode support
- Native protected mode support
- Native mouse support
- An upgrade to Microsoft C 8.0

This guide is designed to give you all the information you need for the installation of CA-Clipper, as well as to introduce you to some of its new features.

Note: The README file installed in the \CLIP53 directory contains important last minute information. This file can be reviewed using the DOS TYPE command or any text editor.

What's New in CA-Clipper?

In addition to the many powerful commands and functions that have been added to the language, CA-Clipper 5.3 provides several language enhancements and improvements to the development environment.

Version 5.3 Features

The following features are new to CA-Clipper:

- **A Windows-Based Workbench**

  The CA-Clipper Workbench provides tools that enable you to create DOS applications in a Microsoft Windows-based integrated development environment (IDE). Graphical editors allow you to *visually* design data entry screens, or *forms*, and menus using point-and-click and drag-and-drop techniques. You can also create data servers and field specifications for each of your applications. These tools let you see the result of your design as it progresses.

  There are also browsers, a debugger, and a built-in repository where the Workbench stores application components and entities. The browsers allow you to facilitate project management, and the integrated debugger lets you debug your CA-Clipper DOS applications from within the Workbench. The Workbench debugger supports a wide variety of standard features, such as single stepping, stepping to the cursor, setting and removing breakpoints and watchpoints, viewing the call stack, viewing work areas, etc.

  The repository is seamlessly integrated into the Workbench, and includes an automatic build and make facility. Dependency information is then used internally to manage the application for you. When possible, it automatically rebuilds only those parts of the current application that have been changed—or are affected by changes—since the application's last build and then does any necessary recompilation and relinking. Note that all compile and link options are configurable.

The Workbench also contains an import and export facility that allows you to export an entire application—including all entities in its Binary Object module and its source code—to a single external file (with a default extension of .cef). Later, you may import that application back into the Workbench to refine it or continue its development.

- **Graphic Mode Support**

  CA-Clipper now has the ability to display output to the screen in graphic mode, as well as in text mode. You can control what is displayed using pixel coordinates, rather than row and column coordinates as in text mode. Bitmaps can be displayed and smooth lines drawn, resulting in more elegant looking applications. CA-Clipper provides the Light Lib Graphics API for graphic mode support.

- **Native Protected Mode Support**

  CA-Clipper/Exospace has been integrated with the base CA-Clipper product. Now you can create and run CA-Clipper applications in *both* protected and real mode. Using protected mode eliminates the DOS 640 KB barrier, and allows your applications to access extended memory directly without requiring swapping of any kind for mouse support.

- **Enhanced Protected Mode Linker**

  In CA-Clipper 5.3, the default linker, CA-Clipper/Exospace, provides a selection of optional built-in packages that provide protected-mode support for low-level operations, such as mouse drivers, direct screen I/O, and network calls.

- **Optional Real Mode Linker**

  CA-Clipper 5.3 provides an optional real mode linker—BLINKER.EXE, a standard command line-driven linker—that supports a number of command line options and environment variables.

- **Native Mouse Support**

  CA-Clipper now has mouse support that is fully integrated with its GET/READ system, which enables all of the forms and menus you create to be mouse-enabled. Additionally, mouse support has been added to the debugger. Other enhancements include event detection for the INKEY() function, manipulation functions, and low-level API for mouse support.

- **New Classes and Objects**

  There are several new predefined classes now supported in CA-Clipper. These include the following: CheckBox class, ListBox class, PushButton class, RadioButto class, RadioGroup class, MenuItem class, TopBarMenu class, PopupMenu class, ScrollBar class, and TBColumn class.

  In addition to such popular objects as TBrowse and Get, there are now many new, reusable menu and form objects. These include check boxes, list boxes, radio buttons and radio button groups, push buttons, scroll bars, top bar menus, pop-up menus, and menu items. Additionally, there is a new column object for defining a data column of a TBrowse object.

- **Get Class Enhancements**

  The GET/READ system has been enhanced to make it aware of the new form and menu objects. Mouse support, accelerator key support, status bar messages, and top bar menu integration are now available.

  New GETs include the following: @...GET CHECKBOX, @...GET LISTBOX, @...GET PUSHBUTTON, @...GET RADIOGROUP, and @...GET TBROWSE.

  New Get instance variables include: caption, capRow, capCol, and message. Also, the colorSpec variable has been modified. Finally, the Get:hitTest() method has been added for mouse support as well.

- **TBrowse Enhancements**

  New TBrowse instance variables include the following: border, mRowPos, and mColPos. Also, the colorSpec variable has been modified. Finally, four new methods have been added: hitTest(), setKey(), applyKey(), and setStyle().

- **Upgrade of Microsoft C**

  CA-Clipper has been upgraded to compile under Microsoft C 8.0 (Microsoft Visual C/C++ 1.0). This upgrade entails a switch from alternate to emulator floating point support, enabling CA-Clipper to utilize a math coprocessor, if present.

  Note that these changes require you to do the following:

  - Any routines written in C need to be recompiled with the Microsoft C 8.x (Microsoft Visual C/C++ 1.x) compiler
  - Replace the /FPa C compiler command line argument with /FPi
  - Any references to LLIBCA.LIB in your make/link scripts need to be changed to LLIBCE.LIB

- **New Replaceable Database Drivers (RDD)**

  Unless otherwise specified, the enhanced DBFNTX driver is automatically linked into your application when you use the CA-Clipper linker. Other RDDs, such as DBFCDX, have also been enhanced.

  CA-Clipper also provides two new RDDs, DBFMEMO and DBFBLOB, for compatibility with FoxPro 2.

In This Guide

This guide, *Getting Started,* is designed to take you the first few steps into CA-Clipper. It contains installation instructions, describes the CA-Clipper development environment and its basic set of programming tools and documentation, and walks you through several tutorials in order to familiarize you with the Workbench and CA-Clipper utilities.

Installation

Chapter 2 gives complete instructions on how to install the CA-Clipper development system on your computer. CA-Clipper is much more environment intensive than its predecessors and, therefore, is supplied with an installation program. In addition, the compiler and linker take advantage of directory structures for organizing files by project and type. This chapter describes the CA-Clipper development environment and how the install program works. It then instructs you how to begin the installation process.

It is important that you read this chapter whether you are a seasoned CA-Clipper developer or a new user.

The Benefits of Using CA-Clipper

Chapter 3 gives you ideas on how to make the most of your time during the development process while maximizing the end-user potential of your applications. It focuses on high-level benefits and then goes on to discuss some of the newer features in the system, including the CA-Clipper Workbench, that will help you achieve those benefits. Whether you are using CA-Clipper for the first time or are a seasoned developer, the information in this chapter will help you in making decisions regarding new program development.

Learning the Basics

Chapter 4 walks you through the process of creating an executable program using the CA-Clipper Workbench. If you have never used a graphical user interface (GUI) before, you should read this chapter.

Debugging a Simple Program

Chapter 5 walks you through the process of debugging a simple program using the CA-Clipper DOS-level debugger, CLD.LIB. Several of the most used debugger features are discussed. You should read this chapter if you are not already familiar with using a source code debugger.

Creating Data Structures

Chapter 6 introduces you to DBU, the CA-Clipper database utility. DBU is one of the many utilities provided as part of the CA-Clipper development system. It is written in CA-Clipper and provides you with an interactive database design environment. For a general overview of DBU, read this chapter.

DOS Online Documentation: The Guide To CA-Clipper

Chapter 7 gives you a first tour through *The Guide To CA-Clipper,* the DOS-level online documentation system for CA-Clipper. This system consists of the popular Instant Access Engine and several databases containing most of the CA-Clipper documentation. To learn how to use the online documentation and see what is available to you, read this chapter.

Where to Go from Here

Chapter 8 is a guide to and overview of CA-Clipper programming tools and documentation. It can be used as the jumping-off point from the *Getting Started* guide into the CA-Clipper development system, or as a point of reference when you do not know where else to look.

Chapter 2
Installation

In This Chapter

This chapter discusses requirements and procedures for installing and starting CA-Clipper. Whether you're a new or experienced CA-Clipper user, the installation procedure is a required step for copying and organizing the necessary files on your hard disk. The files on the distribution disks are compressed into self-extracting archive files that cannot be used and/or executed otherwise.

To get you up and running with the CA-Clipper development system, this chapter leads you through the following steps:

- Determining if your computer meets the system requirements for the CA-Clipper development system (memory and disk space requirements)
- Making a backup copy of your distribution disks
- Running the installation program
- Learning about the environment created by the install program and how to operate within it

System Requirements

The minimum and recommended system requirements needed to install and run CA-Clipper are as follows:

Component	Minimum	Recommended
DOS	Version 3.1	Version 6.x
MS-Windows	Version 3.1	Version 3.11
Hard disk space	3 MB	20 MB
RAM	640 KB	16 MB

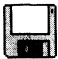
Note: The recommended hard disk space is the amount required for the full installation of CA-Clipper—partial installations will require less space. The actual amount of disk space required will be indicated by the CA-Installer when you install the product. These requirements refer only to the CA-Clipper development system and do not reflect the requirements for your CA-Clipper-compiled and -linked programs. The actual requirements for your application programs depend almost entirely on the architecture you build and the code and data resources you use.

Tip: Before beginning the installation process, it is important to make backup copies of your distribution disks. You can use the DOS DISKCOPY utility, the DOS COPY command, or another preferable backup utility. After you have successfully installed CA-Clipper onto your hard disk, store your original and your backup disks in *safe* and *separate* locations.

Installation Procedures

If your system meets the requirements, you are ready to begin the installation procedure. Note that CA-Clipper 5.3 may be installed either from within MS-Windows or from DOS.

Installing CA-Clipper from Windows

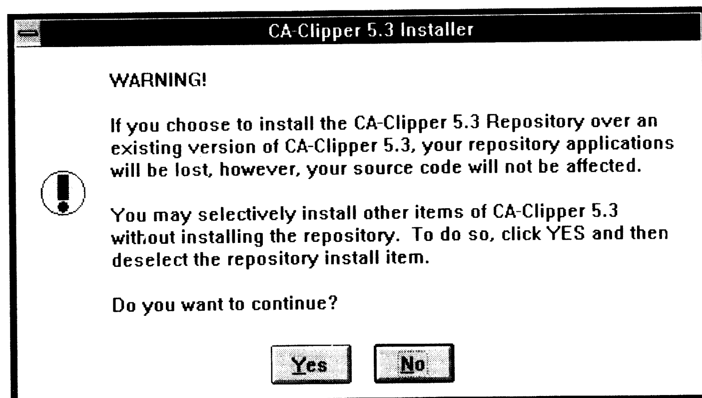


To install CA-Clipper on your hard drive from within MS-Windows, use INSTALL.EXE, on disk 1:

1. Insert the installation disk in drive A (or B).
2. From the Windows Program Manager, select the File Run command.
3. Type **A:\INSTALL** (or **B:\INSTALL**) and press Enter.

You will then be prompted to enter the directory in which you want to install CA-Clipper 5.3. The default directory is C:\CLIP53.

Note that if you have a previous version of CA-Clipper 5.3 installed, you will receive the following warning message:

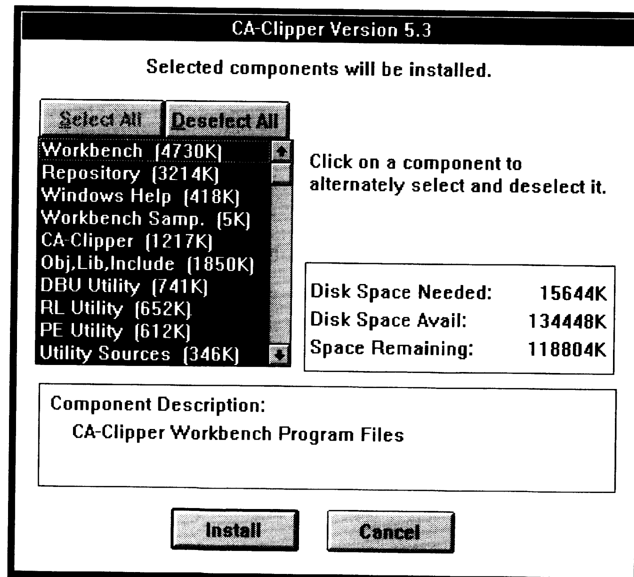


Important! *If you have a previous version of Clipper 5.3 installed, all data in the forms, menus, and data servers will be lost if you choose to install a new repository over your existing one. These items are stored in the "Binary Objects" module, and installing a new repository will overwrite this information. In addition, the dependencies between different program (.prg) and header files will be lost. Your source code itself, however, will not be affected. To prevent losing this data,* ***deselect*** *the Repository option. For more information about the Repository option, see the Partial Installation section.*

Tip: If your repository becomes corrupted, it is not always necessary to re-install it. Instead, you can run a special utility entitled "CA-Clipper Repository Reindex Utility" (CACIRIDX.EXE) and use it to rebuild the necessary indices.

The Windows CA-Installer

If you are installing from within MS-Windows, this version of the CA-Installer appears:



The CA-Installer is self-prompting, with online instructions which lead you through each step.

From here, you have two options. You can either install all of the components that are part of the CA-Clipper package (i.e., the full installation) or select only those components you know you are going to need (i.e., a partial installation).

Full Installation

By default, all options are selected, so for a *new*, full installation, simply choose the Install push button. The CA-Installer will begin the installation procedure, prompting you for any information necessary. Follow the onscreen prompts to proceed with the installation; CA-Clipper will be installed in the drive and directory you specify.

Partial Installation

If you choose to selectively install CA-Clipper, you should first click on the Deselect All button and then proceed to click on only those components that you wish to install, or you can leave everything selected and click on only those components that you do not wish to include. Either way, before you choose Install, the components that you want to install will be highlighted in the list box.

The various components are described below to help you decide what you need to install. Note that if you choose not to install a particular component at this time and later find that you need it, you can always run the CA-Installer again and select only that component, without affecting the rest of your current installation.

Minimum Installation

The minimum installation for *CA-Clipper* must include the following options: CA-Clipper; Obj, Lib, Include; and System Files. This will install all the system-defined libraries as well as other components that are common to all DOS applications.

For the CA-Clipper *Workbench*, the first two options are basic requirements—Workbench and Repository. The rest are arbitrary.

Workbench

This option installs all of the necessary files for the Workbench, the CA-Clipper interactive development environment. If you plan to use the interactive development environment, you must select this option.

Repository

The *repository* is where the CA-Clipper Workbench stores all application components and entities. It automatically manages the relationships between the various components, including the dependencies between different program (.prg) and header files for each application. The Repository option installs a new, *empty* repository that is used by the Workbench.

Windows Help

This option installs the Windows-style online help system associated with the CA-Clipper Workbench. If you plan to use context-sensitive help, you must choose this option.

Workbench Samp.

This option installs the sample files that are used in the various hands-on lessons in the printed documentation for the CA-Clipper Workbench. Choose this option if you plan to follow through the Workbench documentation examples.

CA-Clipper

The CA-Clipper option installs the CA-Clipper compiler and all necessary files for building applications. This option *must* be installed in order to create any CA-Clipper application.

Obj, Lib, Include Files

This option installs other files used by CA-Clipper including library and header files.

DBU Utility

This option installs the CA-Clipper database utility, DBU.EXE, which 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.

RL Utility

This option installs the CA-Clipper report and label utility, RL.EXE, which gives you the ability to create and modify standard dBASE III PLUS report and label definitions and use them in your applications.

PE Utility

This option installs a DOS-level program editor, PE.EXE, that gives you the ability to create and modify source code and header files. (Note that the CA-Clipper Workbench has its own source code editor.)

Utility Sources

This option installs the source code used by the database (DBU), report and label (RL), and program editor (PE) utilities.

System Files

This option installs files used by various subsystems in CA-Clipper, such as the Get, Menu, Error, and TBrowse systems.

Samples

This option installs the sample files that are used in the various hands-on lessons in the printed documentation. For example, the tutorial in this guide for debugging a simple program using the DOS-level debugger, CLD.LIB, assumes you have installed the sample files. Choose this option if you plan to follow through the DOS-level documentation examples.

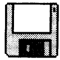
DOS Help

This option installs the Norton Guide Instant Access Engine and the DOS-level online help databases, *The Guide To CA-Clipper*.

Moving On

Once you have selected all of the components you wish to install, simply click on the Install button. The CA-Installer will begin the installation procedure, prompting you for any information necessary. Simply follow the onscreen prompts to proceed with the installation; CA-Clipper will be installed in the drive and directory you specify.

Installing CA-Clipper from DOS



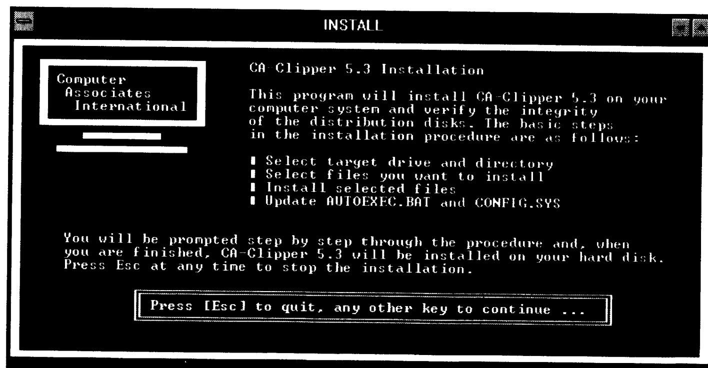
To install CA-Clipper on a hard disk from DOS, use the installation program, INSTALL.EXE, on *disk 5*:

1. Insert disk 5 in drive A.
2. Type **A:** and press Return to change the default drive to A.
3. Type **INSTALL** and press Return to start the installation procedure.

Note: The CA-Clipper Workbench files *cannot* be installed using the DOS installer. To install the Workbench or any of its components, you must use the Windows installer on disk 1.

The DOS CA-Installer

If you are installing CA-Clipper from DOS, this version of the CA-Installer appears:



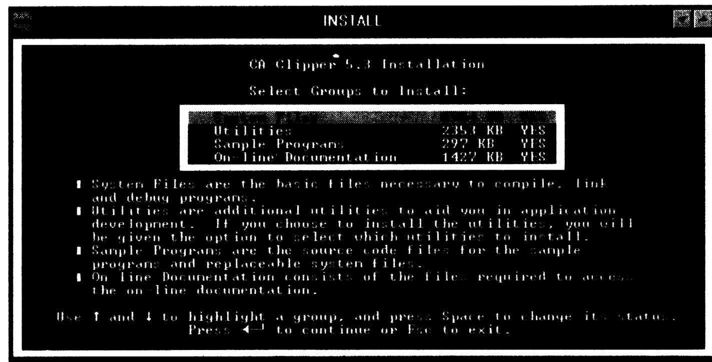
The CA-Installer is self-prompting, with online instructions that lead you through each step. The CA-Installer will begin the installation procedure, prompting you for any information necessary. Simply follow the onscreen prompts to proceed with the installation; CA-Clipper will be installed in the drive and directory you specify.

Note that if you have a previous version of CA-Clipper 5.3 installed, you will receive the following warning message when prompted to select the target directory:

Warning! *The installation will overwrite the files in the target directory. If another release of CA-Clipper is installed in the target directory that is displayed, you may want to install CA-Clipper 5.3 into a different directory to prevent losing files. Please note that \CLIPPER53 is not a valid directory name, as it will be truncated to \CLIP53.*

Full Installation or Partial Installation

After selecting the target directory, you will be prompted to specify the groups of components to be installed:



From here, you have two options. You can either install all of the components that are part of the CA-Clipper package (i.e., the full installation) or select only those groups of components you know you are going to need (i.e., a partial installation).

Full Installation

For the full installation, simply press Enter as all groups are selected by default.

Partial Installation

If you choose to selectively install CA-Clipper, you should highlight those groups you do not want to install using the arrow keys, and then press the spacebar to change their status from YES to NO. Then, press Enter.

The various groups are described below to help you decide what you need to install. Note that if you choose not to install a particular group at this time and later find that you need it, you can always run the CA-Installer again and select only that group, without affecting the rest of your current installation.

Minimum Installation

The minimum installation for CA-Clipper is the System Files group. This option will install all the system-defined libraries as well as other components that are common to all DOS applications.

System Files

This option installs the CA-Clipper compiler and all other system files and libraries associated with CA-Clipper.

Utilities

This option installs the files for the CA-Clipper database utility, DBU.EXE; the report and label utility, RL.EXE; and the DOS-level program editor, PE.EXE.

Sample Programs

This option installs the sample files that are used in the various hands-on lessons in the printed documentation for CA-Clipper.

Online Documentation

This option installs the Norton Guide Instant Access Engine and the DOS-level online help databases, *The Guide To CA-Clipper*.

The CA-Clipper Development Environment

The installation procedure sets up an environment to organize the files that make up the CA-Clipper development system. It is designed to make CA-Clipper easy to use by allowing you to operate all of the system components (e.g., the DOS-level online documentation system and the linker) from any drive or directory.

Note: The installation program creates a default configuration (e.g., directory structure, environment variables) that is used for discussion purposes in this chapter and throughout the documentation. Your actual configuration may be slightly different depending on your operating environment and your instructions to the install program.

The Directory Structure

Once CA-Clipper is installed, all the files necessary for compiling and linking your CA-Clipper programs are on the drive and directory designated by you (the default is C:\CLIP53) during the installation procedure. The following table shows a typical directory structure created by the install program to organize the files.

Directory	Description
\CLIP53	Master CA-Clipper directory
\CLIP53\BIN	Executable (.EXE) and batch (.BAT) files
\CLIP53\CACI	Dynamic link library (.DLL) and other files used by the Workbench

Continued

Continued

Subdirectory	Description
\CLIP53\CACI\DATA	Repository files
\CLIP53\CACI\DATA\AUTOMAKE	Files used for building CA-Clipper applications from within the Workbench
\CLIP53\CACI\DATA\AUTOMAKE \BACKUP	Used by the Workbench build system to back up files
\CLIP53\CACI\SAMPLES	Workbench sample source files
\CLIP53\INCLUDE	Header (.ch and .h) files
\CLIP53\LIB	Library (.LIB) files
\CLIP53\NG	*The Guide To CA-Clipper* files
\CLIP53\OBJ	Object (.OBJ) files
\CLIP53\SOURCE	Master directory for source (.prg) files
\CLIP53\SOURCE\DBU	Database utility source files
\CLIP53\SOURCE\PE	Program editor source files
\CLIP53\SOURCE\RL	Report and label utility source files
\CLIP53\SOURCE\SAMPLE	Sample program source files
\CLIP53\SOURCE\SYS	Subsystem source files

When you install CA-Clipper, the \CLIP53\BIN subdirectory is added to the SET PATH list in your AUTOEXEC.BAT. This is so you can access all CA-Clipper utilities from any drive or directory.

Warning! *The file Std.ch in the \CLIP53\INCLUDE subdirectory is a header file containing the definitions for all CA-Clipper commands. You should never modify this file directly. If you wish to explore or experiment, copy Std.ch to a separate file and make your modifications to the copy. To compile using your modified header file, use the /U compiler switch as shown below:*

```
CLIPPER Myfile /U Mystd.ch
```

Environment Variables

The DOS environment variable definitions—INCLUDE, OBJ, and LIB—are used by the compiler and linker when searching for certain types of files. Each variable is set to the directory name that contains the indicated file type. For example, SET LIB=C:\CLIP53\LIB causes CA-Clipper to search for libraries in the designated directory. This way, you do not have to include the directory name when you want to link a library with your application as long as the .LIB file is located in the LIB directory.

Note that the installation program optionally adds these environment variables to your AUTOEXEC.BAT file. You must, however, add the following environment variable manually:

```
SET PIPEHANDLE=0
```

This variable is used by the Workbench debugger. Other settings that may need to be adjusted are discussed in the next section.

Before Moving On

Before continuing on with the tutorial portions of this guide, you must make sure your AUTOEXEC.BAT and CONFIG.SYS files are updated with the necessary changes to operate in the CA-Clipper development environment.

If you allowed the installation procedure to update the AUTOEXEC.BAT and CONFIG.SYS files for you, no further changes should be necessary unless you had an older version of CA-Clipper installed elsewhere on your computer. In this case, you should edit the AUTOEXEC.BAT file and remove any environment variable and path settings that referred to the old version location.

If, however, you did not allow the CA-Installer to add the CA-Clipper directory to your AUTOEXEC.BAT path setting, you will need to make these changes manually. You may also need to adjust other settings before starting CA-Clipper, depending on your individual requirements.

AUTOEXEC.BAT

If you did not allow the installer to make the changes for you, you should add the following lines to your AUTOEXEC.BAT file:

```
SET PATH=C:\CLIP53\BIN;%PATH%
SET OBJ=C:\CLIP53\OBJ;%OBJ%
SET LIB=C:\CLIP53\LIB;%LIB%
SET INCLUDE=C:\CLIP53\INCLUDE;%INCLUDE%
SET PIPEHANDLE=0
```

Furthermore, if you do not have an environment variable named TMP, you need to create one. To do so, add:

```
SET TMP=<TempDir>
```

to your AUTOEXEC.BAT file. (Note that *<TempDir>* is any valid directory name.)

Note: If this environment variable does not exist or does not point to a valid directory, CA-Clipper applications will not be built properly from the Workbench.

SYSTEM.INI

To debug any CA-Clipper application from within the Workbench, please add the following to the [386Enh] section of your SYSTEM.INI file which is located in your Windows directory:

```
Device = CAWDCI.386
```

The CAWDCI.386 device driver is placed in your Windows system directory upon installation of CA-Clipper 5.3. If the CAWDCI.386 device driver is not loaded when Windows is started, any attempt to debug an application will receive the following error message: "Unable to connect to Windows/DOS communications pipe." You will then be unable to use the CA-Clipper Workbench debugger.

FILES Option

The recommended setting for the FILES option in CONFIG.SYS is 60 or greater. For example:

```
FILES=60
```

Loading SHARE.EXE

If you are using Windows version 3.10, it is necessary to load the DOS SHARE.EXE program to install file sharing and locking capabilities on your system. To do this, include a line similar to the following in your CONFIG.SYS file:

```
INSTALL=C:\DOS\SHARE.EXE /F:4096 /L:500
```

The values you specify for the /F and /L options will depend on your system (see your DOS manual for details).

Note: If you are using Windows version 3.11 or greater, SHARE.EXE is installed automatically.

Swap File Size

Depending on the amount of RAM in your system, you may want to increase the swap file size. We recommend using a swap file size at least as big as the amount of RAM you have, and slightly more if possible. For example, if you have 8 MB RAM, you might want to consider a swap file size between 8 and 10 MB.

Of course, this depends on the amount of free disk space you have to spare and what other applications you are running. As always, system-specific requirements can only be recommended, not dictated. Refer to your Microsoft Windows documentation for more information on setting the swap file size and on managing memory and performance, in general.

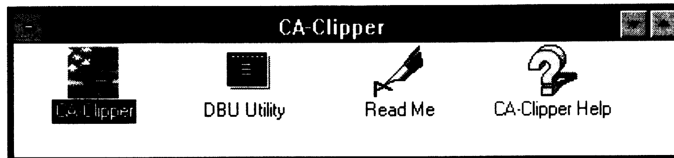
Important! *After making changes to AUTOEXEC.BAT or CONFIG.SYS, you must restart your computer for the changes to take effect. This is true regardless of whether the changes were made by the CA-Installer or by you. Therefore, as a final step before starting CA-Clipper, reboot your computer.*

Starting the CA-Clipper Workbench

CA-Clipper Program Group

Once you have rebooted, you can start the CA-Clipper Workbench from the Windows Program Manager.

To start CA-Clipper from Windows, double-click on the CA-Clipper icon that appears in the Windows program group which was created when you installed CA-Clipper. This program group should look similar to the following:



Note: The contents of this program group will differ depending on which components you chose to install. This example represents the full installation.

Double-clicking on any of the icons in this program group gives you direct access to that component.

Viewing the Read Me File

After the installation procedure is complete, you will want to view the Read Me file before doing anything else. It contains important information that is not included in the printed documentation.

Summary

Now that you have successfully installed CA-Clipper, you should read any other chapters in this book appropriate to your level of expertise. Each has an introductory section to let you know if it is for you. You are now ready to explore the CA-Clipper development system.

Chapter 3 Making the Transition to CA-Clipper

The introductory chapter gave you an overview of the new features in CA-Clipper 5.3. This chapter gives you step-by-step instructions to get your existing applications up and running using the enhanced compiler and the new, protected mode linker, and discusses the immediate benefits of so doing. It also suggests some changes that you may want to make in your source code to reap even bigger benefits. Lastly, it introduces you to the CA-Clipper Workbench, a graphical user interface (GUI) for creating DOS applications.

If you are coming to CA-Clipper 5.3 from another dBASE-style language (Xbase is a generic term used to cover all products that fit this category), this chapter may not be of much use to you. Unfortunately, the level of detail required to cover this subject adequately is beyond the scope of this chapter.

In This Chapter

The following topics are discussed in this chapter:

- How to compile and link your existing CA-Clipper applications with CA-Clipper 5.3
- The immediate benefits of using the new version
- Suggested source code changes that take advantage of new features
- Overview of the CA-Clipper Workbench

Requirements for This Chapter

Before compiling and linking your programs, you must update your AUTOEXEC.BAT and CONFIG.SYS files according to the instructions in the Before Moving On section in the "Installation" chapter of this guide. Otherwise, the operating system will not know where to locate the compiler, linker, or any of the CA-Clipper utilities you might be asked to use; and the utilities themselves may not be able to locate needed files.

Note: You must reboot your computer for the changes in AUTOEXEC.BAT and CONFIG.SYS to take effect.

Compile, Link, and Go

For the most part, your earlier CA-Clipper programs should compile, link, and run with no changes using the new version of CA-Clipper. This section tells you how and discusses any minor changes that might be necessary.

Runtime Issues

Several areas of code optimization have been added to the CA-Clipper compiler, and one of them, shortcutting logical expression evaluation, may affect how your program runs. CA-Clipper takes a shortcut when evaluating complex logical expressions linked with .AND. and .OR. by examining only as much of the expression as necessary to determine the final result.

For .OR., as soon as any expression in the list evaluates to true (.T.), the program knows that the entire expression is true (.T.) and evaluation stops. Similarly for .AND., as soon as any expression in the list evaluates to false (.F.), the entire expression is known to be false (.F.).

If your program depends on the evaluation of all expressions in a complex logical expression, compile using the /Z option to suppress logical shortcutting:

```
CLIPPER <sourceFile> /Z
```

Compiling

Although the compiler has been significantly improved (including the addition of a preprocessor), you compile, as always, by typing:

```
CLIPPER <sourceFile>
```

where *<sourceFile>* is the name of the program (.prg) file you want to compile. As before, all other .prg files referenced in *<sourceFile>* are automatically compiled and included in the resulting object (.OBJ) file. The /M option suppresses this behavior, compiling only the named .prg file.

