

# Repertoire<sup>®</sup>

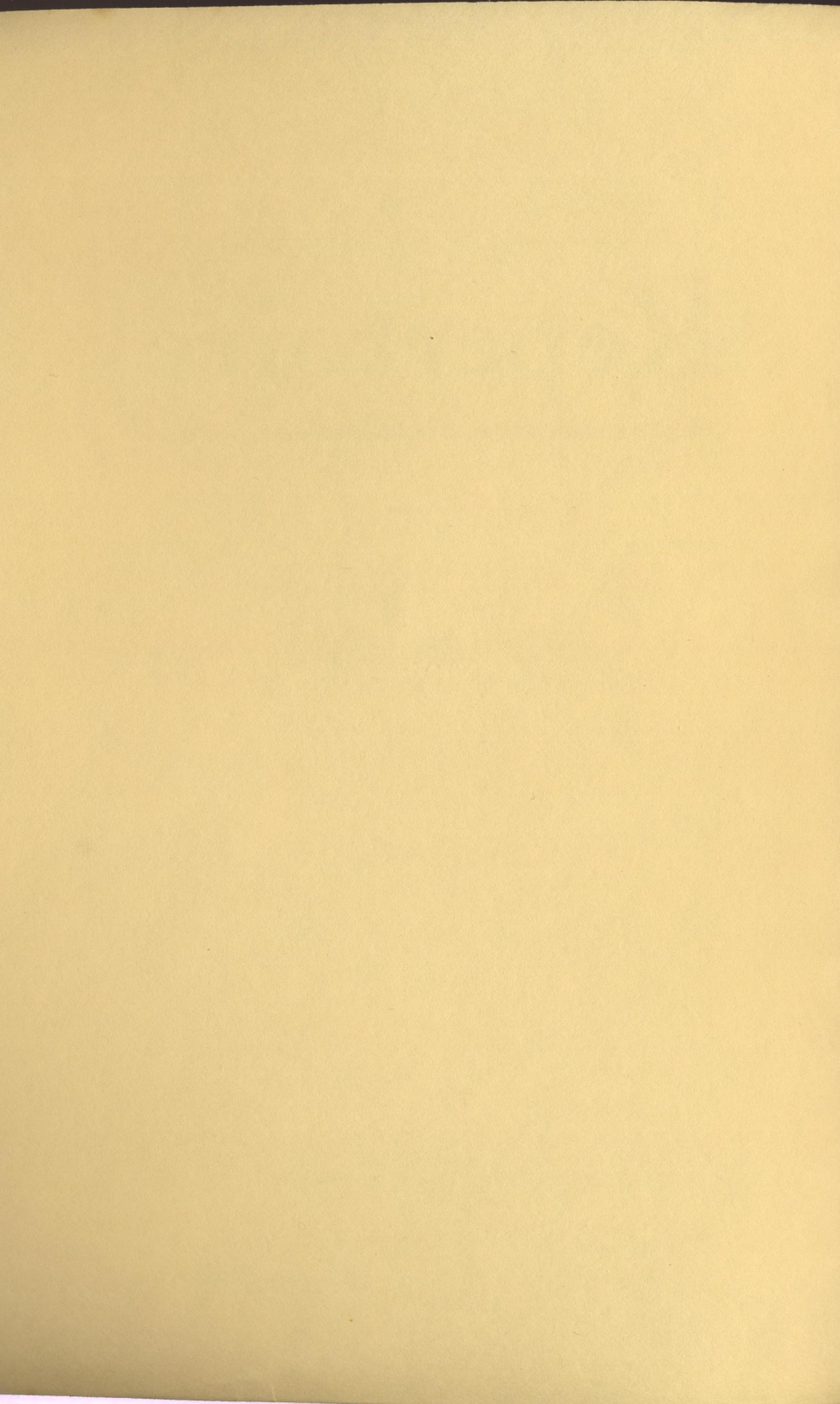
Version 1.6

Users Guide

by

Charles Bradford • Cole Brecheen







# Repertoire<sup>®</sup>

Version 1.6  
Users Guide

by  
Charles Bradford  
Cole Brecheen



Green Bay, Wisconsin

# Repertoire

Version 1.0  
Users Guide

by  
Charles B. Borchers  
Cole Borchers



Copyright 1999



# TABLE OF CONTENTS

|                                                       |    |
|-------------------------------------------------------|----|
| TABLE OF CONTENTS .....                               | 1  |
| PMI Software License Agreement.....                   | 5  |
| New In Version 1.6 .....                              | 8  |
| 1. Introduction .....                                 | 10 |
| 1.1. Indexed File System.....                         | 10 |
| 1.2. Screen Design/Display System.....                | 11 |
| 1.3. Multi-Window Editor.....                         | 12 |
| 1.4. Natural-Language Analysis Routines.....          | 13 |
| 1.5. Overview of Low-Level Utilities.....             | 13 |
| 1.5.1. DOS File Services .....                        | 14 |
| 1.5.2. Screen Output.....                             | 14 |
| 1.5.3. Keyboard Input.....                            | 14 |
| 1.5.4. Program parameter and environment access ..... | 15 |
| 1.5.5. Other PC and DOS services .....                | 15 |
| 1.5.6. String and list manipulation.....              | 15 |
| 1.6. Supported Compilers.....                         | 16 |
| 1.7. Object-Only and Full-Source Versions .....       | 17 |
| 2. Generic List Handling.....                         | 18 |
| 2.1. Quick Start Examples.....                        | 18 |
| 2.1.1. Retrieving a List Element .....                | 19 |
| 2.1.2. Inserting a List Element.....                  | 19 |
| 2.1.3. Deleting List Elements .....                   | 19 |
| 2.1.4. Creating a List of Strings.....                | 20 |
| 2.2. How a GenList Works .....                        | 20 |
| 2.3. Error Handling .....                             | 22 |
| 2.4. Type Checking .....                              | 23 |
| 2.5. Trouble Shooting.....                            | 24 |
| 3. Screen Design/Display System.....                  | 25 |
| 3.1. Overview.....                                    | 25 |
| 3.2. Your Screen-Creation Editor .....                | 26 |
| 3.3. Creating the Screens.....                        | 26 |
| 3.3.1. Command Area .....                             | 27 |
| 3.3.1.1. Field Commands .....                         | 29 |
| 3.3.1.2. Global Definitions .....                     | 35 |
| 3.3.1.3. Links to Other Frames .....                  | 36 |
| 3.3.1.4. Help.....                                    | 38 |
| 3.3.1.5. Data.....                                    | 39 |
| 3.3.2. Window Commands .....                          | 39 |
| 3.3.2.1. Colors.....                                  | 42 |
| 3.3.2.2. Automatic Screen Mode Detection.....         | 46 |
| 3.3.2.3. Using Repertoire in Graphics Modes.....      | 46 |

## TABLE OF CONTENTS

|                                                      |    |
|------------------------------------------------------|----|
| 3.3.3. Frame Text .....                              | 47 |
| 3.4. Compiling Frames .....                          | 47 |
| 3.5. Displaying Frames in a Program .....            | 49 |
| 3.5.1. The DisplayFile Type.....                     | 53 |
| 3.5.2. Opening and Closing DisplayFiles .....        | 53 |
| 3.5.3. ReadFrame.....                                | 53 |
| 3.5.4. ShowDisplayFrame .....                        | 54 |
| 3.5.5. ControlFrame.....                             | 54 |
| 3.5.5.1. Input Validation and Error Checking.....    | 56 |
| 3.5.5.2. Screen Creation by Programs .....           | 57 |
| 3.5.6. Keys Recognized in Repertoire .....           | 58 |
| 3.5.6.1. Entry Keys .....                            | 58 |
| 3.5.6.2. Special Keys .....                          | 59 |
| 3.5.7. Reading Input From a Frame.....               | 60 |
| 3.5.7.1. String and numeric input .....              | 61 |
| 3.5.7.2. Choice fields and menu items.....           | 62 |
| 3.5.8. Natural-Language Analysis of Input.....       | 63 |
| 3.6. Example .....                                   | 64 |
| 3.6.1. The Editable Frame.....                       | 64 |
| 3.6.2. The Processing Program.....                   | 67 |
| 3.7. Advanced Use of the Display System.....         | 68 |
| 3.7.1. The DisplayFrame and DisplayFile Records..... | 69 |
| 3.7.1.1. The DisplayFile Record.....                 | 69 |
| 3.7.1.2. The DisplayFrame Record.....                | 69 |
| 3.7.2. The FieldList and the ImageList.....          | 73 |
| 3.7.3. Pushing and Popping Frames.....               | 77 |
| 3.7.4. Saving Screen Input in an NdxFile .....       | 78 |
| 3.7.5. Detecting Normal Screen Submission. ....      | 79 |
| 3.7.6. Programs Changing Frame Data.....             | 80 |
| 3.7.6.1. Modifying Frame Fields .....                | 80 |
| 3.7.6.2. Adding To Frames.....                       | 81 |
| 3.7.7. Interpreting GoTo values differently .....    | 83 |
| 3.7.8. Automatic Frame Scrolling .....               | 84 |
| 3.7.9. Changing the Specially Handled Keys.....      | 84 |
| 3.7.9.1. Changing the Special Keys.....              | 85 |
| 3.7.9.2. Changing the Field Entry Keys.....          | 86 |
| 3.7.9.3. Changing the Cursor Movement Keys .....     | 86 |
| 3.7.9.4. Changing Field and Key Handling.....        | 87 |
| 3.7.10. Changing the Input Prompts .....             | 88 |
| 3.7.10.1. Whether to Prompt .....                    | 89 |
| 3.7.10.2. Prompting Size and Position .....          | 89 |
| 3.7.10.3. The Prompts .....                          | 89 |
| 3.7.10.4. Prompt Colors .....                        | 90 |
| 3.7.11. Writing in Small Windows.....                | 90 |



## TABLE OF CONTENTS

|                                                    |     |
|----------------------------------------------------|-----|
| 3.7.12. Using Many Frames at Once .....            | 90  |
| 3.7.13. Using Multiple Screen Files .....          | 93  |
| 4. Indexed File Subsystem .....                    | 95  |
| 4.1. Supported Features .....                      | 95  |
| 4.2. How to Use the File System .....              | 95  |
| 4.2.1. Creating Files .....                        | 96  |
| 4.2.2. Opening and Closing Files .....             | 97  |
| 4.2.3. Writing to Files .....                      | 98  |
| 4.2.3.1. PutField: .....                           | 98  |
| 4.2.3.2. WriteRecord: .....                        | 99  |
| 4.2.3.3. DeleteRecord: .....                       | 99  |
| 4.2.4. Reading Files .....                         | 100 |
| 4.2.4.1. GetField .....                            | 100 |
| 4.2.4.2. High-Level File Searching .....           | 103 |
| 4.2.5. Safety .....                                | 108 |
| 4.3. An Example .....                              | 108 |
| 4.4. Advanced Topics .....                         | 110 |
| 4.4.1. Using NdxFiles With the Screen System ..... | 110 |
| 4.4.1.1. Creating the Index File .....             | 110 |
| 4.4.1.2. Saving data from a Screen .....           | 111 |
| 4.4.1.3. Filling a Screen with NdxFile Data .....  | 112 |
| 4.4.1.4. Example .....                             | 112 |
| 4.4.2. Space Usage: Crunching Files .....          | 115 |
| 4.4.3. Rebuilding Damaged Files .....              | 116 |
| 4.4.4. Error Checking .....                        | 117 |
| 4.4.5. Type checking .....                         | 117 |
| 4.4.6. Lists and Sublists .....                    | 118 |
| 4.4.7. Changing File Structures .....              | 120 |
| 4.4.8. Adding NdxFile Features .....               | 120 |
| 4.4.9. Record and File Size Limits .....           | 121 |
| 4.5. How the File System Works .....               | 122 |
| 4.5.1. Byte Map of an NdxFile .....                | 124 |
| 5. The Editor .....                                | 127 |
| 5.1. Overview .....                                | 127 |
| 5.2. How to Call It .....                          | 127 |
| 5.3. The User Interface .....                      | 130 |
| 5.3.1. User Instructions .....                     | 131 |
| 5.4. Line and Character Counters .....             | 133 |
| 5.5. Editor Function keys .....                    | 133 |
| 5.6. Editing To/From Files .....                   | 133 |
| 5.7. Editing in Windows .....                      | 134 |
| 5.8. Multiwindow Editing .....                     | 134 |
| 5.8.1. Simple Multiple Windows .....               | 135 |
| 5.8.2. Separate Processes .....                    | 136 |

TABLE OF CONTENTS

6. Natural-Language Analysis: StrLogic ..... 139

    6.1. Suggested Applications..... 139

    6.2. How to Use It ..... 140

Appendix A: Extended Scan Codes ..... 142

Index ..... 143



# PMI Software License Agreement

PMI provides this library and licenses it for use. You, the licensee, assume responsibility for the selection of the library to achieve your intended results, and for the installation, use and results obtained from the library.

This is a single-user license. The license may be assigned to an individual person or to an individual computer. If assigned to an individual computer, the library must be used only on that computer. If assigned to an individual person, only that person may use the library.

## UNDER THIS LICENSE YOU MAY:

- \* Use the library on any single computer at any given time.
- \* Make two copies of the library for backup purposes.
- \* Link into your own programs copies of the library's implementation modules in compiled (object-code) form.

In their compiled, linked forms, you may distribute copies of the library's implementation modules and program modules to others, but you may not in any way make available to others the source code or the unlinked object versions of any of the library's implementation modules or program modules.

YOU MAY NOT USE, COPY, MODIFY OR TRANSFER THE LIBRARY OR ANY COPY, MODIFICATION OR PORTION, IN WHOLE OR IN PART, EXCEPT AS EXPRESSLY PROVIDED FOR IN THIS LICENSE. IF YOU TRANSFER POSSESSION OF ANY COPY OF THE LIBRARY TO ANOTHER PARTY, YOU MUST DESTROY ALL COPIES REMAINING IN YOUR POSSESSION.

LIMITED WARRANTY: PMI warrants that the library will perform substantially as described in the accompanying User's Manual on a standard IBM Personal Computer with Color Graphics Adapter running PC-DOS 3.3. The library is provided without any other warranty of any kind, either expressed or implied, including but not limited to the implied warranties of merchantability or fitness for a particular purpose. The entire risk as to the quality and performance of the library is yours. PMI does not warrant that the functions contained in the library will meet your requirements or that the operation of programs you write with the library will be uninterrupted or error free. PMI warrants the diskettes on which the library is furnished to be free of defects in materials and workmanship under normal use for a period of sixty days from the date of delivery.

If you believe that the library does not meet the conditions of the above Limited Warranty, you must notify PMI within sixty days of the date of

## PMI Software License Agreement

purchase.

**LIMITATION OF REMEDIES:** PMI 's entire liability and your only remedy is:

1. The replacement of any diskette not meeting PMI's "Limited Warranty" which is returned to PMI with evidence of date of purchase.
2. If notified within the warranty period of significant errors in the library that fail to meet the conditions of the above Limited Warranty, PMI will at its option, (a) correct such errors within ninety days or (b) authorize a refund of your purchase price upon return of all library materials to PMI.

IN NO EVENT WILL PMI BE LIABLE FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY EVEN IF PMI OR AN AUTHORIZED DEALER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

You acknowledge that you have read this Agreement, understand it and agree to be bound by its terms and conditions. You further agree that it is the complete and exclusive statement of the Agreement between us which supersedes any proposal or prior agreement, oral or written, and any other prior communications between us relating to the subject matter of this Agreement.



"Industry in art is a necessity, not a virtue, and any evidence of the same, in the production, is a blemish, not a quality; a proof, not of achievement, but of absolutely insufficient work, for work alone will efface the footsteps of work."

-- James McNeill Whistler

The Gentle Art of Making Enemies

[1890]

## New In Version 1.6

Release 1.6 wasn't in the original plan for Repertoire's evolution, but it's a significant step in the right direction. We realized that it was going to be impossible to maintain separate DOS and OS/2 versions. I tried for several months to do it the C way (i.e., with a preprocessor and conditional compilation), and I tried using vmerge and vdiff to maintain two independent sets of files, but nothing worked. We had to have a single version.

It was also clear that we couldn't stop supporting real-mode-only executables. Few of our customers have access to Microsoft's BIND.EXE and the MS and compiler-specific libraries that it requires, at least one of the M2 compilers we have to support generates executables that cannot be bound, and we knew DOS developers would object to the size of bound executables, even if their load sizes didn't exceed those of real-mode-only executables. We thought about requiring DOS developers to use Microsoft's API.LIB, which contains real-mode emulations of OS/2's Family API subset, but of course there were access problems, and API.LIB turned out not to be a linkable library anyway; it is full of assumptions that are satisfied only by the BIND.EXE link postprocessor.

So I convinced Chuck to do a clean, linkable, real-mode, assembly-level emulation of the parts of the OS/2 API that Repertoire depends on. Its interface is identical to OS/2's DOSCALLS.LIB, but you can link it into a DOS executable with a normal linker. It means that we can have a single version not only of the source, but of the object code as well. Link it with OS/2's DOSCALLS.LIB and the executable runs in protected mode with unlimited memory; link the same program with our FAPI.LIB and you get a native real-mode DOS executable that runs identically. That makes it easy to develop and debug real-mode applications in protected mode, and it lets DOS-only developers hedge their bets by using the OS/2 API to communicate with DOS.

FAPI.DEF is a literal translation of the OS/2 api interface. Under DOS FAPI.LIB is linked in and these functions are a part of the runtime system of the TopSpeed Extender and DLL's that are part of OS/2 under OS/2. Therefor under OS/2 and the TopSpeed environment you do **not** link in FAPI.LIB but you must link in OS2INIT.OBJ or OS2INIT.LIB that contain a dummy procedure that is for the initialization procedure in FAPI.LIB. Explanations of the functions in FAPI.DEF would require hundreds of pages in itself therefor we will not attempt to cover it. If you need to call any OS/2 api functions directly it is recommended that you consult a text on OS/2 programming. Also remember that all the various OS/2 api definition modules are equivalent. Use which ever one you prefer. FAPI.DEF is a



mechanical translation and is not very Modula-2 friendly and will be revised.

The main purpose of this release is therefore to consolidate the DOS and OS/2 versions. Upgrading to it from the DOS version of release 1.5 should be painless. Very few argument lists or procedure names have changed. We did change some module names to prevent conflicts with compiler-provided modules: FileIO is now HandleIO, and Speaker is now Spkr. The StringIO.ErrorMessage type is now CARDINAL. FindFirstFile and FindNextFile now have additional DirectoryHandle parameters. We've added a very nice NumInput module written and contributed by Mike Carter and Jonathan March. There are a lot of new string- and list-handling routines, but they won't get linked in if you don't use them. The best reason to upgrade is to begin protected-mode debugging of your DOS application. The protected mode version of CodeView works fairly well with both Stony Brook and JPI. It lets you forget about memory limits and bugs in the debuggers themselves, and finally get on to real work.

# I. Introduction

*Repertoire* distinguishes itself from the horde of toolkits available for C and Pascal by providing a powerful, integrated solution to problems associated with screen handling, text handling, list handling, structured-file handling, and input analysis.

## I.1. Indexed File System.

*Repertoire's* indexed file system provides full support for true variable-length keyed records, as well as very rapid record access. It supports record deletion, insertion, garbage collection, recovery of damaged files, and automatic storage and retrieval of linked lists (including lists of lists) ala *Repertoire's* GenLists module, explained in Chapters 2 and the *Repertoire* reference manual. It stores a true binary image of your data, and thus avoids the need for the cumbersome data conversion operations required in other file systems.

A *Repertoire* NdxFile stores its index of record names, sizes, and file positions in the same file that stores the data. That gives *Repertoire* several advantages over file systems that maintain separate data and index files:

- It means that your data files cannot be rendered unusable, and cannot cause your program to misbehave, by damage to (or accidental erasure of) your index files.
- It reduces the likelihood that you will encounter the DOS limit of 20 open files per process--each open data file takes up one DOS handle, not two.
- It reduces the complexity of developing and managing an application program. You never have to wonder whether a misbehaving database is using the correct index file, and the user does not have to worry about keeping all index files properly paired in the same subdirectories or on the same disks with the corresponding data and program files.

Moreover, because all NdxFile records are of variable length, the system is quite flexible. You can still use it for fixed length records, of course, and gain all of the record deletion, insertion, and file compaction advantages not provided in the minimal file-handling systems intrinsic to Pascal, C, and Modula-2. But if you choose to employ variable-length, mixed-type records, different possibilities present themselves:

- You could use a *Repertoire* NdxFile to store and retrieve pieces of text: electronic mail for a bulletin-board system, perhaps, or long



comments imbedded in a customer database.

- You could use a *Repertoire* NdxFile to store bitmaps, fonts, etc., for graphics applications.
- You could use a *Repertoire* NdxFile as the file system for an outline processor, or for any other program that employs hierarchically structured data.
- You could use a *Repertoire* NdxFile to consolidate an application's data files into a single, master file, for the convenience of your users. Many applications require a host of small data files because Pascal, C, and Modula-2 do not provide built-in support for data files of mixed type. Thus, for every different data type that your program must keep on disk, a separate file is required. An NdxFile, however, can contain mixed data types.

The most significant advantage of *Repertoire's* indexed file system, however, is that it is list-oriented. The concept of lists gives *Repertoire* an unusually elegant way of structuring records. Each field can be either a list or an "element," and each element can, in turn, be either a list or an element. *Repertoire* does not impose any limits on the level to which an NdxFile's lists are nested. The CreateNdxFile procedure (in *Repertoire's* NdxFiles module) takes, as one of its parameters, a string that contains all of the "node names" in a record. If a node name is followed by a bracketed list of other names, *Repertoire* knows it is a list, and allows reference to its elements like this:

Address.ZipCode

(where ZipCode is one of field names that make up the field Address). You could retrieve and manipulate the entire list with a single operation, however, by using the field name 'Address'. Arbitrarily complex structured lists can be saved and retrieved under single node names in an NdxFile.

List-handling with this much power is made possible by GenLists, *Repertoire's* generic list handler. It provides a standard way of referring to and manipulating lists of any data type, including other lists. All of *Repertoire's* major subsystems use GenLists; that allows tight, transparent integration at an unusually high level--arbitrarily complex structured lists can be passed between subsystems (e.g., from the editor to the file system to the screen system and back again in any order) in an intuitive way.

## 1.2. Screen Design/Display System.

*Repertoire's* screen system differs from other systems in several respects.

- *Repertoire's* screen system does not require you to learn a new editor designed solely for painting screens. Instead, it lets you conveniently



## Introduction

create and edit attractive, full-color screens with your favorite ASCII editor--probably the same editor you use for programming.

- *Repertoire's* screen system does not generate hard-coded screens (i.e., it does not generate a file of source code that displays a screen). Instead, it generates a file of densely encoded bytes and symbols that a program reads during execution and uses to construct the screen. This approach has several advantages: (1) it reduces the bulkiness of an application's code when many screens are used; (2) it allows screens to be changed without re-linking; (3) it allows the developer to distribute multiple versions of an application's screens without having to deal with multiple versions of the application itself--the user can determine at run time whether screens are in, say, English, French, German, or Spanish; and (4) it makes programs easier to organize because they do not have to incorporate foreign screen code.
- *Repertoire's* screen system is easier to use in a team-programming environment. Its screens can be printed in an intelligible form on cheap, black & white printers with no loss of information about colors and attributes. Its screens can also be transmitted in standard format between members of your programming team.
- *Repertoire's* screen system allows remote application updates and seamless DBMS integration. Any *Repertoire* application can dynamically insert and delete compiled screens from its own screen database--an essential feature for applications that must be updated often, or by modem.
- *Repertoire's* screen system is powerful. It provides features, like natural-language analysis of input and multi-line fields in which input text wraps and scrolls, that are remarkably difficult to hand code. It also controls program execution with minimal effort on the programmer's part, but (unlike several other authoring and prototyping systems) it does not try to be a programming language unto itself. No language we could write could compete with the elegance and power of Modula-2. *Repertoire* therefore merely supplements Modula-2 programming; it does not try to insulate you from it.

## 1.3. Multi-Window Editor.

*Repertoire's* editor is a simple screen-oriented editor. It is not intended for full-scale word-processing uses. It is quite useful, however, for applications that require input from users in the form of long strings and/or paragraphs. It is also useful in applications that need to allow simple editing of pre-existing data. *Repertoire's* editor is quite flexible, in that it can be used in windows of any size, and multiple copies of the editor can be used concur-



rently.

*Repertoire's* editor is also flexible in handling its input and output. It can take its input or write its output from or to a file an ASCII file or an Ndx-File, and it can pass data back to your program in the form of linked lists that can be manipulated outside the editor or that can be used in discontinuous sessions with the editor. It uses the same form of list that every other part of *Repertoire* uses, so all of *Repertoire's* list manipulation utilities are always available for manipulating the editor's text.

## 1.4. Natural-Language Analysis Routines.

These routines perform complex boolean operations on strings and allow string analysis to be wholly separated from the code of the program. String analysis commands can even be imbedded in *Repertoire* screens to automatically analyze unstructured user input. *Repertoire's* NdxFile system also provides a utility that uses the basic string-expression evaluation to search a database.

## 1.5. Overview of Low-Level Utilities

*Repertoire* provides about 350 low-level routines, including: fast, assembly-level routines for directly manipulating bytes and addresses; extensive string- and list-handling tools; and relatively complete access to interrupt 21H (DOS) and 10H (BIOS video) service routines.

*Repertoire* does all its DOS-dependent I/O directly through DOS "handles," and allows your programs to do the same. Unlike the handle-oriented routines in the libraries supplied with your compiler, *REPETOIRE's* handle-oriented I/O routines are buffered, and can therefore conveniently be used for common chores, such as reading variable-length (CR/LF delimited) strings from any file or device associated with a handle. That makes *Repertoire's* DOS-oriented I/O libraries (1) faster and more flexible than the compiler's; (2) easier to integrate with esoteric DOS functions (extended error codes, byte- and record-locking, device drivers, etc.); and (3) easier to translate into other programming languages (only one routine -- the DOS "gate" -- has to be rewritten). All of *Repertoire's* exported routines are explained individually in the *Repertoire Reference Manual*.

You might categorize *Repertoire's* low-level services as follows:

1. DOS file and directory services
2. Screen output
3. Keyboard input
4. Program environment access



## Introduction

5. Other PC and DOS services

6. String and list manipulation

### 1.5.1. DOS File Services

HandleIO, StringIO, Drectory

The HandleIO module includes routines to create and check for the existence of files; open and close them; force physical disk updates; read and write blocks of data; and read, move, and set file pointers. Simple string I/O is handled in the StringIO module. StringIO routines can be used with file handles representing the PC screen and keyboard as well as disk files.

The Drectory module provides complete access to the DOS directory functions. It improves on the directory-access routines supplied with your compiler in both its convenience and its comprehensiveness. Every directory-oriented service that DOS supports has a corresponding routine in Drectory. Drectory also provides services that have no corresponding interrupt 21H primitive. It allows you to copy files, for example, and gives you access to a disk's volume label.

### 1.5.2. Screen Output

StringIO, SmartScreen, WindowPrims

SmartScreen gets its name from its ability to adapt at runtime to the user's choice of operating environments, and to the target machine's level of IBM-compatibility, as well as from its ability to map monochrome and color attributes onto each other in a flexible, intelligent way.

The WindowPrims (for Window "Primitives") module allows you to draw boxes; read, save, and restore the cursor position; push and pop the cursor size and current screen colors; save, restore, move and scroll any portion of the screen; and pass messages to the user in *Repertoire* "Message Boxes" that save and restore any existing display.

*Repertoire*'s high-level screen design/display system does not depend directly on SmartScreen and WindowPrims, however. It uses only procedure variables exported by a module called VWindows, which are assigned by default to SmartScreen and WindowPrims procedures. If you need to use the screen/design display system in conjunction with a different window manager, you can simply assign the window manager's screen writing procedures to the VWindows procedure variables, and the screen system will run as if it were designed specifically for that window manager.

### 1.5.3. Keyboard Input

StringIO, KbdInput, StrInput, NumInput



StringIO's ReadStr provides a simple, convenient, and flexible way to read strings. It works equally well from the keyboard or from a file. More advanced single key entry is made possible by KbdInput. StrInput provides routines that allow entry of strings with full editing capabilities.

#### **1.5.4. Program parameter and environment access**

##### **EnvironUtils**

Routines like GetProgParams, ParsedParam, GetExeName, and ReadEnvironment allow your programs to access these DOS data areas. You can also directly manipulate DOS's Program Segment Prefix for lower-level purposes.

#### **1.5.5. Other PC and DOS services**

##### **EnvironUtils, Timing, Speaker, LowLevel, FAPI**

EnvironUtils provides routines to get and set DOS's date and time, and to read the DOS version. To time events in your programs, call the routines BeginTimer, ReadTimer, and ShowUsage in the module Timing. To produce sounds on the PC's speaker, you may use the high-level routine, Noise, in the module Speaker. Other routines in Speaker give lower level access to the speaker. LowLevel gives you direct access to DOS's service routine interrupt 21H, and the BIOS video interrupt 10H. LowLevel, also lets you directly peek and poke PC memory. FAPI is an interface into the OS/2 api.

#### **1.5.6. String and list manipulation**

##### **PosUtils, ScanUtils, StrEdit, StrConv, StrConv1, LowLevel, ListUtils, GenList,**

Since much of what Modula-2 programs must do is essentially string manipulation, *Repertoire* provides four types of string handling modules. The most sophisticated of these, StrLogic, is explained in The Repertoire reference manual and in the StrLogic section of the Repertoire Reference Manual. PosUtils provides improved string-search routines. ScanUtils is similar to PosUtils but allows specification of case sensitivity for each procedure. StrEdit provides routines that modify strings in a variety of ways. StrConv converts CARDINALs, LongInts, dates, and times to and from strings or parts of strings. StrConv's AppendDecoded routine provides a convenient way to imbed nonprintable (e.g., graphics) characters in strings.

LowLevel provides many routines written in assembly language that fill, scan, and move sequences of bytes. Other operations include bit-wise shift-left and shift-right routines, and full-array shifting operations.



## Introduction

GenLists is a remarkably powerful and reliable list handler. It allows Modula-2 programs to manipulate linked lists with simple analogues of the standard operations on strings: insert, delete, copy, etc. A GenList can contain elements of any data type, including mixed types, and including other lists. All GenList routines call themselves recursively to deal with lists that contain sublists. GenLists gives Modula-2 much of the list-handling power of special-purpose languages like LISP and LOGO. ListUtils has higher level routines based on GenLists

## 1.6. Supported Compilers

This is the manual for all Modula-2 versions of *Repertoire*. *Repertoire* now uses precisely the same source files for all supported compilers; only the LowLevel module, written largely in assembler or inline machine code, is different. To achieve this, it has to take some extreme steps. *Repertoire* now fully qualifies all imported identifiers, because different compilers disagree about whether it is legal to import the same identifier into both the definition and the implementation of a module. It also uses TSIZE instead of SIZE because the compilers don't agree about whether SIZE can, cannot, or must be imported from SYSTEM. And it no longer uses inline address arithmetic.

*Repertoire* eliminates all direct references to compiler-specific library modules so as to make the object code compiler independent. Even *Repertoire*'s dynamic memory allocation is compiler independent, because it now does all allocations and deallocations through the procedure variables in VStorage, and it does no dynamic allocations at all during the module initialization sequence. To make it work, you merely assign your compiler's ALLOCATE, DEALLOCATE, and Available procedures to three VStorage procedures (see InitCompilerMods, explained in the *Repertoire* Reference Manual). If you are using *Repertoire* from C, you assign pointers to procedures that call the corresponding stdlib procedures.

Most of the source files are common to all compilers. The following files are unique for each compiler: FAPI.DEF, NumTypes.DEF, NumTypes.MOD, InitComp.MOD, LowLevel.ASM and LowLevel.OBJ. While the particulars of the distribution disks is subject to change, currently the compiler dependent parts along with a *Repertoire* library are in separate ZIP files with files grouped in appropriate directories such as DEF, MOD and OBJ. Check the read.me file on the distribution disk for details,

Because of the wide variety of directory structures recommended by different compiler companies, and the even wider variety of directory structures that programmers actually use, we cannot provide installation programs to copy *Repertoire* files from the distribution disks to your hard disk. If you are a new Modula-2 programmer, or if you are learning to use a new



Modula-2 compiler, we strongly recommend that you thoroughly acquaint yourself with the compiler and its file-search strategies before trying to install *Repertoire*. Decide, for example, whether you want to organize your files by project (e.g., all files related to a particular project in the same directory) or by function (e.g., all .DEF files in the same subdirectory regardless of project), and then copy the appropriate *Repertoire* files into the directories you have chosen. If you can't figure out what goes where, you probably need a great deal more experience writing simple programs with your compiler before trying to install *Repertoire*.

## 1.7. Object-Only and Full-Source Versions

The full-source version of *Repertoire* comes with a nicely printed and bound manual, as well as commented source for every part of the system, including the screen compiler. Every full-source copy comes with all the files necessary to use *Repertoire* with all supported compilers.

The object-only version of *Repertoire* comes with all the .DEF (and .SYM) and .OBJ (or .LIB) files needed to use all the facilities described in this manual. Because of differences in how the compilers generate public symbols, however, there is a separate object-only version of *Repertoire* for each supported compiler. The object-only version is quite valuable, and works well, but developers of commercial-quality software will need the full-source version for at least four reasons:

1. Having the source code gives you an indispensable form of documentation. A manual simply cannot provide all the information you will need in implementing a finished application. Moreover, the source code will always be there to answer your questions; *Repertoire's* developers may not be.
2. Having the source code allows you to delete the procedure bodies of *Repertoire* routines that you do not call, thereby saving memory and disk space.
3. Having *Repertoire's* source code allows you to customize the system. You can feel more secure in changing *Repertoire* than you would in changing a typical Pascal or C library, for example, because Modula-2 itself protects you from making mistakes that introduce subtle, hard-to-trace bugs.
4. Having *Repertoire's* source code means that you will be able to move your programs to other machines, other operating systems, and perhaps even other languages, if that becomes necessary. It is almost always a serious mistake to invest substantial amounts of time and money in a program that depends on libraries for which you do not have source code.



## 2. Generic List Handling

*Repertoire's* GenLists module brings to Modula-2 the fully elaborated concept of lists. Its lists can consist of anything--most notably, other lists. It supports indefinitely nested lists of lists, and allows your programs to manipulate them with simple analogues of string operations such as Insert and Delete. It makes list-oriented code short, clear, and reliable, and gives Modula-2 many of the characteristics that developers seek in special purpose list-handling languages, such as LISP and LOGO.

The module is named GenLists because it is a "generic" list handling system. See Namir Clement Shammas, "Generic Routines in Ada and Modula-2," *Computer Language*, August 1986; K. John Gough, "Writing Generic Utilities in Modula-2," *Journal of Pascal, Ada, & Modula-2*, May/June 1986. A Modula-2 program can pass any variable or constant to a procedure with an open ARRAY OF BYTE parameter, and the procedure can determine the size of that variable without requiring the programmer to indicate the variable's size in another parameter. GenLists can therefore allocate just enough memory to hold each variable inserted into a list, and copy bytes between that untyped data area and open ARRAY OF BYTE parameters when the user wants to change or access the data.

We think you will find GenLists helpful enough to warrant using it for almost all list-handling purposes. Operations on GenLists tend to be faster than similar operations on hand coded lists because GenLists uses techniques that would be too complicated to recode for every list a program uses (see descriptions of individual routines in the *Repertoire Reference Manual*). A GenList's per-list memory overhead is only 28 bytes, and its per-element memory overhead is only 10 bytes. When the module is used to store lists of strings, it more than compensates for its per-element overhead by allocating only enough memory to store the string's present length (i.e., it does not allocate memory for the part of the string variable that follows the null terminator). And when PMI's EmsStorage product is linked in, GenLists automatically stores all data in expanded memory. GenLists therefore saves memory not only by eliminating duplicative list handling code, but also by storing data more efficiently than is normally possible.

### 2.1. Quick Start Examples

To use *Repertoire* effectively, you have to understand GenLists. We begin with a few examples of simple things you can do with GenLists, as well as a simple explanation of how it works. It is important also to become familiar with the descriptions of individual procedures in the ListUtils and GenLists sections of the *Repertoire Reference Manual*.



### 2.1.1. Retrieving a List Element

Suppose you have a GenList that contains only strings, such as the contents of an editor window, and you want to examine the fifth element of the list. This is one way to do it:

```
ListUtils.GetStr( EditorList, 5, YourStrVariable );
```

To retrieve elements from a list of binary objects, use the lower-level GetElmt and GetElmtAdr procedures in GenLists. For example:

```
VAR
  YourRecordVar:
    RECORD
      PartNumber: CARDINAL;
      PartName: ARRAY [0..15] OF CHAR;
      StressPoints: ARRAY [0..255] OF REAL;
    END; BEGIN
  ....
  GetElmt( TheList, 5, YourRecordVar, TypeCode );
```

### 2.1.2. Inserting a List Element

To insert a string into a list as the fifth element of the list, do something like this:

```
GenLists.ListInsert( 'This is now the fifth line.',
  GenLists.StrCode, EditorList, 5 );
```

You can insert any object into any GenList, even if the object is completely binary, and even if you don't know the size of the object at compile time. To insert a bitmapped image from a graphic display as the fifth element of a list of bitmaps, do this:

```
GenLists.ListInsertAdr( BitMapAddress, SizeOfBitMap, 0,
  BitMapList, 5 );
```

### 2.1.3. Deleting List Elements

Suppose you have a list of 5 strings:

```
"This is line 1."
"This is line 2."
"This is line 3."
"This is line 4."
"This is line 5."
```

To delete the third and fourth elements, do this:

```
GenLiasts.ListDelete( TheList, 3, 2 );
```

The list now looks like this:

```
"This is line 1."
"This is line 2."
"This is line 5."
```

### 2.1.4. Creating a List of Strings

This is a simple program that uses GenLists to make a list of strings and write them out in reverse order.

```
MODULE ListExample;
  IMPORT GenLists;
  IMPORT StrEdit;
  IMPORT StringIO;
  IMPORT InitCompilerMods;
VAR
  StringList: GenLists.GenList;
  cnt, TypeCode: CARDINAL;
  TmpStr: ARRAY [0..79] OF CHAR;
BEGIN
  GenLists.NewList( StringList );
  TmpStr := '';
  FOR cnt := 1 TO 5 DO
    StrEdit.Append( TmpStr, 'a' );
    GenLists.ListInsert( TmpStr,
      GenLists.StrCode,
      StringList, 1 );
  END;
  FOR cnt := 1 TO 5 DO
    GenLists.GetElmt( StringList, cnt, TmpStr,
      TypeCode );
    StringIO.WriteEol( StringIO.outp, TmpStr );
  END;
  GenLists.DisposeList( StringList );
END ListExample.
```

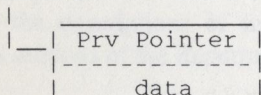
The program's output will be:

```
aaaaa
aaaa
aaa
aa
a
```

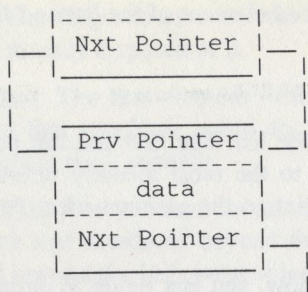
## 2.2. How a GenList Works

All GenLists are doubly linked; each element contains a pointer both to the previous element and to the next element so that a program can traverse the list in both directions. The first element of a GenList has a NIL prv pointer. The last has a NIL nxt pointer.

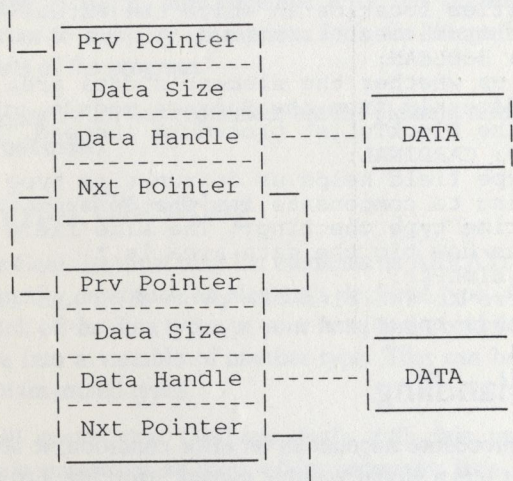
That is the only similarity between a GenList and a hand coded list. A normal hand coded linked list intermixes data and inter-element links. It looks like this:







A GenList, on the other hand, separates data from inter-element links. It looks like this:



The GenList type is actually an opaque pointer to a record (defined in the GenLists implementation module) with the elements listed below. Your programs do not need to concern themselves with the structure of a GenList (and in fact do not even have access to it), but the structure helps explain how a GenList works.

```
ParentList: GenList;
```

A pointer to the list into which this list has been inserted as a sublist. If it hasn't been inserted into any other list (i.e., if it is a root list) the ParentList pointer is NIL.

```
InitCheck: CARDINAL;
```

Used to help GenLists remind programmers to initialize their lists using GenLists's NewList procedure.

```
current, lngth: CARDINAL;
```

## Generic List Handling

These variables track the last referenced element of the list and the present length of the list.

```
first, now, last: ElmtPtr;
```

These are the three entry points into the list. The first and last pointers are self-explanatory; now always points to the most recently referenced element of the list. It should always point to the element identified sequentially by the 'current' field, above.

The ElmtPtr type, used for the first, now, and last fields, is defined as follows:

```
RECORD
  elem: VStorage.MemHandle;
    (*Identifies location in which the actual data for
      this element is stored.*)
  FromBlock: BOOLEAN;
    (*Tells us whether the element's data area came
      directly from the Storage module, or from
      the BlockToList procedure instead.*)
  type, size: CARDINAL;
    (*The type field helps us do run-time type check-
      ing to compensate for the loss of compile-
      time type checking. The size field tells
      us how big the data area is.*)
  prv, nxt: ElmtPtr;
    (*Pointers to the elements preceding and following
      this one.*)
END;
```

## 2.3. Error Handling

When a GenList procedure encounters an error condition, it always does two things. First, it sets a global variable named ErrorFlag. ErrorFlag is of type ListError, defined as follows:

```
ListError = (NoListError, underflow, overflow,
              NoInit, RefToZero, RefPastEnd, InsuffMem,
              corruption, NoSuchList);
```

Although the meaning of each element of the ListError enumerated type can vary according to the context, this is generally what they mean:

**NoListError:** No error occurred.

**underflow:** The variable into which GetElmt or NextElmt copied its data was larger than necessary. If the list element was a string, the procedure should nevertheless have returned a TRUE, because underflows do not matter in strings--GenLists copies the null terminator into the variable.

**overflow:** The variable into which the procedure copied its data was too small. GenLists actually does not allow overflows--it truncates to prevent damage to surrounding parts of memory.



NoInit: You forgot to initialize the list, or some other procedure or subsystem has already disposed of it.

RefToZero: The first element of a GenList is numbered 1. This message indicates that you have asked for the list's 0 element, or tried to insert something at the 0 position.

RefPastEnd: You've asked for an element beyond the present length of the list. Note that insertions beyond the present length are allowed--elements inserted beyond the list's present length are appended to the list.

InsuffMem: The Storage module was unable to allocate enough memory for the list or for one of its elements.

corruption: A weird, unexplainable error has occurred. We attribute all such errors to memory corruption; look for things like wild pointers in other parts of the program.

NoSuchList: A list was requested, but the element referred to was not a list, or does not exist.

## 2.4. Type Checking

Because GenLists must treat list elements as ARRAYs OF BYTE, it cannot automatically determine an element's declared type. As a result, you must keep track of the data types in your lists, and avoid copying an element of one type into a variable of another type. That can be difficult when your lists contain mixed types.

You will notice that GenLists deals with this problem by adding a TypeCode parameter to most of its parameter lists. GenLists' system of numeric type codes is meant not only to help your programs manage that problem, but also to give its routines information that they need to operate correctly. In general, however, use of the type code system is optional--if you don't need it, you can pass 0's as type codes.

Use of the type code system is mandatory only when you insert one list into another. In that case, you must use the constant ListCode, which is exported from GenLists. The ListCode is the only thing that tells a GenList routine that it has encountered a sublist. Without that information, GenLists could not properly dispose of its lists, and could not traverse structured lists. ListCode's value is 65535.

Use of the type code system is strongly suggested, but is not mandatory, when you insert strings (ARRAYs OF CHAR) into lists. If GenLists can determine from the type code that the element is a string, it can avoid allocating memory for data past the end of the string's null terminator, and it can relax its type checking for string variables of different sizes. GenLists'



## Generic List Handling

StrCode constant is what tells it that the element is a string. StrCode's value is 65534.

Programs that use many lists, or that need to mix data types in a single list, may need to use the type code system more extensively. The simplest way to use it is to assign numeric codes to all the types going into a GenList, and then check each element's type code before copying it into one of your variables. The most efficient way to check the type code is with the following routine:

```
PROCEDURE TypeCheck( TheList: GenList;  
    TheElmt: CARDINAL ): CARDINAL;  
VAR TypeCode, DummySize: CARDINAL;  
    DummyAddr: POINTER TO CARDINAL;  
BEGIN  
    GetElmtAdr( TheList, TheElmt,  
        DummyAddr, DummySize, TypeCode );  
    RETURN TypeCode;  
END TypeCheck;
```

You may want to hide the details of your type checking scheme by writing separate Insert and Retrieve procedures for each type going into a GenList.

## 2.5. Trouble Shooting

The problem most commonly encountered by people first learning to use GenLists involves knowing when and when not to dispose of a list with GenLists' DisposeList procedure. The general rule is that, once a list has become a sublist of another list, it cannot be disposed of before the parent list (the list into which it was inserted) is disposed of; GenLists deallocates sublists automatically when it deallocates parent lists.

If you are not sure whether one list contains another, you can test for it with GenLists' SameList procedure. SameList returns TRUE if the two lists passed as its parameters point to the same area of memory, or contain elements that point to the same area of memory. It does not actually compare the data in the two lists, so it returns FALSE if one list is a true independent copy of the other. And if you want to be really careful, as you might if you were tracking down a bug related to premature list deallocation, you can turn on the GenLists DiagMode switch. DiagMode is a boolean variable exported from GenLists. Its value is FALSE by default. If you set it to TRUE, GenLists calls an internal diagnostic procedure every time it is asked to insert one list into another. The diagnostic procedure makes sure that the list being inserted has not already been inserted at some other point, and that the two lists do not share any sublists.



## 3. Screen Design/Display System

### 3.1. Overview

*Repertoire's* screen design/display system is a powerful, flexible tool. You can use it with or without the *Repertoire* screen compiler that produces fast-access databases of compressed screens. You can also use it as a supplement to most of the conventional window managers presently being sold for DOS machines. Using the screen system without the screen compiler is simple and adds features like horizontal and vertical scrolling, input management, context sensitive help, etc., to a conventional window management system. Using the screen system with the screen compiler provides the same advantages, and a great many more. We begin with instructions for using the screen system with the screen compiler, since that is how most people use it.