Similarly, compiling with a script file (.clp) is unchanged:

```
CLIPPER @<scriptFile>
```

One compiler command line option, /P, has been changed. In previous CA-Clipper versions, this option allowed you to switch disks after loading the compiler from a floppy disk. In CA-Clipper, this was deemed an obsolete and unnecessary feature. The /P option is now used to generate a preprocessed output (.ppo) file.

Linking

Linking with this version is slightly different because of the new protected mode linker, CA-Clipper/Exospace, but the basic syntax you use is the same:

```
EXOSPACE F1 MyFile LI MyLibs
```

The libraries CLIPPER.LIB, EXTEND.LIB, DBFNTX.LIB, and TERMINAL.LIB are automatically linked without specifying a LIB option.

Using RMAKE

The new make system, RMAKE.EXE, replaces MAKE.EXE, the Summer'87 make system, and provides you with many new features, but you can continue to use your old make files as follows:

```
RMAKE <makeFile>.mak /U
```

This syntax tells RMAKE to look for a make file with a .mak extension rather than the default (.rmk) and to treat the hash symbol (#) as a comment indicator.

Alternatively, you can rename each of your .mak files to have an .rmk extension (RENAME *.mak *.rmk) and edit the comment lines to use the double-slash symbol (//) instead of the hash symbol (#). Then, you would use:

```
RMAKE <makeFile>
```

Immediate Benefits

By simply compiling and linking your old programs with the new version of CA-Clipper, you reap several immediate benefits, including:

- LIM 4.0 compliance
- RSIS compliance
- Virtual Memory Management
- Graphic mode

Each of these points is discussed below, including how it benefits you as a CA-Clipper developer.

LIM 4.0 Compliance

When an EMS (Expanded Memory Service) driver that complies with the LIM 4.0 standard is loaded, your application will automatically take full advantage. The result is a break in the 8-megabyte expanded memory limit (up to 32 MB), faster expanded memory access, and greater compatibility with other programs that use expanded memory, such as disk caching software and TSRs.

RSIS Compliance

The system also complies with the Relocatable Screen Interface Specification (RSIS). This compliance means that your CA-Clipper applications automatically recognize the additional high memory made available when an RSIS driver is loaded, resulting in faster screen updates and overall better program performance.

Virtual Memory Management

The Virtual Memory Management (VMM) system allows you to work with strings and arrays that are much larger than the available memory on your computer by swapping data into and out of conventional memory, giving the effect of a much larger virtual memory space. As with dynamic overlays, expanded memory is used if it is available. VMM results in faster data access through use of expanded memory. It also increases the size of memory variables and array storage areas.

Two of the CLIPPER environment variable settings are affected by the VMM system.

Although the /E setting is still supported, you should try running your application without it since limiting the amount of expanded memory available may slow down your application.

The /V setting is now obsolete since the method of storing and retrieving memory variables has been redefined with VMM. If used, this setting is ignored.

Graphic Mode

CA-Clipper 5.3 provides you with the capability of switching easily from text mode to graphic mode. 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. 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.

Note that the Light Lib Graphics API for graphic mode support is included with this release.

Source Code Changes to Consider

As you have seen, you get several immediate benefits by using CA-Clipper to compile and link your application. These benefits are available to you without making any changes to your source code, and depending on your situation you may or may not want to continue to improve your programs using CA-Clipper features.

Without considering the new features, or their advantages, this section presents some source code changes that you may want to consider to improve the overall performance of your existing applications. In each case, you will have to weigh the benefit of the suggested change against the time and effort it will take to make it and base your decision on your own analysis.

Use Manifest Constants

The CA-Clipper compiler has a built-in preprocessor that is invoked each time you compile a program. The preprocessor scans your source code for special instructions that tell it what to do. These instructions, called directives, begin with the hash symbol (#) and cause the preprocessor to perform specific tasks before the compilation process begins.

One of the directives that you might want to consider using right away is #define. In its simplest form, shown below, you can use it to replace constant values with symbolic names:

```
#define <idConstant> <resultText>
```

When the preprocessor encounters a #define directive in this form, it replaces all subsequent occurrences of *<idConstant>* with *<resultText>* in the file. The search and replace is case-sensitive so that "Esc," "ESC," and "esc" are three distinct manifest constant names.

Depending on how your programs are written, using #define instead of constant values may be quick and easy or time-consuming. If you already use the practice of declaring memory variables to store constant values, the change will be simple. For example, the following Summer '87 code,

```
Esc = 27
.
.
.
IF LASTKEY() = Esc
       .
       .
       .
ENDIF
```

could be replaced with:

```
#define Esc 27
.
.
.
IF LASTKEY() = Esc
       .
       .
       .
ENDIF
```

If you do not use this programming practice, you will have to analyze your programs to determine where constant values are used repeatedly and where memory variables are used to store such constant values. You will need to place the necessary #define statement in your program, remove any memory variable declaration and assignment statements, and put the manifest constant name into your code wherever you want its value to be used.

One specific thing to look for is the use of constant array subscripts. Using descriptive symbol names instead will make your code more readable and your array structure more flexible should you decide to change it later. It is much easier to insert a new row into an array if you only have to change the numbers once.

The concept of using manifest constants for array subscripting is well-illustrated in the Dbstruct.ch and Directry.ch header files supplied with CA-Clipper. These header files are designed for use with the new DBSTRUCT() and DIRECTORY() functions and are located in your INCLUDE subdirectory (e.g., \CLIP53\INCLUDE in the default installation). There is also an example in the Update Array Usage section later on in this chapter.

Note that #define is limited in scope to the current file (i.e., lexically scoped). If your program uses PUBLIC variables that are known application wide, you have several choices. You can put the #define statements in all your files, or in a header file (discussed later on in this section) that you #include in all your files.

At first glance this might seem futile. But when you consider that for each memory variable you eliminate, you save at least 22 bytes of memory, it doesn't seem so futile anymore. Manifest constants have no memory overhead as they are completely resolved at compile time. This means that your executable file will be smaller and will not require as much memory. Additionally, since the variable names that you eliminate do not have to be resolved at runtime, your program will run faster.

Remember the following points when deciding whether or not to use manifest constants in your existing programs:

- Manifest constants require no additional memory
- Manifest constants are completely resolved at compile time
- Symbol names are easier to read and remember than literal values
- Programs that use symbol names are easier to write and maintain

Use Pseudofunctions

Another use of the #define directive is to create pseudofunctions. This form of #define is specified as follows:

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

The preprocessor replaces calls to *<idFunction>* with the replacement expression, *<exp>*, and transports the values in the function call argument list into the replacement expression. Note that you can create replacement expressions that are quite complex by connecting several expressions using semicolons and taking advantage of the CA-Clipper inline assignment operator (:=).

A simple example of a pseudofunction replacing a function call follows. In Summer '87, you used a function call:

```
SETVAR(x, y)
.
.
.
FUNCTION SETVAR
   PARAMETERS set, get
   set = get
   RETURN (set)
```

In CA-Clipper, use a pseudofunction:

```
#define SETVAR(set, get) (set := get)
.
.
.
SETVAR(x, y)
```

If you have user-defined functions (UDFs) in your programs that lend themselves to replacement by a pseudofunction, it is something that you will want to think about doing. Functions that are simple calculations can be readily replaced. Functions that are more complicated are probably better left alone.

Using pseudofunctions gives you the same benefits as using manifest constants, including making your applications smaller and faster; however, they also have several drawbacks of which you should be aware:

- As with manifest constants, pseudofunction names are case-sensitive
- Pseudofunctions have a filewide scope which is different from declared functions
- Automatic parameter skipping is not supported in pseudofunctions

To overcome these deficiencies, you can define pseudofunctions using the #translate directive. The syntax for using this directive is slightly more complicated than #define, but its flexibility is greater. The following example illustrates how you would implement the SETVAR() pseudofunction using #translate:

```
#translate SETVAR(<set>, <get>) (<set> := <get>)
.
.
.
SETVAR(x, y)
```

Use Header Files

Another feature of the preprocessor is the ability to read the contents of an entire file into your program file at a specified location. You do this with the #include directive which has the following form:

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

Follow these guidelines when using header files:

- Limit the header file contents to preprocessor directives and EXTERNAL statements. Other statements such as commands and functions are not allowed.
- Use a .ch extension in header file names to follow the established CA-Clipper convention. The extension must be specified as part of the *<headerFileSpec>* of #include.
- Store header files in your INCLUDE subdirectory (e.g., \CLIP53\INCLUDE in the default installation). The installation program sets the value of the INCLUDE environment variable to the appropriate subdirectory so that CA-Clipper can easily locate header files.

One of the most common uses of #include is to read a file containing several related #define statements that are used repeatedly. It is much easier to type a single #include statement than to type several #define statements.

If you have taken the suggestions above and are using manifest constants and pseudofunctions in your programs, consolidating them into a header file that you #include in your source code files will save you a lot of time and make you more productive. Consider these points:

- Header files are reusable
- Header files can save numerous source code changes
- Header files make your programs more generic and modular

In addition to creating and using your own header files, you can also use the ones supplied with CA-Clipper. If you use the command or function for which the header file is designed, your programs will be more readable and maintainable when you #include the appropriate header file and use symbol names instead of constants.

For example, in Summer '87 you might have used the following to open a file with read/write access:

```
nHandle = FOPEN("Temp.txt", 2)
```

In CA-Clipper, you would use the following, more readable code:

```
#include "Fileio.ch"
.
.
.
nHandle = FOPEN("Temp.txt", FO_READWRITE)
```

Note: All of the supplied header files are located in \CLIP53\INCLUDE in the default installation.

Use Lexically Scoped Variables

In Summer '87, all variables (private, public, and field) are dynamically scoped which means they are created and maintained at runtime. They are automatically inherited by called procedures and functions which, if you take advantage of it, violates one of the main principles of modular programming. This type of variable is characteristic of interpreted languages and exists in the CA-Clipper language for compatibility with those dialects.

Lexically scoped variables, functions, and procedures were introduced in CA-Clipper to encourage modular programming and to offer you increased program efficiency. The term *lexically scoped* means that the scope of the item (i.e., its lifetime and visibility) is limited by the lexical unit in which it is defined. A lexical unit can be an entire file, a procedure or function, or a code block.

The LOCAL and STATIC statements are used to declare lexically scoped variables. They can also be declared (as local) using formal parameters in a function, procedure, or code block definition.

The following table summarizes the difference between these and dynamically scoped variables:

Declaration	**Lifetime**	**Visibility**
PUBLIC	Application or until released	Application
PRIVATE	Until creator returns or until released	Creator and called routines
STATIC	Application	Creator
Filewide STATIC	Application	Program file
LOCAL	Until creator returns	Creator

Using lexically scoped instead of dynamically scoped variables saves you execution time and program space as follows. Lexically scoped variables do not take up any space in the symbol table, allowing your application to run using less memory. Dynamically scoped variables have to be resolved at runtime each time they are referenced whereas lexically scoped variables need only be resolved once at compile time. This makes your program smaller and faster.

There are a number of cases where you may be able to replace dynamically scoped with lexically scoped variables, several of which are discussed in the following sections. All of these suggested changes will result in faster, more compact code that is more modular in nature. When making the changes suggested in this section, you may find it useful to compile with the /W option to warn you of undeclared variable references. Having a list of these may remind you of variables that you forgot to redeclare.

Replace Privates with Locals

Depending on how your programs are currently structured, you may or may not want to replace your private variable declarations with local ones. These variables have the same lifetime but different visibility, so a strict replacement will not always work.

If you use modular programming principles, the change will be easy. If you rely on the fact that the values of the variables in your program are automatically inherited, the change will be tedious and may not be worthwhile.

For example, look at the following Summer '87 function skeleton:

```
FUNCTION Dummy
   PARAMETERS a, b, c
   .
   . <some function operations>
   .
   Other(a)
   .
   . <more function operations>
   .
   RETURN (result)
```

In this code, the variables *a*, *b*, and *c* are made private by the PARAMETERS statement. Thus, even though *a* would be visible to the UDF Other(), Dummy() passes *a* as a parameter because it is good programming practice to do so. This code could be easily replaced with:

```
FUNCTION Dummy(a, b, c)
   .
   . <some function operations>
   .
   Other(a)
   .
   . <more function operations>
   .
   RETURN (result)
```

Note the use of formal parameters in the function definition. As stated earlier, this usage automatically declares *a*, *b*, and *c* as local.

If, on the other hand, your Summer '87 code is not so nicely structured, the function might look like this:

```
FUNCTION Dummy
   PARAMETERS a, b, c
   .
   . <some function operations>
   .
   Other()
   .
   . <more function operations>
   .
   RETURN (result)
```

Even though in this example Other() is called with no arguments, the function still has access to *a*, *b*, and *c*. Presumably, Other() uses this fact and depends on it to access the variable *a*. In this case, replacing the PARAMETERS statement with a formal parameter list would result in an error when Other() tries to access *a* since the visibility of this variable would be local to Dummy().

In this case, in addition to having to change the Dummy() function definition to use local parameters, you would have to change the Other() function call to pass *a* as a parameter and change the Other() function definition to accept a single argument. You would also have to change all calls to Other() in your application to pass a parameter. Thus, you can see that this may not be an easy change to make.

Use Statics Instead of SAVE and RESTORE

Static variables that are declared inside a function or procedure definition are like locals in their visibility; they are only visible inside the function or procedure. However, their lifetime is significantly different. The value of a static variable is retained when the routine that created it returns to its caller even though, at that point, the variable is no longer visible. The next time the routine is called, the static variable is again visible with its old value.

If you have an application that uses SAVE and RESTORE to remember the value of a variable, you can replace the code by declaring a static variable. For example, this Summer '87 code,

```
PROCEDURE SaveIt
   PARAMETERS a, b, c
   PRIVATE SaveIt
   RESTORE FROM SaveIt
   .
   . <statements updating value of SaveIt>
   .
   SAVE ALL LIKE SaveIt TO SaveIt
```

could be replaced with:

```
PROCEDURE SaveIt(a, b, c)
   STATIC SaveIt
   .
   . <statements updating value of SaveIt>
   .
```

In addition to the savings you get by replacing the private variables with locals, the CA-Clipper version is faster because it does no disk access.

Replacing SAVE/RESTORE code with static variable declarations only works for values that you need to save while your application is running. If you require values to be retained after the program returns control to the operating system, you will still have to SAVE and RESTORE them (although in multi-user systems, using a database (.dbf) file is preferable since .mem files cannot be shared).

Replace Publics with Filewide Statics

Static variables can also have filewide scope if you declare them at the beginning of your .prg file before any other function or procedure declarations. Filewide statics are visible to all routines defined in the .prg file and are available as long as the application is running.

You may want to replace your use of public variables with filewide statics. These variables have the same lifetime but different visibility, so a strict replacement will not always work.

In general, if you do not rely on the fact that publics are inherited outside of the current .prg file, you can make this change. If you rely on this inheritance property, the change will not be possible without significant program restructuring.

Declare Everything

One of the advantages of using lexically scoped variables is that they do not take up space in the symbol table. This means that your program will be smaller and use less memory (i.e., it will execute faster) than if you had used dynamically scoped variables. However, you may not find it feasible to change your variable usage techniques to take advantage of this feature.

CA-Clipper provides another feature that allows you to declare your existing dynamically scoped variables. Declaring the variables, although it will not save on the memory used by them, will result in compile-time resolution which saves on the code generated by the compiler each time a variable is referenced. Because runtime resolution is not required when you declare variables, your program will be smaller and run faster.

Use FIELD and MEMVAR Declarations

Two statements, FIELD and MEMVAR, allow you to declare lists of variable names to the compiler. The special aliases, FIELD-> and MEMVAR->, let you qualify variable names in your code without making declarations.

FIELD causes the compiler to treat unaliased references to the variable names it declares as though they were preceded by the FIELD-> alias or another alias that you specify. The syntax is:

```
FIELD <idField list> [IN <idAlias>]
```

MEMVAR causes the compiler to resolve unaliased references to the variable names it declares as though they were preceded by the MEMVAR-> alias. The syntax is:

```
MEMVAR <idMemvar list>
```

Declaring all the variables with these two declarations is equivalent to going through your code and putting in the proper alias (i.e., FIELD->, MEMVAR->, or <idAlias>->) before each variable name. This can be tedious if you are using a lot of variables. The CA-Clipper compiler provides you with several compiler options that might help.

Use New Compiler Options

Compiling with /A automatically declares as MEMVAR all variables included in a PUBLIC, PRIVATE, or PARAMETERS statement. Thus, if you are careful about variable declarations in your existing programs, use of a single compiler option can significantly increase your program performance and decrease its overall size.

Finally, after you have declared everything, you can use the /W option to check for undeclared variable references. This option causes the compiler to generate a warning message for each undeclared and unaliased variable name in your program. In addition to showing you variable names that you forgot to declare, this option may also indicate areas of your program where you inadvertently misspelled a variable name.

Use Lexically Scoped Functions and Procedures

One form of lexical scoping, the filewide static, can also apply to function and procedure definitions. You simply put the keyword STATIC in front of the FUNCTION or PROCEDURE statement to make the routine invisible to all but the routines in the current .prg file. This feature is useful if you have a function library and want to inhibit access to service routines (i.e., routines whose only purpose is to service other functions).

In previous releases, you had to be careful about naming such routines in your library so that they did not clash with function names that your users might pick. You may have even gone so far as to repeat useful code in several function definitions to avoid having service routines in your library code all together, resulting in a larger, less modular library.

With CA-Clipper, you can make your library more modular and readable by breaking out reusable code into static routines that are called by the end-user functions. When your library is linked into an application, the user will not stumble onto the service routines inadvertently since you declared them to be static.

For example, the following Summer '87 code,

```
FUNCTION One
   PARAMETERS a, b, c
   .
   . <statements to save environment>
   .

   .
   . <function statements>
   .

   .
   . <statements to restore environment>
   .

   RETURN (lSuccess)

FUNCTION Two
   PARAMETERS x, y, z
   .
   . <statements to save environment>
   .

   .
   . <function statements>
   .

   .
   . <statements to restore environment>
   .

   RETURN (nValue)
```

could be replaced with the following, more efficient and readable CA-Clipper code:

```
STATIC PROCEDURE SaveEnv    // Service routine to save
                            // environment
   .
   . <statements to save environment>
   .
   RETURN

STATIC PROCEDURE RestEnv    // Service routine to
                            // restore environment
   .
   . <statements to restore environment>
   .
   RETURN
```

```
FUNCTION One(a, b, c)

    SaveEnv()          // save environment

    .
    . <function statements>
    .

    RestEnv()          // restore environment

    RETURN (lSuccess)

FUNCTION Two(x, y, z)

    SaveEnv()          // save environment

    .
    . <function statements>
    .

    RestEnv()          // restore environment

    RETURN (nValue)
```

If you already have service routines to which you have assigned weird names, you can declare them as static and give them meaningful names without worry. For example, the following Summer '87 code,

```
PROCEDURE _z57935abc
    * Service routine to save environment
    .
    . <procedure statements>
    .
    RETURN

PROCEDURE _z57936abc
    * Service routine to restore environment
    .
    . <procedure statements>
    .
    RETURN
```

```
FUNCTION PrintIt
   PARAMETERS a, b, c
   * save environment
   DO _z57935abc
   .
   . <function statements>
   .
   * restore environment
   DO _z57936abc
   RETURN (lSuccess)
```

could be replaced with the following, more readable CA-Clipper code:

```
STATIC PROCEDURE SaveEnv   // Service routine to save
                           // environment
      .
      . <statements to save environment>
      .
   RETURN

STATIC PROCEDURE RestEnv   // Service routine
                           // to restore environment
   .
   . <statements to restore environment>
   .
   RETURN

FUNCTION PrintIt(a, b, c)
   SaveEnv()        // save environment
   .
   . <function statements>
   .
   RestEnv()        // restore environment
   RETURN (lSuccess)
```

Replace Macros with Code Blocks

Code blocks are a new and powerful data type in CA-Clipper that allow you to store compiled code as data. Like other data types, you can assign code blocks to variables, pass them as arguments, and return them from functions. In many cases, you can use code blocks to replace the macros in your existing programs, making them more efficient in the process.

The main difference between macros and code blocks is the time at which they are compiled. Macros are compiled "on the fly" at runtime, whereas code blocks are compiled at compile time along with all other program code. The fact that there is nothing to figure out about code blocks at runtime is what makes them execute faster than macros. Since code blocks effectively break the single compile and run function of the macro operator into two discrete steps, you save execution time especially in cases where a macro is evaluated repeatedly.

Use of code blocks is not limited to compile-time resolution. You can store a code block as a character string (in a database field or a memory variable) and evaluate it at runtime using the macro operator to convert it from a string to a code block. This feature is useful in situations where the data needed to form the code block is not available until runtime.

Use the following syntax to create a code block:

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

The argument list is optional, but the vertical bars surrounding it are required to distinguish the code block from a literal array definition. The expression list is comma-separated, the last one in the list being the value that is returned when the code block is evaluated. When the compiler encounters a code block definition, the expression list is compiled and saved if you assign the result to a variable.

To evaluate the compiled code stored in a code block, use the EVAL() function:

```
EVAL( <bBlock>, [<BlockArg list>]) → LastBlockValue
```

The following Summer '87 code skeleton illustrates a case when you would want to use a code block instead of a macro. The program uses the macro operator to compile and run expressions stored in a database file. Since the expressions are dynamic (i.e., their values change), it is necessary to use the macro operator to compile each time you evaluate. In this example, you have to compile the expression three times:

```
PROCEDURE Main
   PRIVATE cExp
   USE Exp_File
   DO WHILE .NOT. EOF()
      cExp = fExp
      .
      . <first expression involving &cExp>
      .
      . <second expression involving &cExp>
      .
      . <third expression involving &cExp>
      .
      SKIP
   ENDDO
   CLOSE
   RETURN
```

The equivalent CA-Clipper code using code blocks is shown below. In this example, the information needed to construct the code block is not available until runtime which means that you still have to use the macro operator to build the code block. This version executes faster than the Summer '87 code because the expression has to be compiled only once when the code block is constructed:

```
PROCEDURE Main
   LOCAL bExp
   USE Exp_File
   DO WHILE .NOT. EOF()
      // construct and compile code block
      bExp : &("{ || " + fExp + "}")
      .
      . <first expression involving EVAL(bExp)>
      .
      . <second expression involving EVAL(bExp)>
      .
      . <third expression involving EVAL(bExp)>
      .
      SKIP
   ENDDO
   CLOSE
   RETURN
```

Update Array Usage

Previous releases supported one-dimensional arrays only. You may have gone to great lengths to simulate them in your applications because of the lack of multidimensional array support. Simulating them would have meant using a database file where you would have liked a two-dimensional array, or using macro substitution to give the appearance of an array with several dimensions.

CA-Clipper fully supports multidimensional and nested arrays that you may want to use in place of your own unique solution. In most cases, implementing a solution requires many more lines of code than using the solution that is built into the language. Thus, your program will be smaller. Also, using an array instead of a database file saves a lot of disk I/O and, therefore, a lot of time. Finally, anytime you can get away without using a macro you should do so because you will save execution time.

The following example shows a menu that depends on the contents of a database file. This Summer '87 example assumes that the database file, MenuFile.dbf, contains the appropriate menu information:

```
FUNCTION Menu
   PRIVATE nChoice, cFunc
   USE MenuFile
   SET WRAP ON
   SET MESSAGE TO 23 CENTER
   DO WHILE .T.
      DO WHILE .NOT. EOF()
         @ f_x, f_y PROMPT f_prompt MESSAGE ;
         f_message
         SKIP
      ENDDO
      MENU TO nChoice
      DO CASE
      CASE nChoice = 0
         CLOSE
         RETURN (.T.)
      OTHERWISE
         GO nChoice
         cFunc = f_func
      ENDCASE
      DO &cFunc
   ENDDO
```

The following CA-Clipper code does the same thing as the above example using an array instead of a database file. Of course, just as you had to create the database file for use with the Summer '87 program, you have to create the array for use with the new menu function. This example includes the array declaration and initialization:

```
#include "menusubs.ch"      // contains subscript
                            // names for menu array

LOCAL aMenu[3, 5]
aMenu[1, ME_X] := 6
aMenu[1, ME_Y] := 10
aMenu[1, ME_PROMPT] := "Add"
aMenu[1, ME_MESSAGE] := "New Acct"
aMenu[1, ME_FUNCTION] := "{ || NewAccount() }"

aMenu[2, ME_X] := 7
aMenu[2, ME_Y] := 10
aMenu[2, ME_PROMPT] := "Edit"
aMenu[2, ME_MESSAGE] := "Change Acct"
aMenu[2, ME_FUNCTION] := "{ || ChangeAccount() }"
```

```
aMenu[3, ME_X] := 9
aMenu[3, ME_Y] := 10
aMenu[3, ME_PROMPT] := "Exit"
aMenu[3, ME_MESSAGE] := "Return to DOS"
aMenu[3, ME_FUNCTION] := "{ || WrapUp() }"

Menu(aMenu)

FUNCTION Menu(aOptions)
   LOCAL i, nChoice, bFunc
   SET WRAP ON
   SET MESSAGE TO 23 CENTER
   DO WHILE .T.
         FOR i := 1 TO LEN(aOptions)
            @ aOptions[i, ME_X], aOptions[i, ME_Y] ;
             PROMPT aOptions[i, ME_PROMPT] MESSAGE ;
                   aOptions;[i, ME_MESSAGE]

         NEXT
         MENU TO nChoice
         DO CASE
         CASE nChoice = 0
            RETURN NIL
         OTHERWISE
            bFunc = &(aOptions[nChoice,;
                  ME_FUNCTION])
      ENDCASE
      EVAL(bFunc)
   ENDDO
```

Another difference between this example and the Summer '87 code is the use of code blocks instead of the macro operator and the use of a header file, Menusubs.ch, to define menu subscripts. For completion of the example, the header file is listed below:

```
// Menusubs.ch
// contains subscript names for menu array

#define ME_X            1
#define ME_Y            2
#define ME_PROMPT       3
#define ME_MESSAGE      4
#define ME_FUNCTION     5
```

Now let's completely "shift gears" and take a look at the Workbench, CA-Clipper's new graphical user interface.

An Overview of the Workbench

This section presents an overview of some of the features of the CA-Clipper Workbench. Its purpose is to help you gain an understanding of what features are available to you, as well as a familiarity with the basics of working in the Workbench. This information will help you complete the sample application introduced in the next chapter.

Note: This chapter only touches upon some of the tools provided by the CA-Clipper Workbench. For complete details, please refer to the *Workbench User Guide*.

The Repository

In the CA-Clipper Workbench, applications, modules, and entities are stored in a *repository*. Source code files are stored outside the repository on disk.

Note: The CA-Clipper Workbench provides File Import and Export commands that you can use for maintaining external backup files.

An Internal, Automated MAKE Facility

The repository manages all of the pieces of an application for you. It automatically maintains the relationships between the various entities of an application. Each time you build an application, the repository "knows" what to compile based on changes that you have made, and builds the application in the most efficient way.

Applications

Modules

Entities

The repository is based on a hierarchical view of an application. Applications (like "DISKCAT") consist of modules (such as "DC_MENU_PRG") which in turn consist of entities, such as procedures and functions like "DC_MENUCREATE."
The highest level in the hierarchy is the *application*, which is exportable as an executable (.EXE) file.

Modules, which form the second level in the hierarchy, are similar to .prg and header files. They contain a group of logically related parts of the application, and may be used to limit the visibility of variables, functions, etc. defined in the module.

Similarly, just as a typical .prg file contains function and procedure declarations, modules in the Workbench contain *entities*. Entities form the third level in the hierarchy. An entity is any part of your application that has a name and can be edited. Some of the available entity types are:

- forms
- menus
- data servers
- field specs
- text blocks
- procedures
- functions

The Workbench Tools

The CA-Clipper Workbench features an integrated development environment (IDE) that provides you with a flexible, intuitive, and powerful environment for creating applications.

Within this single desktop, you can access almost any of the Workbench's features from any window at any time. You can also open and simultaneously work with multiple windows and editors.

The Workbench provides a rich set of *tools* that can be used to create sophisticated GUI applications. Like a hammer or ruler, these tools allow you to create things. For example, *browsers* let you organize and view your data, and *editors* allow you to create forms, menus, data servers, field specifications (or *field specs*), and source code.

Visual Editors

Many of the editors in the CA-Clipper Workbench are *visual*. Their point-and-click, drag-and-drop design approach and WYSIWYG environment allow you to develop an application *visually*, thereby improving the quality of the application and reducing the total development time.

In almost all cases, the flexibility and ease-of-use provided by the visual editors can help you work more efficiently. Instead of working directly in programs with the CA-Clipper language, you can lay out the visual aspects of the application and much of its functionality, providing ongoing evaluation of the application as it is created, as well as meaningful feedback about the design.

For example, to add GUI controls (like check boxes or list boxes) to a form using the Form Editor, you simply click on an icon in a tool palette and click in the form to place it. You can then manipulate and define the control as desired (for example, resize, change colors and fonts, or add code to handle events).

Likewise, when designing a menu in the Menu Editor, it is displayed in a partially operational "preview" menu bar, so you can view what your menus look like as you create them. This preview area is continually updated as you work, providing immediate visual feedback.

Creating an application in a visual fashion improves the quality of the application and reduces the total development time, as it leads to a better definition of what is needed and thereby provides for an application that best meets the user's needs.

Generating Code

When you are finished designing in any of these editors and have saved your work, the CA-Clipper Workbench generates powerful and straightforward code based on the underlying classes in the CA-Clipper language, such as CheckBox, ListBox, and TopBarMenu.

The generated code is not only efficient and powerful, it is clean and maintainable and forms a solid foundation for the future evolution of the application.

A Complete Development Environment

Of course, the CA-Clipper Workbench provides a host of other complementary tools to complete the development environment, including a source code editor, a compiler, and a debugger.

The CA-Clipper Workbench is designed to provide a productive framework for developing all kinds of applications and is specifically designed to support the iterative development paradigm.

All development tools provided in the CA-Clipper Workbench are closely integrated with the repository. In fact, all aspects of working with application components are done from the repository. This ensures efficient development and protects the integrity of your applications.