Use of *Repertoire's* screen design/display system involves four basic steps:

1. Using your favorite editor, or the editor in `scrncomp.exe`, create a file of screens that look just the way you want them to be displayed. A command area at the top of each screen tells *Repertoire* about the input fields on the screen: what kind of fields they are, what range of values is acceptable, etc.
2. Execute *Repertoire's* screen compiler (`SCRNCOMP.EXE`) to transform the file of editable screens into an encoded file of displayable screens.
3. In your program, when you want to display a specific screen and accept user input into that screen, call one of *Repertoire's* display routines, passing it that screen's name. The screen named will be displayed. Next, call the *Repertoire* `ControlFrame` procedure. It will control the user's interaction with this screen.
4. The user can make entries in various kinds of input fields, choose between designated "choice" fields, and press the arrow keys to move between fields. Certain keys allow the user to ask for Help, to go back to the last screen, to exit the program, and to perform other functions that you specify. Extensive field-editing functions are handled automatically, including insert/overwrite modes and full scrolling and word-wrap in multi-line (editor) fields. The screens also contain logic that lets one of *Repertoire's* higher level procedures (`FrameSequence`) interpret user input and thereby determine the next screen to display. That logic might indicate, for example, that if a given field is selected, a screen named 'EntryForm' is to be displayed next. *Repertoire* will display this next screen and control its input, proceeding until the



user exits the screen sequence. Your program can interrupt this sequence and examine user input at any point.

*Repertoire* contains routines that allow your program to be involved at any level in the process described above. At the highest level, you can call various procedures from the ControlUtils module that deal with *groups* of screens. You can call the routine FrameSequence, for example, passing it a starting screen name, and *Repertoire* will display and control user interactions with all screens until the user exits. At a lower level, you can display a screen from a file with the Display routine, and then pass it to a routine named Input. After Input returns, your program can call other *Repertoire* routines to examine and process the user input from each field, by field name. At a still lower level you may, for example, call one *Repertoire* routine to read a screen from a file, then change that screen in your program, and then call another routine to display the resulting screen.

The sections below explain the process in stepwise order.

### 3.2. Your Screen-Creation Editor

*Repertoire's* screen compiler includes an integrated environment in which you may create, edit, display and run frames and sequences of frames. The editor in SCRNCOMP.EXE is primitive compared to editors like Brief, Point, Microsoft Word, PC-Write, etc. We recommend that you find a heavy-duty professional editor for most of your screen creation work.

You can use a standard editor because *Repertoire's* screen compiler uses plain ASCII files. It allows most of the "non-printing" characters (i.e., those with ordinal values below 32 or above 127) as well. You can use it with any editor that produces plain ASCII files, but we recommend an editor that displays all non-printing characters, that allows entry of non-printing characters from the keyboard, and that includes convenient support at least for line and box drawing with character graphics. Most editors do, and most word processing programs support non-document (i.e., plain ASCII) modes that will work as well.

### 3.3. Creating the Screens

Two kinds of files are involved when you use *Repertoire's* screen system in conjunction with the screen compiler (ScrnComp.EXE): .EDT files and .DSP files. Files with the .EDT suffix contain the equivalent of source code for the screen compiler. Files with the .DSP suffix are actually fast-access databases of compiled screens and screen control information. Under your program's control, *Repertoire* pulls screens out of .DSP files at runtime and displays them in any way you desire.

Although we sometimes use the terms interchangeably, we generally try to



speak of each "screen" defined in a *Repertoire* screen file as a "frame," and we refer to the area of the screen in which the frame is written as its "window." By "screen" we mean the face of the monitor--the entire physical display presented to the user. A "frame" is simply any group of text and menus, etc., that the screen system identifies with a single name and treats as a unit.

Although *Repertoire* makes it easy to use multiple .EDT and .DSP files, most users keep all of the frames for a particular program in a single .EDT file. Each frame contains two parts: the command area, and the frame text.

### 3.3.1. Command Area

The source text for a frame always begins with a Command Area, which is delimited by a pair of strings called command delimiters. By default, the command delimiter that marks the start of the command area is:

```
=====
```

The delimiter that marks the end of the command area is:

```
-----
```

The actual strings used must be at least 20 characters long, and can be longer they normally extend across the entire screen in examples given in this manual. The defaults can be changed, as explained in section 4.3.1.2 below. The lines in the Command Area will not appear on the screen when the frame is displayed.

Generally the first line inside the command area contains the name of the frame. Your programs should pass this name to the display routines to identify a frame when you want to display it. The frame name can actually appear anywhere in the command area; *Repertoire* recognizes it as the frame name by a colon in the first column. The colon is not a part of the name. The name can be up to 20 characters long, and can contain any combination of alphabetic or numeric characters. Spaces and punctuation marks are not allowed. *Repertoire* is sensitive to case in frame names.

This is a very simple example of a frame containing no input fields:

```
=====
:HelpTextOne
HitAnyKey
-----
```

```
The interest rates used in this
computation are for purposes of
illustration only. They may not
be accurate for your business or
your area of the country.
```

To change the program's interest

## Screen Design/Display System

rate assumptions, please pull down the "Options" selection on the main menu, and then select "Rates."

The Command Area tells *Repertoire* what to do with this frame. In the example above, the command area tells *Repertoire* to display the frame with a prompt line asking the user to "Press any key to continue," and to retain control of user input until the user does so.

You can use any of the following types of commands in a command area:

1. Fields: the control-information for the fields in the frame. 2.

Global Definitions: you may change *Repertoire*'s defaults for some of the special identifiers it searches for in processing screen files. Specifically, you may change:

The Field Mark: the symbol that indicates the start and end of a field in the text of the frame.

Command Begin/End Strings: the strings that delimit Command areas in a *Repertoire* frame.

3. Links to other frames. Several high-level *Repertoire* procedures can handle sequences of frames with no intervention from your program. They do so by using frame link values optionally defined in each frame and simply returned by the lower-level procedures. Frame links have labels like 'NormalNext,' 'ParentFrame,' 'FrameAbove,' 'FrameLeft,' etc. If your program uses only the low level procedures, it can ignore the frame links.
4. Help: a message or the name of a help frame that will be shown when the user presses the help key in this frame. If you use a frame name here, it will actually designate the entry point in a sequence of help frames that *Repertoire* will execute until one of them returns '0' as its interframe link value.
5. Data: any data that your program is likely to need while the frame is displayed. The data is accessible as a linked list (see the explanation of GenLists in the *Repertoire Reference Manual*).
6. Window: The window commands indicate features of the physical display window. The currently available commands indicate where to display the frame, whether a box should be drawn around this frame, what sort of box should be drawn, and which line of the window is the bottom of the heading when scrolling is enabled.
7. Colors: *Repertoire* allows you to assign colors and monochrome attributes to any area of the frame. Monochrome attributes can be mapped onto colors in any way you please--you do not have to be



content with the odd system employed by IBM display adapters. The Colors section of the command area provides a powerful system of defining colors and delimiters for colored areas of the frame.

8. Action: Tells *Repertoire* whether to control the user's interaction with the frame or simply display it. The keyword 'DisplayOnly' at the top level of a command area, for example, will cause a frame merely to be displayed; the program will not stop for user input. The keyword 'HitAnyKey' at the top level will cause the program to stop after displaying the frame and prompt the user to press any key before continuing.

On first reading of the following discussion, you may want to refer to section 3.6, which contains an example.

### 3.3.1.1. Field Commands

Field commands indicate the type of fields found in the frame text. In the frame text (i.e., the area between the line of dashes and the beginning of the next frame), each field begins and ends with the Field Mark (see section 3.3.1.2). There is one field command for every field, and the commands must be in the same order as the fields in the frame. These commands tell the *Repertoire* input routines what kind of input is acceptable in these fields. There are five additional options for any field: the selection character, the name, the 'required' designation, a help frame name, and a special prompt for this field.

The combined format for these options is:

```
(ch) 'Name' FieldType; REQUIRED; PROMPT; HELP HelpName
```

Example:

```
=====
:ExampleFrame
Fields:{
  () 'FileName' string; required; help Frame45

  (c) 'comments' editor; 3 lines; prompt
  Enter comments about the file in this window.
}
```

```
-----
                        File Name Entry Form
```

```
#
```

```
#
```

```
#
```

```
#
```

```
=====
```

## Screen Design/Display System

The first field in this example requires entry of a string. If the user presses the help key while the cursor is in this field, *Repertoire* shows one or more frames beginning at Frame45. Your Modula-2 program can use the name 'FileName' to refer to the user's input in this field.

The second field allows multi-line input. Text within the field will wrap and scroll. The field has no help frame sequence specifically associated with it, but it has a special prompt message ("Enter comments about the file . . .") that will be displayed while the cursor is in the field. The user can move freely in and out of this field, but can reach it directly by pressing the selection character *c* while in any other non-string-entry field.

The row of '=====' at the bottom of the example is not necessary; it shows where the next frame would start in the .EDT file.

The Selection Character: Each field command starts with a pair of parentheses, optionally enclosing a character. If the user enters that character while this frame is displayed, InputManager.ControlFrame will move the input cursor (the PointerBar) to that field. In the formats shown below, the selection character is "x". Selection characters are not case sensitive. Any alpha-numeric character can be used as a selection character. Extended keys (function keys, arrow keys, etc.) may not be used as selection characters.

To let the user know that the selection character is available, *Repertoire* automatically highlights the first character of the field's text that matches the selection character. Case DOES matter in this process; if you enter the selection character as an uppercase 'A', for example, the first uppercase 'A' in the menu item's text will be highlighted, and any lowercase 'a' that precedes it will not be highlighted.

The Name: After the selection character in parentheses comes the field name, enclosed in single quotation marks. Field names may contain any characters except blanks, colons, and braces. Each name may be up to 16 characters long. If you do not want to give a field a name, you must include a pair of empty quotation marks after the selection character. In the example formats below, the name is shown as either 'name' or none.

Example:

```
() 'FileName' string
```

Immediately following the field name you must designate the field type. The various field type designators are explained below.

The next three options are KEYWORDS that may optionally follow the field type designation:

Required: Any field may contain a semicolon and the word "required" after



it. If it is "required," the field must be filled in before the user submits the frame. Required fields are not checked if the user is exiting from the program or backing up to the previous frame. If a required field is not filled, *Repertoire* will prompt for it again.

**Help Frame:** Any field may contain a semicolon and the word "help" after it. There must be a space after the word help followed by a frame name. By specifying a "Help Frame" for a specific field, you can have *Repertoire* go to that frame in the file if the user asks for help (by pressing a help key that the programmer may define) while "in" this field. This help frame name actually designates the entry point in a sequence of help frames, so it is important to use '0' (ScrnTypes.ProgramExit) for the ParentFrame and NormalNext link values in the last frame of the sequence. This is what tells *Repertoire* that the help sequence has ended.

**Prompt:** Any field may contain a semicolon and the word "prompt" after it. The line following any field line that contains "prompt" is the field-specific prompt line for this field. When *Repertoire* is controlling the user input to a frame, a prompt line is shown centered at the bottom of the screen. This prompt will by default be determined by what kind of field the user is in at the time. For example, if the field is numeric, the prompt line tells the user to enter a number between X and Y. By specifying "prompt" for a field, however, you can override the default and force any text you enter on the next line to replace the default prompt for this field type. The entire Fields command area starts with "{" and ends with "}".

**Format:**

```
Fields:{
(Select) 'Name' FldType; REQUIRED; PROMPT; HELP HelpNum
*
*
}
```

**Field Types:** There are ten types of field entries:

1. **String:** allows entry of a string of characters.

Format: (x) 'name' String; KEYWORDS

**Example:**

```
(c) 'children' string; prompt; help ChildHelp
Enter the names of your children here.
```

2. **Integer:** allows entry of a whole number between min and max. The range clause is mandatory.

Format: (x) 'name' Integer [min..max]; KEYWORDS

**Example:**

## Screen Design/Display System

```
() 'hatsize' INTEGER [4..12]
```

**3. Real:** allows entry of any real number between the real numbers min and max. Negative numbers and numbers with fractional parts are allowed. The range clause is mandatory. The specification of the number of decimal places is optional.

Format: (x) 'name' Real.decplaces [min..max]; KEYWORDS

Example:

```
() 'velocity' real [0..99999]; required  
( ) 'amount'    real.2 [0..99999];
```

**4. Date:** Allows entry and validation of dates in the following formats, mm/dd/yyyy or mm/dd/yy the separator can be any non-digit. Conversion and validation is done by DateFunctions.StrToDate. Remember to leave 8 or 10 for the date.

Format: (x) 'name' Date; KEYWORDS

Example:

```
() 'StartDate' date; required
```

**5. Editor:** allows multi-line string input. In an editor field, *Repertoire* is actually calling its full-screen text editor in a small window and making it look like a field. Text wraps and scrolls in that window. An unlimited amount of text can be entered (unless you explicitly limit it). The vertical dimensions of the window are indicated immediately after the word 'editor' with a number, followed by the word 'lines.' Its position and horizontal dimensions are indicated in the text of the frame, just as they are with any other kind of field. If you do not want to let the user change the text of the field, you can add the keyword 'ReadOnly' after designating the number of lines in the editor window. And if you want to set an upper limit on the number of lines the user may enter in an editor field, you can use the keyword 'MaxLines' followed by a number designating the limit.

Format:

```
(x) 'name' Editor; x lines; ReadOnly; MaxLines x
```

Examples:

- ```
( ) 'descrip' editor; 4 lines; help DscHlp; prompt  
Describe the accident in your own words.  
  
( ) 'comments' editor; 3 lines; MaxLines 20; prompt  
Please tell us what you think of our program.  
  
( ) 'HelpText' editor; 6 lines; ReadOnly; prompt  
You cannot change the text in this window.
```



ReadOnly editor fields have two purposes. First, they allow a good equivalent for multiline menu options. Normal menu options (see the explanation of Goto fields, below) can have only one line. If you need more than that, you can fill ReadOnly editor fields with the text of your menu option1. So long as the text is not wider than the field's window and does not have more lines than the window, the text will not scroll while the pointer bar is on it. The user will perceive the field simply as a multi-line menu option.

First to fill an editor field with text, you must use the ListToEdField procedure from the ScrnUtil module. You can also put text in the field's data list; it will automatically be loaded into the editor field when the frame is read from the .DSP file.

Second, they give you a convenient way to display large amounts of information. The user can page up and down or scroll left and right in a ReadOnly editor field, but can't change any of the text. Of course, you could also use a normal DisplayFrame to do the same thing, since *Repertoire* can scroll DisplayFrames as easily as editor text. But there are often situations where the text has to be created dynamically. In those cases it is much easier to use ReadOnly editor fields. Editor fields also consume a slightly smaller amount of memory.

**6. Goto:** This is *Repertoire*'s idiosyncratic name for a menu option. A goto field allows no entry, but highlights the marked portion of the frame when the user moves the pointer bar into this field. If the user exits the frame with this field highlighted, ControlFrame returns the NxtFrame value (below). Also, if the user presses the selection character for a Goto field, ControlFrame exits and returns the NxtFrame value.

Format: (x) '' GoTo NxtFrame; KEYWORDS

### Example:

```
(e) '' GoTo TaxForm; prompt; help TaxHelp
This option allows you to edit your 1976 return.
```

The reason *Repertoire* calls these fields 'Goto' fields is that, in most cases, the user's selection of a menu item indicates that your program should branch to a different frame. Goto fields allow you to specify the name of that frame. This name is returned through a variable in the ControlFrame procedure's parameter list. High level routines such as FrameSequence and ControlFrameGroup in the ControlUtils module use this value automatically to display and control the next frame. If you call ControlFrame directly, however, you may still want to use this value to determine which frame should next be processed. If you want selection of this field to cause one of the high level procedures controlling a sequence of frames to terminate, use '0' as NextFrame. Otherwise, use the name of the frame to be called if this field is chosen. In the second frame of the example in section

## Screen Design/Display System

3.6, the fields are all Goto fields.

7. **Group**: defines the following N fields as all members of a "group." All group members must be Choice fields. When the user moves into a group, *Repertoire* lets the user select a single member of the group as the "choice." For example, a yes or no question could have "Yes" and "No" as the choices, as in frame 3 in the example in section 3.6. "Group" really is not a field itself, but indicates that the following fields are grouped. Therefore, a group field does not have a matching set of field markers in the frame text, as all other fields have. Similarly, specifying a selection character for a group field has no effect.

Format: () '' Group N

8. **Choice**: represents one choice in a group field.

Format: (x) 'name' Choice; KEYWORDS

Example:

```
() '' Group 2
(y) 'yes' choice
(n) 'no' choice
```

The selection character of a choice field will be highlighted in the visible text of the field if *Repertoire* finds a character that matches it in case. You can use the fact that this match is case sensitive to control from the .EDT file whether the selection character is highlighted.

9. **Boolean**: represents a Boolean value.

Format: (x) 'name' Boolean; KEYWORDS

Example:

```
() 'picked' Boolean
```

A "√" will be displayed in the first position if the field when selected.

```
# Picked#
```

Will display as:

```
√Picked
```

or

```
Picked
```

10. **Display**: represents a "field" that the user cannot even move into, but that programs may want to change. We identify such fields only so we can give them a name and thus allow the controlling program to find them eas-



ily. Thus, if there is a place in the frame where you want your program to print a different value every time the program is run, but that users are not allowed to change, it is a "Display" field.

```
Format: () 'name' Display
```

In summary, the fields are of three basic types: entry fields, in which the user enters something (String, Editor, Integer, and Real); the fields that handle choices (Goto, Group, Choice); and display fields. In choices, the user enters nothing, but selects a field by moving the PointerBar.

### 3.3.1.2. Global Definitions

You may redefine three special markers that *Repertoire* looks for in processing screen files: the field mark, the delimiter string that marks the start of the command area (BeginCmdLine), and the delimiter string that marks the end of the command area (EndCmdLine).

The field mark delimits the start and end of a field in the text of a frame. Since this symbol is imbedded in the normal frame text, it must be unique; it only appears when a field is being defined. The default is '#'. As noted above, for every field in the Fields section of the command area (except a group designator), a pair of field markers must appear in the frame. The area between the markers will be highlighted when the user moves into that field. For integer, string, and real fields, the marked field is usually just a pair of markers separated only by spaces. Alternatively, you could put a default value in such a field.

To set the Field Mark character, include this statement at the beginning of a command area:

```
Field Mark: X [replace]
```

(where X is the character you want to use for field marks). This character can be any ASCII or extended ASCII value. For example, you could use the Greek capital Sigma (ordinal value 228). The field mark designation remains in effect for all following frames until you set it again.

Immediately after designating the Field Mark character, you may optionally append the word 'replace' (separated from the field mark by a semicolon). This tells the screen compiler that it is to replace the field mark with blanks when it processes the text of the frame; otherwise, the screen compiler will delete field marks from the frame as it processes them. Using the replace option makes it easier to see how frames with multiple columns will be displayed, but it reduces your control over the precise placement of fields. The Replace option does not stay in effect across frames.

The BeginCmdLine tells *Repertoire* where a Command area starts; the EndCmdLine marks the end. Any other text on one of these lines will be

## Screen Design/Display System

ignored. The default BeginCmdLine is the string:

```
=====
```

(at least 20 chars long)

The default EndCmdLine is the string:

```
-----
```

(at least 20 chars long)

To change them, include the command BeginCmdLine= on a line by itself in the Command Area. The following line will be read and stored as the BeginCmdLine henceforth in this file. The command EndCmdLine= will cause the following line to be used as the EndCmdLine. For example, to set the BeginCmdLine to a row of "\*", and the end to a row of '- ' (alternating dashes and blanks), include the following in your first command area:

```
=====
BeginCmdLine=
*****
EndCmdLine=
- - - - -
.
.
.
-----
```

Notice that you must end the Command Area in which you set a new EndCmdLine with the OLD EndCmdLine. The new end line can be used for the next frame.

### 3.3.1.3. Links to Other Frames

These commands tell high-level *Repertoire* procedures such as FrameSequence and ControlFrameGroup the name of the frame to be displayed next. Even lower-level procedures such as ControlFrame always return a name that can represent the next frame to be displayed, if you choose to do so. Each Frame Link keyword governs a different situation, and determines what the NextFrame value will be only in that situation. Frame links always start with a keyword followed by a colon, and then the name of a frame. These are the frame link keywords presently supported:

FrameAbove:

FrameBelow:

FrameLeft:

FrameRight:

NormalNext:



**ParentFrame:**

This is what they do:

**FrameAbove:** If the user presses the up arrow while the pointer bar is at the top of a frame in which you have declared a `FrameAbove`, `InputManager.ControlFrame` returns the frame name defined for `FrameAbove` in its `NextFrame` parameter. That allows the high-level procedure `ControlFrameGroup` to find the `FrameAbove` in its list of preloaded frames, and to pass that frame to `ControlFrame`, placing the pointer bar at the bottom of the new frame. The user perceives the two frames as one.

**FrameBelow:** If the user presses the down arrow while the pointer bar is at the bottom of a frame in which you have declared a `FrameBelow`, `InputManager's ControlFrame` procedure returns (in its `NextFrame` parameter) the frame name defined for `FrameBelow`. The `ControlUtils.ControlFrameGroup` procedure finds that name in its list of preloaded frames, passes the frame variable to `ControlFrame`, and places the pointer bar at the top of the new frame.

**FrameLeft:** If the user presses the left arrow while the pointer bar is at the left edge of a frame in which you have declared a `FrameLeft`, `InputManager's ControlFrame` returns the frame name defined for `FrameLeft`. The `ControlFrameGroup` procedure finds that name in its list of preloaded frames, passes the frame variable to `ControlFrame`, and tells `ControlFrame` to place the pointer bar at the bottom of the new frame.

**FrameRight:** If the user presses the right arrow while the pointer bar is at the right edge of a frame in which you have declared a `FrameRight`, `InputManager's ControlFrame` returns the frame name defined for `FrameRight`. `ControlFrameGroup` passes the corresponding frame variable to `ControlFrame`, and tells `ControlFrame` to place the pointer bar at the bottom of the new frame.

**NormalNext:** represents the frame that will follow by default. `Goto` fields override `NormalNext`; if the pointer bar is on a `Goto` field when the user presses `Return` or the submit key, `ControlFrame` will return the "next" frame name listed in that `Goto` field definition (see section 3.3.1.1). If the user presses the submit key or the exit key when any other kind of field is highlighted, `ControlFrame` returns the frame's `NormalNext` frame name.

**ParentFrame:** If the user presses the key you have defined as the `BackUp` key, `ControlFrame` will return the frame's `ParentFrame` string.

If your frame designates a `ParentFrame` but not a `NormalNext`, `NormalNext` defaults to "" (a null string). Conversely, if your frame has a `NormalNext` but not a `ParentFrame`, `ParentFrame` defaults to "".



## Screen Design/Display System

After a frame is loaded into memory (using `DspFiles.ReadFrame`) you can determine its `ParentFrame` and `NormalNext` from the `.parent` and `.normlnext` fields of the `DisplayFrame` record. That allows you to write loops like this:

```
REPEAT
    Control( NextFrame );
    (* Code to process the user's input would
       go here. *)
UNTIL Equal(NextFrame, YourFrame^.normlnext);
(* Means we will stay in this loop until
   the user presses the submit key. *)
```

### 3.3.1.4. Help

The help command designates the default help string for this frame. Help processing is done during user input when the user presses the help key (F1 by default). If a help frame has been specified for the field that the cursor is in when help is requested, *Repertoire* uses that field-specific help frame; otherwise it uses the frame-specific help specified here. Help processing takes two forms:

**Help Frames:** If the help string is a valid frame name (i.e., a single string less than 20 characters long), *Repertoire* (actually, the `FrameSequence` procedure in the `ControlUtils` module) presents and controls the sequence of linked frames beginning with the named help frame (if the help frame has frame links) until some frame in the sequence returns '0' (i.e., `ScrnTypes.ProgramExit`) as its `NextFrame` value. *Repertoire* uses '0' to detect the end of the sequence so that the same sequence of frames can be called from any point in a program. When `FrameSequence` encounters a '0', it completely restores the appearance of the current frame, and returns the cursor or pointer bar to the position it was in when the user pressed the help key.

`FrameSequence` considers it an error for a frame in a help sequence to return the name of the frame in which the help sequence was called. It checks for this condition to prevent one of the most common mistakes made by users of the Release 1.4 screen system. People would terminate help sequences with explicit returns to the calling frame (i.e., they would make the calling frame the `ParentFrame` or `NormalNext` of the help frame). This caused *Repertoire* to think every screen in the application was part of the help sequence. This was the "Ghost Application" phenomenon; it looked like the application was almost working, but none of the programmer's Modula-2 code was being executed at all.

**Help Messages:** If the help string is not a valid frame name (i.e., it contains spaces or is longer than 20 characters), *Repertoire* will show it to the user in a window when the user presses the help key. Like the `Fields` command,



Help must start with "{" and end with "}". Everything in between is the help string.

```
Format:  Help:{
          *
          *
        }
```

### 3.3.1.5. Data

Each line in the Data section of the Command Area is loaded into a list accessible through the DisplayFrameRec associated with the frame. Your programs can refer to the list as YourFrame^.DataList. See Chapter 2 for an explanation of how to manipulate lists of this sort.

*Repertoire* provides special support for using the data section with its natural language analysis tools to perform complex analysis of user input. You can use it to pass various language processing tests that can be applied to any field in the frame by the calling program. Each test can return a value indicating, for example, whether a word or phrase is present or where it occurs in relation to other phrases. Any boolean combination of such conditions can be specified.

This allows your programs to respond to english questions or commands by jumping to an appropriate frame. The StrLogic Section explains *Repertoire's* language analyzer and how to construct these tests.

```
Format:  Data:{
          *
          *
        }
```

### 3.3.2. Window Commands

These commands describe the physical window for Display and ControlFrame. The Windows section must begin with "{" and end with "}".

```
Format:  Window:{
          *
          *
        }
```

The current window commands are: Position, Box, EntryBox, ExitBox, Caption, Heading Line, and Overlay.

Examples of Window Command Sections:

```
Window:{
  Position: ( 20, 10, 75, 15 )
  Single Box
  (*This frame will be drawn in a long
  horizontal box near the top of the
  screen, outlined with a single box.*)
}
```

## Screen Design/Display System

```
}  
  
Window: {  
    Overlay  
    No Box  
    (*This frame will occupy the entire  
    screen and will not be outlined with a  
    box; the screen will not be cleared  
    before it is displayed.*)  
}
```

**Position:** tells *Repertoire* where to display this frame on the monitor. You indicate the column and row of the upper left and lower right corners of the window to use. All counting starts at 1, not 0. The corners of a normal 25 by 80 frame are specified as (1, 1, 80, 25). If the Position command is omitted, default coordinates of (1,1,80,25) are used. Programs that call the lower-level *Repertoire* routines have the option of overriding the coordinates defined here. The format is:

```
Position: (UprLeftCol, UprLeftRow, LwrRghtCol,  
          LwrRghtRow)
```

You can use values approaching 256 to specify bottom and right edges of the main window if you want your program to adapt to unusual screen dimensions at runtime. For example, if you specify a bottom row of 254, and *Repertoire* determines at runtime that the screen has 43 lines, the actual bottom row will be on line 41.

**Box Commands:** Release 1.5 added a powerful new method of controlling the appearance of the box surrounding each frame. Earlier releases provided only three options: NoBox, SingleBox, and DoubleBox. Those options are still supported for compatibility with earlier releases. When *ControlFrame* is invoked for frames surrounded by a single-lined box, *ControlFrame* now changes it to a double-lined box while the frame is active. What is actually happening is that *ControlFrame* is drawing the *DisplayFrame^.EntryBox* on entry, and *DisplayFrame^.ExitBox* on exit (*FramePainter* draws the *^.ExitBox* when the frame is initially displayed). You can change these strings so as to have any sort of border on entry and exit. You can also set them in your .EDT files using new *EntryBox* and *ExitBox* commands in the Window section of the command area, and you can designate a Caption that will be centered on the top border of the window.

The *EntryBox* command allows you to specify every character in the box that surrounds a frame. It must be followed by a colon and a string containing exactly eight characters. The first character in the string will be used for the top left corner of the box; the second will be used for the top horizontal edge; and so on, in accordance with this diagram:

0-----1-----2



```

|           |
3           4
|           |
5-----6-----7

```

The ExitBox command does the same thing for frames that are not active.

The FramePainter module will use the border rows and columns of the defined frame to draw these boxes. Therefore, DO NOT put text in the first column or row if the frame will be enclosed in a box.

Heading Line: tells *Repertoire* the lowest line in the frame that is a "heading line" and should not be scrolled. ControlFrame has the ability to display a frame in a window smaller than the defined frame and scroll through the window as the user presses the arrow keys to move around in it. If each of your frames fits inside the window in which it will be displayed, you do not need scrolling, so don't worry about the Heading Line.

If you do need scrolling, ControlFrame will scroll the entire window up or down when needed, except for lines 1 through HeadingLine.

Format: HeadingLine: N

Overlay: When your program tells *Repertoire* to display a frame, the window in which the frame will be displayed must be defined, either in the screen file or in the parameters passed to the ShowDisplayFrame routine. *Repertoire* will automatically clear this window before displaying the frame. If you want *Repertoire* to overlay the frame on top of what is already on the screen, use the window command Overlay. For example, you may want to ask the user to enter a file name. If the file does not exist, you may want to overlay a prompt at the bottom of the screen that asks, "This file does not exist. Do you wish to create it [Yes/No]?".

Format: OverLay

The OverLay command actually has an opposite named ClearFirst, but since the frame's window area will be cleared first by default, you should never need to use the ClearFirst command. Omission of the OverLay command accomplishes the same thing.

ClearAfter: the ClearAfter command is new with Release 1.5b. It tells *Repertoire* that the window area occupied by this frame is to be cleared after ControlFrame finishes operating on the frame. If the frame happened to be obscuring another frame associated with the same physical window, the underlying frame (or frames) will be restored. *Repertoire* does not accomplish this through screen save/restore operations because the possibility of multiple processes prevents a frame from knowing that the obscured frame has not changed while it was in the background.

Caption: The Caption command lets you define a string that will be cen-

## Screen Design/Display System

tered on the top border of the frame's window. You must enclose it in quotation marks (single or double). The quotation marks will not appear when the caption is written.

Example:

```
Window:{  
    Position: ( 20, 10, 75, 15 )  
    Caption: 'File Saving Options'  
}
```

### 3.3.2.1. Colors

*Repertoire's* screen display system includes several integrated features to display frames in color:

1. By default, all frames will be displayed in color when the terminal is operating in a color mode. If you do not specify colors for different areas of the frame, *Repertoire* uses defaults that are set in the *ScrnType* module's *InitDisplayFrame* procedure.
2. The programmer may create color delimiters in the editable screen file and use them to paint areas of the screen and pieces of text in different colors.
3. Commands in the editable screen file may force areas of the frame that *Repertoire* controls (e.g., the pointer bar that indicates the current choice in the *ControlFrame* routine) to be shown in specified colors. It includes an option that selects a default color for all text in the frame not specifically given another color.
4. When displayed on a monochrome monitor, these same frames will be shown in a mix of monochrome attributes (e.g., bold, reverse video, plain) chosen by the programmer. The colors chosen do not have to translate to monochrome attributes in the restricted manner that IBM translates colors to black and white. Rather, specific monochrome attributes can be specified for each color combination. *Repertoire* will automatically detect whether the display adapter is in a color or a black & white mode and use the appropriate character attributes.

The Color commands are similar to the Windows or Fields commands in that they are set off from other commands in a command area by a key word. It looks like this:

```
COLORS: {  
    .  
    .  
    .  
}
```

A color sub-command consists of a special character which the program-



mer wishes to define as a color delimiter, and a set of numbers or color names in parentheses indicating the color combination that the character represents. The numbers or color names represent two things: the foreground (text) color, and the background color. After the color designation, the programmer should specify the monochrome attribute that this region should have on monochrome monitors.

Colors are indicated by name or by the cardinal number of the color in IBM's color scheme (*Repertoire* supports color numbers only for purposes of compatibility with its earlier releases). See the list of colors in the table below. One restriction in color choices must be noted: IBM-compatible display adapters use only the first eight colors (0-7) as background colors. If one of the second set of eight colors is put in the background, the adapter will use the color that is eight places lower in the list, and will blink the resulting text. Thus, you can use the second set of eight backgrounds when you want to blink colored text. The monochrome attributes are designated by key words shown in Table 2.

Table 1: Color Numbers and Names

Number Color

|    |              |
|----|--------------|
| 0  | Black        |
| 1  | Blue         |
| 2  | Green        |
| 3  | Cyan         |
| 4  | Red          |
| 5  | Magenta      |
| 6  | Brown        |
| 7  | LightGrey    |
| 8  | DarkGrey     |
| 9  | LightBlue    |
| 10 | LightGreen   |
| 11 | LightCyan    |
| 12 | Pink         |
| 13 | LightMagenta |
| 14 | Yellow       |
| 15 | BrightWhite  |

Table 2: Monochrome Key Words

|              |
|--------------|
| Plain        |
| ReverseVideo |
| Bold         |
| UnderScored  |
| Blinking     |

## Screen Design/Display System

For example, to define the dollar sign as a delimiter for areas of yellow text on a blue background that will be shown in plain text on monochrome monitors, the color section of the command area would read:

```
Colors: {
    $ = (YELLOW, BLUE) PLAIN
}
```

Any number of these color delimiters may be defined in one color section of the command area. These codes may be used in any subsequent frame in the file, so they need be defined only once, in or before the first frame in which they are used.

To use one of these delimiters, put it before and after a text string to be printed with the special color. Delimiters may be nested. For example, the second frame of the demonstration program named PMIDBMS includes the following color designations:

```
=====
:Welcome
Colors:{
    % = (BrightWhite, blue) Bold
    (* % delimits areas in which Bright White
        characters appear on a blue background on
        color monitors; bold characters on monochrome
        monitors.*)
    & = ( yellow, cyan ) UnderScored
    (* & delimits areas in which Yellow characters
        appear on a Cyan background on color monitors;
        UnderScored characters on monochrome.*)
}
```

```
-----
%REPertoire Screen Display System Demonstration%
```

```
& Welcome to a demonstration of REPERTOIRE's &
& Screen Display System. REPERTOIRE supports &
& the types of screen interactions that you &
& will see on the next few screens. &
```

### Standard Frame Parts.

*Repertoire* controls certain areas of the frame for which you may wish to choose colors. This is especially important for the InputManager routine that controls user interactions with the frame. For example, InputManager uses a Pointer Bar to indicate which field the user is currently "in," Prompts to show what entries are expected in this field, and Messages to give the user help.

The colors to paint these areas may be specified in the Colors section of the command area of the editable screen file, similar to the way color delimit-



ers are specified above. First, a frame part key phrase is entered, followed by an equals sign and the color designations. The Color key words are:

Table 3: Frame Part Key Phrases

| <u>Key Phrase</u> | <u>Meaning</u>                                                                                                                                                                                                         |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Normal Text       | All text in the frame not given another designation will be shown in this color. In addition, all blank areas of the window for this frame will have this background color.<br><br>Default colors = (pink, blue) plain |
| Border            | If a box is chosen to go around the frame, it will be shown in this color.<br><br>Default colors= (LightCyan, blue) plain                                                                                              |
| Pointer Bar       | The bar that InputManager uses to show the current field-during user input.<br><br>Default colors = (black, red) ReverseVideo                                                                                          |
| Prompts           | The line that InputManager displays (at the bottom of the screen by default) describing the valid entries for the current field.<br><br>Default colors = (black, cyan) ReverseVideo                                    |
| Messages          | Any message box chosen by a user in InputManager (e.g., the help window).<br><br>Default colors = (black, brown) plain                                                                                                 |
| Selected Text     | Text in a <i>Repertoire</i> choice group (see section 3.3.1.1 of the <i>Repertoire</i> manual) that is currently the selected field in that group.<br><br>Default colors = (LightMagenta, blue) bold                   |

The colors of these frame parts are set in the command area in the same way that delimiter definitions are set. Frame-part definitions and delimiter definitions may be combined in a single Color section of the command area. For example, the frame from the *Repertoire* demonstration shown in the previous example could use:

```
Colors:{
  NormalText = (red, blue) plain
  Messages = (black, green) ReverseVideo
  % = (BrightWhite, blue) plain
    (* % delimits areas in which Bright White
      characters appear on a blue background.*)
  & = ( yellow, cyan ) UnderScored
}
```

Note that color definitions do not take effect until the frame containing the definitions is processed. However, the color delimiters are handled differently than the frame-part color designations.

Definitions of color delimiters are handled by the screen compiler, which processes frames sequentially from the top of the file. Color delimiter definitions should therefore be placed in the first frame of the screen file, or at least before they are used.

### 3.3.2.2. Automatic Screen Mode Detection

*Repertoire* will detect the current screen mode and, based on what it finds, choose color or monochrome attributes. Most color video adapter boards (e.g., the IBM color graphics card) automatically boot in color mode. However, some boards that mimic the color adapter board for monochrome screens (such as the Paradise graphics board), will boot in black and white mode. To ensure that the screen is being used as you intended, you can call the *ScreenMode* routine in the *SmartScreen* module. With it, you can set the video mode before displaying frames. Color screens normally should be set to the color80x25 mode. For further explanation, see the *SmartScreen* section of the *Repertoire Reference Manual*.

### 3.3.2.3. Using *Repertoire* in Graphics Modes

The present version of *Repertoire* is primarily intended for use in text-only modes. These are the modes referred to in the *SmartScreen* module as BW40x25, color40x25, BW80x25, and color80x25.

The best way to use Release 1.6 in graphics mode is to write an implementation of the *VWindows* module that uses the primitives of a graphics package such as *MetaWINDOW* or *Halo*. *VWindows* supplies every screen handling function used in the *Repertoire* screen system, and it does so through procedure variables. The high level parts of *Repertoire's* screen system, therefore, do not assume anything about the screen that is incompatible with graphics mode operation.

If you choose not to write an alternate implementation of *VWindows*, you can still use *Repertoire's* screen system in graphics modes under certain constraints.

The constraints are as follows:

1. Characters from IBM's extended character set (those with ordinal values between 128 and 255) will not appear in graphics modes unless you define a table of graphics characters and set interrupt vector 31 (hex 1F) to point to it. Graphics-mode environments such as Microsoft Windows do this for you automatically.



2. The CGA ROM routines do not translate monochrome attributes into appropriately drawn high-resolution graphics characters. The result is that, unless the program is running in a graphics-mode environment, you will not see an identifiable pointer bar on the screen in high-resolution graphics modes. You must instead use selection characters (see section 3.3.1.1) or other forms of user input.

### 3.3.3. Frame Text

After the EndCmdLine that marks the end of the command area comes the actual frame text. This should look exactly as you want it displayed on the screen. The first row after EndCmdLine will be written on the first row of the defined window, starting at the defined starting column. If you requested a Box to be drawn around the window, leave the first line blank; it will be used for the box. Remember to similarly leave the first column blank if you request a box.

The only characters in the frame text that will not look exactly as you type them are the Field Marks and the color/attribute delimiters. These will be omitted, and the adjacent text will be moved into the position formerly occupied by the mark (in other words, the marks are deleted, not replaced with blanks). You can make the screen compiler replace field marks with blanks if the resulting shifting of text makes it hard to align columns. (See the explanation of Field Marks above.) There are no other restrictions on what the rest of the frame may contain. The frame does not have to fit within the window defined for it because *Repertoire* will automatically allow the user to scroll the frame if parts of it are not visible.

## 3.4. Compiling Frames

*Repertoire* provides a screen compiler named SCRNCOMP.EXE. ScrnComp integrates the full display system with the editor and with the modules that actually process .EDT files and produce displayable .DSP files from them. As a result, when SCRNCOMP.EXE finds an error, it now lets you invoke the editor, with the cursor positioned to the source of the problem in the .EDT file. It also lets you switch rapidly between working on the editable form of a screen and viewing it as it will appear when displayed. You can execute a sequence of frames inside SCRNCOMP.EXE with the Run command. You can also delete screens from a .DSP file (since a .DSP file is really just an NdxFile), you can crunch a .DSP file to eliminate holes caused by deletions or size changes, and you can insert screens from multiple .EDT files into a single .DSP file.

You can still use the screen compiler from the DOS command line by using a trailing /c after the .edt and .dsp file names, or by removing SCRNCOMP.DSP from your hard disk. For example:



## Screen Design/Display System

```
scrncomp helpfile.edt helpfile.dsp/c
```

If file extensions are not specified the first will default to .edt and the second will be .dsp. Therefore the following will be equivalent to the previous example.

```
scrncomp helpfile helpfile/c
```

This causes ScrnComp to add every frame in helpfile.edt to helpfile.dsp. If you want to replace every frame in helpfile.dsp with frames in helpfile.edt, you should first erase helpfile.dsp.

ScrnComp's integrated environment uses menus with informative prompts and context sensitive help, and generally takes very little instruction. It uses three names that may require explanation: the current .EDT file, the current .DSP file, and the current frame. The current .EDT file means the last .EDT file loaded with the "Open .EDT file" selection under the Files option of the main menu. You can use this option at any time to change .EDT files. If you forget to open an .EDT file, and you attempt some other operation that requires an open .EDT file, ScrnComp will prompt you for the name of an .EDT file.

The current .DSP file means the last .DSP file opened with the "Open .DSP file" selection under the Files option of the main menu. The current .DSP file is where ScrnComp writes the output of the MakeFrame module every time it successfully compiles a frame. It is also the file from which ScrnComp retrieves frames when you ask it to display a specific frame, or to run a sequence of frames.

The "current frame" has meaning only after you have compiled or attempted to compile one or more frames. If you choose to Edit the Current Frame, ScrnComp invokes the editor, passing it the editable form of the last frame it tried to compile. If you choose to Display the Current Frame, ScrnComp displays the result of the last successful compilation of a frame.

Note that when you compile frames one at a time with the "Current Frame" and "Specific Frame" selections under the Compile menu option, color delimiter definitions and default color settings that are meant to be global may or may not be in effect, depending on whether the frame in which they are defined has been compiled earlier in the same execution of ScrnComp. You may want to place all color definitions and declarations in one frame, probably at the top of the .EDT file, so that you can compile that frame once before working on other individual frames. This will give ScrnComp (actually MakeFrame) all the color information it may need in compiling individual frames.

ScrnComp's Run command provides an excellent way to preview the appearance of screens and to prototype entire applications. It prompts for the



name of the first frame in the sequence, and then continues to execute the sequence until it encounters '0' (ScrnTypes.ProgramExit). If a frame returns a NextFrame value that is not in the file, ScrnComp prompts for a new entry point.

ScrnComp's .EDT file and source code are meant to be used as examples of new features provided in Release 1.5. When you need an example, look there first.

### 3.5. Displaying Frames in a Program

*Repertoire* has four basic levels of program access to the screen display and control facilities. At all levels, you must import the type DisplayFile, and the routines to open and close screen files, OpenDisplayFile and CloseDisplayFile. You must define a variable of type DisplayFile that will identify your screen file and you must pass this variable along with the file name to OpenDisplayFile. A separate DisplayFile variable must be declared for every screen file you want to open simultaneously in your program. Unless it improves the organization of your program to do otherwise, it is usually sufficient to create only one file for all the frames you use in a program. The basic shell is thus:

```
FROM ScrnTypes IMPORT DisplayFile;
FROM DspFiles IMPORT OpenDisplayFile,
    CloseDisplayFile;

VAR
    MyScreenFile: DisplayFile;
BEGIN
    OpenDisplayFile(MyScreenFile,
        'FileName.dsp');
(* ... *)
    CloseDisplayFile( MyScreenFile);
END ProgramID.
```

At this point, you may use any combination of the following routines, depending on what level of control your program needs:

1. At the highest level, you may use the routine FrameSequence, imported from ControlUtils, to display and control input to a series of frames. The only parameter is the frame name to be displayed first. *Repertoire* will display the frame, accept user input, and determine which frame to display next based on the user input and the frame links in the screen file. *Repertoire* will continue to display frames until the user exits. A program that does nothing else could look like:

```
MODULE ProgName;
FROM DspFiles IMPORT OpenDisplayFile, CloseDisplayFile;
FROM ControlUtils IMPORT FrameSequence;
FROM ScrnTypes IMPORT AFrameName, DisplayFile, MainFrame;
FROM VWindows IMPORT CurrentWindow;
```

## Screen Design/Display System

```
IMPORT InitCompilerMods;
VAR
  MyScreenFile: DisplayFile;
  NextFrame: AFrameName;
BEGIN
  IF OpenDisplayFile(MyScreenFile,
    'FileName.dsp') THEN
    NextFrame := 'FirstFrame';
    FrameSequence( MainFrame, NextFrame );
    CloseDisplayFile(MyScreenFile);
  END;
END ProgName.
```

The ControlUtils.ControlFrameGroup procedure does much the same thing, except that it preloads into memory all frames linked to the opening frame through its FrameAbove, FrameBelow, FrameLeft, and FrameRight links.

2. At the next lower level, you may call the ControlUtils procedure named Input, which displays a single frame and controls user input to it. Your program must contain a loop of some sort which calls Input for each frame. The parameter to Input is a VAR parameter of type AFrameName that must contain the name of the frame to display and control. Input will change the value of the parameter to the next frame the user has selected.

Input returns the value ProgramExit (i.e., '0'), imported from the ScrnTypes module, when the user chooses to exit, so your loop should look like:

```
MODULE Example2;
FROM DspFiles IMPORT OpenDisplayFile, CloseDisplayFile;
FROM ScrnTypes IMPORT
  DisplayFile, DisplayFrame, AFrameName;
FROM ControlUtils IMPORT ReadShowAndCntrl;
FROM ScrnTypes IMPORT ProgramExit;
FROM PosUtils IMPORT Equal;
IMPORT InitCompilerMods;

VAR
  NextFrame: AFrameName;
  MyScreenFile: DisplayFile;
  MyFrame: DisplayFrame;
BEGIN
  IF NOT OpenDisplayFile(MyScreenFile,
    'FileName.dsp') THEN
    RETURN;
  END;
  NextFrame := 'FirstFrame';
  REPEAT
    ReadShowAndCntrl(MyScreenFile, MyFrame, NextFrame );
  UNTIL Equal( NextFrame, ProgramExit );
  CloseDisplayFile(MyScreenFile);
END Example2.
```

The added power at this level is that you may examine and process user input after any frame is filled by putting an IF statement in the processing loop. For example:



```

REPEAT
  ReadShowAndCntrl(MyDisplayFile, MyFrame, NextFrame
    );
  IF Equal( NextFrame, SpecialFrame ) THEN
    ProcessInput();  (* Your own routine *)
  END;
UNTIL Equal( NextFrame, ProgramExit );

```

*Repertoire* stores the user's input in a linked list that your programs may access, and it provides the routines *ReadInput* and *ReadNextInput*, among many others, to give you convenient access to the data. (See section 3.5.7).