The Browsers

Browsers provide a convenient and organized way to view the data that is currently stored in your repository. In the CA-Clipper Workbench, you can browse:

- Applications
- Modules
- Entities
- Errors

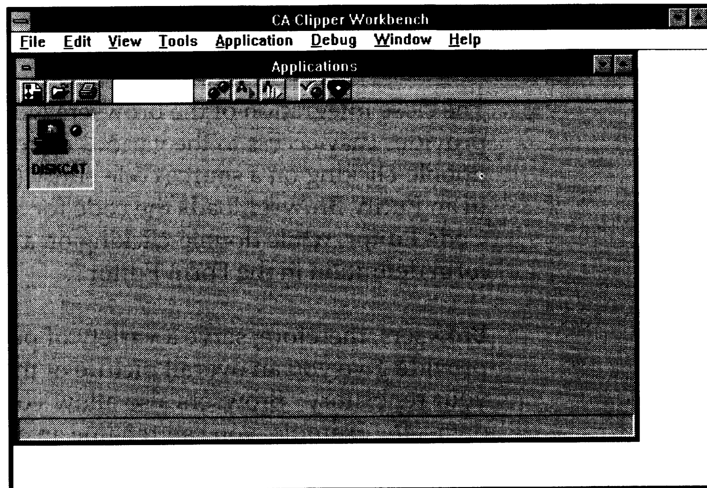
The close integration of the browsers with the repository provides easy access to the various editors. For example, double-clicking on a source code entity (such as a FUNCTION) in an Entity Browser loads the code for that entity in the Source Code Editor, while double-clicking on a form entity loads the form definition in the Form Editor.

Browsers, therefore, serve a variety of purposes. The views they provide give you an overall picture of the data that is stored in your repository. Browsers also allow you to manipulate that data—for example, you can rename an application or delete a module. Finally, they provide access to the various the Workbench editors.

Application Browser

As you have already learned, in the Workbench hierarchy, applications are composed of modules that contain entities. The primary browsers, therefore, follow this top-down hierarchy: Application Browser, Module Browser, and Entity Browser.

The *Application Browser* appears when you first start the Workbench:



This window allows you to view what is currently stored and maintained in the repository. Each button in the Application Browser represents an application.

Initially, the Application Browser is blank. As you start to add to the repository, creating your own applications, the Application Browser will display them all. (The DISKCAT application shown here is one that you will create following the steps in the tutorial in the next chapter.)

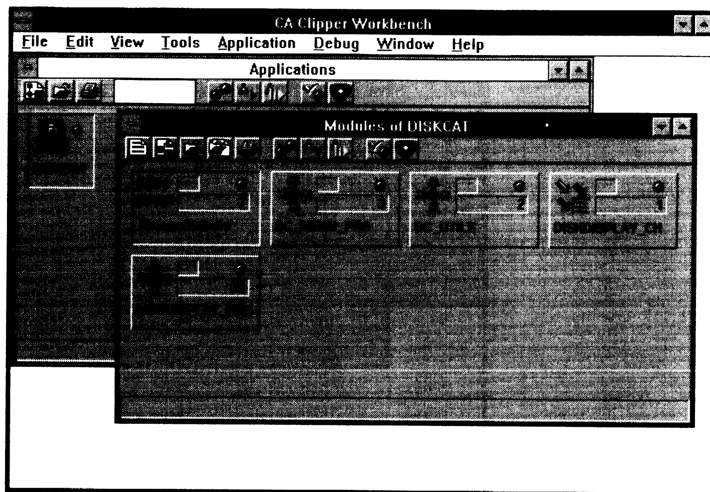
If desired, you can customize this browser as you work. For example, you can use the Name Filter area on the toolbar to limit the display of applications to those with a certain name.

Double-clicking on one of the buttons in the Application Browser "opens" it, displaying the modules associated with that application in a Module Browser.

Note: You will soon see that you can have many different Module, Entity, and Error Browsers open at the same time. However, there is only one Application Browser.

Module Browser

After double-clicking on a button in the Application Browser, the Workbench displays a *Module Browser*. For example, double-clicking on the DISKCAT button displays the following:



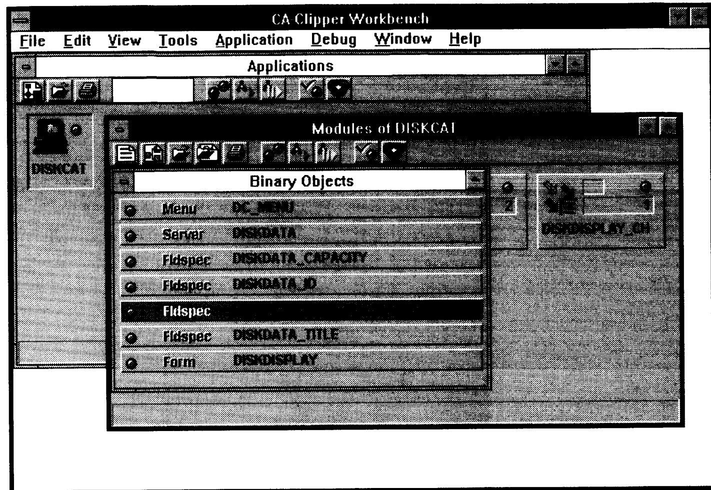
Like the Application Browser, a Module Browser displays each module as a button. You can display a Module Browser for every application in your repository.

Buttons in Module Browsers are arranged in alphabetical order. Double-clicking on one of these buttons opens it, displaying its entities in an Entity Browser.

Entity Browser

Double-clicking on a module brings up a new window that displays all the entities defined in that module. This window is called an *Entity Browser*.

For example, double-clicking on the Binary Objects module in the DISKCAT application displays the following:

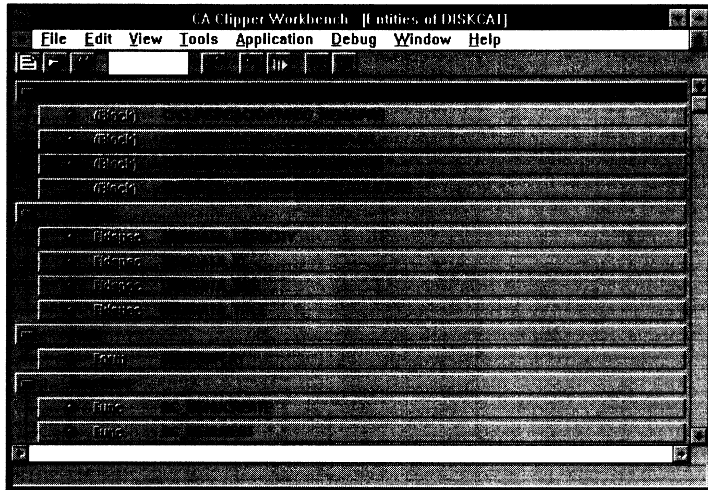


Entity Browsers display each entity as a sculpted 3-D bar, grouped by entity type in alphabetical order.

Like its predecessors, double-clicking on a binary entity causes an action. However, rather than loading another browser, double-clicking on something at the entity level starts an *editor*. For example, double-clicking on a menu entity invokes the Menu Editor. On the other hand, double-clicking on a non-binary entity, like a function or text block, activates the Source Code Editor.

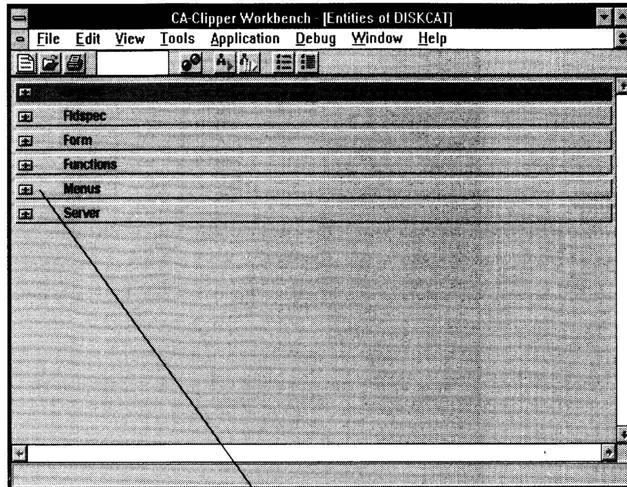
Note that the CA-Clipper Workbench allows you to browse entities on two levels: you can either view all the entities within a *module* (as described above) or all of the entities within an *application*.

To view all entities within an application, choose the Tools Entity Browser command from within the Module Browser. For example, choosing this menu command for the DISKCAT application would display the following:



Note that the items in this Entity Browser are displayed in a collapsible/expandable tree structure (grouped by entity type) that allows selective viewing of entities.

For example, collapse all branches in the entire tree by choosing the View Collapse All menu command; you could then expand only one branch to view its entities in more detail by clicking on the + icon to the left of the branch:



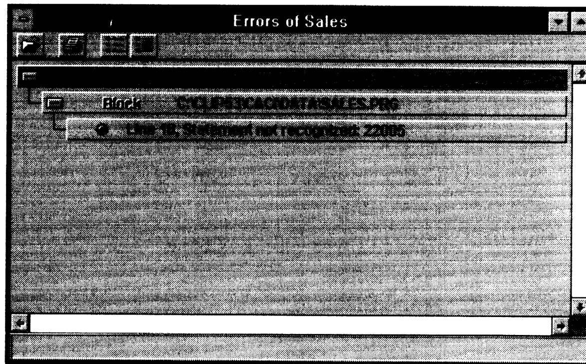
Click to expand just this branch

Error Browser

During the development cycle, compiling (or *building*) an application often results in errors. To help you locate and correct errors and warnings quickly and efficiently, the CA-Clipper Workbench provides an *Error Browser*.

As you may have noticed by now, applications, modules, and entities displayed in the various browsers you have seen so far include a small, LED-style icon. These icons indicate compilation status. For example, green means compiled successfully, while red denotes that the item needs to be compiled.

Therefore, when you build an application and a module remains red, the icon lets you know that the module contains one or more entities that have errors. To quickly view and go to these errors, choose the Tools Error Browser command. Choosing this command lists all the entities in the application that have errors or warnings:



The Error Browser displays the entities in a collapsible/expandable tree structure. If you then double-click on an error, you are brought directly to the line in the source code that contains the error.

The Editors

The CA-Clipper Workbench provides the following types of editors:

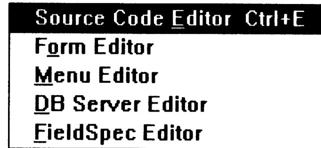
Editor	Creates
Source Code Editor	Source code entities, like functions and procedures
DB Server Editor	Data server entities
FieldSpec Editor	Field spec entities
Form Editor	Form entities (like data forms and dialog boxes)
Menu Editor	Menu entities (like menus and shortcut keys)

These editors can be used to create all the components of a sophisticated GUI application. All editors can be started by choosing a command from the Tools menu:

Tools
Application Browser
Module Browser Ctrl+F12
Entity Browser Ctrl+F2
Error Browser
Source Code Editor Ctrl+E
Form Editor
Menu Editor
DB Server Editor
FieldSpec Editor

Select an editor appropriate for the new entity

Alternatively, you can also start an editor by clicking on the New Entity toolbar button in any Module Browser and choosing an editor from a local pop-up menu:



And of course, as described earlier in this chapter, you can access an editor from any Entity Browser or Error Browser.

Source Code Editor

The Source Code Editor provides a powerful environment for writing and editing code. For example, you can cut, copy, paste, delete, search for, and replace text, as well as undo and redo editing actions, using standard Windows techniques.

The Source Code Editor also provides visual feedback by continually parsing each keystroke as you enter source code (or paste text) to color-code text based on its structure. Keywords, literals, and comments, for example, are all displayed in different colors, while each entity is separated from the next by a horizontal marker.

The collapse/expand icons, available for every entity loaded, allow you to collapse entities that you are not currently editing to provide a cleaner view of the source code and to expand them again when you need to work with them.

Data Server Editors

One of the primary tasks of any GUI database application is to enter, modify, view, and utilize the information stored in databases. This is facilitated by the use of ancillary information, like index files in the Xbase model.

The DB Server Editor

The Workbench provides the *DB Server Editor* that lets you create and modify *data servers*. A data server is a high-level, abstract entity designed to give you a consistent interface for your database. The DB Server Editor creates data servers based on the traditional Xbase model of a database file. It does not, however, physically change any database files on disk.

You can import an existing database structure and generate a default set of field specifications (explained below in The FieldSpec Editor section) that you can optionally modify.

Using data servers offers you some significant benefits. For example, many of the properties that you define for a data server and its field specifications are designed to be used by data forms that you create using the Form Editor. Thus, you need only define the attributes for a data server once, and they will be automatically inherited and used by any data form that is linked to that data server.

Similarly, changes to a data server (such as the validation rules or picture formats for one or more fields) need only be made in one place, the data server itself. Resources that use the data server will automatically inherit those changes.

Using a data server also provides an integrated view of all the pieces of information related to it. Without this comprehensive entity, you would have to create and maintain the various pieces (tables, index files, relations, and field specifications) independently. Additionally, creating data servers for your database tables allows them to be easily viewed and manipulated within the Workbench (for example, using the Entity Browser).

The FieldSpec Editor

In many cases, the different data servers your application uses contain similar, if not identical, fields (for example, all zip code fields are typically the same, regardless of where they are used). You can either define the properties of these common fields (such as validation and formatting rules) each time you create a new data server, or you can create a single field specification and reuse it in each data server that needs it.

A field specification (or *field spec*) created in the FieldSpec Editor is essentially a set of properties that are related to a field but are *independent* of any particular data server. Thus, multiple data servers can access the same property values for common fields (for example, if you create a Salary field specification, you can simply reuse its properties when creating an EmpSalary field in a data server for an Employee database). Additionally, if you change a field specification, the change will automatically propagate to all appropriate places.

Form Editor

The Form Editor is used for the interactive design of the various forms of your application.

Tool Palette

To design these forms, the Form Editor features a floating tool palette. To place a control on a form (such as a push button, list box, or radio button group), just click on a button in the tool palette and click in the form.

You can then go on to define *properties* for your forms and the various controls you place on them (for example, you may want to specify the text that should appear in the status bar when a form or control is selected).

Controls and Actions

One important property of certain controls is an *action type*. This is because certain types of controls initiate actions (for example, when the user clicks the OK button in a dialog box, the program processes the information entered in the dialog box and closes it).

The Form Editor makes it easy for you to associate actions with these types of controls by allowing you to specify an action type as property. You have the option of using either a predefined action or a custom function call.

There are many different types of controls that you can define in the Form Editor, but before going on to describe them, a few words about data forms are in order.

Data-Aware Forms

The integration of the various tools in the CA-Clipper Workbench provides some powerful benefits, one of which is the ability to create data forms. Data forms are *data-aware* because they "know" about the data server(s) upon which they are intended to operate.

A data form knows about a data server by a link that you establish between it and one or more data servers. Once a data form and a data server are linked, you can actually link individual controls in the form (such as single-line edit controls and check boxes) with fields in the data server.

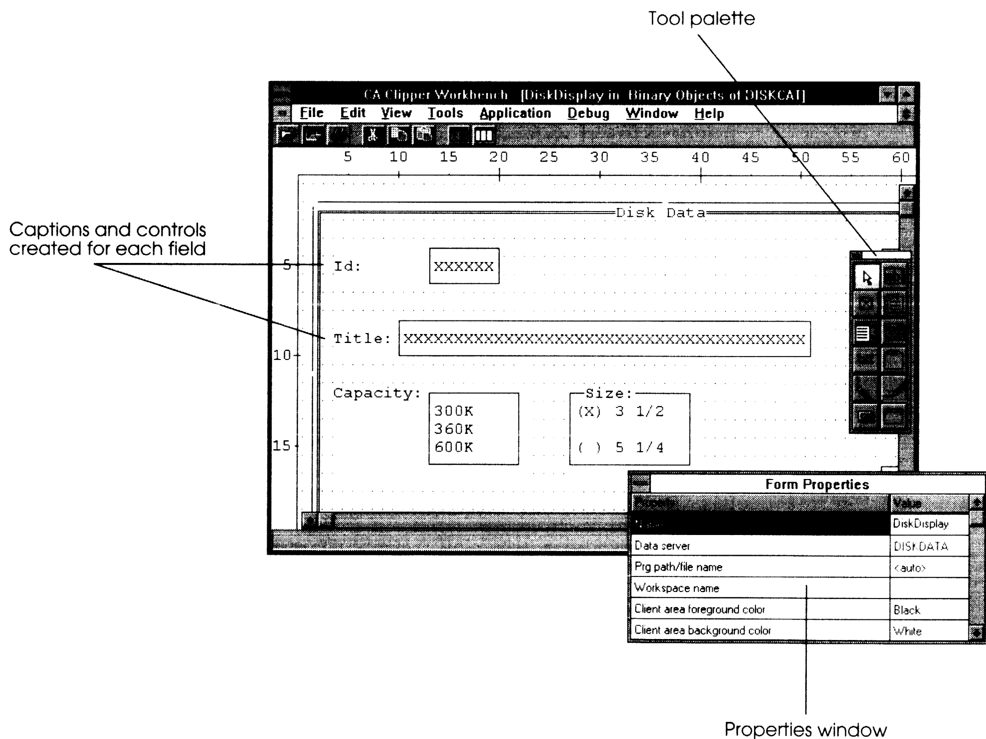
When you link a form control to a field, you are actually linking it with the field specification associated with that field—the control, therefore, automatically inherits and uses all of the field specification's properties (for example, its validation and formatting rules).

By their very nature, data forms are capable of interacting intelligently with data servers. For example, data forms can easily display the contents of a data server and have preprogrammed functions for moving among the records and manipulating the data in a data server (i.e., Go to Top and Delete Record).

Not only are data forms powerful additions to your applications, but they are also easy to create. Using the Form Editor's Auto Layout feature, you can quickly link a data form with one or two data servers, creating either a single-server or master-detail data form, respectively.

When you use Auto Layout, the Workbench automatically creates a fixed text caption and single-line edit control for every available field in the associated data server(s).

For instance, here is the DISKCAT data form that you will create in the next chapter using the Auto Layout feature:



Types of Controls

The tool palette in the Form Editor contains a host of buttons representing different graphical user interface (GUI) controls. (If you prefer, the Form Editor also features an Edit Select from Palette menu command, which allows you to place controls by choosing commands from a menu.)

Let's briefly review the types of controls that you can place on your forms using the tool palette in the Form Editor. (If you prefer, the Form Editor also features an Edit Select from Palette menu command, which allows you to place controls by choosing commands from a menu.)

Check Boxes

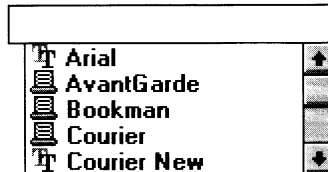
Check boxes indicate a set of options that are either on or off. If more than one check box is present on a form, the user can select as many as are applicable. The state of a check box is indicated in the box to its left. If there is an X in the box, it is selected; otherwise, it is not.

For example, the following check box from the Workbench's Compiler Options dialog box utilizes a logical field for the inclusion of debugging information in your applications. Checking the box would indicate a value of true (.T.), while unchecking it would indicate a value of false (.F.).

Combo Boxes

Combo boxes are list boxes with a single-line edit control attached at the top. The user can either type a value directly into the edit control, or click on the Down arrow button to the right to open a list box from which to make a selection. The selection is used to fill in the edit control, which can then be edited.

In a data entry form, you can use a combo box instead of a list box when the field value has more possibilities than you care to list. By placing the most commonly used values in the associated list box, you give the user a quick way to make a selection without removing the flexibility of entering values that are not listed. Shown here is an example from the Workbench's Browsers Font dialog box:



Fixed Text

Fixed text displays a caption or label anywhere within a form. A common use of this type of control is to create a caption for a single-line edit control, a feature that is utilized by the Form Editor's Auto Layout feature. Below is an example from the Workbench's System Options dialog box:

Path for CA-Clipper EXE Files:
C:\CLIP53\CACI\DATA

Frames

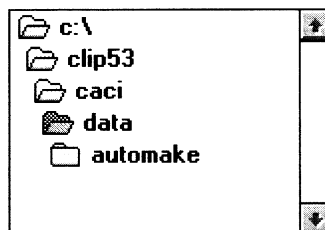
Frames visually indicate a set of related controls. They provide a caption to describe the controls, but serve no other purpose. They are most often used to display a group of related check boxes. Below is an example from the Workbench's System Options dialog box which allows you to choose various system default options:

System Default Options

- [X] Show Entity on Status Bar
- [ ] Create Default PRG Module
- [ ] New Module Debug
- [X] Color LED's
- [X] OEM to ANSI

List Boxes

List boxes display a list of choices to the user and allow the user to scroll through them and select one. In a dialog box, you might use list boxes to allow the user to select a file name and a path. On a data entry form, you might use a list box to display all possible values for a particular field. This example is from the Workbench's Browse dialog box:



Push Buttons

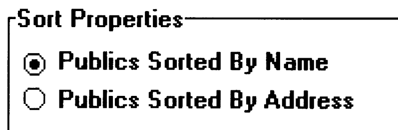
Push buttons react when the user chooses them by generating an event. Some examples of push buttons are the standard OK and Cancel push buttons, shown below, used to close a dialog box:

Radio Button Groups

Radio button groups visually indicate a group of *radio buttons.* Like a frame, they provide a descriptive caption for the controls they contain, but they have another special purpose—only one of the radio buttons within a radio button group can be selected at a time. When the user chooses a new radio button in the group box, the previously selected one is deselected.

Each radio button group behaves independently. In other words, you can place several groups of radio buttons on the same form, and the user can select one radio button in each radio button group.

Radio button groups let you use radio buttons to present a set of choices to the user. The following example from the Workbench's Link Map Options dialog box allows you to sort public symbols either by address or by name:



Single-Line Edit Controls

Single-line edit controls are probably the most commonly used controls on data entry forms and are often used to represent fields into which the user may type almost any value:

Subforms

Subforms are simply data forms that you place on other data forms as controls. Typically you would use a subform to show a master-detail relationship between two related data servers. For more information, refer to Data Forms in the "Using the Form Editor" chapter of the *Workbench User Guide.*

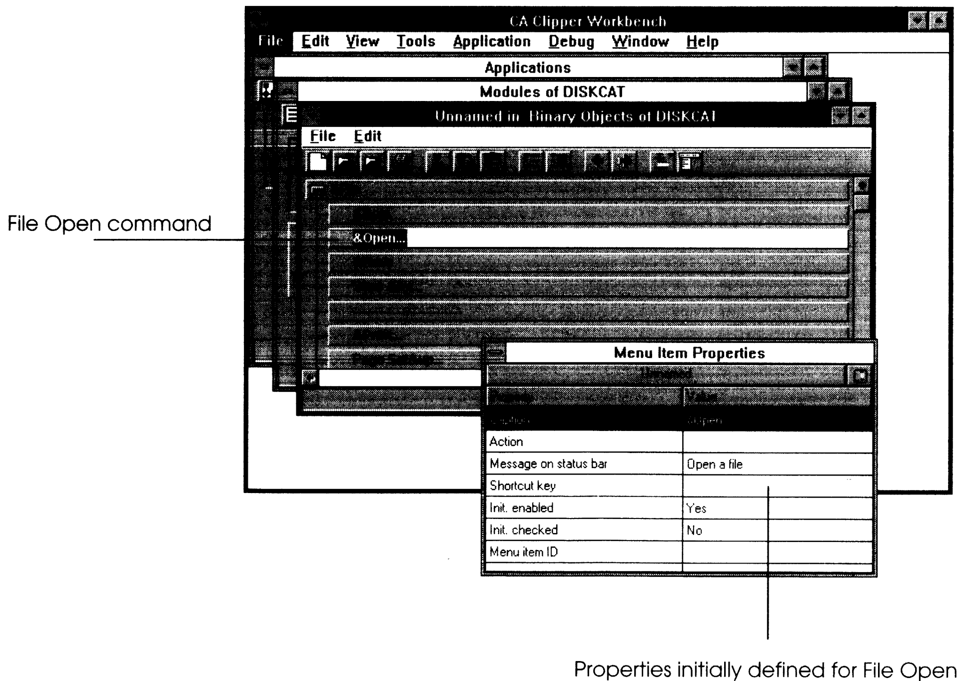
Menu Editor

The Menu Editor, shown below, provides a powerful yet easy way to create menus and toolbars for your applications.

Auto Layout

First of all, like the Form Editor, the Menu Editor features an Auto Layout feature. In the Menu Editor, however, Auto Layout is used to add one or more predefined, standard menus to an application. For example, at the touch of a button, you can add File, Edit, View, Window, and Help menus to your application. In addition, each of these predefined menus (like File) contains a set of default menu items (for example, New, Open, and Save), for which default properties are already supplied, including action types.

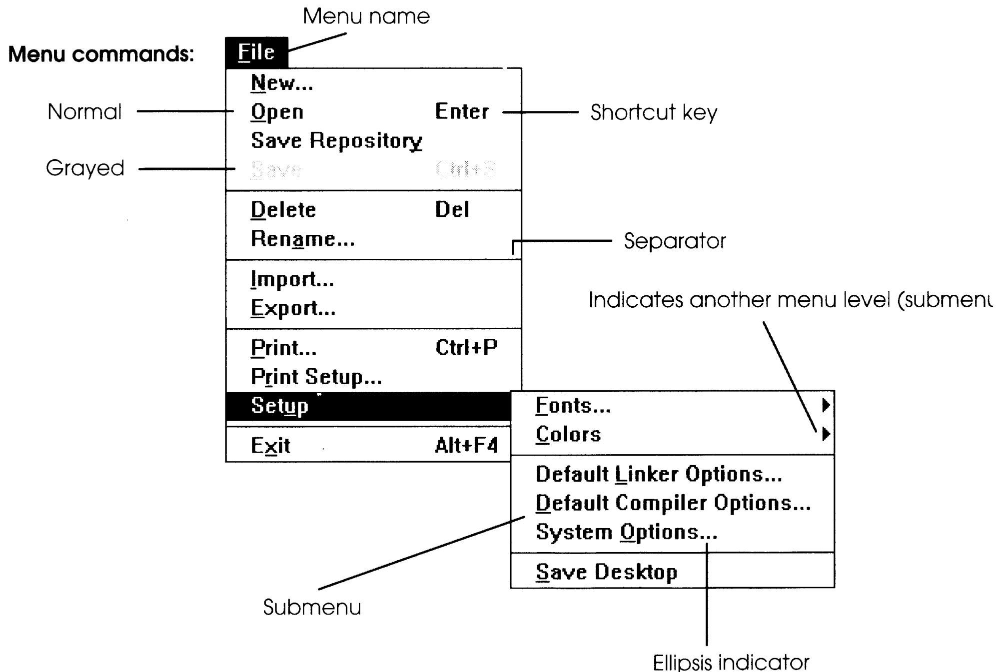
For example, the following shows the properties initially set for the predefined File Open menu command:



Auto Layout provides a quick way to get started with your menu structures—you can use the resulting menus as is, or you can customize them as desired to fit your application. Of course, you can easily create your own custom menu structures in the Menu Editor.

Menu Items

The various types of menu items that you can include in a menu's structure are illustrated in the Workbench example below:



A *grayed* menu command is one that is not currently available, and a *checked* menu command (not shown) indicates a default toggle setting.

Notice that the "F" in the "File" menu is underscored. This indicates that it is an *accelerator key,* which allows fast access to the pull-down menu itself by pressing Alt and the specified key together (or just the specified key in the case of a menu item on a pull-down menu).

Additionally, *shortcut keys*—like Del, Ctrl+P, and Alt+F4—also allow fast access to menu items. Unlike accelerator keys, the menu need not be open for the shortcut key to be active.

The *ellipsis indicator* (...) indicates that a dialog box is displayed when this type of menu command is chosen.

Like some form controls, an important property of items on a menu is an action. This is because menu items, like certain form controls, initiate actions. The Menu Editor makes it easy for you to associate actions with menu items using the Action property, exactly as previously described for the Form Editor.

For more detailed information about menus, menu items, and the Action property see the *Workbench User Guide*. Also refer to the following classes in the *Reference Guide*: TopBarMenu, PopUpMenu and MenuItem.

The Debugger

The Workbench's debugger provides advanced tools for tracking and correcting errors that occur at runtime. For example, you can:

- Use one of several execution modes to control the execution of your application while viewing the source code in the Debug source code window
- Evaluate and trace expressions
- Set, reset, and clear breakpoints
- View and modify variables
- Create watch expressions
- View the call stack
- View database and other work area information in a separate window and modify database field values
- View and modify system settings

In addition, the CA-Clipper Workbench allows you to set debugging options at either the application or module level, overriding the current system default setting.

This means, for example, that if you have a successful, stable application and decide to add new features to it, you can save valuable time by testing and debugging only the new module. It also means that, with new applications, you can debug the application piece-meal by setting the application-level debug flag on, and selectively turning off the debug flags for modules that you are not currently interested in debugging.

Summary

In this chapter, you have seen a high-level overview of many new CA-Clipper features and their benefits. Some of the benefits require a fair amount of work to achieve and may or may not be worth your while, but the important thing to keep in mind is that you will get substantial benefits immediately by simply compiling and linking with the new version.

Many of the topics presented in this chapter are complex and are not fully explained. Also, the list of topics discussed is by no means exhaustive (see the "Introduction" chapter of this guide for a complete list of new features). There are probably other areas of program improvement that might be beneficial to you, and certainly there are many more new features for you to take advantage of when doing new development. For additional information regarding any topic in this guide, there are a number of places that you can look.

In *The Guide To CA-Clipper*, the Release Notes database provides a detailed change summary. Use this database if you want to know the new and changed features in the product, including language enhancements.

In the "Basic Concepts" chapter of the *Programming and Utilities Guide*, you will find a section on Variables that will help you to better understand variable scoping and declaration. Also, there is a section on Code Blocks in this same chapter that will help you with that subject.

The compiler, linker, and make utility are fully documented, each in its own chapter, in the *Programming and Utilities Guide*. The preprocessor is discussed in detail in the compiler chapter.

Additionally, you can always look to the alphabetical listing of the *Reference Guide* for additional information on any language item mentioned in this chapter (e.g., #define, #translate, #include, LOCAL, STATIC, MEMVAR).

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.

Finally, you have had an introduction to the browsers and visual editors that make up the CA-Clipper Workbench. These tools provide an immediate means for you to examine and control your applications and will become even more useful as your application increases in sophistication. Building upon this overview, the next chapter teaches you how to use the CA-Clipper Workbench with a "hands-on" tutorial.

Chapter 4
Learning the Basics

In This Chapter

This chapter walks you through the process of creating and executing a simple CA-Clipper application using the various tools provided by the CA-Clipper 5.3 Workbench. If you have never used a GUI interface before or are simply new to CA-Clipper, just follow the easy, hands-on lessons in the tutorial.