3. At the next lower level, you may call one *Repertoire* routine to display the frame and another to accept and control user input. Basically, the function *Input* above is split into two routines, *Display* and *Control*. This allows you to check user input and call *Control* again with the frame as it was filled in by the user.

If *Display* was called before *Control* was called again, the frame would be read anew from the file, wiping out the user's input. These functions are also convenient when you want to leave either the *Display* or *Control* functions intact but split up the other with the functions at the next lower level.

4. At the lowest normal level of program access, you may call separate routines to read the frame from the screen file, display it on the screen, and control the user input. This allows you to change the frame's contents before it is displayed and do other more flexible screen handling. The routines at this level, (*ReadFrame*, *ShowDisplayFrame*, and *ControlFrame*) also contain additional parameters that you may use.

The most significant parameter is the *DisplayFrame*, a record (actually a pointer to a record) that contains frame data and the control information for the frame currently in use. *Repertoire* sets up a global *DisplayFrame* called *MainFrame* that the higher level routines all use. However, it is better for purposes of clarity to declare and initialize your own *DisplayFrame* variable. You can even set up several *DisplayFrames* and keep them all active at once. It is necessary, in fact, to have separate *DisplayFrame* variables for frames that are to be simultaneously displayed if you want the *FrameManager* module always to be able to restore underlying images after a window or message pops up.

The three routines at this level are *ReadFrame*, *ShowDisplayFrame*, and *ControlFrame*. You pass *ReadFrame* the *DisplayFrame* and *DisplayFile* to use, plus the frame name to read. You pass *ShowDisplayFrame* the *DisplayFrame* and the coordinates of the window in which to display the frame. If you use (0, 0, 0, 0) as the coordinates, the default screen position stored in the screen file will be used.

*ControlFrame*'s parameters include the *DisplayFrame*; a *BOOLEAN* that

## Screen Design/Display System

indicates whether the frame should be redrawn on entry to the procedure; a **CARDINAL** that indicates where **ControlFrame** should initially place the **PointerBar**; and a string that may optionally contain a special message to the user. These facilities are primarily for giving the user feedback on application-specific input errors. Our simple example program now becomes:

```
MODULE Example3;
FROM DspFiles IMPORT
    ReadFrame, OpenDisplayFile, CloseDisplayFile;
FROM FramePainter IMPORT ShowDisplayFrame;
FROM InputManager IMPORT ControlFrame;
FROM PosUtils IMPORT Equal;
FROM ScrnTypes IMPORT AFrameName,
    ProgramExit, DisplayFile, DisplayFrame;
IMPORT InitCompilerMods;

VAR
    NextFrame: AFrameName;
    MyFrame: DisplayFrame;
    MyScreenFile: DisplayFile;
BEGIN
    IF NOT OpenDisplayFile(MyScreenFile,
        'FileName.dsp') THEN
        RETURN;
    END;
    NextFrame := 'StartFrame';
    REPEAT
        ReadFrame(MyScreenFile, MyFrame,
            NextFrame );
        ShowDisplayFrame(MyFrame, 0, 0, 0, 0);
        ControlFrame( MyFrame, 0, '',
            FALSE, NextFrame );
    UNTIL Equal( NextFrame, ProgramExit );
    CloseDisplayFile(MyScreenFile);
END Example3.
```

You might want that capability for several reasons. If, for example, a user makes errors that cannot be prevented in *Repertoire's* default input routines (unintelligible commands, mistyped record names, etc.), the most natural response is generally to place the pointer bar on the field containing the invalid input and prompt the user to enter it again. To do so, call **ControlFrame** with its **LandingField** parameter set to the number of the field in error, and pass an explanation of the error in **ControlFrame's** last parameter. Example:

```
ControlFrame( MyFrame, 3, 'Invalid password. Please re-
    enter or press ESC.', FALSE, NextFrame );
```

**FALSE** in the next to last parameter tells **ControlFrame** not to redisplay the frame before asking for input. If we had changed any of the fields since the last time the frame was shown, we would want it redisplayed. However, when user input has just been processed, the frame need not be reshown. See section 3.5.5 for details on **ControlFrame**; see section 3.5.5.1 in par-



ticular for details on error checking and input validation techniques.

The following sections explain individually the lowest level *Repertoire* routines and types. The higher level services are simply combinations of the low level services. The combinations are:

| <u>High Level Routine</u> | <u>Lower Level Routines Used</u> |
|---------------------------|----------------------------------|
| FrameSequence             | Input (for many frames)          |
| Input                     | Control, Display                 |
| Control                   | ControlFrame                     |
| Display                   | ReadFrame, ShowDisplayFrame      |

Thus, statements about the effect of ControlFrame, for example, will also interest users of Control, Input, and FrameSequence.

### 3.5.1. The DisplayFile Type

The first step in displaying frames created with ScrnComp.EXE is to declare a DisplayFile. A DisplayFile is really only an NdxFile (see Chapter 4), so you can use it to store things other than screen frames if you care to do so. For reasons related mostly to *Repertoire*'s history, most programs keep screen frames in separate files.

### 3.5.2. Opening and Closing DisplayFiles

```
PROCEDURE OpenDisplayFile(VAR dfile : DisplayFile);
```

OpenDisplayFile uses OpenNdxFile (see Chapter 4) to open the .DSP file and load its index into memory. The file name can be qualified with drive and path names, but that is generally unnecessary because OpenNdxFile uses the HandleIO.FindFile routine to locate the file. Thus, it begins its search for the file in the current directory of the current default drive if no drive or path qualifiers are present. Then it searches the drive and directory from which the program was executed, and finally it examines all the drives and directories listed in the DOS PATH setting.

For the convenience of older *Repertoire* programs, the first call to OpenDisplayFile automatically initializes DspFiles.MainFrame and associates it with the DisplayFile passed to OpenDisplayFile.

More than one DisplayFile can be open at the same time. Of course, a unique DisplayFile variable should be used for each file. The CloseDisplayFile routine closes the file and disposes of the DisplayFile record.

### 3.5.3. ReadFrame

```
PROCEDURE ReadFrame(VAR dfile : DisplayFile; VAR
    FrameRec : DisplayFrame; FrameName: ARRAY
    OF CHAR );
```



## Screen Design/Display System

This procedure reads the frame identified by `FrameName` from `dfile` and loads it into the `DisplayFrame` record. You must open the screenfile with `OpenDisplayFile` before calling `ReadFrame`. After you have loaded the frame into memory with `ReadFrame`, you can use the `ChangeField` routine (from `ControlUtils`) to modify or load data into the frame.

### 3.5.4. ShowDisplayFrame

```
PROCEDURE ShowDisplayFrame(VAR FrameRec : Display-  
Frame; Col1, Row1, ColX, RowX : CARDINAL);
```

This procedure displays the frame last read into the indicated `DisplayFrame`. The last four parameters define the window position where the frame will be written; `col1`, `row1` define the coordinates of the upper left corner of the window, and `col2`, `row2` define the lower right window corner. Thus, the character in column 1 of row 1 on your edited frame will be written at location `col1`, `row1`. Only the portion of the frame that fits in the window will be written on the monitor.

If the frame definition includes a single or double box, the box is drawn in `col1`, `col2`, `row1`, and `row2`. The box appears just before the frame text appears, so any text in the first row, the first column, or in the row or column corresponding to `col2`, `row2` will be overwritten by a box.

The frame must be read with `ReadFrame` before `ShowDisplayFrame` is called.

### 3.5.5. ControlFrame

```
PROCEDURE ControlFrame(VAR FrameRec : DisplayFrame;  
LandingField : CARDINAL; message : ARRAY OF  
CHAR; RedrawFirst: BOOLEAN; VAR NextFrame:  
AFrameName);
```

`ControlFrame` completely handles user input to a frame that has been displayed. `ControlFrame` does not clear the frame's window when it exits, so programs may call `ControlFrame` again to request re-entry of an input value. `ControlFrame` only allows user input in the screen positions in which input is defined, and checks the input according to the criteria in the screen file. `ControlFrame` will not let the user submit the frame with any "required" field empty (the user is allowed to back-up or exit, however).

In the simplest case, one could pass `ControlFrame` an initialized `DisplayFrame`, and set the next three parameters to 0, "", and `FALSE`. When thus called, `ControlFrame` will highlight the first field in the frame. As the user presses the up, down, right, or left arrows, `ControlFrame` will highlight the closest field in the appropriate direction. When a string or numeric input field is highlighted, the user can press keys defined as valid for that field. When the user presses a specially defined help key, (see section 3.7.9 for



instructions on defining special keys) help is displayed. When the user is finished, he/she presses a special submit key (F10 by default).

For frames that serve only to determine the next frame to be displayed, the NextFrame value returned in ControlFrame's last parameter is the only result of interest to calling programs. See frames 1 and 2 in the example in section 3.7.

The returned frame name is normally used to represent the next frame to be displayed. If the user presses the "backup" key (Esc by default), ControlFrame exits and returns the parent frame name defined in the editable screen file for this frame. If the user pressed the submit key when a GoTo field was highlighted, ControlFrame returns the goto frame name defined for that field. If the user pressed submit when any other field was highlighted, the Normal Next Frame is returned. Thus, programs can branch and display frames depending on the results they receive.

In table form, this is how ControlFrame determines the FrameKey it returns. The keys listed below are the only keys the user can press to get out of a frame, and they only cause ControlFrame to exit if the PointerBar is in the kind of field listed in the second column:

| If the<br>user<br><u>presses</u> | ...and the<br>current<br><u>field is</u> | ...FrameKey<br>returned by<br><u>ControlFrame is</u> |
|----------------------------------|------------------------------------------|------------------------------------------------------|
| exit                             | any field                                | ProgramExit                                          |
| submit                           | non-Goto                                 | DispFrame^.normlnext                                 |
| submit                           | Goto                                     | specified by Goto                                    |
| return                           | Goto                                     | specified by Goto                                    |
| backup                           | any field                                | DispFrame^.parent                                    |

ControlFrame is the routine that draws and positions the PointerBar--the reverse-video area that indicates the presently selected field. Its second parameter, a CARDINAL variable called LandingField, determines the field on which ControlFrame will initially place the PointerBar. (See the discussion of FieldNum in the introduction to section 3.5 for an explanation of how to convert field names into field numbers.) Higher-level routines such as Control, Input, and FrameSequence (all from the ControlUtils module) call ControlFrame with the LandingField parameter set to 0, which places the PointerBar on the first input field.

When ControlFrame exits, it erases the PointerBar. This saves your programs from having to explicitly erase the PointerBar in one window when a new one becomes active. ControlFrame also changes the appearance of the



## Screen Design/Display System

surrounding box (if there is one) as a way of indicating to the user that the frame is no longer active. It does so by drawing the DisplayFrame's Exit-Box on top of the box it drew on entry with the DisplayFrame's EntryBox. If the frame has its ClearAfter flag set to TRUE, ControlFrame also dispatches a message to the window manager telling it to restore underlying frames and windows so that their appearance reflects whatever their current status happens to be.

ControlFrame does not reset the frame's CurrentField value when it exits. As a result, you can use it for direct access to the field that was selected when the user submitted the frame. Say, for example, that you want to display a list of file names and allow the user to select one by moving the PointerBar to it and pressing return. Begin by using the AddMenuItem procedure from ControlUtils in a loop with the Directory module's FindFirstFile and FindNextFile procedures to fill the frame with matching file names. Display the frame with ShowDisplayFrame, and then call ControlFrame as follows:

```
ControlFrame( MyFrame, 0, '', FALSE, NextFrame );
  (*Now you can determine which file was
   selected like this: *)
IF NOT Equal(NextFrame, MyFrame^.parent) THEN
  (*User isn't backing up.*/)
  ScrnUtil1.GetFieldText( MyFrame,
    MyFrame^.CurrentField, FileName );
  (*FileName now contains the name of the
   selected file.*/)
END;
```

### 3.5.5.1. Input Validation and Error Checking

Occasionally, we do not want ControlFrame to highlight the first field. We want it to go to a specific field and tell the user something about the entry in that field.

For example, we may have called ControlFrame previously, examined the input, and determined that there is some application-level error in the user response (specification of a non-existent file, for example). If we set the parameter LandingField to the number of the field in error, ControlFrame will begin by placing the PointerBar on that field. If we pass a string in ControlFrame's message parameter, ControlFrame will print the string in a Message Box.

We would normally check these error conditions immediately after a call to ControlFrame. If an error occurs, we call ControlFrame again with LandingField and message set, and the user sees instantaneous error feedback. The user is never aware that we exited the frame. We can put the ControlFrame call inside a REPEAT loop along with our error checking so that we repeatedly call ControlFrame and check for errors until none are found.



It may at first seem odd that `ControlFrame` requires you to designate the erroneous field by number rather than name. The reason is that *Repertoire* does not require you to name all of your fields, nor does it require that field names be unique. To get the number of a field when you know only its name, use the procedure `FieldNum` in `ScrnUtil1`.

You may occasionally want to check for errors on a field-by-field basis instead of waiting for the user to submit the entire frame. The best way to do so is to insert your own error-checking procedure variable into the default version of `TheFieldHandler`. (The default is a procedure in the implementation of the `UserOps` module named `DefaultFieldHandler`.) You put your error checker in all the places where the user could potentially exit the field; those places are clearly marked in the `UserOps` source code. Your error checker can be set to a null procedure in most parts of your program, but in parts where you want to do field-by-field error checking, you can assign it a routine that examines the name of the field being processed, and then examines the data entered if the name is of interest. If the data is invalid, your error checker beeps or does something else to signal the error, returns a `FALSE`, and `TheFieldHandler` continues. Otherwise your error checker returns a `TRUE` and `TheFieldHandler` allows a normal exit from the field.

### 3.5.5.2. Screen Creation by Programs

*Repertoire* makes it easy for your programs to modify frames retrieved from a `DisplayFile`. It does so by separating the frame reading operation from the frame displaying operation. You can read the frame, modify it, and display it in three intuitive steps. Example:

```
ReadFrame( DisplayFile, YourFrame, 'PhoneList' );
(*Reads the frame named 'PhoneList' from
 DisplayFile and stores it in the YourFrame
 record.*)
```

```
ChangeField(YourFrame, 'Pat Collins', 'name', TRUE);
(*Inserts 'Pat Collins' into the 'name' field of Your-
 Frame. The TRUE in the last parameter
 indicates that the former length of the
 field is to be preserved by padding blanks
 onto the end of the new field contents, if
 necessary.*)
```

```
ChangeField(YourFrame, '617-495-5000', 'phone', TRUE);
(*Inserts the new phone number into the 'phone' field of
 YourFrame.*)
```

```
ShowDisplayFrame( YourFrame, 0, 0, 0, 0 );
(*Writes YourFrame to the screen, using the
 window coordinates defined in the file.*)
```

Remember that *Repertoire's* `Scrn2Ndx` module provides very high-level



## Screen Design/Display System

functions for modifying frames with data retrieved from the NdxFile system. See the Scrn2Ndx section of the Repertoire Reference Manual.

### 3.5.6. Keys Recognized in Repertoire

#### 3.5.6.1. Entry Keys

These are the keys that, by default, allow string editing operations, as well as movement between and within fields:

- Ins:** In string-entry fields, toggles the "insert mode" on and off. By default, *Repertoire* is always in "overwrite mode"--new characters are typed over old ones. When you press the insert key, however, the cursor gets fatter and new characters push old ones to the right.
- Del:** In string-entry fields, deletes the character at the cursor position and pulls remaining characters in from the right.
- Alt-D:** In string-entry fields, deletes the entire line.
- Alt-E:** In string-entry fields, erases to the end of the line.
- TAB:** Highlight the next field in the frame. (We also say "move to the next field"). If a group member is currently highlighted, that member is shown in bold, and the next field outside this group is highlighted. If the last field in the frame is currently highlighted, *Repertoire* wraps around to the first field. These keys differ in only one way: if there is only one field in the frame, or the fields are all GoTo or Choice fields, <CR> submits the frame (ControlFrame exits).

**BACKTAB:** Functions like the above keys, except that it moves back to the prior field in the frame.

**Right & Left Arrows:** Work like the TAB and <CR> in Goto fields. In group fields, right and left highlight the next and prior group members, respectively. If the last (or first) member is currently highlighted, these arrows wrap to the first (or last) group member. In string entry fields, right and left move the cursor in the string nondestructively; when the cursor reaches the end of the field, the pointer bar moves to the next field in the frame. Right and left work much the same way in editor fields.

**Ctrl-Right & Ctrl-Left:** Move right and left to the first character of the next or preceding word in string and in editor fields.

**Up & Down Arrows:** When not in a group field, these arrows move to the field most directly above (or below) the current one. This field is found by locating the nearest line that contains any fields and selecting the field on this line that starts in the column closest to



the column where the current field starts. In a group field, these arrows function just as the TAB and <CR> (they move out of the group). In Editor fields, up and down move the cursor around in the editor window unless they are pressed on the top or bottom lines of the filled area of the window, in which case they cause the pointer bar to move to the next field.

Notice that inside group fields the above positioning keys ignore the real horizontal/vertical relationships of fields in the frame. The left and right arrows always move between group members, even if the members are on different lines. Consequently, you may prefer to put all members of a group on the same line.

**Home:** Moves the cursor to the beginning of the current line if the field is a string or editor field. Does nothing in other fields.

**End:** Moves the cursor to the end of the current line if the field is a string or editor field. Does nothing in other fields.

**SPACE..'~':** The valid keys for entry into a STRING field. These are the ASCII values from 32 to 126. Certain keys in the range above 126 are also valid to allow entry of mathematical symbols and characters from alphabets for other languages.

**'0'..'9', SPACE:** The valid keys within an INTEGER field.

**'0'..'9', SPACE, '.' :** The valid input keys for a REAL field.

### 3.5.6.2. Special Keys

The entry keys allow the user to move around and enter data in the frame, but that is all. Most conspicuously, there is no entry key that allows a user to exit the frame. Exiting and customized functions are handled by "special" keys. If you are satisfied with the default special keys, no programming changes are needed. If you want to change the special keys, see Section 3.7.9. In any case, you will probably want to display the meaning of the special keys somewhere on the screen. In the example in section 3.6, we display the special keys at the bottom of the screen. *Repertoire* is shipped with the following special key definitions:

**<CR>:** Submits the frame if the present field is a Goto field or if all fields are choice fields; in string-entry fields, moves the cursor to the next field; in editor fields, creates a new line in the editor window.

**F1:** Asks for Help. The help frame for this field or frame is called, or the string defined for this frame (see section 3.3.1.4) is displayed in a Message box. (See the MsgBox1 and MsgBox2 sections of the *Repertoire Reference Manual* for an explanation of "Message Boxes".)



## Screen Design/Display System

- F2: Exits the entire program. If the user presses this key, *ControlFrame* asks for confirmation that the user wants to exit. If the user still wants to exit, *Repertoire* calls the *StraightToDOS* procedure from *ErrorManager* after executing *ErrorManager's* *TermProcList*. In earlier releases, *ControlFrame* merely returned the *ProgramExit* value, and it was up to the calling procedures to find their way back to DOS. Because *Repertoire* now exits directly, it is important that you add any critical housekeeping code (i.e., screen restoration or file closing) to *ErrorManager's* *TermProcList*, unless you plan to disable F2 as a Program Exit key.
- F3: Displays as much information as is available about the program's memory usage.
- Esc: Backs-up to the last frame. *ControlFrame* will exit and return as its result the *ParentFrame* defined for this frame. All user input is still accessible to the calling program. Required fields are not checked.
- F10: Submits the frame. By pressing F10, the user indicates that he/she is finished with this frame and wants it processed. *ControlFrame* exits and returns to the calling program. If a *GoTo* field is highlighted, *ControlFrame's* result is the *GoTo* frame name defined for the field. If any other type of field is highlighted, *NormalNext* for this frame (see Section 3.3.1.3) is returned.

### 3.5.7. Reading Input From a Frame

After the user submits or in some way exits from a frame (i.e., just after your program finishes executing the *InputManager* module's *ControlFrame* procedure), you have many ways of retrieving the user's input. For simple help screens or other read-only applications of the screen system, you don't need to analyze user input, so you can skip this section if that sort of screen is all your program contains.

The user's input is stored in the *DisplayFrame* variable (actually, in the *GenLists* associated with the *DisplayFrame* variable), so all procedures for reading input have a *DisplayFrame* as their first parameter. Although you don't have to know precisely where in that *DisplayFrame* variable the various kinds of field data are stored to be able to retrieve them, it helps to have a general sense of it so that you know what the procedures below are doing. Understanding where the different kinds of user input are stored also gives you the ability to use lower-level procedures to retrieve them, if you feel the need to do so.



With one exception,<sup>1</sup> every part of a frame that you can see on the screen is stored in the frame's image list (i.e., `TheFrame^.ImageList`). Thus, if a user enters 'John Q. Smith' in a frame's `CustomerName` field, you know that the string 'John Q. Smith' will be stored somewhere in the frame's `ImageList`, because you can see it. The field's name and `TypeCode`, on the other hand, are not visible, so you know that they are stored in the frame's `FieldList`. Likewise, if your frame contains a group of two choice fields between which the user selects by moving the pointer bar between 'Proceed' and 'Cancel', the strings 'Proceed' and 'Cancel' will be stored somewhere in the frame's `ImageList`, and the frame's `FieldList` will contain elements which contain the names, `TypeCodes`, selection characters, etc., that are associated with these choices. The `FieldList` and the `ImageList` are cross referenced (see the declarations of the `InputFieldRecord` and `ImageElement` record types in the `ScrnTypes` definition module) so that, given a field number, it is easy to locate the visible text associated with the field, even though it is not stored in the same place.

Since you know that there is exactly one element of the frame's `FieldList` for every input field in the frame, you could theoretically use nothing but low-level `GenList` procedures to retrieve `InputFieldRecords` and `ImageElement` records from these two `GenLists`. That would be tedious, however, so the higher level screen system modules provide procedures that combine steps and use shorter parameter lists. These procedures have names that clearly suggest their purposes, but they are scattered between the modules `ScrnUtil1`, `ControlUtils`, and `InputManager`. The most important are explained below. Please refer to the documentation for `ScrnUtil1`, `ControlUtils`, and `InputManager` in the *Repertoire Reference Manual* for a complete description of any routines you find interesting that are not covered below.

### 3.5.7.1. String and numeric input

The simplest and most traditional way to get most forms of user input is to call the `ReadInput` procedure from the `ControlUtils` module:

```
PROCEDURE ReadInput (VAR TheFrame : DisplayFrame; VAR
    TheStr : ARRAY OF CHAR; FieldName : ARRAY
    OF CHAR);
```

You pass it the `DisplayFrame` variable upon which `ControlFrame` just finished operating, as well as the name of the field from which you want to read, and it returns whatever the user entered in that field. The major difference between it and the `ReadNextInput` procedure, below, is that you

---

<sup>1</sup>The exemption is that the contents of editor fields are stored in the `DisplayFrame`'s `EdFieldList`, which is a list of strings.



## Screen Design/Display System

pass a field name to ReadInput. Because none of *Repertoire's* field name are procedures are case sensitive, it doesn't matter how you capitalize the field name. If the none of the frame's fields have a name that matches, *Repertoire* will exit and write an error message on the screen.

```
PROCEDURE ReadNextInput (VAR TheFrame : DisplayFrame;  
    VAR TheStr : ARRAY OF CHAR; skip : CAR-  
    DINAL);
```

ReadNextInput returns in its TheStr parameter the text in the next field in the frame. 'Skip' indicates how many fields to skip over since the last call to this routine. Notice that Display fields are not counted. We no longer recommend the use of ReadNextInput in most cases because it involves too many little exceptions. (For example, how do you read the first input field if the procedure always reads the next input field?)

For situations in which you do not know or it is not convenient to use the name of the field, we now recommend the procedures GetFieldText and GetFieldImageRec from ScrnUtil1.

All of these procedures return the text of the user's input as a string, no matter what sort of TypeCode the field has (goto, choice, string, integer, real). This implies that none of them will work with Editor fields--the ScrnUtil1 module provides a procedure named ListFromEdField for that purpose. The returned string is always padded with blanks, if necessary, so that its length equals the number of characters between the field marks in the .EDT file. If the field is of GoTo or Choice type, the text is simply the text defined in the editable screen file.

To save you the trouble of converting these strings into CARDINAL, INTEGER, and LONGINT values, the ScrnUtil1 module also provides the procedures GetCardField, GetIntField and GetLongIntField. Pass them a DisplayFrame and a field number, and they convert the contents of the field into a binary value and return it in the last parameter of the procedure. ScrnUtil1 does not provide a GetRealField procedure to avoid certain complications involved in scattering real number types across definition modules, but you can easily write one by examining the source for the procedures above.

### 3.5.7.2. Choice fields and menu items.

In processing input that arrives from choice and GoTo fields, you are normally interested only in determining which choice was made or which item was selected, rather than what the text of that field contains. Indeed, the text of the field is normally just what you typed into the .EDT file.

The most convenient way to determine which choice has been selected from a group of choice fields is to use the WhichChoiceNum and Which-



ChoiceKey procedures from ScrnUtil1. They require only that you know the field number of one of the members of the choice group--any member will do. WhichChoiceNum returns the group number of the member that was selected. The group number of the middle choice in a group of three choices will always be 2, regardless of the position of the group in the frame's FieldList.

To further insulate your code from the arrangement of fields in the .EDT file, you can use the WhichChoiceKey procedure to refer to choices by the selection key (i.e., the ordinal value of the field's selection character) associated with them. This will always be the same, no matter how the choices are arranged within the group.

Remember that the ScrnUtil1 module's FieldNum procedure returns a field's number when you pass it a name, so you can write your code like this:

```
CASE WhichChoiceKey(MyFrame, FieldNum('Yes')) OF
  ORD('Y'): (*Do something*)|
  ORD('N'): (*Do something else*)
...

```

Processing input from GoTo fields is easy. If, for example, the field section of the frame's command area contains 'GoTo EditMenu' for some item in a menu, then ControlFrame will return 'EditMenu' in its last parameter (i.e., its NextFrame parameter) if the user pressed Return or the frame-submit key (normally F10) while the pointer bar was on that menu item. But there are other ways to get essentially the same information. You can pass DisplayFrame^.CurrentField to the GetFieldText procedure to get the text of field on which the pointer bar was resting when the user exited. The ScrnUtil1 module's SelectedChar procedure also returns the selection key (i.e., the ordinal value of the field's selection character -- the letter in parentheses preceding the name of the field in the .EDT file) of whatever field the pointer bar happened to be on when the user exited the frame. It is handy when you know that the frame contains nothing but GoTo fields, but it doesn't tell you whether the user exited from the frame by pressing escape, so you have to test explicitly for that condition (if you care about it) by asking whether ControlFrame's NormalNext parameter returned a value equal to DisplayFrame^.ParentFrame.

### 3.5.8. Natural-Language Analysis of Input

*Repertoire* contains a special facility that ties the Screen Display System to the Natural Language Analyzer. As detailed in section 3.3.1.5 above, each editable frame may contain a DataList. This list can contain expressions that will allow *Repertoire*'s StrLogic module operate on the contents of a single entry field at a time. The expressions can contain a series of tests, each with a result that will be returned if that test is satisfied. These results



## Screen Design/Display System

can correspond to frame names. Thus, each test could mean "if I am TRUE, go to frame XX next". After calling `ControlFrame`, you can do this sort of analysis by calling the routine `WhatTheySaid`, imported from the `LogicInput` module.

```
PROCEDURE WhatTheySaid( VAR TheFrame: DisplayFrame;  
                        VAR FieldName: ARRAY OF CHAR) : CARDINAL;
```

The parameters are the `DisplayFrame` pointer and the name of the field to be analyzed. If the field name is null, the next field of type `String` will be used. `WhatTheySaid` returns the result of the string analysis as defined by `Repertoire's StrLogic` module. (See the `StrLogic` section of the `Repertoire Reference Manual`.)

## 3.6. Example

### 3.6.1. The Editable Frame

The following example is only a partial excerpt from a real editable (.EDT) *Repertoire* screen file. The lines in the frame have been shortened to fit in the manual. The actual lines could be of any width, since *Repertoire* supports horizontal scrolling.

The example also uses only numeric frame names for simplicity. Frame names can actually contain any combination of alphabetic or numeric characters.

The first frame contains a single line of text that we will write at the bottom of the screen to remind users what special keys are available. (A program can display any frame at any location on the screen) We will never erase or write over this portion of the screen with later displays, so the user can always see this function key prompt area. You may choose to have a bigger or smaller function key prompt area, you may put it at the top of the screen, or you may omit it entirely. This prompt conforms with *REPERTOIRE's* default set of special keys.

In the second frame, the user can choose among three actions. Depending on the action chosen, frames "Branch", "AddSummary", or "RequestStat" will be shown next (frame "RequestStat" is not shown as part of this excerpt, but it must exist in the real application) Frame "Branch" is similar to the second one. The fourth frame contains 4 fields for entry of alphanumeric strings, one numeric field, and one choice group. The user can enter data in the string and numeric fields and use the left and right arrows to make a choice in the group.

The parent frame entries allow the user to back-up from frame "Branch" to frame "MainMenu" and from frame "AddSummary" to frame "MainMenu". If the user tries to back-up from frame "MainMenu", it is re-displayed.



## Example 1: The Editable Screen File

```

=====
:Start
  NormalNext: MainMenu
  DisplayOnly
  Window:{
    No Box
    Position:( 1, 256, 80, 256)
  }
-----
F1:HELP F2:EXIT F3:MEMORY Esc:BACK-UP F10:FORWARD
=====
:MainMenu
  ParentFrame: MainMenu
  Fields: {
    (S) '' Goto Branch
    (E) '' Goto AddSummary
    (R) '' Goto RequestStat
    (Q) '' Goto 0 }
  Window:{
    Position:( 1, 1, 80, 255)
  }
  Help: {
Searching allows you to read summaries meeting
specified criteria.  Entering updates the summary
files.  A Statistics request will tabulate summary
counts according to entered criteria.}
-----

```

Welcome to the Supreme Judicial Court  
Case Summary System

Do You Want To:

(\*This is a comment; it won't appear on the screen.  
Lines it occupies are deleted, not replaced with blank  
lines.\*)

```

#Search for a summary#

#Enter a new summary#

#Request statistics#

#Quit#

```

```

=====
:Branch
  ParentFrame: MainMenu
  Fields:{
    (D) '' Goto SearchDecided; prompt; help DecidedHelp
           Press F10 now to examine Decided cases
    (P) '' Goto 7; SearchPending;prompt; help
           PendingHelp
           Press F10 now to examine Pending cases
  }

```

## Screen Design/Display System

```
Window:{
  Position:( 1, 1, 80, 255 )
}
Help:{
This system contains all cases tried since 1980.
All pending cases at least 30 days old are included.
  You must look through these two groups separately.
}
```

-----

Case Summary System: Searching for a Summary

Do You want to search:

#Decided Cases#

or

#Pending Cases#

=====

:AddSummary

ParentFrame: MainMenu  
NormalNext: ExceptData

Fields:{

(\*Note that 'Fields:{' must be on a line by itself.  
The '{' cannot be on the same line with the first  
field entry.\*)

() '' string; required; prompt

Enter the name of the case

() '' string; required; prompt

Enter the court in which the appeal originated

() '' integer [8000..9999]; required; prompt

Enter a Docket number between 8000 and 9999

() '' string; required; prompt

Enter the standard one digit department code

() '' string; prompt

Enter the Judge's last name only

() '' group 3

(c) '' choice

(a) '' choice

(r) '' choice }

Window:{

Position:( 1, 1, 80, 255 )

}

Help: {

Use the official docket number and case name.

Origin is the prior court of review.

Use the one-character dept. code and the Judge's last  
name. }

-----

Case Summary System  
Entering a Preliminary Summary



```

Case name:
#
#
Origin: # # Docket Number: # #
Department: # # Assigned to Judge: # #
Disposition: #Cert Denied# #Affirmed# #Reversed#

```

### Example 1: The Processing Program

```

MODULE Supreme;

FROM DspFiles IMPORT MainFrame, OpenDisplayFile,
    CloseDisplayFile;
FROM ControlUtils IMPORT ReadShowAndCntrl;
FROM ScrnTypes IMPORT AFrameName, DisplayFile,
    DisplayFrame, InitDisplayFrame;
FROM VWindows IMPORT CurrentWindow;
IMPORT InitCompilerMods;

VAR
    ScreenFile : DisplayFile;
    NextFrame : AFrameName;
    MyFrame : DisplayFrame;
BEGIN
    IF OpenDisplayFile( ScreenFile,
        "SUPREME.DSP") THEN
        InitDisplayFrame(MyFrame, CurrentWindow);
        NextFrame := 'Start';
        ReadShowAndCntrl( ScreenFile, MyFrame, NextFrame );
        CloseDisplayFile( ScreenFile);
    END;
END Supreme.

```

### 3.6.2. The Processing Program

The example processing program controls the frame in a very simple manner—it leaves everything to *Repertoire*. It opens the *DisplayFile* and calls *ReadShowAndCntrl* to display the frames as they are selected by the user. When the user chooses to exit, it closes the screen file, restores the cursor, clears the screen, and exits.

Notice that simply calling *ReadShowAndCntrl* does not store the user input in frame "AddSummary". *ReadShowAndCntrl* only displays frames and determines which frame to show next. A real data entry program would read the input data from frame "AddSummary" and process it. Thus we would modify our program to use lower level *Repertoire* routines:

```

MODULE Supreme;

FROM DspFiles IMPORT MainFrame, OpenDisplayFile,
    CloseDisplayFile;

```

## Screen Design/Display System

```
FROM ControlUtils IMPORT ReadShowAndCntrl;
FROM PosUtils IMPORT Equal;
FROM ScrnTypes IMPORT DisplayFile, AFrameName,
    ProgramExit, DisplayFrame, InitDisplayFrame;
FROM VWindows IMPORT CurrentWindow;
IMPORT InitCompilerMods;

VAR
    ScreenFile : DisplayFile;
    MyFrame:DisplayFrame;
    NextFrame : AFrameName;
BEGIN
    IF NOT OpenDisplayFile( ScreenFile,
        "SUPREME.DSP") THEN
        RETURN;
    END;
    InitDisplayFrame(MyFrame, CurrentWindow);
    NextFrame:='Start';
    REPEAT
        ReadShowAndCntrl(ScreenFile, MyFrame, NextFrame );
        IF Equal(NextFrame, 'AddSummary') THEN
            (*ProcessData(ScreenFile);*)
        END;
    UNTIL Equal(NextFrame, ProgramExit);
    CloseDisplayFile( ScreenFile);
END Supreme.
```

ProcessData would be a user-written routine that takes the data in the frame and writes it to a file. If you want to store the frame data in a *Repertoire* NdxFile, *Repertoire's* Scrn2Ndx module (see section 4.4.1) provides special support for doing so. For applications desiring variable-length data storage, each field can be stripped of blanks and separators inserted before storage (see the StrEditSection of the *Repertoire* Reference Manual for *Repertoire's* CrunchBlanks routine to strip blanks). Of course, if you use *Repertoire's* NdxFile system, you can simply strip the field of blanks and pass it to the NdxFiles.PutField procedure; any necessary separators will be inserted for you automatically.

## 3.7. Advanced Use of the Display System

The material below explains the fundamental data structures used in *Repertoire's* screen system. In this context, we are more careful to distinguish between the terms "screen" and "frame." The material above tends to use those terms interchangeably, but here it is important to keep them straight. By "screen" we simply mean the face of the monitor--the entire physical image presented to the user. By "frame" we mean each named group of menus, graphics, text, etc., defined as a single unit in an EDT file (i.e., a file submitted to *Repertoire's* screen compiler). A frame could occupy the entire screen, but it could also occupy a smaller area. After a frame has been read from the DSP file, it is associated with a DisplayFrame record (actually, a pointer to a DisplayFrame record). We sometimes refer to those



pointers as frames in themselves, since they provide the only way of referring to a frame once it has been read into memory.

### 3.7.1. The DisplayFrame and DisplayFile Records

#### 3.7.1.1. The DisplayFile Record

Several *Repertoire* procedures, such as `OpenDisplayFile` and `ReadFrame`, require you to specify a `DisplayFile` in their parameter lists. Even procedures which require specification only of a `DisplayFrame` implicitly refer to a `DisplayFile` through a field in the `DisplayFrame` record named `FromFile`. This is how the type `DisplayFile` is declared:

```
TYPE
  DisplayFile = NdxFileType;
```

In other words, a `DisplayFile` is just a *Repertoire* `NdxFile`, as described in Chapter 4. Anything you can do with an `NdxFile`, you can do with a `DisplayFile`.

#### 3.7.1.2. The DisplayFrame Record

*Repertoire* keeps all information connected with each frame in a record pointed to by a `DisplayFrame` variable. You will not normally see the `DisplayFrame` pointer at all. For purposes of compatibility with older applications, *Repertoire* provides a default `DisplayFrame` variable called `MainFrame` (exported from `DspFiles`). If you want to use additional `DisplayFrames`, you must declare them and pass them to the routine `InitDisplayFrame` before using them. A `DisplayFrame` variable may be used repeatedly, for all frames in your file, but at any one time its record contains data for only one frame.

`DisplayFrame` variables are passed to the lower level routines in *Repertoire*'s screen system (for example, `ReadFrame`, `ShowDisplayFrame`, `ControlFrame`). A `DisplayFrame` is a pointer to a record containing the following fields:

```
ThisFrame:  AFrameName;
```

This is the name of the frame presently in this `DisplayFrame` record.

```
WindowHandle: AWindowHandle;
```

This is what allows you to associate *Repertoire*'s frames with particular windows in other window managers. In the standard version of *Repertoire*, the `WindowHandle` is always `NIL` and is ignored, because the `SmartScreen` and `WindowPrims` modules assume they own the entire screen. If you connect *Repertoire*'s screen system with a package like JPI's window manager,

## Screen Design/Display System

however, the window manager's low level writing procedures expect to be passed a window identifier. The WindowHandle provides a place to store it.

PushedFrame: DisplayFrame;

A pointer to the next frame down in the frame stack. (There is no global frame stack; each DisplayFrame variable has its own.) If no frames have been pushed underneath this frame, its value will be NIL. See the discussions of PushFrame and PopFrame above and in the ScrnUtl2 section of the Repertoire Reference Manual.

clear: BOOLEAN;

Indicates whether this window is to be cleared before the frame is written. The default is TRUE.

box: CHAR;

Indicates the type of box to be drawn around the window:

D = double line box  
S = Single line Box  
N = None

VirtualWidth, VirtualHeight, RowsScrolled, ColsScrolled, headline: CARDINAL;

These variables help *Repertoire* implement horizontal and vertical scrolling. VirtualWidth and VirtualHeight track the true width and height of the frame, as opposed to the width and height of the window in which the frame is displayed. RowsScrolled and ColsScrolled track the number of rows and columns above and to the left (respectively) of the visible window. The headline variable represents the lowest line that will not be scrolled if scrolling is needed.

startcol, endcol, startrow, endrow: CARDINAL;

These fields define the area where the current frame was written. Note that these values include the columns and rows overwritten by the box, if the frame has one. They are offset from 0, not 1. In other words, if a frame occupies the entire screen, these values will be 0, 79, 0, and 24, respectively.

action: CHAR;

The type of user action allowed on this frame:

I = normal user input frame.

D = the frame is displayed and the normal next frame called immediately--no input is allowed.

W = the frame is displayed until any key is pressed.



```
HelpFrame: AFrameName;
```

If HelpFrame is not null, it represents the help frame to be displayed when the user presses the help key in a field that does not have field-specific help. (Actually, it represents the entry point in a sequence of helpframes to be displayed, since pressing the help key in this situation invokes the FrameSequence routine.) If HelpFrame is null, the HelpList will be used.

```
parent, normlnext: AFrameName;
```

The parent frame and normal next frame for the currently displayed frame. See section 3.7.5 for an example of why you might want to access these fields.

```
pbfor, pbbak, selfor, selbak, normfor, normbak,  
bordfor, bordbak, msgfor, msgbak, promfor,  
prombak : AColor;
```

These are the foreground and background colors that will be used for this frame, for specified parts of the screen that *Repertoire* controls: the pointer bar that indicates the current field, selected group members, normal text, the frame border or box, message boxes that the user may see, and the prompt line. (see section 3.3.1.2).

```
pbattrb, selattrb, normattrb, bordattrb, msgattrb, pro-  
matrb : AMonoAttribute;
```

These are the monochrome attributes for the screen parts listed above (pointer bar, selected group members, normal text, border, message boxes, and prompt line). *Repertoire* will use the colors above on color screens and these attributes on monochrome screens.

```
FromFile: DisplayFile;
```

A pointer to the DisplayFile record with which this frame is associated.

```
self: GenList;
```

This is the parent of all the other GenLists in the frame. You should never have to do anything with it directly. The only reason it exists is to reduce the number of steps *Repertoire* has to take when it transfers a frame into or out of an NdxFile record.

```
LinkedFrames: GenList;
```

This is a list of four strings. If the frame's .EDT file source code defines FrameAbove, FrameBelow, FrameLeft, or FrameRight, the name defined will occupy the corresponding position in the LinkedFrames list. The list contains a null string for every link not defined.

```
DataList: GenList;
```

## Screen Design/Display System

This is where information entered in the data section of a frame's command area is stored. See section 3.3.1.5. Each line of the data section occupies a separate element of the `DataList`. All elements of the list are strings (i.e., have `StrCode` as their type codes).

```
PromptList: GenList;
```

`PromptList` stores any field-specific prompts that you have defined for this frame. It is used in conjunction with the `PromptNum` field of the `InputFieldRecord`; any field with a field-specific prompt contains a number greater than zero in its `PromptNum` field. That number identifies the string in the `PromptList` with which the field is connected. More than one field can be connected with a single element of the `PromptList`.

```
HelpList: GenList;
```

If the field in which the pointer bar or cursor is located when the user presses the help key does not have a help frame associated with it, and if the frame as a whole does not have a help frame, then this list will contain any help message you have defined in the help section of the frame's command area (in the `.EDT` file).

```
EdFieldList: GenList;
```

If the frame has any editor fields, each one has an entry in this list. The `EdFieldList` is a list of lists of strings—one list of strings for every editor field in the frame. The strings comprise the text that you see in an editor field.

```
FieldList: GenList;
```

The `FieldList` is a list of `InputFieldRecords`. An `InputFieldRecord` contains quite a lot of field control data (see below), but it does not contain the string that actually appears on the screen. Instead, it contains an `ImageNum`, which identifies the string in the `ImageList` that corresponds with this field. Every field in the frame has one entry in the `FieldList`.

```
ImageList: GenList;
```

This is where *Repertoire* stores all of the strings that actually appear on the screen. It is the only list that the screen system normally has to reference when it displays or redraws a portion of the screen. Each element of the `ImageList` is actually a record of type `ImageElement`, defined in the `ScrnTypes` module, but it does not necessarily occupy as much memory as a separately declared variable of type `ImageElement`. The reason is that the `GenLists` module considers the entire `ImageElement` a string, so it only allocates space for the part of the `ImageElement` preceding the first null byte.



### 3.7.2. The FieldList and the ImageList

Internally, *Repertoire* stores the image of the current frame in a GenList named ImageList. You can access it through the DisplayFrame record. You will not normally need to deal directly with the elements of this list, but in unusual cases you may want to read or change the list without the support of routines like ReadInput, ChangeField, or the numerous utility procedures in ScrnUtil1. To remove text from the frame, for example, you must delete elements directly from the ImageList. Each element of the ImageList is a record of type ImageElement (defined in ScrnTypes) and represents some visible component of the frame. The rule determining what makes up a "component" is complicated, but is essentially that a new element starts where ever the screen color changes, or where three or more spaces in a row share the frame's default background color or attribute (DisplayFrame^.normbak).

If you must manipulate the ImageList directly, remember that the CARDINAL values in an ImageElement record are not valid while they are in the ImageList. They are encoded so as to guarantee that the only null byte in the ImageElement is the one that terminates the string. As a result, if you use procedures from ScrnUtil1 such as GetFieldImagePtr or GetImagePtr, you cannot dereference the pointer and use any of the CARDINAL values in the record, because the pointer is pointing directly into the ImageList. You can, however, dereference the pointer to the string part of the record, because it is not encoded. You can also use procedures from ScrnUtil1 such as GetFieldImageRec and GetImageRec to examine the CARDINAL values in the ImageElement, because these procedures copy the ImageElement record out of the ImageList and decode it in the process.

Each ImageElement record has the following fields:

```
row, col: CARDINAL;
```

The position of this part of the frame relative to the top of the window in which the frame is displayed. Both row and col start at 1, for the first line/column in a window.

```
field: CARDINAL;
```

If the text is the image of an input field, this will identify the corresponding element of the FieldList.

```
foreg, backg: AColor;
```

The type AColor is imported from MsColors; foreg and backg specify the color in which this part of the frame will be displayed on color monitors.

```
atrb: AMonoAttribute;
```

## Screen Design/Display System

The type `AMonoAttribute` is imported from `MsColors`; this field specifies the display mode (plain, bold, underlined, blinking, reversevideo) to be used when this part of the frame is displayed on a monochrome monitor.

```
text: ARRAY [0..79] OF CHAR;
```

This is the string that the user actually sees on the screen.

You are much more likely to directly manipulate the other major list in a `DisplayFrame` record: the `FieldList`. The `FieldList` has one element for each input field in the frame. Each element is of type `InputFieldRecord`, which is defined in the `ScrnTypes` module. The `FieldList` is not encoded, so you can safely dereference pointers to `InputFieldRecords` that are still in the `FieldList`. Procedures such as `GetFieldPtr` in the `ScrnUtil` module provide a good way to obtain such pointers. It is safer, however, to use `GetFieldRec` (also from `ScrnUtil`) if your program needs access to the `InputFieldRecord` for very long because of the possibility that the record may be in expanded memory. See the discussion of expanded memory in the `GenLists` and `VStorage` sections of the `Repertoire Reference Manual`.