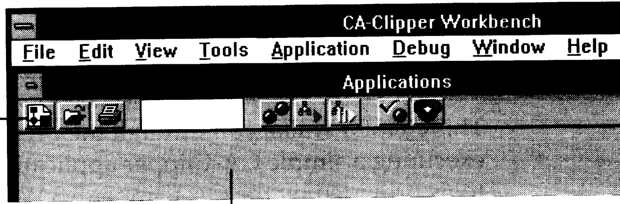
You will learn how to:

- Create a new application—a disk catalog system called "DISKCAT"
- Create a menu structure with the Menu Editor
- Use the DB Server Editor to create a data server and define field specifications
- Create a data entry screen with the Form Editor
- Import an external file
- Access the Workbench's Source Code Editor to modify code
- Select compiler and linker options
- Build and execute the DISKCAT application

LESSON 1: Creating an Application

Creating a new application with the CA-Clipper Workbench is a very simple process. To begin, simply click the New Application toolbar button from within the Application Browser:

New Application toolbar button



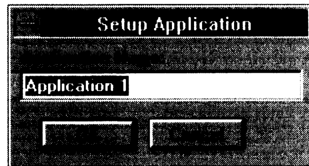
Application Browser

Clicking this button is the same as choosing the File New menu command:

File
New...
Open Enter
Save Repository
Save Ctrl+S
Delete Del
Rename...
Import...
Export...
Print... Ctrl+P
Print Setup...
Setup
Exit Alt+F4

In the Workbench, almost every window contains toolbar buttons that serve as shortcuts for commonly used menu commands. To quickly find out what a toolbar button does, position the mouse over it—a description appears in the status bar. For example, in the case of the New Application toolbar button illustrated above, the status bar message reads "Create a new application."

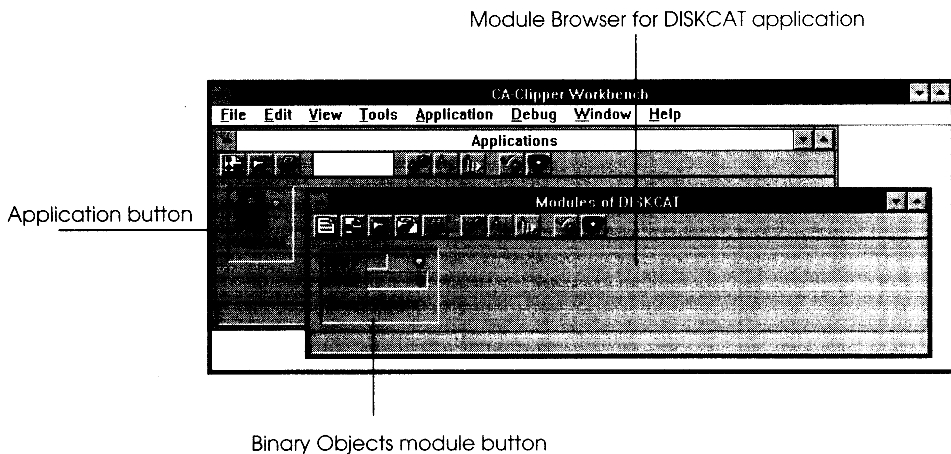
Clicking the New Application toolbar button displays the Setup Application dialog box:



To give your sample application a name:

1. Enter **DISKCAT** in the Application Name edit control.
2. Click OK.

An empty application named DISKCAT has now been created with its own application button:



Notice that a Module Browser has also been created. It contains one module (named Binary Objects) which has zero entities in it. This module will contain the binary definitions of all form, menu, data server, and field specification (or *field spec*) entities that you create for DISKCAT using several of the Workbench's editors: the Menu Editor, Form Editor, DB Server Editor, and FieldSpec Editor.

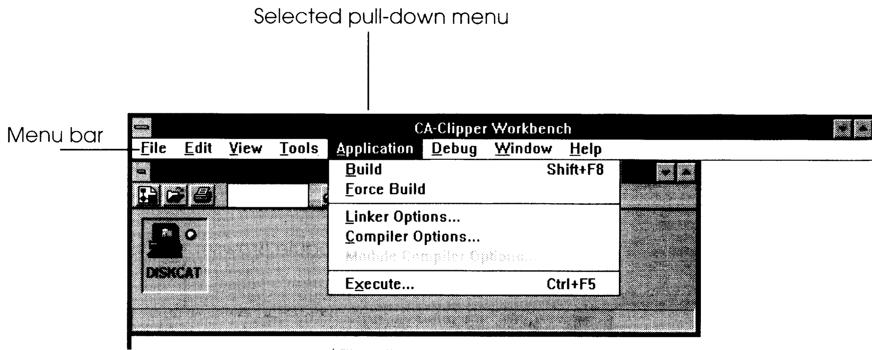
In the next lesson you will create the first entity for the Binary Objects module.

LESSON 2: Adding a Menu Structure

Before creating a menu structure for the DISKCAT application, let's briefly review some basic menu terms used in CA-Clipper.

Menu Terms

A *menu* is a user interface element that presents a list of choices. Menus appear in a window's *menu bar*. For example, below is the Workbench's menu bar:



Typically, when a menu is selected from a menu bar (with the mouse or keyboard), it displays a pull-down menu of *menu items*: menu commands, accelerator and shortcut keys, separators, and additional indicators for submenus and dialog boxes. However, it can also immediately execute an action (for example, the Exit command in the File menu that quits the application when selected).

Local *pop-up menus* also consist of menu items. For example, below is a pop-up menu that appears when you right-click on an application button in the Application Browser:

Start application viewer
Delete application

Compiler options
Linker options

Build the application
Force an application build
Execute the application

Menu Items

The various types of menu items that you can include in a menu's structure are menu commands, shortcut keys, submenus, separators, etc. which were discussed in the previous chapter.

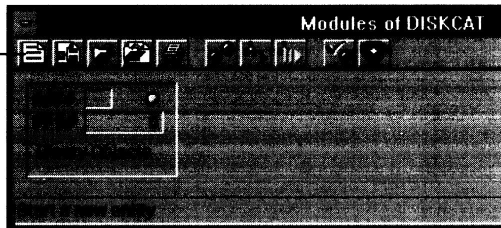
Creating a Menu Entity

Now let's create a menu entity that defines a pull-down menu structure for the DISKCAT application. This menu structure will be made available to the user throughout the application and will allow the user to move among all the records in the database.

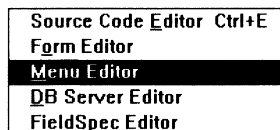
To create a menu entity using the Menu Editor, perform the following steps:

1. With the Binary Objects module selected, click the New Entity toolbar button in the Module Browser:

New Entity toolbar button



2. Choose Menu Editor from the local pop-up menu that appears:

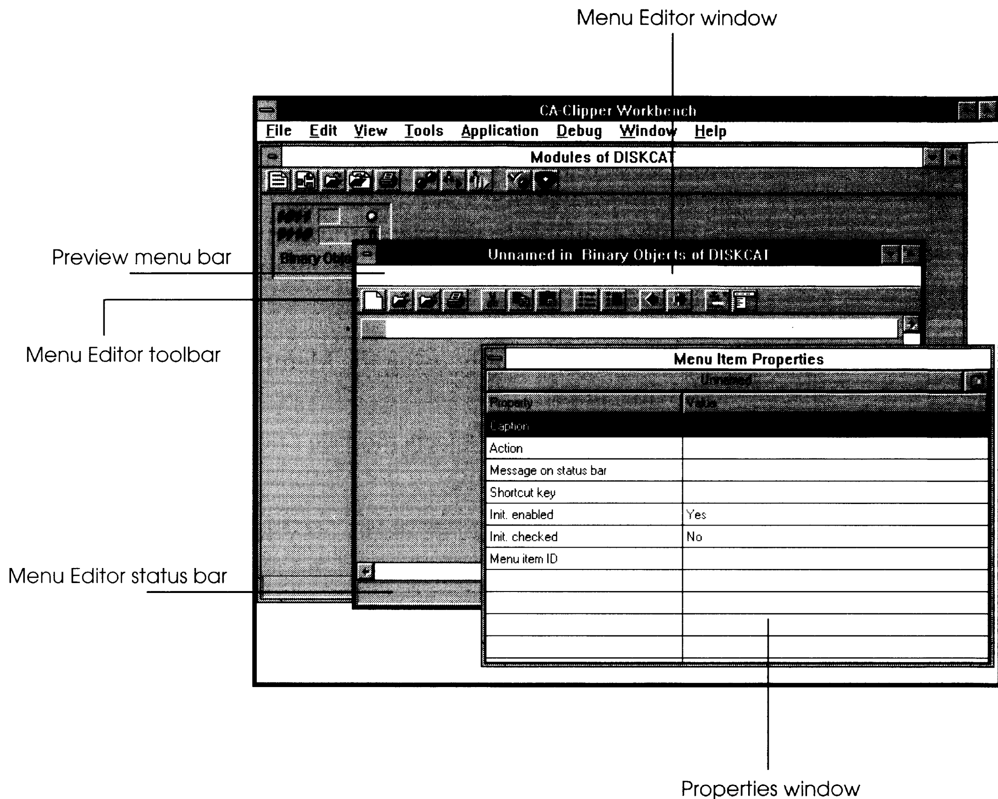


Alternatively, select the Menu Editor command from the Tools menu.

The Menu Editor is displayed. Let's look at its components.

Workspace Overview

The Menu Editor is the primary workspace in the Workbench for creating, viewing, and modifying menu structures:



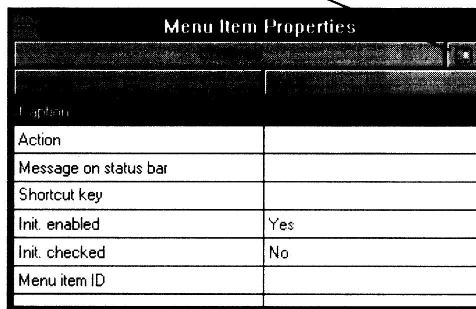
Notice that the Menu Editor has its own toolbar, status bar, floating Properties window, and area for previewing defined menus.

Tip: Like any other Workbench window, the Menu Editor can be maximized or minimized to an icon. Also, windows may be cascaded or tiled.

Naming the Menu Structure

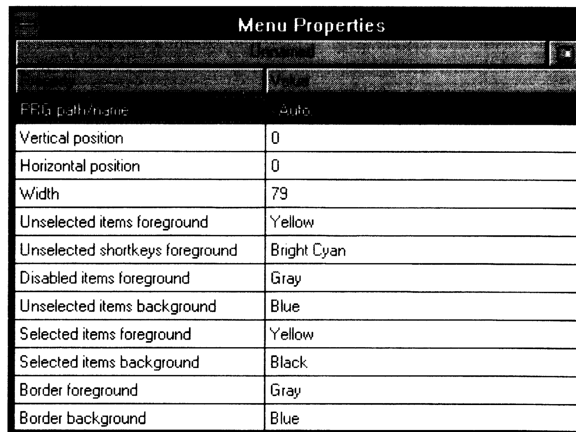
The first step in creating a pull-down menu structure in the Menu Editor is to give it a name. To do this:

1. Switch to the Menu Properties window by clicking the Pencil button to the right of the 3-D bar titled "Unnamed" located at the top of the Property/Value list:



Alternatively, select the Menu Properties command from the Edit menu.

The Menu Item Properties window becomes the Menu Properties window:

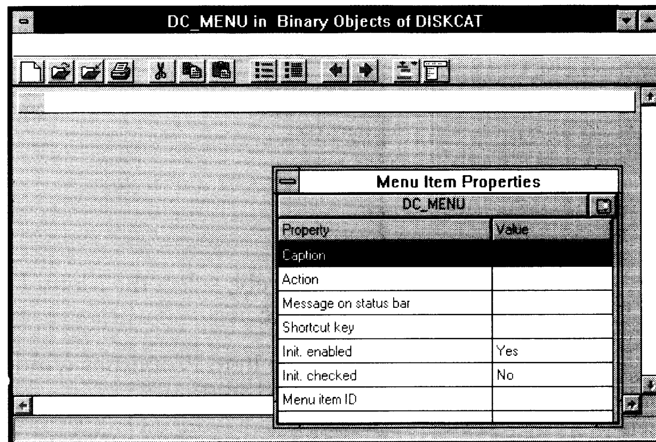


2. Replace "Unnamed" with **DC_Menu** in the edit control that appears.

   This name will be the name of the menu entity in the Binary Objects module. This name, followed by "_PRG," will also be the name of the module that will be created for the generated .prg file. Therefore, it must not conflict with other entity names in your application.

3. Press Enter.

   "DC_MENU" now appears in the title bar of the Menu Editor:

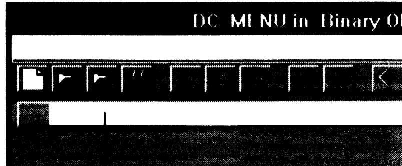


Creating Menus and Defining Menu Items

Adding File and Edit Menus

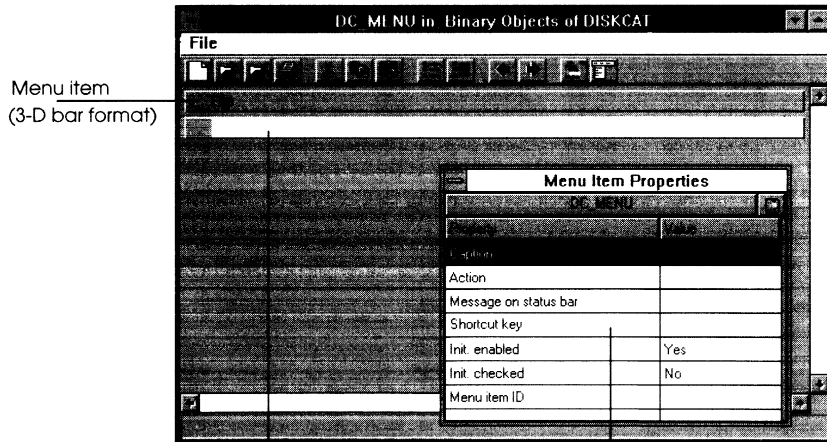
Your next step will be to create two pull-down menus named "File" and "Edit" for the DISKCAT application. To do this:

1. Click on the empty edit control:



2. Type **File**.
3. Press Enter.

   A new edit control and Menu Item Properties window appear for the next menu item:



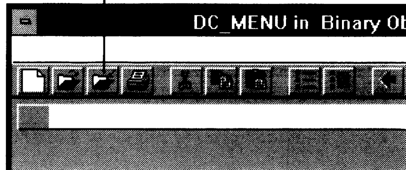
   Notice that defined edit controls in the Menu Editor have a 3-D bar format.

4. Type **Edit** in the new edit control.

5. Press Enter again. At this point, you've created a new, and therefore, empty edit control. Make sure you delete it before you move on to adding menu items.

6. Choose the Save toolbar button to save your work so far:

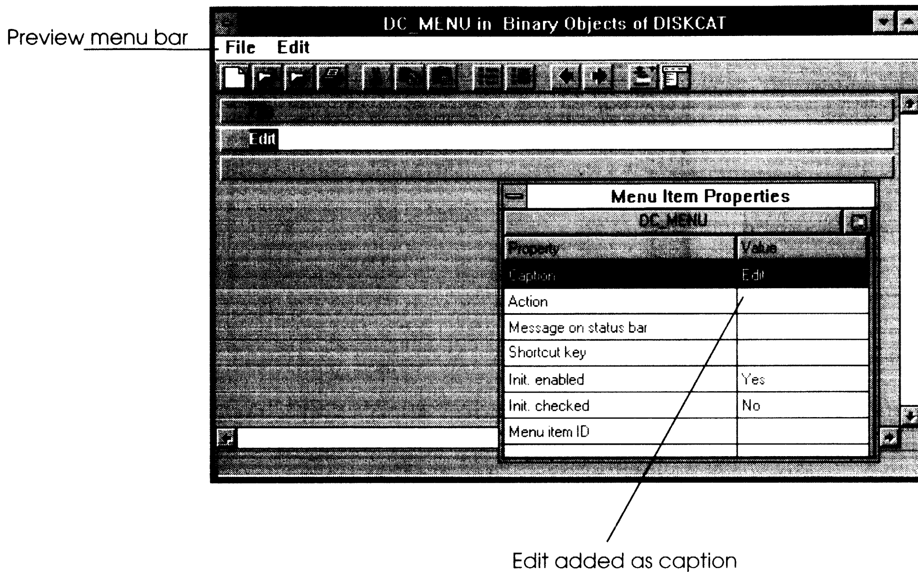
   Save toolbar button

   Alternatively, choose the File Save menu command.

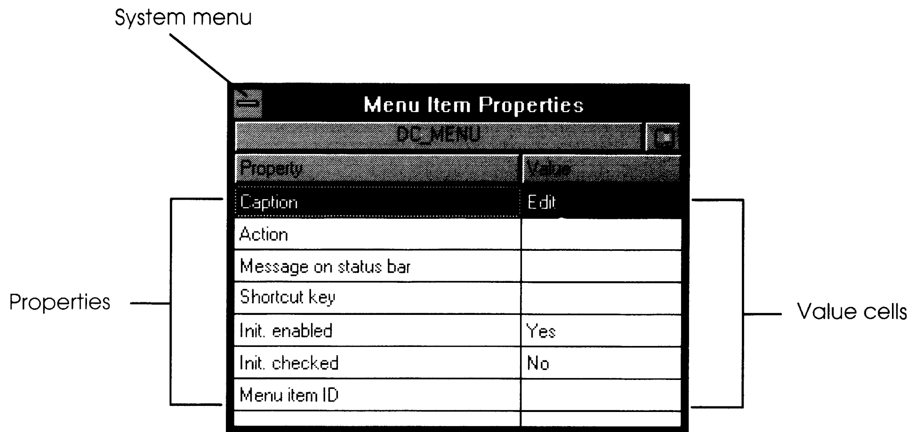
   Note: If you try to save an unnamed menu entity, you will be prompted by a dialog box to input the menu name. Also, you cannot save a menu entity that has a blank menu item. To delete a blank menu item, use the Edit Delete Item menu command.

As each new entry is added to the menu structure, the Menu Item Properties window is also updated behind the scenes using each entry's text as the caption for that menu. For example, if at this point you click on the 3-D bar titled "Edit," you will see the following:



Notice also that File and Edit have been added automatically to the Menu Editor's *preview menu bar*. This prototypical menu bar is partially operational—showing description messages in the status bar and allowing submenus to be pulled down—but nothing actually happens when you make a selection. Its purpose is to provide you with visual feedback while you are designing a menu structure.

Now, you need to define the pull-down menu items that will go under the File and Edit menus just created. To do this, you must use the Menu Item Properties window:



Initially, this window allows you to specify properties for the current menu item. The *Property* column lists all properties that can be specified for the currently selected entry, and the *Value* column contains the corresponding cells where you specify a value.

Note: The Properties window always remains open until explicitly closed (using the system menu) or until its owner, the Menu Editor window, is closed. If explicitly closed, you can reopen it at any time choosing the Open Property Window command from the Window menu. Also, the Properties window is affected by actions to its owner window. For example, if the owner window is minimized to an icon, the Properties window will also be minimized.

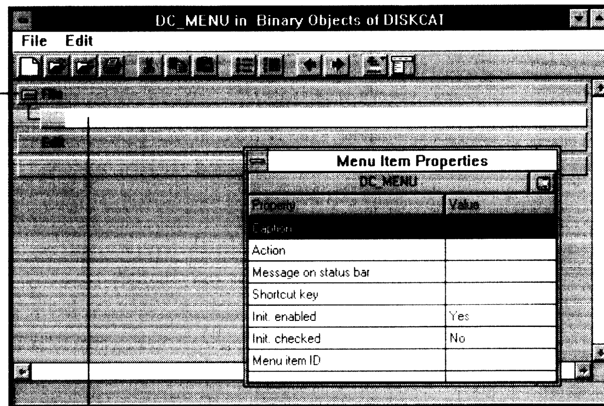
Adding Menu Items to the File Menu

For the purposes of this tutorial, your File menu needs only one menu item—an "Exit" menu command. To insert a menu item hierarchically under the File menu:

1. Click on the 3-D bar titled "File."
2. Select the Insert Child command from the Edit Insert Item menu.

   This opens up a new edit control that is indented, indicating that it is a *child*, or submenu, of File:

Expand/Collapse (+ / -) button



Indented edit control indicating a child menu item (or submenu)

   Notice also that an Expand/Collapse button has been added to the "File" 3-D bar. This button allows you to expand or collapse the menu hierarchy for viewing purposes.

3. Enter **Exit** in the new edit control.

   Alternatively, enter **Exit** in the value cell for the Caption property in the new Menu Item Properties window.

Captions can include text, blanks, punctuation, and the underscore character (_). Additionally, you can use an ampersand (&) in the caption to specify that the character immediately following it be underscored, indicating that it is the menu item's accelerator key. With the exception of the ampersand, all characters will appear exactly as typed.

Note: The shortcut key, check mark, and the right arrow symbol (▶), used to indicate a submenu, are not part of the caption. These indicators are added automatically by the system. However, the standard ellipsis (...) indicator, used to indicate that a menu item opens a dialog box, *must* be included as part of the menu item's Caption property.

4. For the Message on Status Bar property, type **Exit this program**, and press Return.

5. Enter **999** in the value cell for the Menu Item ID property.

   The numeric value of 999 identifies the Exit menu item in the .prg file that CA-Clipper will automatically generate when you are finished defining your menu structure. (Generated code is discussed later in this tutorial.)

6. Delete any blank menu items, using the Edit Delete Item menu command, before saving your work up to this point.

7. Choose the Save toolbar button.

   At this point, the Workbench informs you that an external .prg file already exists and asks whether you wish to overwrite it. This external file was created behind the scenes when you specified a menu name and first clicked on the Save toolbar button. As you continue to add menu items to your menu, choose Yes when prompted by this dialog box.

Tip: If there is not enough room in a property's value cell to type in all of your text, simply resize the Properties window. To do this, click on one of the window's sides and drag it horizontally. Then re-enter the text.

Adding Menu Items to the Edit Menu

Now let's add the following menu items for the Edit menu: "Top," "Bottom," "Next," and "Previous" menu commands, and a separator.

Top Menu Command

To add the first menu command, Top, under the Edit menu:

1. Click on the "Edit" 3-D bar.
2. Select the Insert Child command from the Edit Insert Item menu.

   A new edit control and Properties window appear.

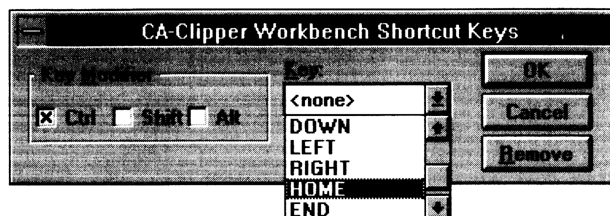
3. In the new Menu Item Properties window, enter **Top** in the value cell for the Caption property.
4. Enter **DiskDispTest(DBGOTOP())** in the value cell for the Action property.

   This is the function call to be performed when this menu item is selected. You will associate the function with the form later in the tutorial using the Form Editor.

5. For the Message on Status Bar property, type **Go to the first record in the database**, and press Enter.
6. Click on the Shortcut key property value cell.

   The Shortcut Keys dialog box appears, allowing you to define a shortcut key.

7. Select Ctrl in the Key Modifier group box and HOME from the Key drop-down list box:



8. Choose OK.

   You are returned to the Menu Editor. Notice that the shortcut key value is added by the system to the Menu Item Properties box:

Menu Item Properties

Property	Value
Caption	Caption
Action	
Message on status bar	
Shortcut key	Ctrl+HOME
Init. enabled	Yes
Init. checked	No
Menu item ID	

9. Again choose the Save toolbar button.

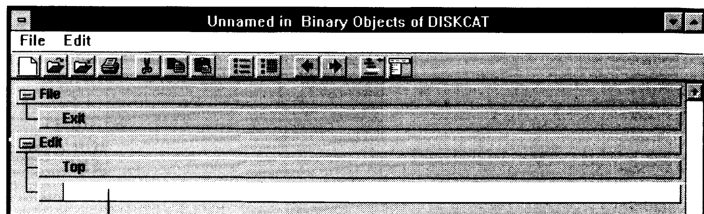
At this point, the Workbench informs you that an external .prg file already exists and asks whether you wish to overwrite it. This external file was created behind the scenes when you specified a menu name and first clicked on the Save toolbar button. As you continue to add menu items to your menu, choose Yes when prompted by this dialog box.

Bottom Menu Command

Now add the Bottom menu command and specify its property values:

1. With the cursor in the Top edit control, select the Insert Sibling command from the Edit Insert Item menu:

   The Insert Sibling command creates a new edit control at the same, or *sibling*, level as the Top menu command:



New edit control at the same level as its sibling, Top

Alternatively, you could hit Enter. But you could wait to do that later where this is dealt with as an alternate method of adding menu items.

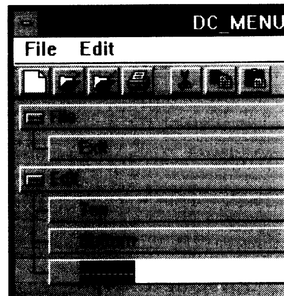
2. In the new Menu Item Properties window, enter **Bottom** in the value cell for the Caption property.
3. Enter **DiskDispTest(DBGOBOTTOM())** in the value cell for the Action property.

   This is the function call to be performed when this menu item is selected. You will associate the function with the form later in the tutorial using the Form Editor.
4. For the Message on Status Bar property, type **Go to the last record in the database**, and press Enter.
5. Click on the Shortcut key property value cell.

   The Shortcut Keys dialog box appears.
6. Select Ctrl in the Key Modifier group box and END from the Key drop-down list box.
7. Choose OK.
8. You are returned to the Menu Editor.

Separator

To separate the Top and Bottom menu commands logically and visually from the Next and Previous menu commands, simply choose the Insert Separator command from the Edit Insert Item menu while the cursor is still in the Bottom edit control. The CA-Clipper Workbench automatically adds a separator to your menu structure:



Separator

Next Menu Command

Now add the Next menu command and specify its property values:

1. With the cursor in the Separator edit control, select the Insert Sibling command from the Edit Insert Item menu.

   A new edit control and Properties window appear.

2. In the new Menu Item Properties window, enter **Next** in the value cell for the Caption property.

3. Enter **DiskDispTest(DBSKIP())** in the value cell for the Action property.

   This is the function call to be performed when this menu item is selected. You will associate the function with the form later in the tutorial using the Form Editor.

4. For the Message on Status Bar property, type **Go to the next record in the database**, and press Enter.

5. Click on the Shortcut key property value cell.

   The Shortcut Keys dialog box appears.

6. Select Ctrl in the Key Modifier group box and N from the Key drop-down list box.

7. Choose OK.

8. You are returned to the Menu Editor.

Previous Menu Command

Finally, add the Previous menu command and specify its property values:

1. With the cursor in the Next edit control, select the Insert Sibling command again.

   A new edit control and Properties window appear.

2. In the new Menu Item Properties window, enter **Previous** in the value cell for the Caption property.

3. Enter **DiskDispTest(DBSKIP(-1))** in the value cell for the Action property.

   This is the function call to be performed when this menu item is selected. You will associate the function with the form later in the tutorial using the Form Editor.

4. For the Message on Status Bar property, type **Go to the previous record in the database**, and press Enter.
5. Click on the Shortcut key property value cell.

   The Shortcut Keys dialog box appears.
6. Select Ctrl in the Key Modifier group box and P from the Key drop-down list box.
7. Choose OK.
8. You are returned to the Menu Editor.

9. Choose the Save toolbar button.

   The Menu Editor displays a dialog box that informs you that an external .prg file already exists and asks you if you want to overwrite it.
10. Choose OK.

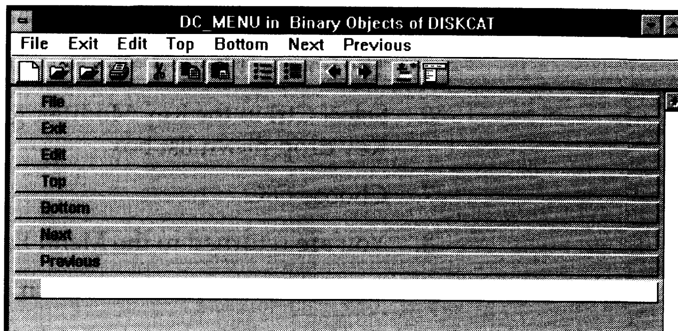
Alternate Method of Adding Menu Items

There is an alternate and faster method of adding menu items to a menu structure and defining its hierarchy.

1. In the initial empty edit control, enter the first item (for example, **File**).
2. Press Enter.

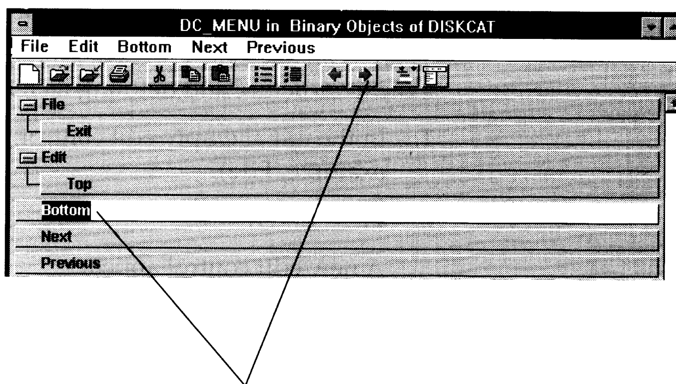
   A new edit control appears.
3. Type in the next item (for example, **Exit**) and press Enter.

4. Continue in this manner until you have entered all of the menu items like so:



Notice that all entries are at the same menu level and, consequently, all appear in the preview menu bar as menus.

5. Now use the Menu Editor's Promote and Demote toolbar buttons to specify whether each menu item is a child or a sibling of the previous menu item. For example:



With the cursor on Bottom, click on the Demote toolbar button to make Bottom both a sibling of Top and a child of Edit.