An `InputFieldRecord` is a case-variant record, so it occupies only as much memory as the largest of its case selectors. This is how it is declared:

```
InputFieldRecord =  
  RECORD  
    fnam : ARRAY [0..15] OF CHAR;  
    ImageNum, PromptNum : CARDINAL;  
    ChangeMade, req : BOOLEAN;  
    HelpFrame : AFrameName;  
    CASE typ : CHAR OF  
      DispCode, StringCode:  
    | GotoCode:  
      MenuKey : CARDINAL;  
      ReturnVal: AFrameName;  
    | GroupMember:  
      ChoiceKey : CARDINAL;  
      GroupSize, GroupID : CARDINAL;  
      selected: BOOLEAN;  
    | EditorCode:  
      EdFieldNum: CARDINAL;  
      Col2, Row2, TextRow1, CursorCol, CursorRow,  
        MaxLines: CARDINAL;  
      ReadOnly: BOOLEAN;  
    | IntCode:  
      iMax, iMin : LONGINT;  
    | RealCode:  
      rMax, rMin : NumTypes.Real8;  
      decimalPlace : DecimalPlaceType;  
    END;  
  END;
```

This is what each field does:

```
fnam: ARRAY [0..15] OF CHAR;
```



This is where the field's name is stored.

```
PromptNum: CARDINAL;
```

If PromptNum is greater than zero, it specifies an element of the Display-Frame's PromptList that is to be used for this field's prompt message. These field-specific prompts come from the prompts defined in the fields section of the frame's command area.

```
ImageNum: CARDINAL;
```

This identifies the element of the frame's ImageList that contains the field's visible component.

```
req: BOOLEAN;
```

Indicates whether this field must be filled in before ControlFrame, Control, Input, or FrameSequence will exit.

```
ChangeMade: BOOLEAN;
```

Indicates if the field has been changed.

```
HelpFrame: AFrameName;
```

If not null, HelpFrame represents the frame name to be displayed (actually the entry point in a sequence of frames to be displayed) if the user asks for help while the pointer bar is in this field.

```
typ: CHAR;
```

A character representing the type of this field (GoTo, String, Integer, Real, Editor, etc.). For details, see ScrnTypes in the Repertoire reference manual.

```
MenuKey: CARDINAL;
```

This is the key number that substitutes for moving the pointer bar to a Goto (menu) field and pressing return; it is the "accelerator" in MS Windows terminology, or what *Repertoire* users have traditionally called the "Selection Character,"

```
ReturnVal: AFrameName;
```

This is where *Repertoire* stores the frame name that follows the word "Goto" in the .EDT file. It is the value that ControlFrame assigns to its NextFrame parameter when the user presses Return on a Goto field.

```
ChoiceKey: CARDINAL;
```

This is the same thing as the MenuKey, but you have to use a different name for it when the field is a choice field (i.e., when the typ field equals ScrnTypes.GroupMember). The reason is that standard Modula-2 requires

## Screen Design/Display System

unique names for all fields in a case-variant record, even if some of the fields serve exactly the same function.<sup>1</sup>

```
GroupSize, GroupID: CARDINAL;
```

You can reference these variables only if the field is a GroupMember. They represent the size of this field's subgroup, and the position of this field in the group.

```
selected: BOOLEAN;
```

Valid only if the field is a GroupMember. Indicates whether this was the highlighted choice in the group.

```
EdFieldNum: CARDINAL;
```

Valid only if the field is an Editor field. EdFieldNum identifies the position of the Editor field's text (i.e., its list of strings) in the EdFieldList.

```
Col2, Row2, TextRow1, CursorCol,  
    CursorRow, MaxLines: CARDINAL;  
ChangeMade, ReadOnly: BOOLEAN;
```

These variables correspond to variables of the same name in VEditor's AnEdControlRec. See the explanation of *Repertoire's* Editor in Chapter 5.

```
iMax, iMin: LONGINT;
```

Valid only if the field is of type Integer. These variables indicate the acceptable range of entries.

```
rMax, rMin: Real8;  
decimalPlace : DecimalPlaceType;
```

Valid only if the field is of type Real. These variables indicate the acceptable range of entries. decimalPlace indicates the number of decimal places.

The CurrentField variable in the DisplayFrame record identifies the most recently referenced member of the FieldList.

The ControlFrame procedure does not reset the CurrentField variable when it exits. It leaves it pointing at the field where the pointer bar was located when the user submitted the screen. This allows you to examine the contents of the menu item selected by the user. To determine which item was selected from a pull down window, for example, you can do the following:

---

<sup>1</sup> At least one recent compiler (JPI's TopSpeed Modula-2) does not impose this restriction, but *Repertoire* has to follow it because the other compilers do.



```

ReadShowAndCntrl(YourFile, YourFrame, NextFrame );
  (*YourFrame's fields pointer now points to
  the selected field.*)
GetFieldImagePtr( YourFrame,
  YourFrame^.CurrentField,
  ImagePtr );
IF Equal( 'Exit', ImagePtr^.text ) THEN
  (*do whatever is necessary to exit*)
END;

```

The big example program on one of your distribution disks uses this feature to load a list of file names into a frame so that the user can select one by moving the pointer bar to it and pressing <RETURN>. The function is actually handled in a fairly generic way by the DirManager module in the PMIdbms program, so your programs may be able to use it without modification.

This implies that you will occasionally have to explicitly set the frame's CurrentField variable to the top of the FieldList; you may want to examine every field in the frame, for example, or you may want to examine a field known only by its position in the list. ResetFieldPtr from the ScrnUtil module provides the most convenient way to do this:

```

PROCEDURE ResetFieldPtr( VAR TheFrame: DisplayFrame);

```

### 3.7.3. Pushing and Popping Frames.

*Repertoire* allows the programmer to save the current frame in a "stack" of frames and return to it later. A separate stack is maintained for each DisplayFrame. In fact, one of the subfields in the DisplayFrame record is a pointer to where the next frame on the stack is stored (see subfield PushedFrame in section 3.7.1). To save the current frame on the stack, you would call PushFrame and pass the DisplayFrame. *Repertoire* will save the current contents of the DisplayFrame and let us now use the DisplayFrame to read and display other frames. PushFrame does not save the screen image. Use SmartScreen.ScreenSave to save an exact image of the screen (see the SmartScreen section of the Repertoire Reference Manual).

When you wish to return to the last saved frame, call PopFrame. PopFrame only fills the displayframe record with the saved data—it does not restore the screen. The declarations of these procedures are:

```

PROCEDURE PushFrame(VAR frecord : DisplayFrame);

```

```

PROCEDURE PopFrame(VAR frecord : DisplayFrame);

```

PushFrame and PopFrame can be handy in limited cases, but they do not work well with Repertoire's FrameManager module; after a frame has been pushed, FrameManager does not know anything about it, and is therefore unable to restore the image of the pushed frame after something pops up

## Screen Design/Display System

over it. It is therefore best to use these routines only when it is practical to erase the frame after pushing it and redraw it after popping it.

As another example, if you want a special frame name to represent an "escape" to another set of frames, you could use:

```
IF NOT OpenDisplayFile( MyScreenFile,
    'FileName.dsp' ) THEN
    RETURN;
END;
NextFrame := '1';
REPEAT
    ReadShowAndCntrl(MyScreenFile, MainFrame,
        NextFrame );
    IF Equal( NextFrame, Special ) THEN
        FrameSequence( MainFrame, Special );
        NextFrame := SpecialNext;
    END;
UNTIL Equal(NextFrame, UserOps.ProgramExit);
CloseDisplayFile( MyScreenFile);
END ProgName.
```

The code for `FrameSequence` in `ControlUtils.MOD` provides a short, concrete example of using push and pop.

Remember that your `DisplayFrame` variable should be considered empty after it has been pushed. To do anything else with it, you should immediately call a routine like `ReadFrame` that refills the `DisplayFrame`. For example:

```
PushFrame( YourFrame );
```

```
(* The data that was in YourFrame is now in Your-
   Frame^.PushedFrame, but everything else in
   YourFrame has been set to 0 or NIL to
   prevent confusion between it and the pushed
   frame. *)
```

```
ReadFrame( ScreenFile, YourFrame, 'EntryForm' );
```

```
(* Now we can treat YourFrame as having meaningful
   values--it contains the frame named
   'EntryForm'.*)
```

### 3.7.4. Saving Screen Input in an NdxFile

*Repertoire* provides a special facility to allow screen data to be stored in a *Repertoire* NdxFile. The complete procedure is explained in section 4.4.1 under NdxFiles. The brief outline is:

1. First, you should create your data entry frame, giving names to all of the fields which you want to store.
2. You should create an NdxFile whose structure includes the names of all of the fields in the frame that you want to store.



3. Now your program that displays the data entry frame can store data entry information. After calling `Input`, `Control`, or `ControlFrame` to accept user input, your program may call the routine `PutScrRec` to automatically put the frame data into an `NdxFile` record. You may either pass a record key to the routine or identify the frame field that contains the key. The `NdxFile` routine `WriteRecord` must subsequently be called to save the record in the file.
4. Your programs may also retrieve data from an `NdxFile` and display it in a frame using the routine `FillScr`. `FillScr` takes data from a named record and moves the fields with the same names as current frame fields onto the screen. It is not case-sensitive in dealing with field names, although it is case sensitive in dealing with record names. You should call `FillScr` after the frame has been read from the file with `ReadFrame` or `Display` (you could also call it after using `Input` or `Control` to read in the record name desired). Then you should call `ShowDisplayFrame` (or `ControlFrame` with the `Reshow` parameter set to `TRUE` if you have already called a show routine) to display the data on the screen.

The file system example program contained on the distribution diskette saves its information from screens with these facilities. It includes add, change, display, and delete transactions. Examination of that code will give you the general idea. A more complete explanation is contained in section 4.4.1.

### 3.7.5. Detecting Normal Screen Submission.

You may prefer to structure programs that only perform certain functions when the user has exited the frame in a normal manner. For example, in a data entry frame, you may decide that the data will not be stored if the user chose to backup to the previous frame, or pressed the `ProgramExit` key to stop the entire program. Thus, in our example 1, we could do the following check after calling `Input` for frame 4:

```
IF (NOT Equal(NextFrame, YourFrame^.parent)) AND
   (NOT Equal(NextFrame, EndCode)) THEN
    ProcessData(YourFrame);
END;
```

Alternatively, you could process only when `NextFrame` is equal to the `NormalNext` value for this frame. The difference in these two checks is that in the first case we will process if the user pressed the "Submit" key while in a `GoTo` field. In the second case, we will not. If there are no `GoTo` fields in the frame, the two checks are equivalent.



### 3.7.6. Programs Changing Frame Data

Since your programs can add and change frame data before calling *Repertoire* routines to display the frames or receive user input, *Repertoire* can potentially handle all user interactions that your programs require. Your programs can create frames from scratch by simply initializing them (with `InitDisplayFrame`), writing to them (with `WriteToFrame`), and displaying them (with `ShowDisplayFrame`). Or you can use `ScrnComp.EXE` to store part of the frame in a .DSP file, and then add to it or change it at runtime. No matter how you create the frame, you can display it with `ShowDisplayFrame`, and control user input to it with `ControlFrame`.

Your programs may want to change frame data in two ways: by changing the data in specific fields, and by adding to a frame.

#### 3.7.6.1. Modifying Frame Fields

The routine `ChangeField` lets a program put a new value into a named field.

##### ChangeField

```
PROCEDURE ChangeField(VAR TheFrame : DisplayFrame;  
    thestr : ARRAY OF CHAR; FieldName : ARRAY  
    OF CHAR; PreserveLength: BOOLEAN);
```

The parameters are the `DisplayFrame`, the new value (an `ARRAY OF CHAR`), the field name, and a `BOOLEAN` that tells `ChangeField` whether you want it to preserve the former length of the field. `ChangeField` works with any kind of field, so `DisplayOnly` fields can be changed if they are named.

The `PreserveLength` parameter was not a part of `ChangeField` in *Repertoire* releases preceding 1.4. It is useful because `ControlFrame` depends on the trailing and/or leading blanks in an input field to know how wide the pointer bar should be when it moves into that field. Passing `TRUE` to `ChangeField`'s `PreserveLength` parameter prevents the `PointerBar` from appearing to be truncated after your program has corrected or modified the user's input in a field.

After you call `ChangeField`, you should next call one of the routines that displays information on the screen (`ShowDisplayFrame`, `RedrawArea`, etc.). If you call one of the routines that reads frames from the `DisplayFile` (`FrameSequence`, `Display`, `ReadFrame`), it will wipe out the current frame in memory, including any changes made with `ChangeField`, and replace it with the frame as stored in the file.

##### FindField

```
PROCEDURE FindField( TheFrame: DisplayFrame; Field-  
    Name: ARRAY OF CHAR; VAR FieldPtr: In-
```



```
putFieldPtr; VAR ImageRec: ImageElement ):
BOOLEAN;
```

```
PROCEDURE FindFieldNum( TheFrame: DisplayFrame; Field-
Num: CARDINAL; VAR FieldPtr: InputFieldPtr;
VAR ImageRec: ImageElement ): BOOLEAN;
```

*Repertoire's* `ScrnUtil` module provides numerous ways to gain direct access to the list element for a field. `FindField` and `FindFieldNum` are likely to be most generally useful. They both return a pointer to the field's `InputFieldRecord`, and a copy of the field's `ImageElement` record. If you dereference the `InputFieldPtr` and change parts of the record, your changes will immediately affect the `FieldList`. (See section 3.7.2 for the complete list of elements in the record). If you change the `ImageElement` record, your changes will go into effect only after you pass the modified `ImageElement` record to `PutFieldImageRec`. For example, you could assign a new color to the field's `ImageElement.backg` value to change the background color of the field. You can also change the field's type, its position in the frame, or any of its other attributes.

For example, to put the words "Request Information" into the field named "action" in a frame named "FirstForm," we could use the following code:

```
ReadFrame( ScreenFile, FrameVar, 'FirstForm');
ChangeField( FrameVar, 'Request Information', 'action',
TRUE );
ShowDisplayFrame( FrameVar, 1, 1, 80, 25);
```

### 3.7.6.2. Adding To Frames

Suppose we want to build a list of the current records in a file, and let the user choose which one to process. If we can get the file data into a frame, we can let *Repertoire* control user selection of a record as if each record name were a menu item.

For convenience, we include much of the frame in our .EDT file. This portion includes the frame's command area (the help message, window position, parent and normal next frames) plus the frame's heading and any explanatory text. Our program calls `ReadFrame` to read the frame, and then adds fields to it. Finally, we call `ShowDisplayFrame` and `ControlFrame`.

Adding fields is simple. The `ControlUtils` module provides two procedures, `AddStrField` and `AddMenuItem`, which do it for you in one step.

```
PROCEDURE AddStrField(VAR TheFrame : DisplayFrame; Col,
Row : CARDINAL; text : ARRAY OF CHAR;
width: CARDINAL );
PROCEDURE AddMenuItem( VAR TheFrame : DisplayFrame; Col,
Row : CARDINAL; text : ARRAY OF CHAR;
width, MenuKey: CARDINAL; ReturnVal : ARRAY
OF CHAR );
```



## Screen Design/Display System

The Col and Row parameters let you designate the position of the field relative to the beginning of TheFrame, not its absolute screen location. They are offset from 1 (i.e., (1,1) represents the top left corner of the frame). The 'text' parameter is the text for the new field.

If you want to add other kinds of input fields to a frame, begin by examining the source code for AddStrField and AddMenuItem. The basic steps are to declare an InputFieldRecord and an ImageElement record, assign new values directly to the InputFieldRecord, and use the ScrnUtil2.CreateImageElement procedure to assign new values to the ImageElement record. Then use the AddImageElement procedure from ScrnUtil2 to add the ImageElement to the frame's ImageList, and use PutFieldRec to add the InputFieldRecord to the frame's FieldList. You can reduce the number of steps required by using AddStrField to put the field in its proper place, and then using GetFieldPtr to change the type of the field or any of its other attributes to the values you desire.

Adding things other than input fields to a frame can normally be accomplished very simply through WriteToFrame:

```
PROCEDURE WriteToFrame( VAR FrameRec: DisplayFrame; col,
                        row: CARDINAL; text: ARRAY OF CHAR );
```

WriteToFrame does not immediately affect the physical screen. It merely adds the string to the frame's ImageList, so that it will be displayed when you pass the frame to ShowDisplayFrame or RedrawArea.

*Repertoire* also provides DisplayOnly fields, which are essentially a powerful way of writing to named areas of a frame.

DisplayOnly fields are just like normal input fields, except that the pointer bar does not move into them. They are used principally to display text in a way that lets *Repertoire* redraw it when necessary and automatically scroll it into and out of the frame's window. For example, you might want to display the current date or time at a particular location in the frame. DisplayOnly fields let you choose the location when you design the frame; thereafter, you can just refer to the location by name when you update the field with the ChangeField procedure.

As an example of adding fields, suppose we want to read data from a file and print each line in a GoTo field on the screen. Then we will let the user choose one to process. We want to skip a line between each field, and start them all in the same column. We start with the editable frame:

```
=====
:DocketList
  NormalNext: DecisionFrame
  ParentFrame: TopMenu
  Help:{
Using the up and down arrows, select a decision
```



to review. Press Return when you have made your selection.

}

-----  
Case Summary System: Choosing Cases to Review

| Docket No. | Case Name | Disposition |
|------------|-----------|-------------|
| -----      | -----     | -----       |

=====

The program fragment below assumes that the file containing our text is open, and that the text in the file is already formatted the way we want to print it (another intermediate routine could handle this). The portion of the program that adds fields to this frame is:

```
ReadFrame( ScreenFile, FrameVar,
'DocketList');
RowCount := 6; (*Start on the 6th row down.*)
REPEAT
  ReadStr( TextFileHandle, thestr);
  AddMenuItem( FrameVar, 2, RowCount,
    thestr, 10, 0, FrameVar^.normlnext );
  INC( RowCount );
UNTIL EndReached( TextFileHandle );
ShowDisplayFrame( FrameVar, 0, 0, 0, 0 );
ControlFrame(FrameVar, 0, '', FALSE, NextFrame);
IF Equal( NextFrame, FrameVar^.normlnext ) THEN
  (*The user has submitted the screen--hasn't
  pressed the backup or program-exit key. Now
  we can examine FrameVar^.CurrentField to
  determine which item the user selected. *)
END;
```

### 3.7.7. Interpreting GoTo values differently

We can use any FrameNames we want for the GoTo values, examine the results in the calling program, and take appropriate action. The GoTo values should differ from the ParentFrame value so as to provide a convenient way to tell whether the user is backing up, exiting, or submitting the frame. Usually, we want to use a unique GoTo value for each GoTo field. *Reper-toire* does not require that the values be unique. If you want to take the same action on several options, you may reuse the GoTo values.

If you do not use GoTo values to indicate the next frame, you can hard code the sequence of frames in your program. If you only have GoTo fields in the frame, ControlFrame does not need a NormalNext value for the frame. You could set NormalNext in the editable screen file. Then before calling the next frame, your program can assign:

```
NextFrame := YourFrame^.normlnext;
```

### 3.7.8. Automatic Frame Scrolling

When a frame is larger than the window defined for Display, *Repertoire* only displays the portion that will fit. In *ControlFrame*, when a user tries to move into fields outside the window, *Repertoire* scrolls the frame either horizontally or vertically enough to guarantee that all of the field fits within the window. When the user tries to move to the last field in a particular direction, *ControlFrame* scrolls the frame as far as possible in that direction so as to show all of the frame.

*Repertoire* also provides a way to scroll frames character-by-character instead of field-by-field. If your frame contains no input fields, and is not a *HitAnyKey* frame, the *ControlFrame* procedure will allow the user to scroll the frame up and down with the arrow keys and with the PgUp and PgDn keys if parts of it extend outside the frame's window.

When using scrolling, you can define a Heading Line in the editable screen file. This is the line below which *Repertoire* will scroll text. If the first line of your frame is blank, and the next three lines are the headings, set Heading Line to 4. The default Heading Line is zero, which causes all lines in the frame to be scrolled. (See Section 3.7.8).

Most likely, you will not prepare frames in an editable screen file that depend on scrolling, although *Repertoire* can handle them. Users may be uncomfortable with frames that have hidden parts, and it is usually simple enough to split the information into more than one frame. Scrolling fits naturally in situations where the user must select an item from a list that is longer than the window. Lists your program may create, as explained in section 3.7.6.2, may require scrolling.

### 3.7.9. Changing the Specially Handled Keys

There are four ways in which you may change the action of keys in *Repertoire* screen input routines. You may change:

1. The special keys that perform actions like help, time, etc., and the actions that these keys perform;
2. The keys that may be typed to enter information in string and editor fields.
3. The cursor control and editing that is done in the input routines for strings and numerics.
4. The screen movement keys that do things like move between fields on the screen.



### 3.7.9.1. Changing the Special Keys

Changing *Repertoire's* set of special keys requires three steps. You must change `UserOps.FunctKeySet` to your new special key set. You must either modify `UserOps.DefaultKeyHandler` or write a new procedure to process the additional keys and assign the new procedure to the `UserOps.TheKeyHandler` procedure variable. Finally, you must set up a way for users to see what your special keys do.

You can make these changes inside the `UserOps` implementation module, or you can import values from `UserOps` and make the new assignments in your own Module. We recommend that you make changes to `UserOps` that you intend to use in every program that uses *Repertoire*. If the change applies only to one program, you should import the variable and set it in your program.

*Repertoire's* `ControlFrame` imports the variable `FunctKeySet` from `UserOps` to see what special keys you want recognized. `FunctKeySet` is a specially encoded representation of IBM PC keys that *Repertoire* calls a `KeyNumSet` (The *BigSets* section of the *Repertoire Reference Manual* explains how to create `KeyNumSets`). You may use any keys for special keys except the keys `ControlFrame` recognizes as valid in `STRING` field entries (these may also be changed--see below). You may use any of the extended key scan codes. The IBM extended key scan codes are listed in several IBM Manuals; this manual includes a list in Appendix B.

`ControlFrame` imports the procedure variable `TheKeyHandler` from `UserOps` to process special keys. When the user presses one of your special keys, control will pass to the routine that is assigned to `TheKeyHandler`. If you write your own procedure, pattern it after `UserOps' DefaultKeyHandler`. `UserOps` also exports the type `AKeyHandler` to help you define and manage multiple `TheKeyHandler` procedures. (`UserOps` globally defines the constants `HelpKey`, `TimeKey`, `BackUpKey`, `SubmitKey`, and `ExitKey` to the standard *Repertoire* values for the F1, F3, Esc, F10, and F2 keys respectively. In changing `DefaultKeyHandler`, you may modify these values and add similar ones as long as you also add them to `FunctKeySet`.)

If you write your own key handling procedure, it must have the same parameters as `DefaultKeyHandler`. The first parameter is the `DisplayFrame` in which `TheKeyHandler` was called. The second is the numeric representation of the key that the user pressed. The last parameter (`NextFrame`) allows `TheKeyHandler` to signal `ControlFrame` that it should exit and return the designated `NextFrame` value; normally, `TheKeyHandler` sets `NextFrame` to 'CONTINUE', which tells `ControlFrame` to keep going. Any other returned value causes `ControlFrame` to exit and pass that same `FrameName` through to the calling program.



## Screen Design/Display System

Notice that TheKeyHandler's LastKey parameter is a VAR instead of a value parameter. This is important because ControlFrame gives TheKeyHandler first access to key strokes. As a result, TheKeyHandler can conveniently control how the rest of the screen system responds to function keys. If it wants to temporarily prevent the up/down arrows from working, for example, it can filter them out before passing the keystrokes on to the rest of the screen system.

Inside DefaultKeyHandler, a CASE statement determines which special key was pressed. The Timing key calls a message box and exits. (the WindowPrims, MsgBox1, and MsgBox2 sections of the Repertoire Reference Manual explain message boxes.) The Help key action depends on whether a HelpFrame or string is set for the current field or frame. (Notice that DefaultKeyHandler calls FrameSequence in the former case, so it can display a sequence of help frames.) "Backup" returns the parent in the DisplayFile record. "Submit" returns normlnext unless the current field is a GoTo field (abbreviated "J" for Jump).