6. Click on the "Bottom" 3-D bar, and choose the Insert Separator command from the Edit Insert Item menu.

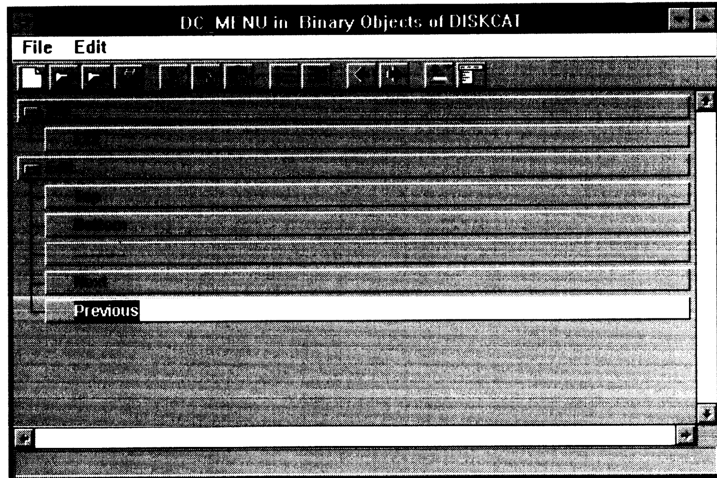
   The Workbench will insert a separator between "Bottom" and "Next."

7. Finally, click on each menu item's 3-D bar and define its various properties in the appropriate Properties window.

Tip: Use the Menu Editor's Auto Layout feature to quickly add standard, predefined File, Edit, View, and Help menus to your menu structure. Each predefined menu has *default* menu items. The functions to be called, however, must be added manually in the Menu Item Properties window. For detailed information, see Adding Predefined Menus in the *Workbench User Guide.*

Previewing Your Menus

After you have finished entering all of the menu items and saved your work, the Menu Editor should look like this:



If you click on "File" in the preview menu bar, you can view the File pull-down menu you just created:

Now click on "Edit." Your Edit pull-down menu should look as follows:

Edit	
Top	Ctrl+HOME
Bottom	Ctrl+END
Next	Ctrl+N
Previous	Ctrl+P

All of the actions that were just entered into the Menu Item Properties window represent the actions that will take place when that particular menu item is selected by the user. For example, when the Top menu item is chosen, the DiskDispTest() function will be run after performing a DBGOTOP().

The CA-Clipper Workbench has now created all the source code that is needed for a complete menu structure. Your pull-down menu structure has been saved as a menu entity in the Binary Objects Module, and a DC_MENU_PRG module with the source code has been generated automatically.

In the next lesson you will learn how to use the DB Server Editor to create a data server entity.

LESSON 3: Creating a Data Server and Field Specs

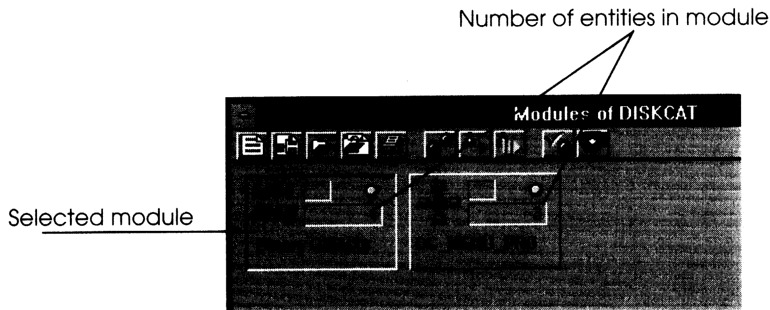
Before you create forms for the DISKCAT application, it is necessary to first create a data server based on an existing database (.dbf) file. This will allow your application to access the appropriate data. To do this, you will use the DB Server Editor.

Defining a Data Server

To access the DB Server Editor and define a data server:

1. First, close the Menu Editor window by clicking on the system menu and selecting the Close command.

   You are returned to the Module Browser for DISKCAT (which now contains the DC_MENU_PRG module created in the previous lesson):



   Notice that the Binary Objects module contains one entity—the menu entity you created in the previous lesson. Also notice that the program module, DC_MENU_PRG, contains three entities representing its text block and the DC_MENUCREATE and DC_MENUTEST functions.

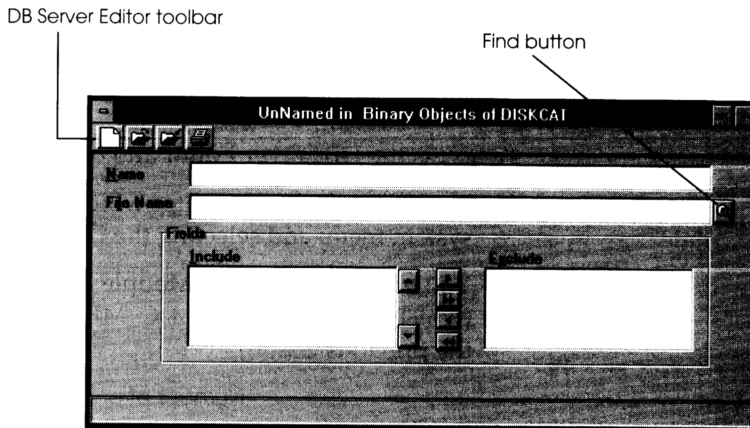
2. With the Binary Objects module selected, click on the New Entity toolbar button.

   A local pop-up menu appears.

3. Choose the DB Server Editor command from the local pop-up menu.

   Alternatively, select the DB Server Editor command from the Tools menu.

   The DB Server Editor appears:



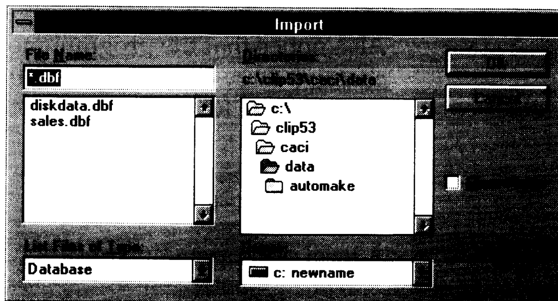
4. Leave the Name edit control blank.

   The name of the associated .dbf file will be used by default.

5. To import an existing database file, click on the Find button to the right of the File Name edit control.

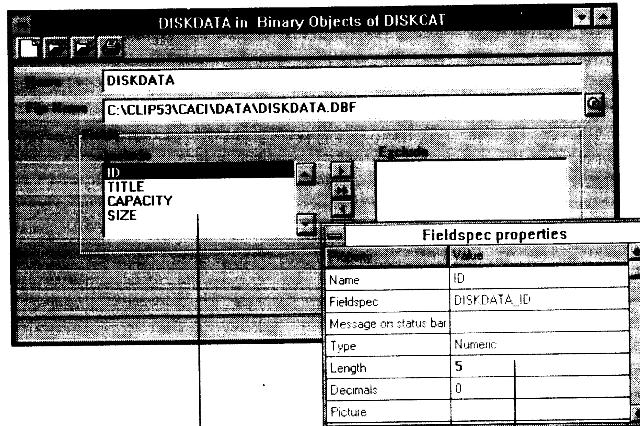
   Alternatively, select the Import command from the File menu.

   The Import dialog box appears:



6. Select the C:\CLIP53\CACI\DATA directory, and double-click on the Diskdata.dbf file.

The Import dialog box closes, and the File Name edit control is updated with the name and path of the associated database file. By default, the name of the .dbf file is also used for the name of the data server, as well as the DB Server Editor's title bar:



FieldSpec Properties window

Database fields

Notice that all of the fields in the Diskdata.dbf database file structure are listed in the Include list box, and that a floating FieldSpec Properties window is displayed.

Note: You can exclude a field from your data server definition, if you wish, by clicking on it and then clicking on the Right arrow button. This will move the specified field to the Exclude list box.

Defining Field Properties

Your next step is to define field properties for each of the four database fields listed in the Include list box. These properties will be used later by the Form Editor. When the source code for the application is generated, all of these details will be appropriately incorporated.

To specify each field's properties, beginning with the ID field, perform the following steps:

ID Field

1. Click on the ID field name in the Include list box.

   The FieldSpec Properties window for the specified field appears:

   Fieldspec properties

Property	Value
Name	ID
Fieldspec	DISKDATA_ID
Message on status bar	.
Type	Numeric
Length	5
Decimals	0
Picture	
Min length	
Required	No
Minimum	
Maximum	
Validation	

   Notice that some property values are already filled in, such as Name, FieldSpec, and Type. The property values of the existing database fields default automatically to the corresponding property value cells in the Properties window of the data server that is being associated with that database file. For example, the default FieldSpec name is *<DataServerName>*_*<FieldName>*, or DISKDATA_ID in this instance.

2. Click in the Message on Status Bar value cell and type **Unique ID number for this disk**.

3. Replace the default value "5" with **6** in the Length value cell.

   Note: All default values can be modified.

4. In the Picture value cell, enter **99,999**.

   This is a standard Xbase picture clause used to format this field.

5. In the Minimum value cell, type **1**.

6. In the Maximum value cell, type **99999**.

TITLE Field

In a similar manner, let's define the properties for the TITLE field:

1. Click on the TITLE field name in the Include list box.

   A new FieldSpec Properties window appears.

2. Click in the Message on Status Bar value cell and type **Title for this disk**.

The other default properties are fine as is.

CAPACITY Field

Now, let's enter the properties for the CAPACITY field:

1. Click on the CAPACITY field name in the Include list box.

   A new FieldSpec Properties window appears.

2. Click in the Message on Status Bar value cell and type **Storage capacity for this disk**.

The other default properties are fine as is.

SIZE Field

Finally, enter the properties for the SIZE field:

1. Click on the SIZE field name in the Include list box.

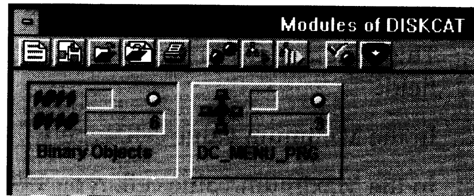
   A new FieldSpec Properties window appears.

2. Click in the Message on Status Bar value cell and type **Size of disk**.

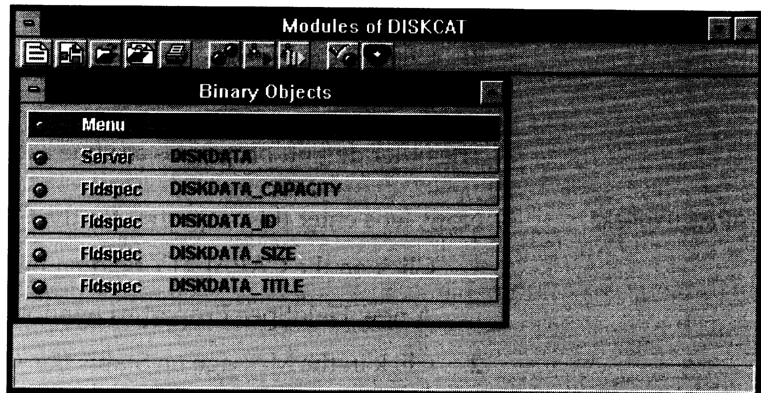
3. In the Validation value cell, type **DISKDATA->SIZE = '3 1/2' .OR. DISKDATA->SIZE = '5 1/4'**.

4. Click on the Save toolbar button.

You may now close the DB Server Editor by clicking on its system menu and selecting the Close command. You are returned to the Module Browser for DISKCAT:



Notice that there are now six entities in the Binary Objects module. If you open the Binary Objects module by double-clicking on it, you can view these entities in a module-specific Entity Browser:



In addition to the menu entity you created earlier for your menu structure, there is now a *server* entity for the DISKCAT data server just created and four *field spec* entities for its corresponding fields.

Now let's create a form for DISKCAT that utilizes the DISKCAT data server you just defined.

LESSON 4: Using the Form Editor

You will now create a data form that will be used for data entry in your DISKCAT application. To do this, you will use the Form Editor.

Creating a Form

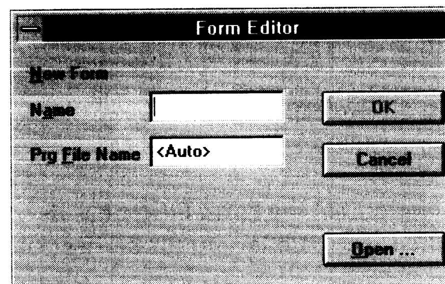
To access the Form Editor and create a data form:

1. First, close any window(s) still open from the last lesson.

   You are returned to the Module Browser for DISKCAT.

2. From within the Binary Objects module, click on the New Entity toolbar button and choose the Form Editor command from the local pop-up menu.

   The Form Editor dialog box appears:



3. In the Name edit control, enter **DiskDisplay**.

   The form name will be used in the source code generated by the Form Editor to create a form entity and the .prg and .ch files and their associated modules, so it must not conflict with other entity names in your application.

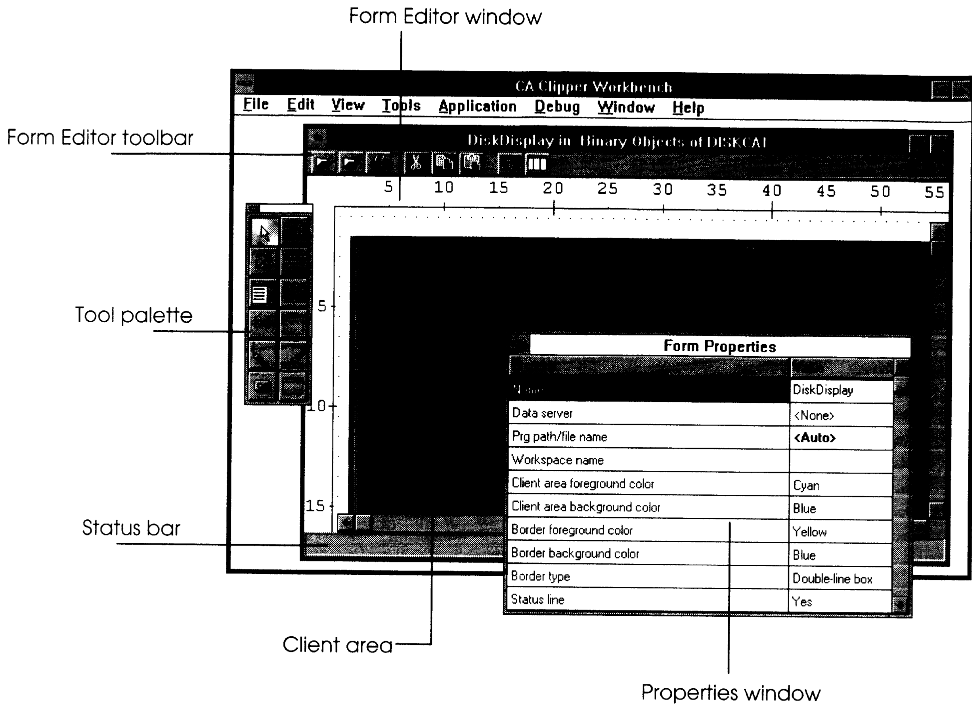
   Note: Blank spaces within the name will be converted to underscores.

4. Choose OK.

   The Form Editor appears. Let's look at its components.

Workspace Overview

The Form Editor is the primary workspace in the Workbench for creating, viewing, and modifying forms:



Notice that the Form Editor has its own toolbar, status bar, client area, tool palette, and Properties window.

Note: For printing reproduction purposes, from this point forward black will be used for the foreground and white for the background for forms, fields, text, and lines in our sample application, DISKCAT, instead of the default CA-Clipper DOS colors.

Specifying Form Properties

Now you need to define two of the data form's properties—its caption and the data server with which it is to be associated.

Caption Property

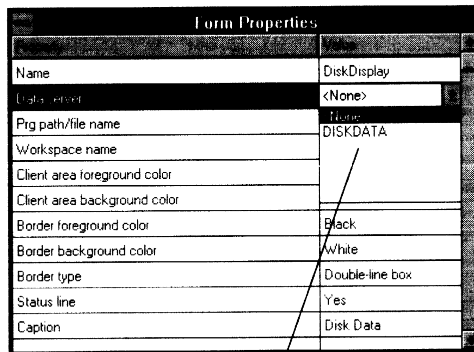
1. Scroll down to the Caption property.
2. Click on the value cell for the Caption property and replace the default caption, "MyForm," with **Disk Data**.
3. Press Enter.

   "Disk Data" now appears as the caption of your form.

Data Server Property

4. Click on the value cell for the Data Server property in the Form Properties window.

   A drop-down list box appears:



Drop-down list of available data servers

5. Select DISKDATA from the drop-down list box of available data servers.

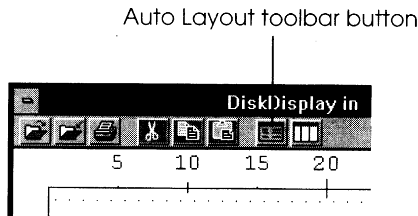
   This is the data server you created in the previous lesson.

Using Auto Layout to Place Controls

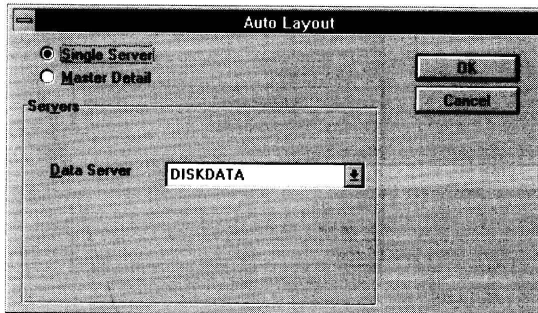
Now let's place some GUI controls on your form. Since you have just assigned the DISKDATA data server to this form, you can use the Form Editor's Auto Layout feature to create a single-line edit control automatically for each of the four fields in Diskdata.dbf. To do this:

1. Click on the Auto Layout toolbar button:

Auto Layout toolbar button



The Auto Layout dialog box appears:

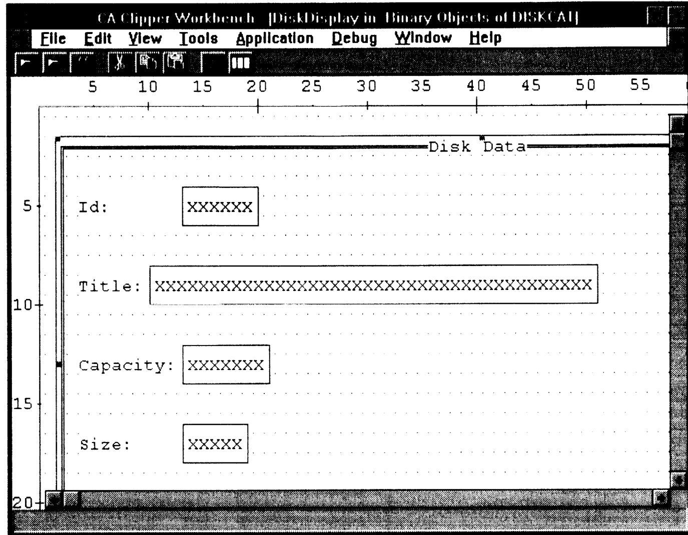


2. Choose the Single Server option.

   Notice that, by default, the DISKDATA data server is already selected in the Data Server edit control.

3. Choose OK.

   The Form Editor should now look like this:



Specifying Control Properties

Now let's modify these single-line edit controls, starting with the ID and the TITLE fields which need only minor changes.

1. Click on the single-line edit control labeled "Id:"

   The Entry Field Properties window replaces the Form Properties window:

   Entry Field Properties

Property	Value
Name	Id
Caption	Id:
Message on status bar	Unique ID number for this disk
Field spec	DISKDATA_ID
Entry type	Edit
Caption foreground color	Black
Caption background color	White
Active edit foreground color	Bright Red
Active edit background color	White
Inactive edit foreground color	Black
Inactive edit background color	White
Frame around entry area	Yes
Memo persistent	No
Default value for field	
Additional entry field properties	

2. In the value cell for the Default Value for Field property, enter **DISKDATA->ID**.

   This ensures that the data that is in the database is displayed immediately when the form is opened.

3. Click on the single-line edit control labeled "Title."

   A new Entry Field Properties window appears.

4. In the value cell for the Default Value for Field property, enter **DISKDATA->TITLE**.

Changing Control Types

The Capacity and Size fields require more extensive changes. First of all, each provides the user with a limited number of options to choose from, so you should change these single-line edit controls to more appropriate control types.

Creating a List Box

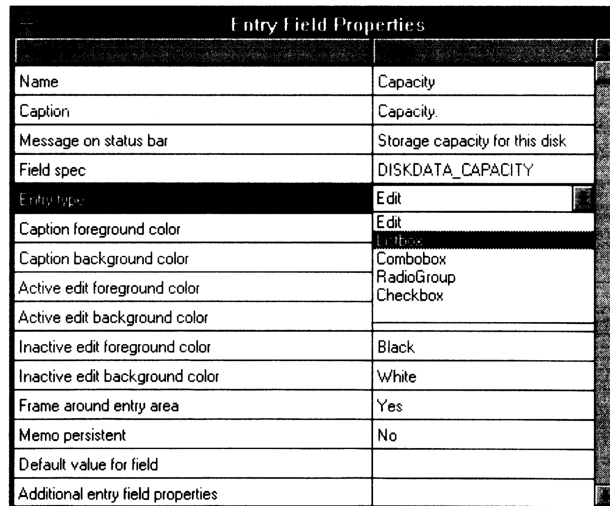
To change the Capacity single-line edit control to a list box:

1. Click on the single-line edit control labeled "Capacity."

   A new Entry Field Properties window appears.

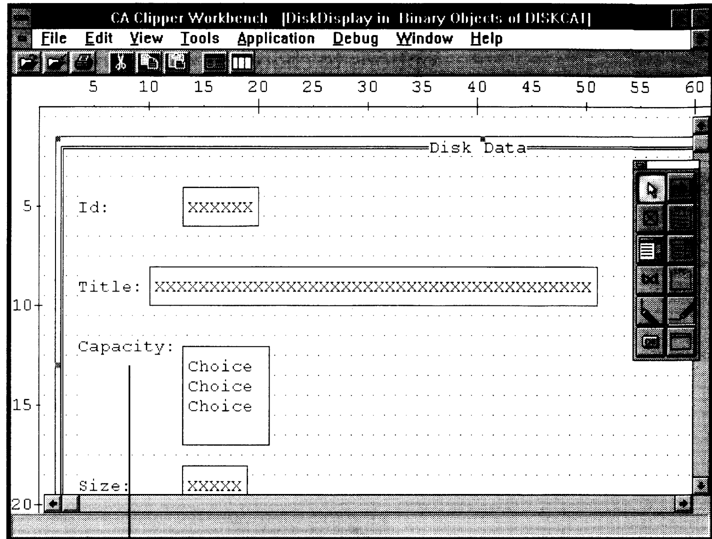
2. Click on the value cell for the Entry Type Properties.

   A drop-down list box appears with a choice of available controls:



3. Select Listbox from this list box and press Enter.

   The Form Editor now reflects the changed control:



Capacity field is now a list box control

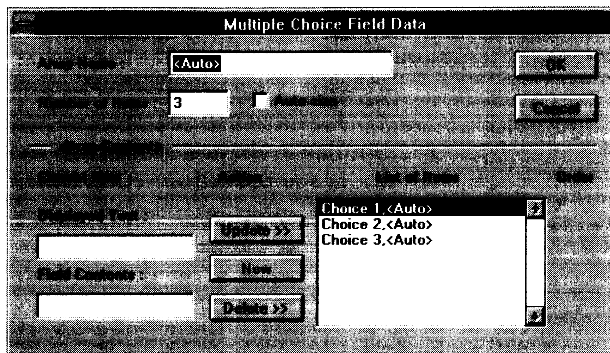
4. In the value cell for the Default Value for Field property, enter **ALLTRIM(DISKDATA->CAPACITY)**.

Customizing a List Box

Since you want specific capacities as your actual choices for the Capacities list box, you need to customize this particular list box control. To do this:

1. Click on the value cell for the Additional Entry Field Properties, and then click on the Ellipsis button that is displayed.

   The Multiple Choice Field Data dialog box appears:

   This dialog box allows you to update the contents of your newly created list box and define an array for it.

2. In the Number of Items edit control, replace the default value of "3" with **6**.

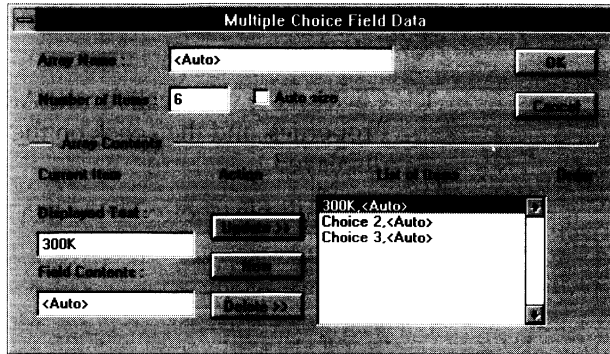
3. Select Choice 1 in the List of Items list box by clicking on it.

   "Choice 1" now appears in the Displayed Text edit control, and "<Auto>" appears in Field Contents, allowing you to change the actual text of these items.

4. Replace "Choice 1" with **300K** and leave the Field Contents as <Auto>.

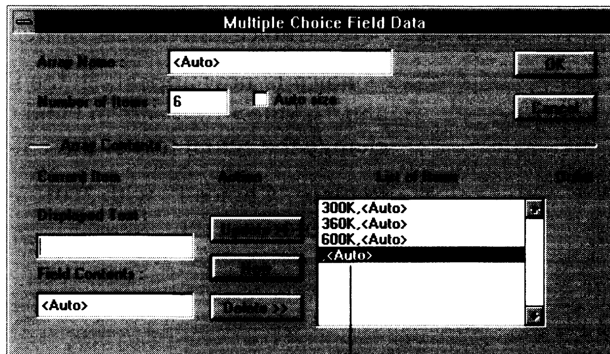
5. Choose the Update >> push button.

   The List of Items list box is updated:



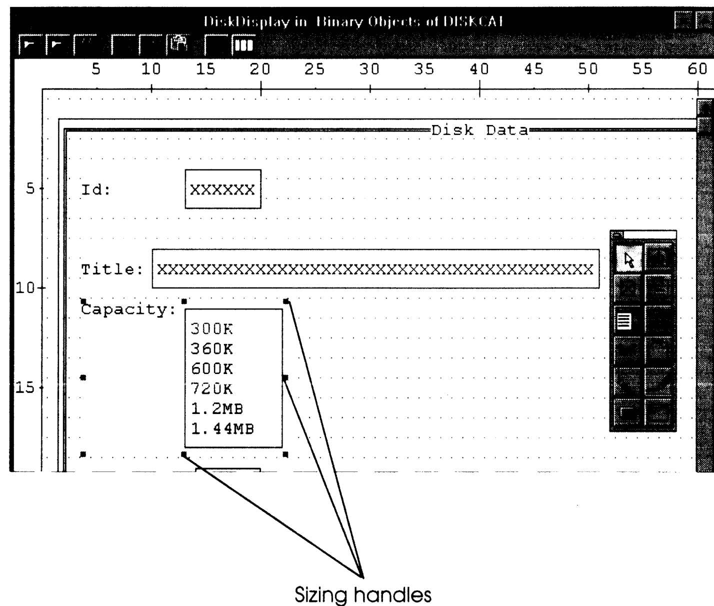
6. Similarly, change "Choice 2" to **360K** and leave the Field Contents as <Auto>.
7. Choose the Update >> push button.
8. Change "Choice 3" to **600K** and leave the Field Contents as <Auto>.
9. Choose Update >> again.
10. Now click on the New push button.

    A fourth array item (,<Auto>) is created and added to the List of Items list box:



11. Type **720K** in the blank Displayed Text edit control and leave the Field Contents as <Auto>.
12. Choose the Update >> push button.
13. Similarly, create two new array items and define their text as **1.2MB** and **1.44MB**, respectively.
14. Choose OK.

The Form Editor should now look something like this:



Sizing handles

The Capacity field now displays a scrollable six-line list box that presents all of the different disk capacities that are currently available. The user of the DISKCAT application will be able to click on one of these choices, thereby greatly enhancing the speed with which he or she works.

Tip: In order to save space, you may change the size of the list box that is displayed in the form by clicking once on it to activate it, and then clicking on the *sizing handles* to resize the list box control. Do not worry if all six items do not appear in the form. When the code for this form is generated, the CA-Clipper Workbench will automatically place a scroll bar on the list box.

Creating a Radio Button Group

Now you need to display a list of user choices for the size of a disk. In this case, the best type of control is not a list box but rather a radio button group that presents the user with mutually exclusive choices. The user will be able to select one size or the other, but not both.

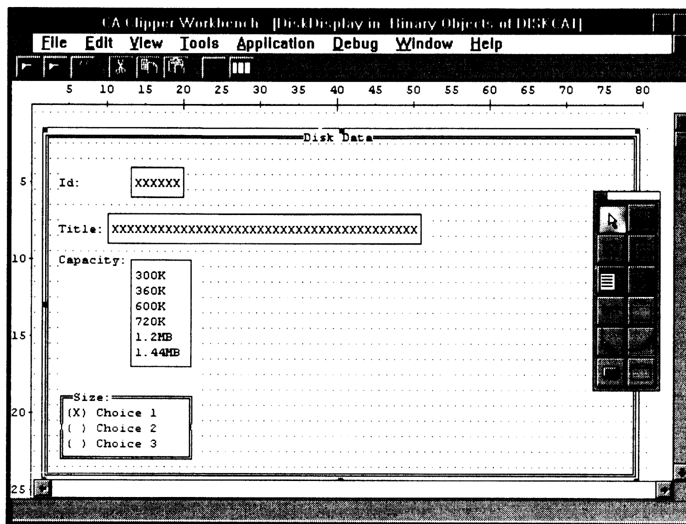
To change the Size single-line edit control to a radio button group:

1. Click on the Size single-line edit control.

   A new Entry Field Properties window appears.

2. Click on the Entry Type value cell.
3. Select RadioGroup from the drop-down list box that appears.
4. In the value cell for the Default Value for Field property and press Enter. Type **ALLTRIM(DISKDATA->SIZE).**

The Form Editor now reflects the changed control for the Size field:



By default, CA-Clipper provides three choices for a radio button group. You will learn how to delete choices and customize the radio button group in the next section.

Customizing a Radio Button Group

Since you want to present specific sizes as your actual choices for the Size radio button group, you need to customize this particular control. To do this:

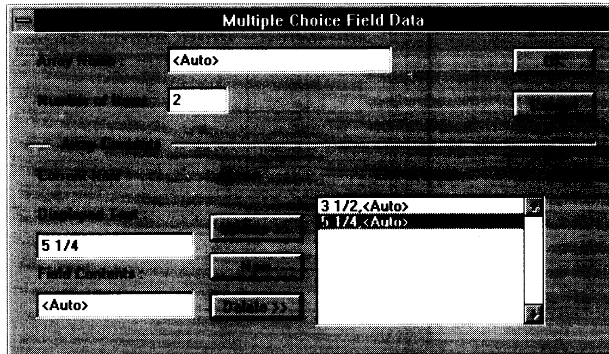
1. Click on the value cell for the Additional Entry Field Properties, and then click on the Ellipsis button that is displayed.

   Again the Multiple Choice Field Data dialog box appears.

2. In the Number of Items edit control, change the value to **2**.
3. Select Choice 1 in the List of Items list box by clicking on it.
4. Replace "Choice 1" in the Displayed Text edit control with **3 1/2** and leave the Field Contents as <Auto>.
5. Choose the Update >> push button.

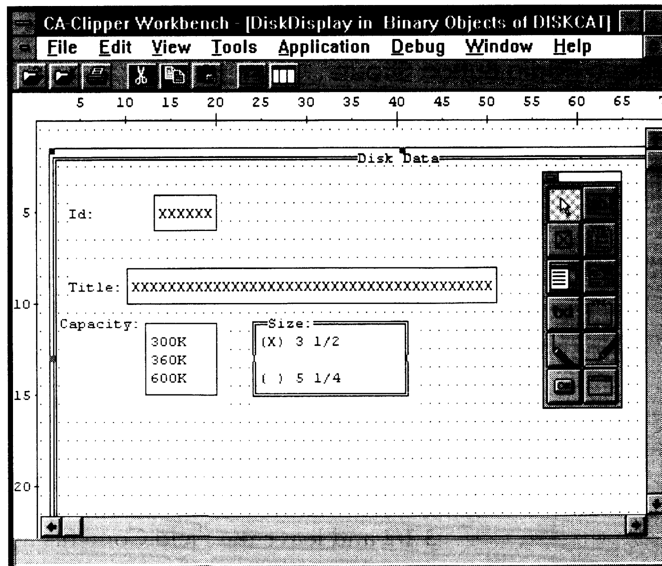
6. Similarly, change "Choice 2" to **5 1/4** and leave the Field Contents as <Auto>.
7. Choose Update >> again.
8. Select Choice 3 in the List of Items list box by clicking on it.
9. Choose the Delete >> push button.

   Choice 3 is removed from the List of Items list box:



10. Choose OK.

The Form Editor should now look something like this:



The Size field now displays a radio button group that presents the user of DISKCAT with two disk sizes, allowing him or her to toggle between these choices.

Tip: You can rearrange the controls on a form using the Form Editor's drag-and-drop feature. Simply place the mouse pointer anywhere on a control, press the left mouse button and hold it down, drag the mouse to move the control to the desired location, and release the mouse button.

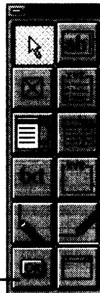
Adding Push Buttons to a Form

To complete the Disk Data form, you need to create push buttons that allow the user of DISKCAT to skip records, create new records, etc. This is done by manually adding push button controls to a form using the tool palette.

Save Button

For example, let us create a Save push button:

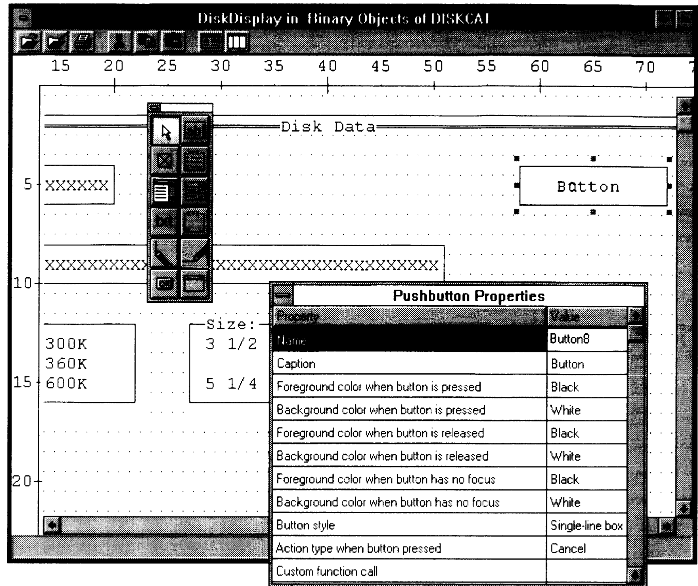
1. Click on the Push Button icon in the tool palette:



Note: See the "Using the Form Editor" chapter in the *Workbench User Guide* for more detailed information about the tool palette and its various icons.

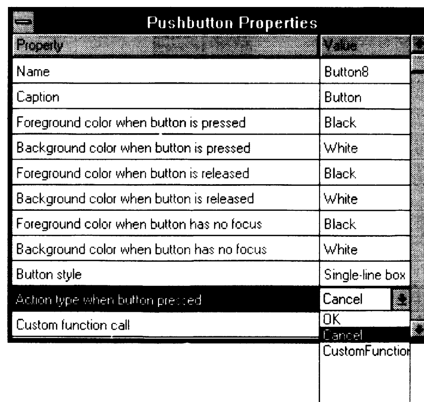
2. Drag-and-drop a push button control at the top-right of the form.

A push button control and its corresponding Push Button Properties window appears:



3. In the Caption value cell, replace "Button" with **Save**.
4. Click on the value cell for the Action Type When Button Pressed property.

   A drop-down list box appears with available actions:



5. Select CustomFunction from this list box, replacing the default Cancel action.
6. Enter **DiskDispSave(aInfo)** in the Custom Function Call value cell.

Done Button

Let's create another button:

1. Drag-and-drop another push button control placing it beneath the first.
2. In the Caption value cell of the new Properties window, replace "Button" with **Done**.
3. Click on the value cell for the Action Type When Button Pressed property.
4. Select OK from this list box.

New Button

Now you need a New button:

1. Drag-and-drop another push button control onto the form.
2. In the Caption value cell of the new Properties window, replace "Button" with **New**.
3. Click on the value cell for the Action Type When Button Pressed property.
4. Select CustomFunction from this list box.
5. Enter **DiskDispAppend(aInfo)** in the Custom Function Call value cell.

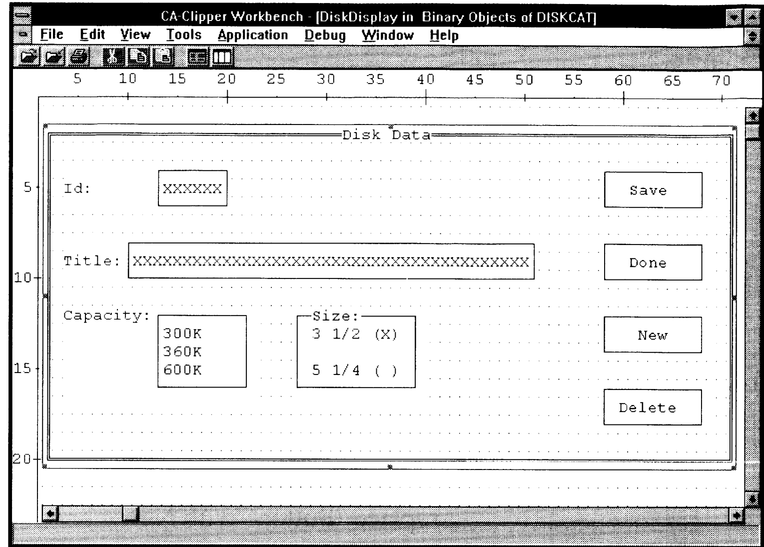
Delete Button

Finally, let's create a Delete button:

1. Drag-and-drop the last push button control onto the form.
2. In the Caption value cell of the new Properties window, replace "Button" with **Delete**.
3. Click on the value cell for the Action Type When Button Pressed property.
4. Select CustomFunction from this list box.

5. Enter **DBDELETE( )** in the Custom Function Call value cell.
6. Click on the Save toolbar button.

Your Disk Data form is now complete:



Before leaving the Form Editor, however, you need to specify the control order for tabbing.

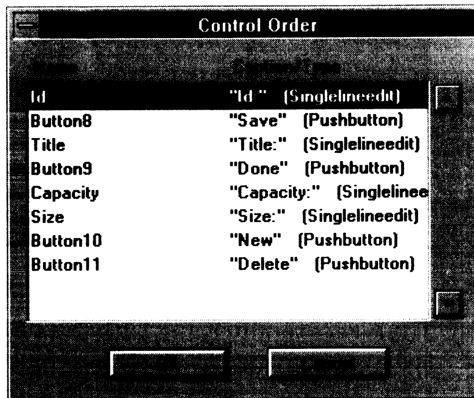
Specifying Control Order

When you add GUI controls to a window, the Workbench automatically creates a tabbing order for the controls. You can, however, rearrange the control order to improve the logical flow of the cursor movement.

For example, in your DISKCAT application you would probably want the user to tab through all of the edit controls first and then through the push buttons. To do this:

1. Select the Control Order command from the Edit menu.

   The Control Order dialog box appears:

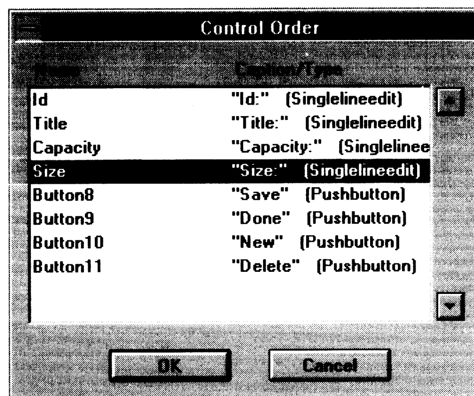


   This dialog box displays the names of all the controls, as well as their captions and control types, in *tab stop* order as they appear in the source code.

2. Click on a control name (for example, Title).

3. Use the Up or Down arrow button to change its control order appropriately.

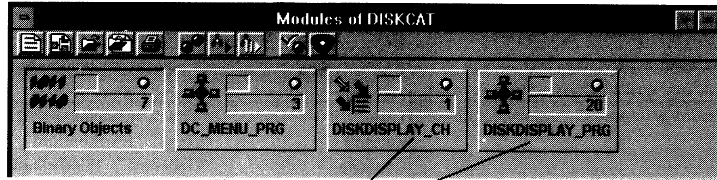
4. Repeat steps 1 and 2 until the Control Order dialog box reflects the desired order:



5. Choose OK.

   You are returned to the Form Editor.

Let's close the Form Editor and return to the Module Browser for DISKCAT. Notice that there are now seven entities in the Binary Objects module:



New modules

The newest entity is the Disk Data form you just created. Notice also that there are two new modules: DISKDISPLAY_CH and DISKDISPLAY_PRG. Each of these modules contains the source code that was generated by the Form Editor when you saved the Disk Data form and exited the Form Editor. The DISKDISPLAY_CH module contains the header, or include, file and the DISKDISPLAY_PRG module contains the procedural code.

LESSON 5: Importing an External File

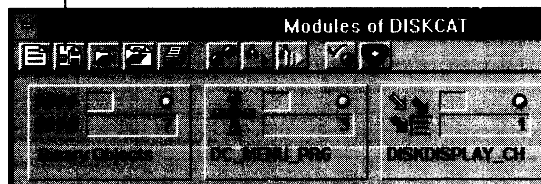
Now you need to add a function that will allow you to use the database file, Diskdata.dbf. Such a function has already been predefined for you and can be found in a program (.prg) file in the C:\CLIP53\CACI\DATA directory. All you need to do is import this program file, DC_UTILS.PRG, as a separate module in the DISKCAT application.

Importing DC_UTILS.PRG

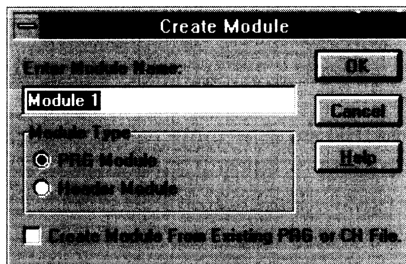
To import the DC_UTILS.PRG external file:

1. First create a new module by clicking on the New Module toolbar button:

   New Module toolbar button

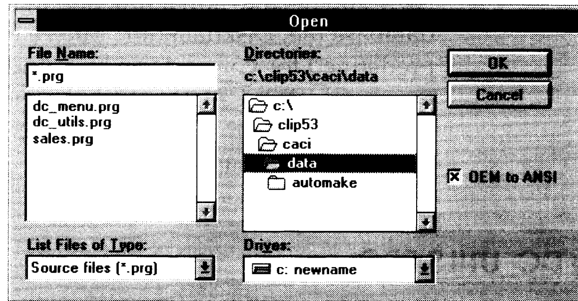
   The Create Module dialog box appears:

2. Select PRG Module from the Module Type group box.
3. To import an existing file, select the Create Module From Existing PRG or CH File check box.

4. Choose OK.

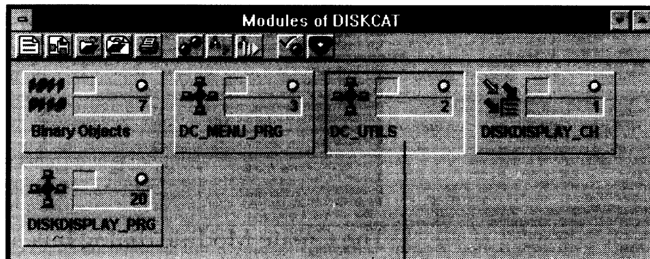
   A standard Open dialog box appears:



5. Select the **DC_UTILS.PRG** file, which is located in the DATA subdirectory.

6. Choose OK.

   A new module is automatically created for this imported file:



Note that the file name of the imported program (.prg) file has been used by default for the name of the new module. You can, of course, rename it, if you wish, using the File Rename Module menu command.

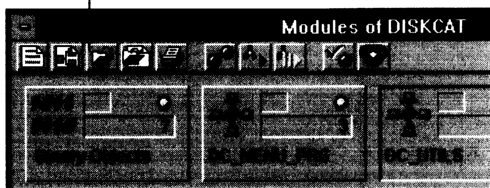
Using an Entity Browser to View DC_UTILS

The Workbench's Entity Browser allows you to view the contents of the new module you just created for the imported file. To view the module's contents:

Accessing an Entity Browser

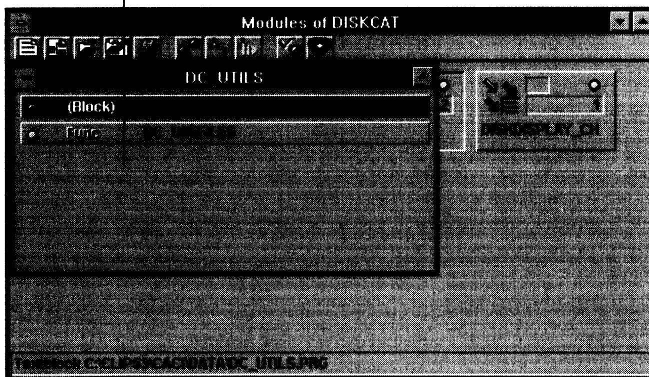
1. Either double-click on the DC_UTILS module button or click on the Open toolbar button:

   Open toolbar button

   A module-specific Entity Browser for the DC_UTILS module appears:

   Entity Browser (module-specific)

   Notice that it displays two entities: a text block and the DC_USEFILES function.

Accessing the Source Code Editor

2. To view the actual code within the module, double-click on the 3-D bar labeled DC_USEFILES. The Source Code Editor appears displaying the text block and the DC_USEFILES() function:

Text block

```
Textblock C:\CLIP53\CACI\DATA\DC_UTILS.PRG
// Created using CA-Clipper Workbench

FUNCTION DC_UseFiles()

DBUSEAREA(.T., "DBFNTX", "DISKDATA")
IF NETERR()
    RETURN .F.
ENDIF
DBSETINDEX("DISKID.NTX")

DBSELECTAREA(1)

RETURN .T.
```

Function

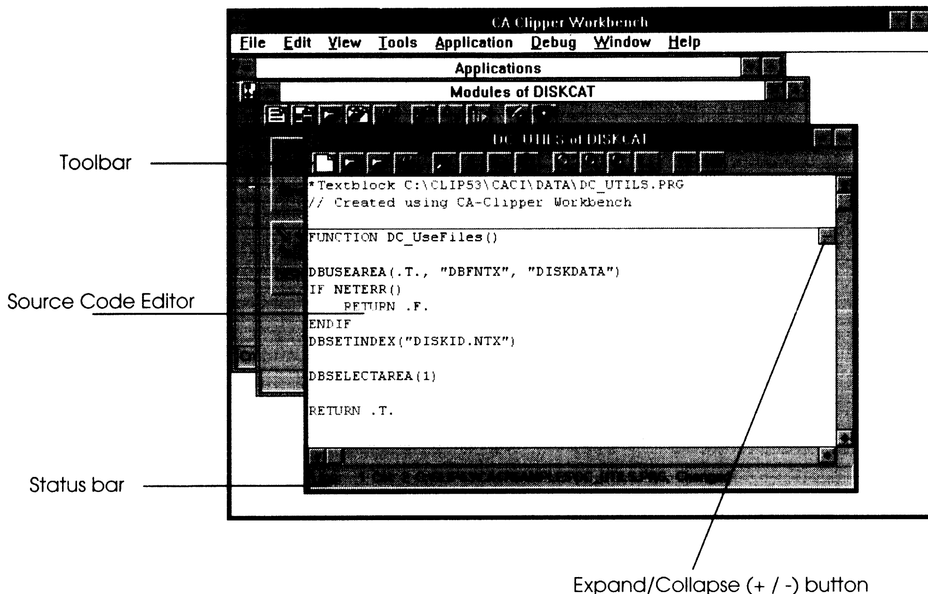
Now that you have just learned how to access the Source Code Editor, in addition to importing an external file, let us proceed to the next lesson, which will show you how to modify code using the Source Code Editor.

LESSON 6: Using the Source Code Editor

Now you will make several slight modifications to the source code that was generated earlier by the Menu and Form Editors. First, you will amend code in the DC_MENU_PRG module, using the Source Code Editor, so that the menu entity can access the database files. Then you will do the same for the form entity.

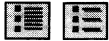
Workspace Overview

The Source Code Editor is the primary workspace in the Workbench for viewing, creating, and modifying source code entities:



Notice that the Source Code Editor, like the other editors, has its own toolbar and status bar.

Also note that, by default, each entity is displayed in its entirety in the Source Code Editor and has a – button to the right of its declaration statement. Clicking on this – button *collapses* the entity, so that only its declaration statement can be seen, and changes the – button to a + button. Conversely, clicking on a + button *expands* an individual entity.



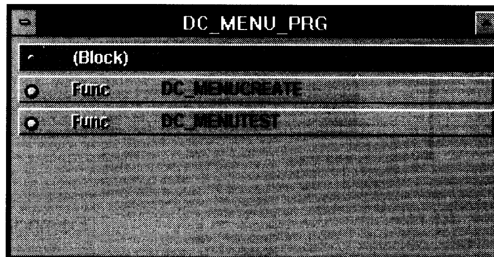
To expand or collapse *all* entities, use the Source Code Editor's Expand and Collapse toolbar buttons, respectively.

Modifying the DC_MENUTest() Function

To modify the source code in the DC_MENU_PRG module:

1. First, close the Source Code Editor for the DC_UTILS module if you have not already done so. Also, close the Entity Browser for DC_UTILS. You are brought back to your DISKCAT Module Browser.
2. Double-click on the DC_MENU_PRG module button in the DISKCAT Module Browser.

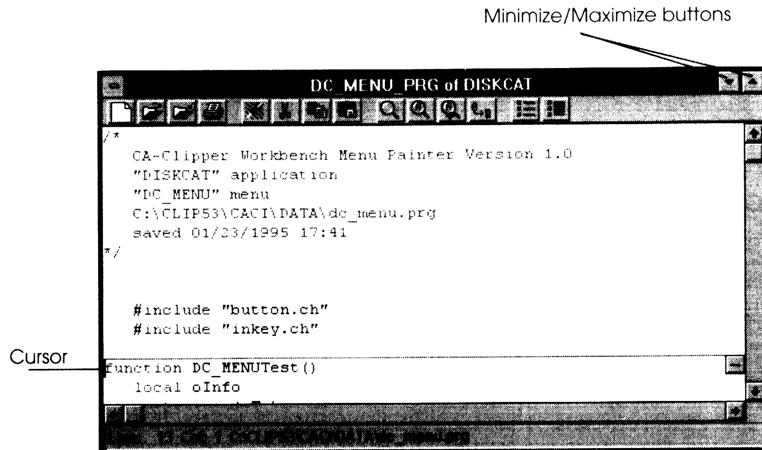
   A module-specific Entity Browser appears:

   Notice that it displays three entities: a text block and two functions, DC_MENUCREATE and DC_MENUTEST.

3. Double-click on the DC_MENUTEST entity.

   The Source Code Editor is now opened with the cursor flashing at the beginning of the DC_MENUTest() function:



Note: Use the scroll bar to view the entire function. You can also click on the Source Code Editor's Maximize button to increase the viewing area.

4. Modify the DC_MENUTest() function, adding the following text in **bold**:

```
Function DC_MENUTest()
    // Change oInfo to public so that the form can
    // use it.
    // LOCAL oInfo
    PUBLIC oInfo
    SET ( _SET_EVENTMASK, INKEY_ALL )
    // Function added to use the database files.
    IF DC_UseFiles()

      MSETCURSOR(.T.)
      CLS
      MEMVAR->oInfo==DC_MENUCreate()

      // Add DO WHILE loop to stay in Menu loop
      // until menu ID 999 is chosen.
      DO WHILE;
        MENUMODAL (MEMVAR->oInfo,1,24,1,79,"R/W");
                    <> 999
      ENDDO

    ENDIF

    RETURN ( NIL )
```

5. Click on the Save toolbar button.

   Your changes to the DC_MENUTest() function in the DC_MENU_PRG module are saved.

6. Close the Source Code Editor.

   You are returned to the DISKCAT Module Browser.

Modifying the DiskDispTest() Function

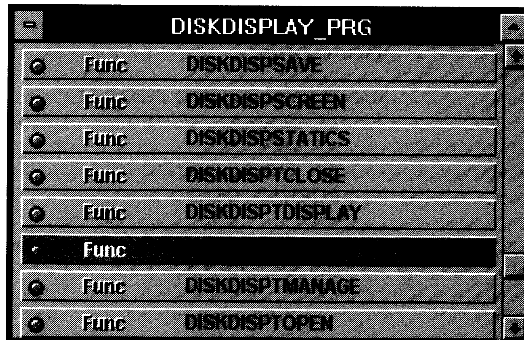
Next you will amend code in the DISKDISPLAY_PRG module, so that the form entity can access the database files.

To modify the source code in the DISKDISPLAY_PRG module:

1. Double-click on the DISKDISPLAY_PRG module button in the DISKCAT Module Browser.

   A module-specific Entity Browser appears.

2. Scroll through the function entities, double-clicking on **DISKDISPTEST** when you locate it:



   The Source Code Editor appears.

3. Modify the DiskDispTest() function, adding the following text in **bold**:

```
FUNCTION DiskDispTest()
   LOCAL aTest := {}, lMouse

   lMouse := MSETCURSOR( .T. )
   SET ( _SET_SCOREBOARD, .F. )
   SET ( _SET_EVENTMASK, INKEY_ALL )

   // Comment the following line because it will
   // be taken care of in the call to
   // DC_UseFiles() function below.
   // DBUSEAREA(.F.,,'DISKDATA')

   aTest := DiskDispOpen ()
   DiskDispInit ( aTest )
   DiskDispLoad ( aTest )

   // Add MEMVAR->oInfo to allow menu items to
   // be chosen when the form is open.
   DiskDispEdit ( aTest , MEMVAR->oInfo )

   DiskDispClose ( aTest )

   MSETCURSOR( lMouse )

RETURN ( NIL )
```

4. Click on the Save toolbar button.

   Your changes to the DiskDispTest() function in the DISKDISPLAY_PRG module are saved.

Modifying the DiskDispAppend() Function

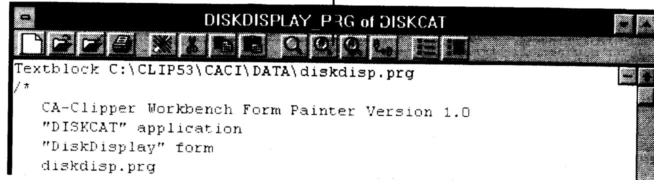
Once these changes have been made to these two source code files, you are just about ready to build the application.

However, you may want to make one extra change before closing the Source Code Editor. If, after clicking on the New button, you would like the information on the form to be set back to an empty record, you will need to amend the DiskDispAppend() function in this same module.

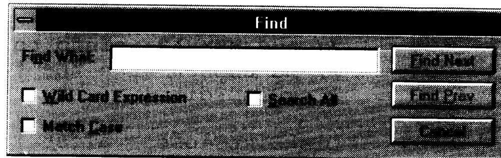
To modify DiskDispAppend() while still in the Source Code Editor:

1. Click on the Source Code Editor's Find button:

Find toolbar button



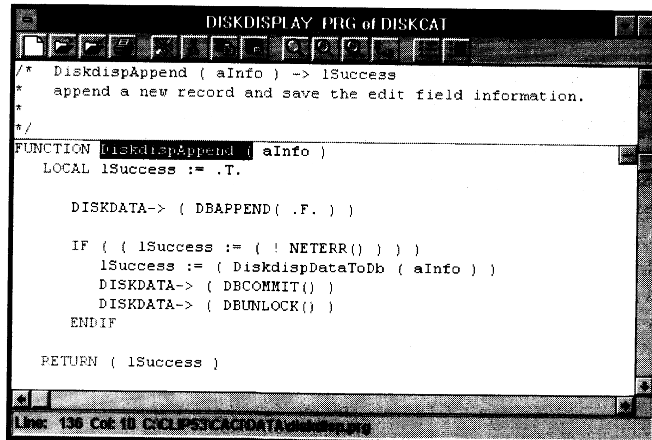
A standard Find dialog box appears:



2. Enter **DiskDispAppend(** in the Find What edit control.
3. Deselect the Match Case check box if it is selected.

4. Choose the Find Next push button.

   You are brought to the correct line of code:



5. Comment out the following line by inserting two forward slashes (//) like so:

```
// lSuccess := ( DiskDispDataToDb ( aInfo ) )
```

6. Then add the following text on the next line:

```
lSuccess := ( DiskDispLoad ( aInfo ) )
```

   This last change will take the information from the current record in the database (a new, empty record) and place it in the form instead of placing the form information into the database.

7. Click on the Save toolbar button.
8. Close the Source Code Editor.
9. Close the DISKCAT Module Browser.
10. Close the Entity Browser. You are returned to the Application Browser.

LESSON 7: Compiler and Linker Options

Now that your entire disk catalog application has been created, all that is left to do is to set up the various compiler and linker options for DISKCAT.

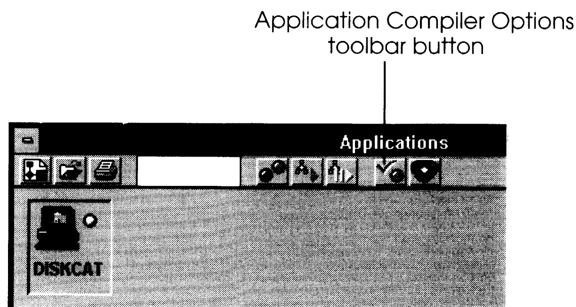
In the CA-Clipper Workbench, you will typically specify the most commonly used compiler and linker settings as permanent system-wide defaults. However, you can also control the manner in which an individual application is built and linked using *application-specific* compiler and linker options instead. That is what you will do now for DISKCAT.

Setting Compiler Options at the Application Level

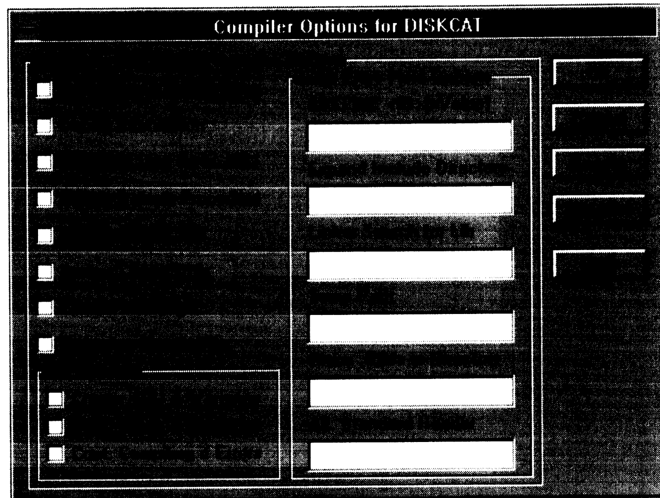
CA-Clipper compiler options allow you to include debug information, select auto MEMVAR declaration, enable warnings, expand the include directory, add additional libraries to the default library search requests, specify a temporary path, etc. for your applications. However, your requirements for DISKCAT are minimal.

To set compiler options specifically for DISKCAT:

1. Choose the Compiler Options command from the Application menu, or click on the Application Compiler Options toolbar button:



The application-specific Compiler Options dialog box appears:



2. Deselect any check boxes in this dialog box that may be already checked, or *selected*.

   Note: See Setting Compiler Options in the "Working in the Workbench" chapter of the *Workbench User Guide* for a full discussion of the available compiler options.

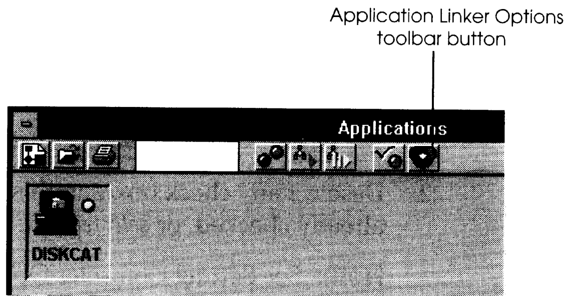
3. Choose the OK push button to save any changes you may have made.

Setting Linker Options at the Application Level

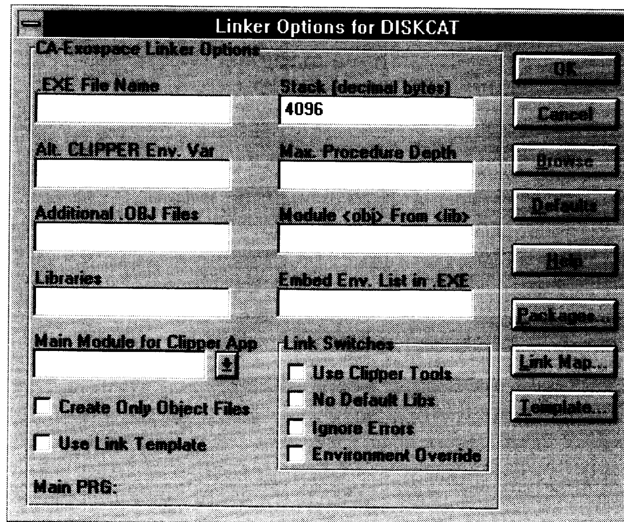
Linker options in CA-Clipper include specifying an executable file name, adding additional object files and libraries, utilizing built-in linker packages, creating link map files, using linker templates, etc. However, once again your requirements for DISKCAT are minimal.

To set linker options specifically for DISKCAT:

1. Choose the Linker Options command from the Application menu, or click on the Application Linker Options toolbar button:



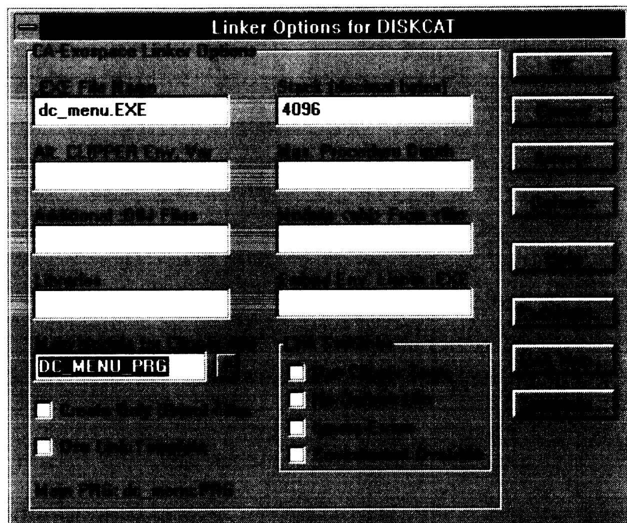
The application-specific Linker Options dialog box appears:



Note: See Setting Linker Options in the "Working in the Workbench" chapter of the *Workbench User Guide* for a full discussion of the available linker options.

2. Select **DC_MENU_PRG** from the drop-down list box for Main Module for Clipper App.

   Notice that when this item is selected, the .EXE File Name edit control and the Main PRG is automatically filled in with a file name based on the name of the main module with the default .EXE extension:



3. In the .EXE File Name edit control, replace DC_MENU.EXE with **DISKCAT.EXE**.

   This is the file name of the executable file to be generated by the linker.

4. Choose OK.

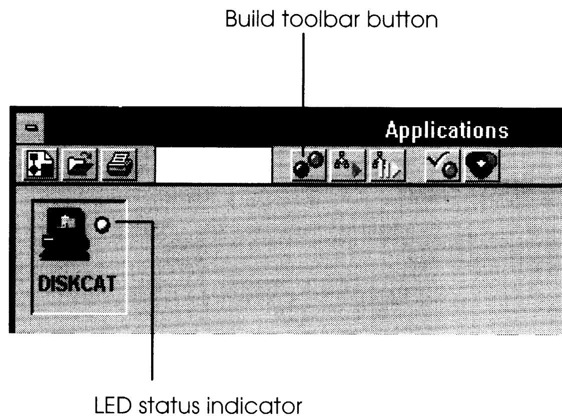
   Your changes are saved.

LESSON 8: Building and Executing DISKCAT

At this point, you have completed the tutorial, but you still have not seen your disk catalog application in action. So let's build the new DISKCAT application now.

Building an Application

To build the application, choose the Build command from the Application menu, or simply click the Build toolbar button:



It's that easy. The application button's LED status indicator changes from yellow to green, indicating that the build is successfully completed. (The Application Browser's status bar also informs you that the build is successful.)

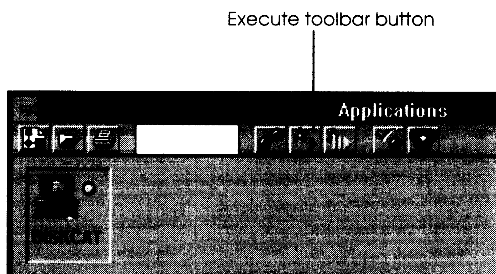
Executing an Application

Executing DISKCAT

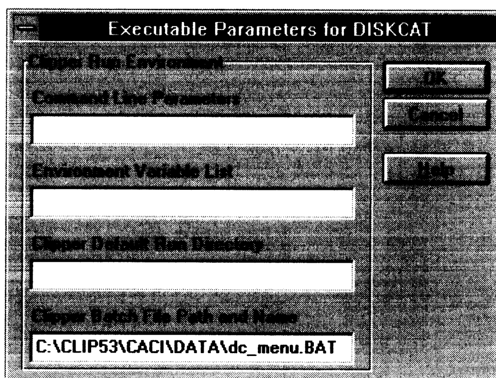
After the build is successfully completed, you can run the DISKCAT application.

To execute the application:

1. Choose the Execute command from the Application menu, or simply click the Execute toolbar button:



The Executable Parameters dialog box appears:

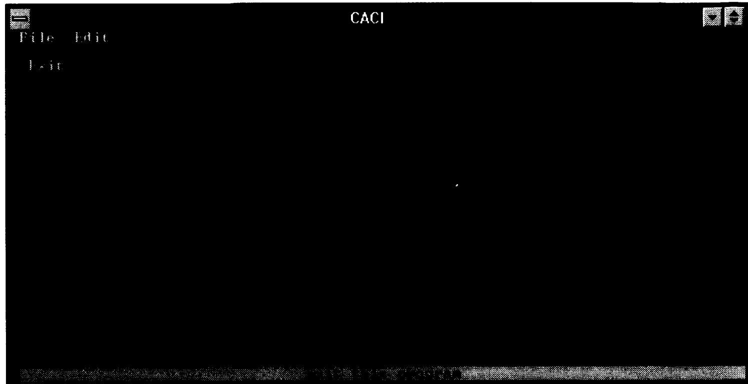


Notice that the Clipper Batch File Path and Name edit control is automatically filled in.

Note: See Executing an Application in the "Working in the Workbench" chapter of the *Workbench User Guide* for more information about defining your CA-Clipper runtime environment.

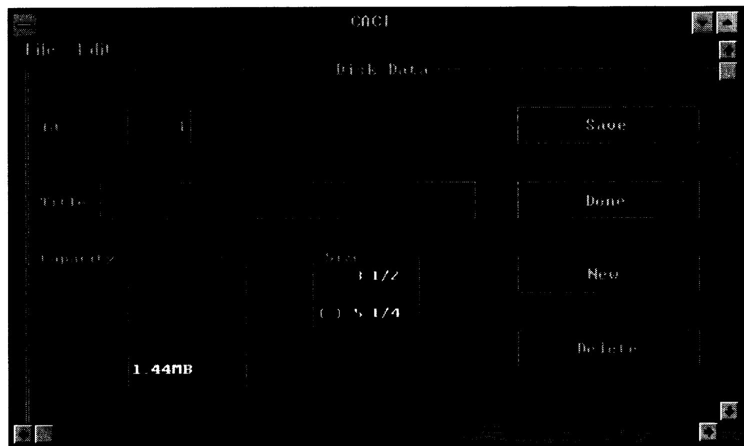
2. Choose OK.

The CACI DOS screen appears:



Viewing DISKCAT

If you now select one of the Edit menu commands, you can view and use DISKCAT:



Summary

In this tutorial, you have seen how simple it is to create a CA-Clipper application using the various tools provided by the CA-Clipper Workbench. You have used the DB Server Editor to prepare data files to be accessed by a form, the Menu Editor to create a complete menu structure, and the Form Editor to generate data entry forms. You have also used the Source Code Editor to amend the code that is generated automatically by the Workbench when you created the data server, menu, and form entities. Finally, you have set the compiler and linker options for an application, built it, and executed it.

The next chapter walks you through the process of debugging a simple program using the CA-Clipper debugger.

Chapter 5
Debugging a Simple Program

In This Chapter

The program that you created in the previous chapter was purposefully over-simplified to demonstrate the features of the Workbench without bogging you down in the details of the CA-Clipper language. The programs that you will write, however, are likely to be much more complex and will, therefore, run the additional risk of errors.

CA-Clipper offers two tools to help you find and correct errors in your applications: the Workbench debugger, and the DOS-level debugger. This chapter walks you through the process of debugging another simple program using the DOS debugger, CLD.LIB. It is intended only as an introduction to the debugger and some of its many features. For a full description of all the DOS-level debugger features, refer to the "CA-Clipper Debugger—CLD.LIB" chapter in the *Programming and Utilities Guide.*

Note: For more information about the Workbench debugger, refer to "Debugging Your Applications" in the *Workbench User Guide.*

In this chapter, you will learn how to:

- Prepare a program for debugging
- Execute the program using the CA-Clipper debugger
- Isolate erroneous lines of code in the Code window
- Use various debugger features to analyze errors
- Correct any errors and continue the debugging process
- Make temporary corrections without leaving the debugger

Requirements for This Chapter

To do the exercises in this chapter, you must update your AUTOEXEC.BAT and CONFIG.SYS files according to the instructions in the section entitled Before Moving On in the "Installation" chapter of this guide. Do not forget that when changes are made to these files, you must reboot your computer for the changes to take effect.

You will also need to access the database file, Sales.dbf, and the index file, Salesrep.ntx, in order to run the sample program. The fields required in Sales.dbf are Salesrep (character) and Amount (numeric), and the index key for Salesrep.ntx is Salesrep.

Finally, this chapter assumes that all files you create (.prg, .dbf, .ntx, etc.) are in the current directory. Otherwise, the utility programs and libraries, such as PE.EXE and CLD.LIB, and the sample program that you will create may not be able to locate the needed files.

Note: The Sales.dbf and Salesrep.ntx files can be found in the CLIP53\CACI\SAMPLES subdirectory, so be sure to move these sample files to the same directory where you create the sample debugging program.

Creating a Program with Errors

To begin this exercise, create the sample debugging program, Dossales.prg, exactly as shown in the steps below.

You can either use the CA-Clipper program editor, PE.EXE, or any text editor or word processor as long as it can create a standard text file with a .prg extension. Here we will use PE.EXE:

1. Type **PE Dossales.**
2. Enter the following lines of CA-Clipper code in the PE window:

```
LOCAL cOldSalesman, nTotalAmount
USE Sales INDEX Salesrep NEW
DO WHILE !EOF()
    cOldSalesman := Sales->Sales
    DO WHILE cOldSalesman = Sales->Salesrep
            ? Sales->Salesrep, Sales->Amount
            nTotalAmount += Sales->Amount
            SKIP
    ENDDO
    ? "Total: ", nTotalAmount, "for", cOldSalesman
ENDDO
CLOSE Sales
```

3. Save Dossales.prg to disk by pressing Ctrl+W.

The Dossales.prg program processes an indexed database file from beginning to end, accumulating the contents of a numeric field named Amount for each Salesrep in the database. The program displays each Amount in the database file and then displays the accumulated amount when a new Salesrep is encountered. After all records are processed, the program closes the database file and quits.

Preparing Your Program for Execution

As in the previous chapter, you need to compile and link the program you have written in order to execute it. This time, however, you are going to assume that the program has errors in it and that a certain amount of debugging will be required. To create DOSSALES.EXE:

1. Access the drive and directory containing Dossales.prg.
2. Type **CLIPPER Dossales /B** and press Enter.

   The /B compiler option includes debugging information in the resulting object file that is necessary for the debugger to access and run the program.
3. Type **EXOSPACE FILE DOSSALES, CLD.LIB** and press Enter.

Executing Your Program Using the Debugger

Although the nature of this particular error may be obvious to you, there may be other errors in the program, and using the debugger can make diagnosing and correcting errors much easier. To execute DOSSALES.EXE using the debugger:

1. Type **Dossales** at the DOS prompt.

   The debugger is displayed on the screen with the contents of Dossales.prg in the Code window as shown below:

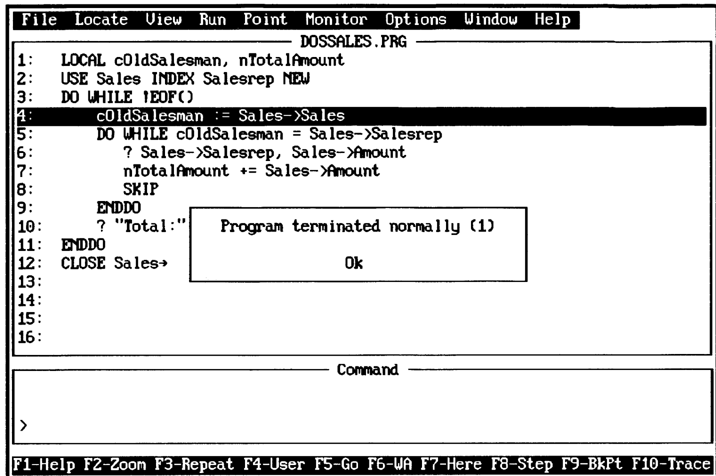
```
File  Locate  View  Run  Point  Monitor  Options  Window  Help
                         DOSSALES.PRG
1:    LOCAL cOldSalesman, nTotalAmount
2:    USE Sales INDEX Salesrep NEW
3:    DO WHILE !EOF()
4:       cOldSalesman := Sales->Sales
5:       DO WHILE cOldSalesman = Sales->Salesrep
6:          ? Sales->Salesrep, Sales->Amount
7:          nTotalAmount += Sales->Amount
8:          SKIP
9:       ENDDO
10:      ? "Total:", nTotalAmount, "for", cOldSalesman
11:   ENDDO
12:   CLOSE Sales→
13:
14:
15:
16:
                         Command
>
F1-Help F2-Zoom F3-Repeat F4-User F5-Go F6-WA F7-Here F8-Step F9-BkPt F10-Trace
```

2. Select Alt+R, G (Run Go) to execute the program in Run Mode.

   Program execution begins and an error dialog is displayed, the same as when DOSSALES.EXE was executed as a stand-alone program.

3. Press Return to select Quit from the error dialog box.

   Control returns to the debugger with the offending line of code highlighted in the Code window as shown below. The dialog box shows that the program terminated with one (1) error:



4. Press Return to select OK from this dialog box.

In the debugger, using the accelerator key—the Alt key followed by a letter—opens the menu beginning with that letter. Once the menu is open, each option has a highlighted letter in its name that you can press to quickly select that option. Thus, pressing Alt+R, G in the steps above selects the Run Go menu option. Rather than having you type the equivalent menu command, the instructions in this lesson always use the accelerator key combination for selecting menu options.

Note: Some debugger menu options have an associated shortcut key that you can press instead of using the accelerator key combination. Shortcut key combinations are listed to the right of menu options for which they are available. Thus, pressing F5 in the steps above is the equivalent of pressing Alt+R, G. If you are interested, you can explore the debugger menus for shortcut key combinations during the next session.

Isolating the First Error

The debugger helps you isolate many errors by highlighting the line of code that was executing when the error occurred which, in most cases, is very revealing. In this example, the offending line of code is cOldSalesman := Sales->Sales, and the error message is "Variable does not exist: SALES." This tells you that your program does not recognize the field name Sales->Sales.

To view the Sales.dbf database structure and determine the correct field name:

1. Select Alt+V, W (View Workareas).

   The View Workareas window opens on the screen to display information about the active database file and work area as shown below:

Notice that the far right window pane shows a list of all the field names in Sales.dbf. As you can easily see, there is no field named Sales. Thus, there is a typographical error in the line of code that caused the error. The field name Sales should have been Salesrep.

2. Press Esc.

   The View Workareas window closes and returns control to the main debugger screen.

Correcting the First Error

Now that you know which line of code caused the error and exactly what the error is, you are ready to correct it. However, the debugger does not allow you to make changes directly to the source code. Instead, you must exit the debugger and repeat the edit, compile, link, and execute cycle to correct the error.

To correct this error:

1. Select Alt+X (File Exit) to exit the debugger.
2. Type **PE Dossales**.
3. Use the arrow keys to move the cursor to the end of the line of code reading cOldSalesman := Sales->Sales.
4. Type **rep** to change the Sales field name to Salesrep.
5. Press Ctrl+W to save the changes and exit.

Starting the Next Debugger Session

Now, you are ready to compile, link, and execute to determine if the correction worked and if there are other errors in the program. By now, you are probably becoming familiar with these steps.

To repeat the compile, link, and execute cycle, type the following, pressing Enter at the end of each line:

1. **CLIPPER Dossales /B**.
2. **EXOSPACE FILE DOSSALES, CLD.LIB**.
3. **Dossales**.

Again, you use the /B option and execute the program, DOSSALES.EXE, using the debugger.

Isolating the Next Error

You can see in the Code window that the line of code that caused the error is now changed to reflect the edit that you just made. The debugger Code window with the new version of Dossales.prg is shown in the figure below:

```
File  Locate  View  Run  Point  Monitor  Options  Window  Help
                         DOSSALES.PRG
1:   LOCAL cOldSalesman, nTotalAmount
2:   USE Sales INDEX Salesrep NEW
3:   DO WHILE !EOF()
4:      cOldSalesman := Sales->Salesrep
5:      DO WHILE cOldSalesman = Sales->Salesrep
6:         ? Sales->Salesrep, Sales->Amount
7:         nTotalAmount += Sales->Amount
8:         SKIP
9:      ENDDO
10:     ? "Total:", nTotalAmount, "for", cOldSalesman
11:  ENDDO
12:  CLOSE Sales→
13:
14:
15:
16:
                         Command
>
F1-Help F2-Zoom F3-Repeat F4-User F5-Go F6-WA F7-Here F8-Step F9-BkPt F10-Trace
```

To make sure that the change worked and to see if there are any more errors in the program, you are going to run the new version of the program using the debugger.

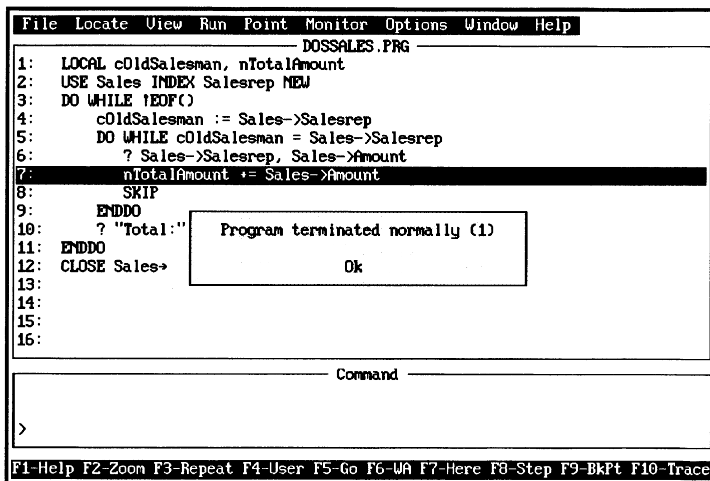
To execute the program in Run Mode:

1. Press Alt+R, G (Run Go).

   This time, the program results in another error, "Argument error: +."

2. Press Return to select Quit.

   The debugger screen is redisplayed as shown in the figure below. The dialog box in the center of the screen tells you that one (1) error was encountered during execution:



3. Press Return to select OK and close the dialog box.

   Now you can clearly see the offending line of code highlighted in the Code window. The line of code reads: nTotalAmount += Sales->Amount. Notice that the program made it past the line of code that you corrected earlier, indicating that the fix worked.

Setting Watchpoints

This line of code seems okay when you first look at it. It simply adds the amount in a database field to a variable and assigns the new value back to the variable. To find out what the problem is, you can set up watchpoints for the two variables in this line of code and view their values as program execution proceeds. To do this:

1. Press Alt+X to exit the debugger, and then type **Dossales** at the DOS prompt to restart the debugger.
2. Press Alt+P, W (Point Watchpoint).

   A dialog box opens on the screen in which you are expected to enter an expression.
3. Type **nTotalAmount** and press Return.

   The Watch window opens at the top of the debugger screen with nTotalAmount displayed as a watchpoint.
4. Press Alt+P, W (Point Watchpoint).
5. Type **Sales->Amount** and press Return.

Sales->Amount is added to the Watch window as a second watchpoint as shown below:

```
 File  Locate  View  Run  Point  Monitor  Options  Window  Help
                                  Watch
0) nTotalAmou <wp, Local, U>: NIL
1) Sales->Amount <wp, U>: Undefined

                                DOSSALES.PRG                          ↑↓
1:   LOCAL cOldSalesman, nTotalAmount
2:   USE Sales INDEX Salesrep NEW
3:   DO WHILE !EOF()
4:       cOldSalesman := Sales->Salesrep
5:       DO WHILE cOldSalesman = Sales->Salesrep
6:          ? Sales->Salesrep, Sales->Amount
7:          nTotalAmount += Sales->Amount
8:          SKIP
9:       ENDDO
10:      ? "Total:", nTotalAmount, "for", cOldSalesman
11:  ENDDO
12:  CLOSE Sales→

                                  Command


>
F1-Help F2-Zoom F3-Repeat F4-User F5-Go F6-WA F7-Here F8-Step F9-BkPt F10-Trace
```

Using Single Step Mode

So that you can see what happens to the value of the two watchpoints, you are going to execute the program one line at a time and carefully view the contents of the Watch window after each step. To do this:

- Press F8 five (5) times to step through the program until the offending line of code is highlighted.

  Using the Watch window as shown below you can see that although Sales->Amount contains a valid numeric value, nTotalAmount is undefined. Given this information, you can look in the Code window and recognize that nTotalAmount was never initialized:

```
File  Locate  View  Run  Point  Monitor  Options  Window  Help
                              Watch
0) nTotalAmou <wp, Local, U>: NIL
1) Sales->Amount <wp, N>:     240.00

                           DOSSALES.PRG
1:   LOCAL cOldSalesman, nTotalAmount
2:   USE Sales INDEX Salesrep NEW
3:   DO WHILE !EOF()
4:      cOldSalesman := Sales->Salesrep
5:      DO WHILE cOldSalesman = Sales->Salesrep
6:         ? Sales->Salesrep, Sales->Amount
7:         nTotalAmount += Sales->Amount
8:         SKIP
9:      ENDDO
10:     ? "Total:", nTotalAmount, "for", cOldSalesman
11:  ENDDO
12:  CLOSE Sales→

                              Command

>
F1-Help F2-Zoom F3-Repeat F4-User F5-Go F6-WA F7-Here F8-Step F9-BkPt F10-Trace
```

Implementing a Temporary Fix

There are several ways to correct this error, including a few temporary fixes that you can make without having to leave the debugger. In order to get past this error and see if there are any more errors in the program, you need to initialize the value of nTotalAmount to zero and continue executing the program.

To initialize the value of nTotalAmount:

1. Type **? nTotalAmount := 0** in the Command window.
2. Press Enter.

Continuing Program Execution

At this point, you can run the remainder of the application in Run Mode. The fix that you just implemented is not logically correct because it introduces an accumulation error in the total for each Salesrep, but it will get you past the argument error and allow you to view your program results. Later on, you can correct the error in your program.

To execute the program in Run Mode starting at the current line of code:

1. Press Alt+R, G (Run Go).

   The debugger again shows the termination dialog box, but this time no errors were encountered during execution.
2. Press Return to close the dialog box.

Viewing Your Program Results

Since you are using the debugger to execute your program, you do not automatically see your program output. To display the results of your program:

1. Press Alt+V, A (View App Screen).

   You will see the contents of your Sales.dbf file broken out by Salesrep with a total each time the Salesrep changes. You can also see the accumulation error with the total since nTotalAmount was not reset to zero for each Salesrep.

2. Press any key to return to the debugger screen.

Correcting the Final Error

Now that you have isolated this error and have determined that there are no more, you will want to make a permanent correction in the Dossales.prg file and regenerate DOSSALES.EXE for one final test. To correct this error:

1. Press Alt+X (File Exit) to exit the debugger.
2. Type **PE Dossales**.
3. Use the directional keys to move the cursor to the end of the line of code reading cOldSalesman := Sales->Salesrep.
4. Press Ins to switch to insert mode.
5. Press Return to open a blank line below the current one.
6. Type **nTotalAmount := 0** to initialize this variable.
7. Press Ctrl+W to save the changes and exit.

Because of the placement of the initialization statement, nTotalAmount will be reset to zero for each Salesrep and the accumulation error that was introduced with the temporary fix in the debugger should disappear.

Running the Final Test

Now, you are ready to compile, link, and execute to determine if the correction worked. Since the last debugger run indicated zero errors, you can be relatively certain that you have corrected all the errors, so this time you will compile without the /B option to save space in the resulting .EXE file.

To repeat the compile, link, and execute cycle, type the following, pressing Enter at the end of each line:

1. **CLIPPER Dossales**.
2. **EXOSPACE FILE DOSSALES, CLD.LIB**.
3. **Dossales**.

This time, there should be no error messages, and the resulting program output should show the correct totals for each Salesrep.

Summary

In this chapter, you learned how to create an executable file that can be executed using the CA-Clipper DOS-level debugger. You learned how to use two of the debugger execution modes, Run and Single Step, to run your application. You learned how to switch back and forth between the application and the debugger screen. You learned how to view a database file structure to determine if a field name had been misused and how to watch several variables to determine which one was causing a particular error. Finally, you learned how to initialize a variable within the debugger to get through the rest of the application and potentially save an edit, compile, link, and execute cycle.

With these basics in hand, you are now ready to use the debugger for your own applications. To discover all of the debugger features that are available to you, refer to the "CA-Clipper Debugger—CLD.LIB" chapter of the *Programming and Utilities Guide.*

Chapter 6
Creating Data Structures

In This Chapter

In CA-Clipper, you can create database files using DBU.EXE, an application written in CA-Clipper designed to provide you with an interactive database design environment. It allows you to build database files, add data to the files, browse existing data, create and attach index files, and construct views within a completely menu-driven system. The Database Utility (DBU) also provides many other features and is fully documented in the "Database Utility—DBU.EXE" chapter of the *Programming and Utilities Guide*; this chapter introduces you to some of its basic features. If you are interested in building data structures to use with your CA-Clipper programs, you should read this chapter and further explore DBU on your own.

In this chapter, you will learn how to:

- Invoke DBU
- Navigate and make selections
- Create a database and an index file
- Add and edit data with the Browse screen
- Leave the DBU utility when you are finished

Requirements for This Chapter

In order to perform the exercises in this chapter, you must first install DBU.EXE.

If you selected the DBU Utility option at the beginning of the installation procedure, this executable file was compiled and linked for you. In the default configuration, DBU.EXE is installed in the \CLIP53\BIN directory.

If you elected not to install DBU.EXE, you must run the installation procedure again and select these options.

Your AUTOEXEC.BAT and CONFIG.SYS files must also be updated according to the instructions in the section entitled Before Moving On in the "Installation" chapter in this guide. Do not forget that any time changes are made to these files, you must reboot your computer for the changes to take effect.

Executing DBU

To use DBU, type **DBU** from the DOS prompt followed by Return. The program loads into memory and leaves you on the main screen.

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.

Navigation

The Menu Bar

Once in DBU, you will notice a menu bar across the top of the screen providing you with several options. Pressing the function key listed directly above a menu name drops the menu down allowing you to select from any of its available items. Unavailable menu items are grayed.

When a menu is open, the Up arrow and Down arrow keys move the highlight bar within the menu and Return selects the currently highlighted item. A shortcut for navigating to a particular menu item is typing the first letter of its name. The Right arrow and Left arrow keys move to the next and previous menus on the menu bar. When no menu is open, you can move the highlight bar around on the main screen with any of the directional keys.

The Main Screen

Without actually using the menu bar, you can perform many basic operations on the main screen. For instance, pressing Return when the Files area is highlighted allows you to open an existing database (.dbf) file either by selecting a file name from a picklist or by typing a file name directly into the dialog box. This action on the main screen is equivalent to selecting F2 Open Database from the menu bar. Up to six database files can be open at one time in DBU. Once a database file is open, the Indexes and Fields areas become available for your use.

Highlighting the Indexes area allows you to open up to seven index files per database file. Opening index files is accomplished in the same manner as opening a database file. This action is equivalent to selecting F2 Open Index from the menu bar. To open more than one index file, simply move the highlight bar below the current index file name and press Return.

Opening database and index files with Return closes any other open files that happen to occupy the currently highlighted slot. Ins, on the other hand, allows you to open a file in much the same way as Return, except that it pushes the other files down one slot and if necessary, closing the file occupying the last slot to make room for the new one. Del closes the file in the currently highlighted slot.

Highlighting the Fields area allows you to delete, insert, and overwrite fields in the field list for the active database file. To delete a field from the list, highlight the field name and press Del.

To insert a field, highlight the proper location for the field and press Ins. A field dialog box opens from which you select the field to insert from a picklist. 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 Return. Similar to pressing Ins, this action opens the field dialog box; however, pressing Return causes the field that you select to overwrite the current field—thus, 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.

In addition to the actions that you can perform on the main screen, the screen serves as an indicator of your current status. It gives a visual indication of the open database files with the index files and field list underneath.

Creating a Database File

Many applications written in CA-Clipper are centered around manipulating one or more database files.

A database file is a collection of rows and columns. The columns, called fields, define the format of each row, or record, in the database file in terms of the following attributes:

- Field name (e.g., Birthday, Zip, Amount)
- Type of data (e.g., Date, Character, Number)
- Maximum width
- Number of decimals (for numbers only)

In DBU, you define the above field attributes for each field that you want to include in the database file. This process is known as creating a database file structure. Later, you can add and change records in the file according to the field attributes that you define.

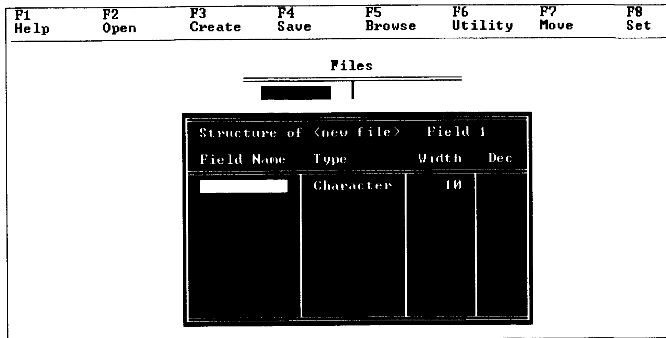
To create a new database file, perform the following steps:

1. Select F3 Create.

   The Create Database menu item is highlighted.

2. Press Return.

   A field box opens on the main screen as shown:



In the field box, you enter the field definitions that make up the database file structure. For this exercise, the file structure you will create is as follows:

Field Name	**Type**	**Length**	**Decimals**
Lname	Character	30	
Fname	Character	30	
Age	Numeric	2	0
Notes	Memo	10	

To create this file:

1. Type the name of a field and press Return.
2. Type the first letter of the data type.
3. Type the length, followed by the Down arrow key.
4. Repeat steps 1–3 for each of the other field definitions.

   For the Notes field, you cannot enter the field length since it is predefined by the system.

When you are done, your screen should look something like this:

```
F1     F2     F3      F4     F5      F6       F7     F8
Help   Open   Create  Save   Browse  Utility  Move   Set

                      Files

Structure of <new file>   Field 4

Field Name    Type        Width   Dec

LNAME         Character     30
FNAME         Character     30
AGE           Numeric        2     0
NOTES         Memo          10
```

To save the file structure, perform the following steps:

1. Select F4 Save.

   The Save Struct menu item is highlighted.

2. Press Return.

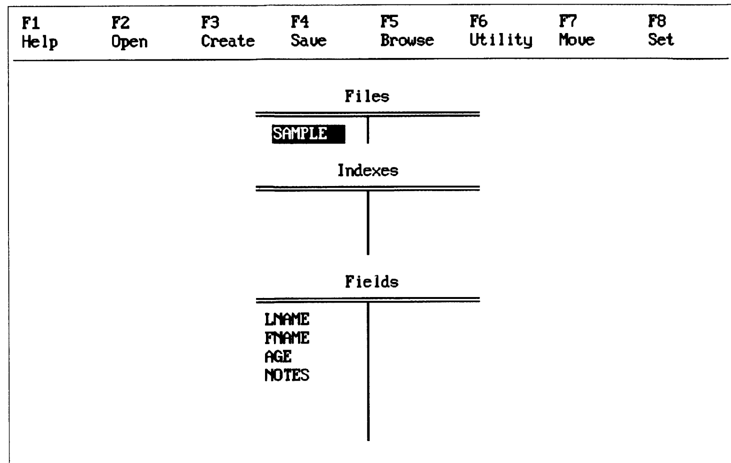
   The Save Struct dialog box opens on the main screen.

3. Type **SAMPLE** and press the Down arrow key.

   The OK button is highlighted.

4. Press Return to confirm.

The database file is created, and its name appears on the main screen:



You can have several database files open simultaneously in DBU. To do this, move the highlight bar to the empty Files column and either create a new file or open an existing one. The active file is the one whose name is currently highlighted.

Creating an Index File

By default, a database file does not have any logical order imposed on its records. Instead, the records are maintained in the order in which you add them to the file. As you add, delete, and make changes to the file, the records remain in the same order without regard to the actual data.

In many applications, it is desirable to maintain and display a particular order for a database file based on the contents of one or more fields. CA-Clipper provides a mechanism for doing this via index files. Index files are separate from the actual database file, and there are many CA-Clipper commands and functions that you can use to do such things as create an index, use a particular index, and search for a specific index key value.

In DBU, you can create and use index files for the active database file. When an index file is used, the records in the database file appear in order according to the columns that you use to define the index key. To create an index file for the active database that imposes alphabetical order by last and first name, perform the following steps:

1. Select F3 Create.

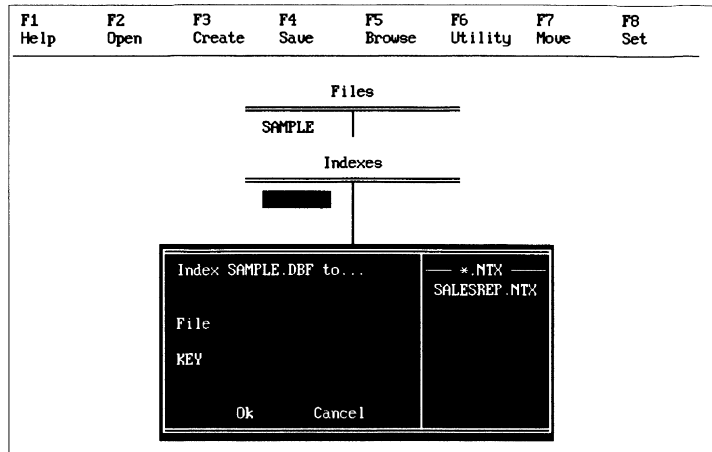
   The Create Database menu item is highlighted.

2. Select Index.

   The Create Index menu item is highlighted.

3. Press Return.

   The Create Index dialog box opens on the main screen:



In the dialog box, enter the name of the index file and the key expression as follows:

1. Type **SAMPLE** after the File prompt and press Return.

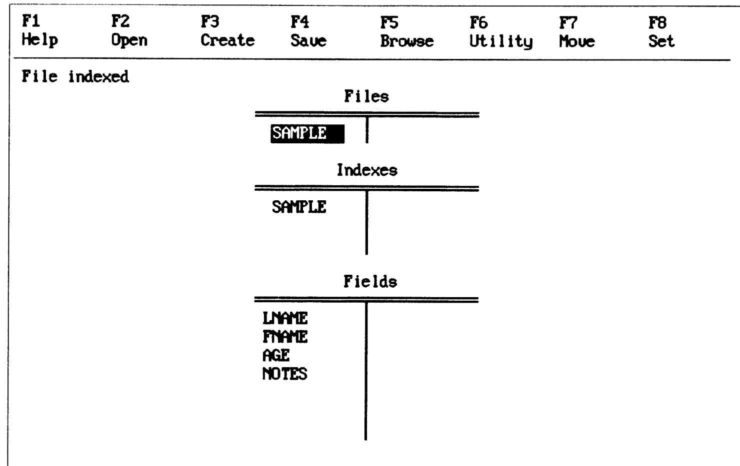
   The highlight bar moves next to the KEY prompt.

2. Type **Lname + Fname** and press the Down arrow key.

   The OK button is highlighted.

3. Press Return to confirm.

The index file is created, and its name appears on the main screen as shown below:



You can have several index files for each database file. To do this, move the highlight bar just below the current index file in the Index column with the Down arrow key, and either create a new file or open an existing one. The controlling index is always first in the Index list.

Adding Data

Once the database file structure is defined, you can add data to the file. Any index files that are currently engaged are automatically updated to reflect the new records. You add records using the Browse menu as follows:

1. Select F5 Browse.

   The Browse Database menu item is highlighted.

2. Press Return.

   The Browse window opens on the main screen:

```
F1     F2     F3      F4    F5      F6       F7    F8
Help   Open   Create  Save  Browse  Utility  Move  Set

                      Files
                      SAMPLE

                                  Record         <new>
   LNAME                FNAME                    AGE
                                                  0
```

Since this is a new file, there are no existing records in the window. To add a new record, start typing the information for the currently highlighted field. When you are finished with one field, press Return to move on to the next one. For example:

1. Type **Smith** and press Return.

   The highlight bar moves to the Fname field.

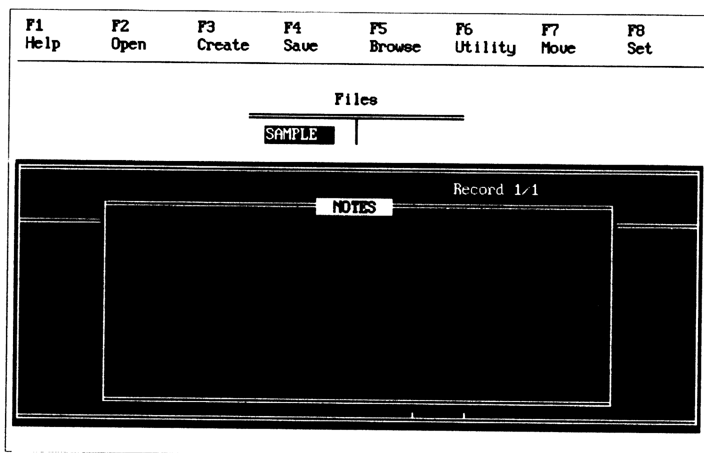
2. Type **Joe** and press Return.

   The highlight bar moves to the Age field.

3. Type **92**.

   The highlight bar automatically moves to the Notes field because the Age field was completely filled.

For memo fields, you can enter data just as you would for any other field. As soon as you start typing, a memo editing window opens. You can also press Return before you start typing to see any existing information without overwriting it. For example:



1. Press Return.

   An empty window opens as shown above.

2. Type **Notes about Joe Smith** and press Ctrl+W.

   The data you entered is saved in the Notes field.

To add another record:

1. Press Ctrl+Home.

   The highlight bar moves back to Lname, the first field.

2. Press the Down arrow key.

   A new blank record appears in the window.

Now, enter the following data into the record:

1. Type **Adams** and press Return.

   The highlight bar moves to the Fname field.

2. Type **Jane** and press Return.

   The highlight bar moves to the Age field.

3. Type **33**.

   The highlight bar automatically moves to the Notes field because the Age field was completely filled.

4. Type **Notes on Jane Adams** and press Ctrl+W.

   The data you entered is saved in the Notes field for Jane Adams.

5. Press Ctrl+Home.

   Notice that the names are sorted automatically in alphabetical order as you type, and that the highlight bar moves back to the first field as shown here:

```
F1      F2      F3      F4      F5      F6       F7     F8
Help    Open    Create  Save    Browse  Utility  Move   Set

                        Files
                     SAMPLE

                                Record 2/2
   LNAME                  FNAME                    AGE
   Adams                  Jane                      33
   Smith                  Joe                       92
```

You can enter as many records as you like using the Browse window. When you are done, press Esc to close the window and return to the main screen.

Editing Data

Now that you have added records to the database file, you can open the Browse window again and see the data you entered:

1. Select F5 Browse.

   The Browse Database menu item is highlighted.

2. Press Return.

   The Browse window opens on the main screen:

```
F1      F2      F3       F4      F5       F6        F7      F8
Help    Open    Create   Save    Browse   Utility   Move    Set

                          Files
                    SAMPLE

                                                Record 2/2
    LNAME                             FNAME                     AGE
    Adams
```

Notice that the entries in the database appear in order by "Lname" and "Fname" as specified by the index file you created earlier. At this point, you can make changes to the existing records by typing over the old information, and you can add new records as previously described. When you are done, press Esc to close the Browse window. All changes are automatically reflected in the database file as well as any open index files.

Feel free to explore the other DBU menu options to see what is available, and experiment with creating database and index files of your own.

Leaving DBU

To exit DBU when you are finished:

1. Press Esc.

   You are prompted with the following message:
   Exit to DOS? (Y/N).

2. Type **Y** and press Enter.

   You are returned to the DOS prompt.

All files that you created and any data that you added or changed is automatically saved.

Summary

In this chapter, you have been introduced to some of the most important features of DBU. You learned how to navigate within the DBU menu system, to create a database file and an index file, and to add and edit data. DBU is intended as a utility to design data structures that will ultimately be used in your CA-Clipper application programs. For more information on DBU, refer to the "Database Utility—DBU.EXE" chapter of the *Programming and Utilities Guide.*

Chapter 7
DOS Online Documentation: The Guide To CA-Clipper

In This Chapter

This chapter introduces you to *The Guide To CA-Clipper*, online documentation at the DOS level provided as part of the CA-Clipper package. It acquaints you with the concept of *The Guide To CA-Clipper* and teaches you the rudiments of loading, using, and leaving the Norton Instant Access Engine.

In this chapter, you will learn:

- The basic concepts of *The Guide To CA-Clipper* and the Instant Access Engine
- How to load and activate the Instant Access Engine
- How to navigate and make selections within the Instant Access Engine
- How to get online assistance and use the cross-referencing system
- How to load a new database
- How to leave the Instant Access Engine and unload it from memory

For complete information, refer to the "Online Documentation—NG.EXE" chapter in the *Programming and Utilities Guide.*

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.

Requirements for This Chapter

In order to perform the exercises in this chapter, you must first install NG.EXE. If you selected the DOS Help option at the beginning of the installation procedure, this executable file was installed in the \NG directory on the drive where CA-Clipper was installed. If you elected not to install the online documentation, run the installation procedure again (see the "Installation" chapter of this guide) and select this option.

You must also update the AUTOEXEC.BAT and CONFIG.SYS files according to the instructions in the Before Moving On section of the "Installation" chapter in this guide. Do not forget that any time you make changes to these files, you must reboot your computer for the changes to take effect.

What Is The Guide To CA-Clipper?

Much of the CA-Clipper documentation is provided in the form of several databases referred to collectively as *The Guide To CA-Clipper*. These databases are accessible via the Instant Access Engine, a memory-resident program also included as part of the CA-Clipper package.

The Instant Access Engine lets you display information from *The Guide To CA-Clipper* databases at the touch of a shortcut key. Other databases (e.g., C, Assembler, and third-party CA-Clipper libraries) designed for use with the Instant Access Engine are available from third-party vendors.

Loading The Guide To CA-Clipper

To use *The Guide To CA-Clipper*, load the Instant Access Engine into memory by typing **NG** at the DOS prompt.

After you load the Instant Access Engine into memory, *The Guide To CA-Clipper* help databases are available to you even though no information appears on your screen. To get help at any time without having to quit what you are doing (perhaps operating from the DOS prompt or using your word processor to edit a program), simply press Shift+F1.

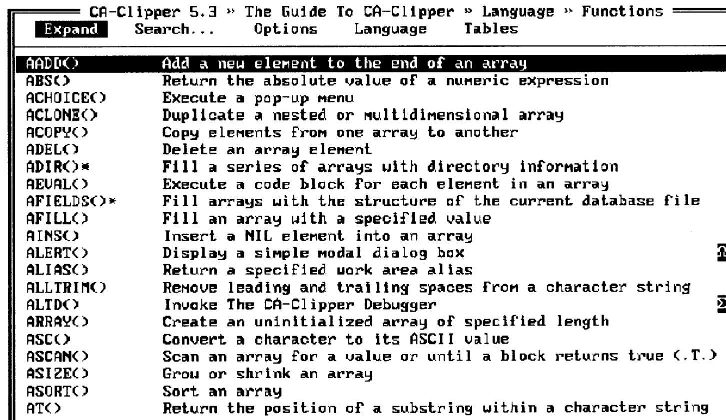
Note: If you are using DOS 5.0 or higher, you must press F1 or Esc immediately after Shift+F1 to activate the Instant Access Engine. On some computers, using Esc here will cause the computer to hang up momentarily after you leave the Norton Guide. Using F1 avoids this. If you place the statement SWITCHES=/k in your CONFIG.SYS file, you will not have to press F1 or Esc after Shift+F1.

Navigation

When the Instant Access Engine is active, there is a menu bar across the top of the screen providing you with several options. Use the Left arrow and Right arrow keys to move the highlight to a new menu option. If the menu bar option has an associated menu, the menu drops down and the first item is highlighted. If there are no pull-down menus open, you can select a menu bar option by typing the first letter of the option.

When a pull-down menu is open, use the Up arrow and Down arrow keys to navigate within the menu and Return to select the currently highlighted item. Again, you can use the first letter shortcut to select a particular item.

Since Expand is the action you will perform most often, it is always the default on the menu bar as shown here:



This menu option is the only one that does not have an associated pull-down menu. When Expand is highlighted, the Up arrow and Down arrow keys allow you to navigate through the alphabetical entries. Return allows you to expand the highlighted entry.

Finding Information

Entries on the main part of the screen are items in the current category. The default category is Functions, but you can select other categories from the Language menu. Commands and Operators are examples of other categories that are available. For example:

1. Select L Language.

   The Language menu is displayed.

2. Press O to select the Language Operators menu item.

   The Operators short entry list is displayed in place of the Functions on the main screen.

To return to the Function entries and continue with the lesson:

1. Select L Language.

   The Language menu is displayed.

2. Press F to select the Language Functions menu item.

   The Functions short entry list is displayed on the main screen.

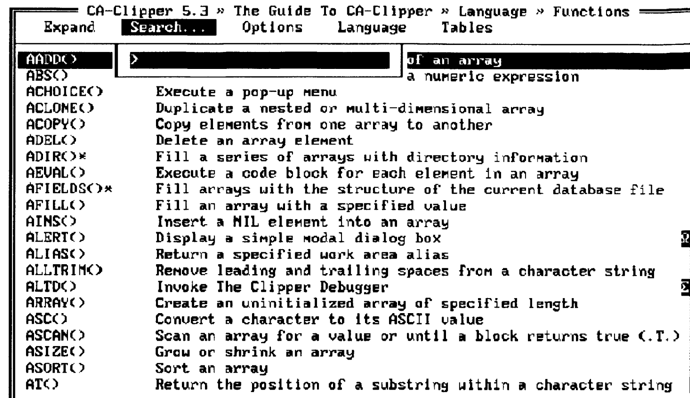
Entries are listed in alphabetical order according to the name on the left, and a short text description is given for each on the right. The name is used to search for the entry, and you can expand the short text for more information.

For example, to obtain information on the SUBSTR() function, navigate to it on the main screen and press Return to expand it. As stated earlier, you can press the Down arrow key to move the highlight to SUBSTR(), but this might take a while since it is near the end of the alphabetical list.

A faster way to find an item is to use the Search menu, as follows:

1. Select S Search.

   The Search menu opens:

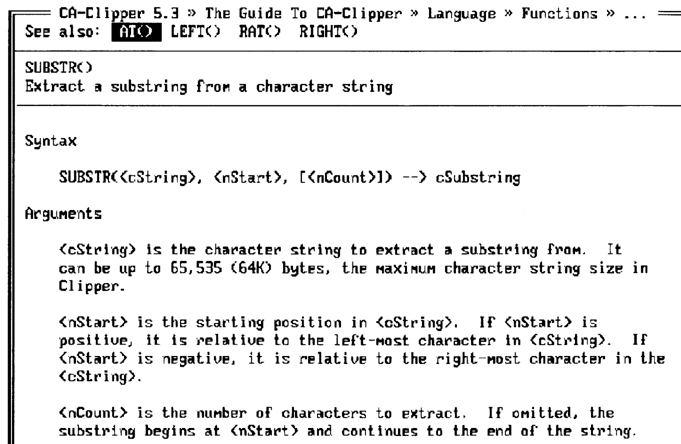


2. Type **substr** and press Return.

   The highlight on the main screen moves to the SUBSTR() entry, and Expand is once again highlighted on the menu bar.

3. Press Return.

   The SUBSTR() entry is expanded as shown:



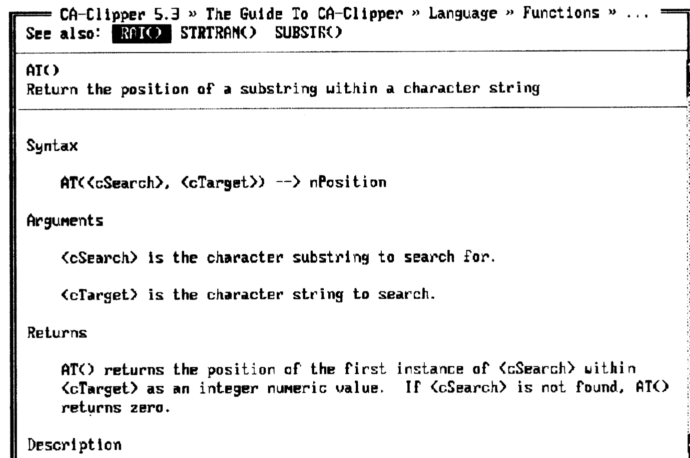
Use PgUp and PgDn to scroll through the expanded entry display on the screen. Press Esc to go back to the alphabetical entries with their short descriptions.

When you expand an entry as you have just done, the menu bar becomes temporarily disabled, and its function is taken over by the See Also line which gives you a cross-reference to related entries.

As with the menu bar, you use the Left arrow and Right arrow keys (or the first letter shortcut) to navigate the See Also list. Pressing Return shows you the expanded text for the currently highlighted See Also reference. For instance, to view the expanded text for AT(), the first See Also item for SUBSTR():

1. Press Return.

   The AT() entry is expanded as shown below:



```
CA-Clipper 5.3 » The Guide To CA-Clipper » Language » Functions » ...
See also: RAT()  STRTRAN()  SUBSTR()

AT()
Return the position of a substring within a character string

Syntax

    AT(<cSearch>, <cTarget>) --> nPosition

Arguments

    <cSearch> is the character substring to search for.

    <cTarget> is the character string to search.

Returns

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

Description
```

2. Press Esc.

The alphabetical entries reappear on the screen and the menu bar becomes active once again.

Selecting a New Documentation Database

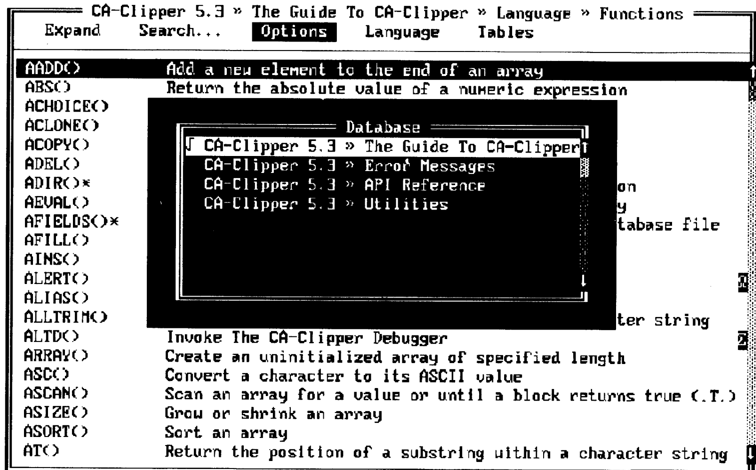
There are several CA-Clipper documentation databases available, including one that describes error messages and one that documents all of the utilities. To select a new database, use the Options Database menu item. For example:

1. Press O Options.

   The Options menu is displayed.

2 Press D to select Options Database.

   The Options Database picklist opens on the screen as shown:



3. Press the Down arrow key until you highlight the Utilities database, and then press Return.

The Utilities database is loaded and its help text replaces that of the current database on the main screen.

At this point, you have seen all of the main features of *The Guide To CA-Clipper*. You may want to take some time now and explore the menu bar to see what else is available.

Leaving The Guide To CA-Clipper

When you have found the information you need in *The Guide To CA-Clipper*, you can return to your prior task by pressing Shift+F1 or F10. You can also exit by pressing Esc when the menu bar is active. Note that pressing Esc may cause you to leave the Instant Access Engine unknowingly from time to time, but this should not be a problem since you can reactivate it by pressing Shift+F1.

If you need to remove the Instant Access Engine from memory, select the Options Uninstall menu item. To do this, activate the Instant Access Engine with Shift+F1 and perform the following steps:

1. Select O Options.

   The Options menu is displayed.

2. Press U to select Options Uninstall.

   You are prompted to confirm your selection.

3. Press Return with the Yes button highlighted.

   The Instant Access Engine is unloaded from memory, and you are returned to DOS.

After uninstalling, you can no longer access *The Guide To CA-Clipper* with Shift+F1 until you reload the Instant Access Engine into memory once again with NG.

Summary

In this chapter, you have been introduced to *The Guide To CA-Clipper* with enough information to allow you to explore it on your own. For more detailed information on the Instant Access Engine and *The Guide To CA-Clipper* databases, refer to "Online Documentation—NG.EXE" chapter in the *Programming and Utilities Guide.*

Chapter 8
Where to Go from Here

The CA-Clipper documentation is modularized for easy use and quick access to the information you need. After reading this guide, you will want to familiarize yourself with the other guides in the set. This chapter gives a description of each guide so that you will know where to look for the answers to your questions.

Note: The README file installed in the \CLIP53 directory contains important last minute information. This file can be reviewed using DOS TYPE or any text editor.

Manual Organization

The CA-Clipper documentation set is divided into seven guides. Each guide is designed to target a specific aspect of the system.

- *Getting Started*

  This is the guide you are reading now. It is designed to give you all the information you will need to install CA-Clipper on your computer as well as introduce you to some of its features and benefits.

- *Reference Guide, Volumes 1 and 2*

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

  Each item in the CA-Clipper language falls into one of the following categories: class, command, directive, function, operator, or statement. The category name follows the item name in the language reference heading.

If you are an experienced CA-Clipper user, you can use these guides to obtain the correct syntax and semantics of the language components. If you are a new user, you can browse through to get an idea of the kinds of things the language has to offer.

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.

- *Programming and Utilities Guide*

  This guide gives you conceptual information on programming in the CA-Clipper environment as well as specific details on how to use the compiler, linker, and other CA-Clipper utilities. It also includes a basic CA-Clipper language concepts chapter.

- *Error Messages and Appendices Guide*

  This guide documents the error messages for the CA-Clipper compiler (CLIPPER.EXE), protected mode linker (EXOSPACE.EXE), real mode linker (BLINKER.EXE), and make program (RMAKE.EXE), as well as the runtime error messages that you may encounter when you execute your CA-Clipper application. The error messages are divided into categories organized by severity. Each error message is given with an explanation and action description. DOS error messages are included as a separate chapter in this guide.

There are also nine appendices in this guide. Appendix A is a complete ASCII character table; Appendix B is a table of color codes used by SETCOLOR() and SET COLOR; Appendix C is a table of INKEY() codes; Appendix D is a series of ASCII character tables, divided into categories; Appendix E is a list of CA-Clipper Reserved Words; Appendix F is a list of dBASE commands and functions, not supported in CA-Clipper; Appendix G contains Categorized Language Tables; Appendix H contains a list of Obsolete Reference Items with recommended replacement items; and Appendix I contains a list of technical specifications.

- *Drivers Guide*

  This guide contains all the information you need to use the replaceable database drivers and alternate terminal drivers provided with CA-Clipper.

- *Quick Reference Guide*

  This is a pocket reference to the CA-Clipper language that gives you the syntax of all language components described in the Reference guide, and command line utilities.

- *Technical Reference Guide*

  This guide contains the alphabetical references for the Extend, Virtual Memory, Fixed Memory, Replaceable Database Driver (RDD), and Terminal API functions. The reference chapters are organized alphabetically to make finding the information you need easy. API is an abbreviation for Application Programming Interface, and the term describes systems (i.e., groups of functions) that give you a means to interface your CA-Clipper applications with other systems.

  If you are an advanced programmer interested in interfacing your CA-Clipper applications with C or Assembly language programs, the information in this guide will be useful to you.

Development Tools

CA-Clipper has a number of tools you will use to create and develop your programs. If you have reviewed all of the chapters in this guide and installed the CA-Clipper development system, you should already be familiar with most of these programs. If not, the following descriptions will give you an introduction as well as point to where the program is installed and where to find more information in the documentation.

- CA-Clipper Compiler—CLIPPER.EXE

  The CA-Clipper compiler is a standard command line driven compiler that supports a number of command line options and environment variables. Invoking it without specifying a source file lists all the available options.

  - **Default Installation:** Executable file in \CLIP53\BIN.
  - **Documentation:** Complete documentation: *Programming and Utilities Guide*; Command line documentation: *The Guide To CA-Clipper* and *Quick Reference Guide*; Tutorial: *Getting Started* guide, "Learning the Basics" chapter.

- CA-Clipper Protected Mode Linker—EXOSPACE.EXE

  The protected mode linker and DOS extender, CA-Clipper/Exospace, is a standard command line driven linker that supports a number of command line options and environment variables.

  - **Default Installation:** Executable file in \CLIP53\BIN.
  - **Documentation:** Complete documentation: *Programming and Utilities Guide*; Command line documentation: *The Guide To CA-Clipper* and *Quick Reference Guide*; Tutorial: *Getting Started* guide, "Learning the Basics" chapter.

- CA-Clipper Real Mode Linker—BLINKER.EXE

  The CA-Clipper real mode linker is an optional, standard command line-driven linker that supports a number of command line options and environment variables.

  - **Default Installation:** Executable file in \CLIP53\BIN.
  - **Documentation:** Complete documentation: *Programming and Utilities Guide;* Command line documentation: *The Guide To CA-Clipper* and *Quick Reference Guide.*

- CA-Clipper Debugger—CLD.LIB

  The CA-Clipper DOS-level debugger is a source code debugger that allows you to debug your programs in a menu-driven environment. Multiple windows are under your control for displaying various aspects of your program while it is executing.

  - **Default Installation:** Executable file in \CLIP53\LIB.
  - **Documentation:** Complete documentation: *Programming and Utilities Guide;* Command line documentation: *The Guide To CA-Clipper* and *Quick Reference Guide;* Tutorial: *Getting Started* guide, "Debugging a Simple Program" chapter.

- Program Maintenance—RMAKE.EXE

  RMAKE.EXE is a program that can be used to maintain files. Essentially it does this by interpreting a text file (*make file*) of commands, comparing dates of source and target files, and performing specified operations if any target files are newer than their dependent source files. It is generally used to force a compile of a source (.prg) file if it has changed since it was last compiled, thereby avoiding unnecessary compilations. It is a standard command line-driven utility.

  - **Default Installation:** Executable file in \CLIP53\BIN.
  - **Documentation:** Complete documentation: *Programming and Utilities Guide;* Command line documentation: *The Guide To CA-Clipper and Quick Reference Guide.*

- Program Editor—PE.EXE

  PE.EXE is a simple text editor written in CA-Clipper that you can use to create small program (.prg) files.

  – **Default Installation:** Source files in \CLIP53\SOURCE\PE; executable file in \CLIP53\BIN.

  – **Documentation:** Complete documentation: *Programming and Utilities Guide;* Command line documentation: *The Guide To CA-Clipper* and *Quick Reference Guide.*

- Database Utility—DBU.EXE

  DBU.EXE is a database creation and modeling utility written in CA-Clipper. It can be used to create database and index files, model relations between database files, and edit data.

  – **Default Installation:** Source files in \CLIP53\SOURCE\DBU; executable file in \CLIP53\BIN.

  – **Documentation:** Complete documentation: *Programming and Utilities Guide;* Command line documentation: *The Guide To CA-Clipper* and *Quick Reference Guide;* Tutorial: *Getting Started* guide, "Creating Data Structures" chapter.

- Report and Label Utility—RL.EXE

  RL.EXE is a report and label creation utility written in CA-Clipper. It can be used to create report form (.frm) and label (.lbl) definitions that can be executed by the REPORT and LABEL FORM commands, respectively.

  – **Default Installation:** Source files in \CLIP53\SOURCE\RL; executable file in \CLIP53\BIN.

  – **Documentation:** Complete documentation: *Programming and Utilities Guide;* Command line documentation: *The Guide To CA-Clipper* and *Quick Reference Guide;* Reference: *Reference Guide,* LABEL FORM and REPORT FORM entries.

- The Guide To CA-Clipper—NG.EXE

  The Guide To CA-Clipper is the DOS online version of the CA-Clipper documentation. It contains the most timely reference information on the CA-Clipper language and utilities. It consists of the Norton Instant Access Engine (NG.EXE) and several databases containing the CA-Clipper documentation.

  - **Default Installation:** Executable and database files in \NG.
  - **Documentation:** Complete documentation: *Programming and Utilities Guide*; Tutorial: *Getting Started* guide, "Online Documentation: The Guide To CA-Clipper" chapter.