The Exit key displays a message box which, by default, prompts the user to press the exit key again to return to DOS. If DefaultKeyHandler determines that the user pressed the Exit key again in the Message box, DefaultKeyHandler calls ErrorManager's StraightToDOS procedure. StraightToDOS executes a list of termination procedures, to which you can add housecleaning code, if necessary. If adding all your file closing and screen restoration code to ErrorManager's TermProcList is too cumbersome, you can remove the call to StraightToDOS and make DefaultKeyHandler instead set NextFrame to the value ScrnTypes.ProgramExit ('0'). Your code will then be responsible for finding its own way back to DOS when it sees that ControlFrame has returned a NextFrame equal to ProgramExit.

UserOps.EndPrompt contains the message that the exit routine writes in the message box; your programs may import and change the EndPrompt, if necessary. By default, EndPrompt reads: 'Press the F2 key again to exit, or any other key to resume.'

### 3.7.9.2. Changing the Field Entry Keys

You may change the keys that *Repertoire* recognizes in String fields by changing the KeyNumSet StringKeySet, exported from UserOps. Any keys that are in StringKeySet can be entered in their respective types of fields.

### 3.7.9.3. Changing the Cursor Movement Keys

In releases of *Repertoire* preceding 1.4, redefinition of arrow and function keys had to be accomplished by modifying the key values associated with the functions, because TheKeyHandler was called after Repertoire did its own special key processing. TheKeyHandler is now called first, and it can



change the key values that it passes on to Repertoire for processing of special keys.

A convenient way of handling special cases on a temporary basis is to save a copy of the default TheKeyHandler procedure variable, and then call it inside the procedure temporarily assigned to TheKeyHandler.

#### 3.7.9.4. Changing Field and Key Handling

UserOps exports a procedure variable similar to TheKeyHandler named TheFieldHandler; ControlFrame calls TheFieldHandler every time it places the pointer bar on an input field. This code is in UserOps so you can change it to meet your personal preferences for user interfaces. You could also build application-specific error-checking ability into a TheFieldHandler routine. This is the procedure assigned to TheFieldHandler by default:

```
PROCEDURE DefaultFieldHandler( FrameRec : DisplayFrame;
                               col1, row1: CARDINAL; VAR LastKey :
                               CARDINAL; VAR NextFrame: AFrameName );
```

The first parameter is the DisplayFrame in which TheFieldHandler was called. It gives TheFieldHandler access to all the information it could possibly need for manipulating the field or the frame itself.

The next two parameters, col1 and row1, represent the absolute position of the field on the screen. These values prevent TheFieldHandler from having to calculate the field's physical screen position if for some reason it needs to directly affect the region of the screen that the field occupies. UserOps' DefaultFieldHandler procedure, for example, uses string input procedures (from the StrInput module) which know nothing at all about Repertoire's high-level screen system, so DefaultFieldHandler has to pass them absolute screen locations. Without the col1 and row1 parameters, it would have to take the field's virtual coordinates (i.e., its coordinates relative to the top left corner of the frame), add to them any offset of the frame's window from the top left corner of the physical screen, and then subtract any columns or rows that the field might have scrolled.

The fourth parameter, LastKey, comes in as the numeric representation of the last key pressed, and should leave as the key that TheFieldHandler wants ControlFrame to think was the last key pressed.

The last parameter, NextFrame, has no meaning coming in, but going out, it must equal the string constant ScrnTypes.ContinueInput ('CONTINUE') if TheFieldHandler wants ControlFrame to continue processing the current frame. If TheFieldHandler sets NextFrame to anything except ContinueInput, ControlFrame will exit and return that value to the calling procedure.

The first question users normally ask in trying to understand TheFieldHandler is how it returns the field itself. The answer is that the field is part of



## Screen Design/Display System

the `DisplayFrame` passed as `TheFieldHandler`'s first parameter. `TheFieldHandler` knows which field it is being asked to handle by examining the `DisplayFrame`'s `CurrentField` value (`TheFrame^.CurrentField`). It must operate directly on the `FieldList` element identified by `CurrentField` because the field may not be a string --it might be a list of strings, for example. `TheFieldHandler` is responsible for determining the type of the field and for using data manipulation routines compatible with that type.

If you need to update other fields from within `TheFieldHandler`, this is the general approach: On entry to `TheKeyHandler` and `TheFieldHandler`, the `DisplayFrame^.CurrentField` value is guaranteed to be correct. If, for example, you want to update field 3 while the `CurrentField` is field 7, use something like

```
Scrnl1.GetFieldImageRec(TheFrame, 3, ImageRec);
```

to get a copy of the `ImageElement` for field 3. Modify `ImageRec.text` in any way you like, and then use `PutFieldImageRec` to put it back in the `TheFrame`'s `ImageList`. Use `RedrawField` to update the screen, but check first to make sure the field hasn't been scrolled out of view by calling the `Scrnl1.EntirelyVisible` function.

Although you could use `GetFieldImagePtr` to modify the text field directly, it is not a good idea unless you thoroughly understand the screen system's innards. The text field is only partially allocated, for example, and the other values in the `ImageElement` record are actually encoded while they are in the `ImageList` (`GetFieldImageRec` decodes as it copies out, and `PutFieldImageRec` encodes).

Be particularly careful in using `Scrnl1` procedures like `FieldNum` or `FindFieldNum` or `FindField` inside `TheKeyHandler` and `TheFieldHandler`, because some of them change the `DisplayFrame`'s `CurrentField` value. That will cause problems if you fail to set `CurrentField` back to its original value before exiting the routine. The safest approach, if you need to use a `Scrnl1` procedure that changes the frame's `CurrentField` value, is to save the `CurrentField` in a cardinal variable at the top of the procedure, and at the bottom of the procedure use `FindField` or `FindFieldNum` so as to get both the `CurrentField` and the field/image list pointers back to their original positions.

### 3.7.10. Changing the Input Prompts

*Repertoire*'s input control routines (`InputManger`'s `ControlFrame` and the higher-level procedures in the `ControlUtils` module) write a prompt line to tell the user what kind of entry is valid in the current field. This prompt is written in a screen position defined by the programmer. If a box line is drawn in this area, the prompts write over the box line. As the user moves



from field to field, this prompt changes. For maximum flexibility, *Repertoire* will even allow the editable screen file to specify a custom prompt for any input field.

UserOps exports several global variables that you can set to affect how *Repertoire* does this prompting.

### 3.7.10.1. Whether to Prompt

The first choice is whether to display any prompts at all. If the UserOps variable DoPrompting is set to FALSE, the prompts are not printed.

### 3.7.10.2. Prompting Size and Position

You can change the screen row and column in which the prompts will start by setting PromptRow and PromptCol. The default position is the first row up from the bottom of the screen (e.g., row 24 in normal text mode, row 42 in the EGA's 43-line mode, etc.). When you display windows in various positions on the screen, you can move the prompts also, or leave the prompts in one position—for example, at the top or bottom of the screen. If you want to move the prompts, you must explicitly assign new values to PromptRow and/or PromptCol. We have found it generally more convenient, for both the user and programmer, to leave the prompts in one position, no matter where the window is placed.

You can import and change the UserOps variable PromptLength to set the length of the prompts. *Repertoire* guarantees that prompts do not extend beyond the allowable length, and it automatically centers prompts within that length.

### 3.7.10.3. The Prompts

You can change the prompts that *Repertoire* uses for each field type by setting the corresponding prompt variables from UserOps. GoTo, Choice, String, Editor, Integer, and Real fields are prompted for with JumpPrompt, ChoicePrompt, StringPrompt, IntegerPrompt, and RealPrompt, respectively. Since *Repertoire*'s defaults tend to be fairly long, we show them on multiple lines below. *Repertoire* actually prints each one on a single line. It substitutes the actual screen characters for the <Arrow> notations below.

JumpPrompt: 'Enter selects the highlighted choice. Use <UpArrow> or <DownArrow> to move to another choice.'

ChoicePrompt: 'Use <LeftArrow> or <RightArrow> to change the highlighted choice, or <UpArrow><DownArrow> to move to another field.'

StringPrompt: 'Fill in the blank and press <UpArrow> or <DownArrow> to change fields, or F10 if finished.'

## Screen Design/Display System

EditorPrompt: 'AltD deletes lines; AltU undeletes; Shift-F7 toggles word wrap.'

IntegerPrompt: 'Enter a number between \_\_\_\_ and \_\_\_\_.'

RealPrompt: (Same as IntegerPrompt)

Any of the above prompts are overridden when a prompt is specified for a particular field in the editable screen file.

### 3.7.10.4. Prompt Colors

The prompt colors are set for each open frame--global variables are not used. The subfields `promfor`, `prombak`, and `promatr` of the `DisplayFrame` record can be set to change the colors and attributes in which the prompt line will be written.

### 3.7.11. Writing in Small Windows

When displaying a frame in a small window, *Repertoire* automatically handles most adjustments. `ShowDisplayFrame` writes only the number of lines that fit in the defined window, and `ControlFrame` scrolls fields and text as needed. You should remember to adjust the Prompt variables (see preceding section), if you want to move or disable input prompts. If you do not move prompts, they will be written on the same screen row, regardless of where you locate the window.

### 3.7.12. Using Many Frames at Once

*Repertoire* allows you to display as many frames as you wish at once. You can display frame A and control input to it; then display and control frame B while frame A is still showing. Then you could display A again. Even though only one frame is actually active in the program at one time, the user sees several at once.

`ControlFrame` operates on the `DisplayFrame` you pass to it. Several routines in the `ControlUtils` module operate on the default frame, `MainFrame`, but it is best to declare and initialize your own `DisplayFrame` variables--at least one for every frame that will be simultaneously visible. In other words, if your program will be displaying a large number of frames, but no more than two of them will be visible at any one time, you need only two `DisplayFrame` variables.

To create and use more than one `DisplayFrame` with a single `DisplayFile`, you must:

1. Import the routine `InitDisplayFrame` from `ScrnTypes`, the procedure `CloseDisplayFrame` from `ScrnUtil2`, and the type `DisplayFrame` from `ScrnTypes`.



2. Declare a variable of type DisplayFrame.
3. Initialize the DisplayFrame variable with InitDisplayFrame.
4. When using this DisplayFrame variable, use ReadFrame instead of ReadFrameFast, and use only the screen system routines that have a DisplayFrame as a parameter (e.g., ReadFrame, ShowDisplayFrame, ControlFrame, etc.). Older procedures such as Control(), which operate only on DspFiles.MainFrame, will not work.

The reason you should use ReadFrame instead of ReadFrameFast is that, for speed purposes, ReadFrameFast does not disconnect the DisplayFrame's lists (e.g. its FieldList, ImageList, etc.) from the DisplayFile's list buffer. Thus, when you read a new frame from the same DisplayFile, any frame in memory that was originally read using ReadFrameFast will disappear (except for frames that have been pushed, popped, or stored in a frame list). ReadFrame, on the other hand, begins by disposing of the frame variable's lists, reads the frame from the DisplayFile, and then copies the DisplayFile's list buffer into the frame instead of aliasing the list buffer to the frame's lists. They will be disposed of the next time you call ReadFrame, or when the frame is closed. ReadFrame is safer, but a little slower, than ReadFrameFast.

5. After you are finished with a frame, call CloseDisplayFrame to deallocate the frame and any pushed frames under it.

In the example below, we have two separate series of frames that we want to switch between at any time. These frames could be from separate files, although below they are from the same file.

```
MODULE TwoFace;
  (*UserOps.OldKeyHandler must be assigned
    to UserOps.TheKeyHandler for this to
    work.*)
FROM BigSets IMPORT AppendSet;
FROM DspFiles IMPORT MainFrame;
FROM ScreenDisplay IMPORT CloseDisplayFile,
  CloseDisplayFrame, OpenDisplayFile,
  ReadFrame, ReadFrame,
  ShowDisplayFrame;
FROM ScreenInput IMPORT ControlFrame;
FROM ScrnTypes IMPORT DisplayFile, DisplayFrame,
  FrameKey, InitDisplayFrame;
FROM SmartScreen IMPORT ClearScreen,
  SetCursorHeight;
FROM StrConv IMPORT ReturnedInt;
FROM UserOps IMPORT AFuncntProc, EndCode,
  FunctKeySet, TheFunctProc;
FROM VWindows IMPORT CurrentWindow;
FROM WindowPrims IMPORT PopCursorHeight,
  PushCursorHeight;
```

## Screen Design/Display System

```
IMPORT InitCompilerMods;

VAR
  Frame1 : DisplayFrame;
  NormlFuncProc : AFuncProc;
  NextFrame1, NextFrame2: FrameKey;
  ScrnFile: DisplayFile;
  NotSwitching : BOOLEAN;

CONST
  SwitchCode = 9999;

PROCEDURE DoFunctions( FrameRec : DisplayFrame;
  VAR LastKey: CARDINAL ) : INTEGER;
BEGIN
  IF (LastKey = 322 (*F8*)) THEN
    RETURN SwitchCode;
  ELSE;
    RETURN NormlFuncProc( FrameRec, LastKey );
  END;
END DoFunctions;

BEGIN
  PushCursorHeight();
  SetCursorHeight(0);
  ClearScreen();
  (* clear screen and save cursor height *)
  AppendSet( FunctKeySet, "{322}" );
  (* add F8 to FunctKeySet as window switching
    key*)
  NormlFuncProc := TheFuncProc;
  TheFuncProc := DoFunctions;
  (* save main TheFuncProc and set TheFuncProc
    to our processing routine *)
  OpenDisplayFile( ScrnFile, "twoface.dsp");
  InitDisplayFrame( Frame1,
    CurrentWindow );
  (* Initialize Frame1 for the first frame set.
    We will use MainFrame for the 2nd frame set.*)
  NextFrame1 := 1;
  (* Start frame set one at frame 1 *)
  ReadFrame( ScrnFile, MainFrame, 10);
  (* Put starting frame of second set (frame 10 in our
    example) in second frame set since we will
    always assume below that frame set 2 is
    loaded when we switch to it. *)
  NotSwitching := TRUE;
  (* used to keep us from re-reading a frame from the
    file when we switch back to a frame set*)
  REPEAT
    REPEAT
      (* show frame set 1 until exit or switch *)
      IF NotSwitching THEN
        (* read frame from file unless we just
          switched back to this frame from the
          other frame set (in which case we want
          to retain the old frame with whatever
          input it contained when we switched
```



```

        out) *)
    ReadFrame( ScrnFile, Frame1,
        NextFrame1);
    (*Note that we use ReadFrame instead of
    ReadFrameFast.*)
END;
ShowDisplayFrame( Frame1, 1, 1, 46, 14);
    (* display this frame set in the upper
    left hand corner *)
NextFrame1 := ControlFrame( Frame1, 0, '',
    FALSE);
NotSwitching := TRUE;
UNTIL (NextFrame1 = EndCode) OR
    (NextFrame1 = SwitchCode);
NotSwitching := FALSE;
IF NextFrame1 = SwitchCode THEN
    NextFrame2 := ReturnedInt(
        MainFrame^.ThisFrame );
    REPEAT
        (* show frame set 2 until exit or switch*)
        ReadFrame( ScrnFile, MainFrame,
            NextFrame2);
        ShowDisplayFrame(MainFrame, 35,10,80,23);
        NextFrame2 :=ControlFrame(MainFrame,0,'',
            FALSE);
        NotSwitching := TRUE;
        UNTIL (NextFrame2 = EndCode) OR
            (NextFrame2 = SwitchCode);
        NotSwitching := FALSE;
        NextFrame1 := ReturnedInt(
            Frame1^.ThisFrame);
    END;
UNTIL (NextFrame1 = EndCode) OR
    (NextFrame2 = EndCode);
CloseDisplayFile( ScrnFile);
CloseDisplayFrame( Frame1);
PopCursorHeight();
ClearScreen();
    (* clear screen and restore cursor height *)
END TwoFace.

```

A more advanced treatment of multiple windowing could employ Modula-2's process creation and transfer capabilities. Section 5.10 explains how to set up this processing for *Repertoire's* editor. You could implement a similar scheme with multiple processes using the screen display system.

### 3.7.13. Using Multiple Screen Files

The only potential complication involved in using more than one screen file at once is in deciding (and then remembering) which of the files is to be associated with `DspFiles.MainFrame`. By default, it is associated with the first `DisplayFile` opened. If you open one `DisplayFile`, close it, and then open another `DisplayFile`, `MainFrame` will be associated with the second file, but the general rule is that, when you have simultaneously open `DisplayFiles`, `MainFrame` is associated with the first one opened. The higher-

## Screen Design/Display System

level routines in *Repertoire's* screen system (Input, Control, FrameSequence, Display, etc.) will all use the DisplayFile associated with MainFrame.

Thus, when you have simultaneously open DisplayFiles, you must access the second one using a separate DisplayFrame variable and using only the newer routines that allow you to specify the DisplayFrame explicitly, such as ReadFrame and ShowDisplayFrame. Alternatively, you can switch MainFrame to the second DisplayFile. To do so, simply assign your DisplayFile variable to MainFrame^.FromFile. You might want to push MainFrame onto its frame stack using the PushFrame procedure in ScrnUtl2 before doing so.

If you employ the first technique (i.e., using ReadFrame directly), remember not to pass MainFrame directly to ReadFrame unless you are certain that you are also passing it the correct DisplayFile variable. The reason is that ReadFrame assigns the DisplayFile variable in its first parameter to the .FromFile field of the frame passed to it in its second parameter.

As you may have gathered, you can save yourself a good deal of trouble by simply not using MainFrame. Like most other global variables, it is convenient in small programs, but tends to cause trouble in larger ones.

Of course, to use multiple DisplayFiles you must also define a separate DisplayFile variable for each file, and you must execute separate calls to OpenDisplayFile and CloseDisplayFile.



## 4. Indexed File Subsystem

### 4.1. Supported Features

*Repertoire's* indexed file system allows:

- Creation of files with a defined structure that names the subfields in each record;
- Storage of indexes and data together in one file.
- Storage of true binary images of your data, allowing any kind of Modula-2 data structure (records, arrays, real numbers, etc.) to be stored and retrieved without cumbersome conversion operations.
- Keyed read, insert, and delete access to entire records or to named subfields of records;
- Records of variable length;
- Garbage collection under program control;
- Recovery of damaged files;
- Subfields that can be linked lists (including indefinitely nested sublists) as supported by *Repertoire's* GenLists module;
- An automatic tie to *Repertoire's* screen system so that data entered on a screen can be stored in a file, or retrieved from it, with a single procedure call.

### 4.2. How to Use the File System

The definition modules for the file system's four major components (AltKeys, NdxFiles, NdxBones, and NdxUtils) contain many routines and data structures that are used only for intermodule communication; they normally need not be accessed by programs that use the indexed file system. At first glance, however, it may not be clear which of the routines are actually meant to be called. A simple scenario for basic use of NdxFiles is given below. The scenario covers all of the services your programs will normally need.

Basic Scenario:

1. Import the type NdxFileType, and the routines referenced below, from modules NdxTypes, NdxBones, and NdxFiles. Declare a variable of type NdxFileType to identify each NdxFile that the program will be using.

## Indexed File Subsystem

2. If the file does not yet exist, create it using the `CreateNdxFile` procedure. A string parameter to `CreateNdxFile` lists all of the fields in the file records that you care to name. When the file already exists, simply open it with the `OpenNdxFile` procedure.
4. Retrieve records and fields with the `GetField` procedure, and create or modify them with the `PutField` procedure. Write all created or modified records to the disk using the `WriteRecord` procedure before accessing any different record.
5. Delete records with the `DeleteRecord` procedure, and occasionally check the ratio of used to unused space in the file with the `SizeInfo` procedure. When the ratio gets low, compress the file with the `CrunchNdxFile` procedure.
6. Close the file with `CloseNdxFile` before exiting the program.

*Repertoire's* screen design/display system is actually built on top of the `NdxFile` system, but uses only the `NdxBones` module. The distribution disk provides an example of a program (`PMIDBMS.MOD`) that uses the complete `NdxFile` system. It provides, in addition to a demonstration of how to use the file system, hints about how you might connect the file system and the screen system.

The following sections describe the elements of this basic scenario in more detail, dividing the steps into creating, opening, closing, writing, reading, and maintaining an `NdxFile`.

### 4.2.1. Creating Files

The routine `CreateNdxFile` from the module `NdxFiles` makes new `NdxFiles`. It has two parameters: an `NdxFileType` variable (`NdxFileType` is imported from `NdxFiles`) and a string defining the file structure. This is how it is declared:

```
PROCEDURE CreateNdxFile( VAR NdxFile: NdxFileType;  
                          fname, Structure: ARRAY OF CHAR );
```

For example:

```
CreateNdxFile( TheFile, 'filename.ndx', 'Name, Age,  
                      HatSize, Planet');
```

**NdxFileType:** For every `NdxFile` that your program uses at one time, a variable of type `NdxFileType` must be declared. An `NdxFileType` variable is actually a pointer to a record with subfields that the `NdxFiles` module uses to track the current status of the file. This variable will also be passed to all other file handling routines to identify the file being used.

**File Structure:** `NdxFiles` assumes that, in the limited sense explained below,



every record in the file has the same structure and every field in the structure has a name. (You can ignore the NdxFile's field-name structure, however, if you need to store different kinds of records in the same file.) When you create the file, the list of names is passed to CreateNdxFile in a string, with the field names separated by commas. Each named field is an entity that you wish to read and write as a unit. Thus, fields can be strings, numbers, records that you define, or even lists. You may determine a file's structure with the routine FileStructure, which returns the same form of structure string used to create a file. This is how it is declared:

```
PROCEDURE FileStructure( NdxFile: NdxFileType; VAR
    Structure: ARRAY OF CHAR );
```

**Fields as Lists:** Any field can be a list (from the GenLists module), so that it is actually many fields. A GenList is simply a general form of linked list that allows many entries, such as a number of strings, to be treated as a single entity. (See Chapter 2 and the GenLists section of the Repertoire Reference Manual for more explanation of Genlists.) Any GenList in an NdxFile record can be accessed as a list by specifying its name. Also, fields within a list can be named if you wish to access them individually. To name subfields in a field that is a list, the list of subfield names is enclosed in braces and placed after the list field name. An example of such a sublist structure is:

```
CreateNdxFile( TheFile, 'filename.ndx', 'Name, Address
    {Street, City, State, Zip}, HatSize');
```

These sublists may in fact contain sublists which contain sublists, ad infinitum. For example:

```
'Name, Addr {Street {Num, StrName, AptNum},
    City, State, Zip}'
```

The point of naming fields is to allow NdxFiles' reading and writing routines to access individual fields by name. Thus, each field name must be unique within the list in which it occurs, but need not be unique within the file structure as a whole. You do not have to use field names at all if you are content to treat each record simply as a list; to do so, you simply use the word 'RECORD' as the field name when calling GetField and PutField. See the descriptions of those routines below for additional information.

### 4.2.2. Opening and Closing Files

A program that creates a file may immediately write data to it. A file that already exists, however, must be opened before it is accessible. Files must also be closed when the program is finished with them. To open a file, pass OpenNdxFile an NdxFileType variable and the file name in an ARRAY OF CHAR. OpenNdxFile will search for the file in the program's home direc-



## Indexed File Subsystem

tory and in the directories listed in the DOS PATH using the routine FindFile from the module Drectory. The routine returns TRUE if the file is successfully found and opened, and FALSE if not. This is how it is exported:

```
PROCEDURE OpenNdxFile( VAR NdxFile: NdxFileType; VAR  
    FileName: ARRAY OF CHAR): BOOLEAN;
```

To close an NdxFile, call this routine:

```
PROCEDURE CloseNdxFile(VAR NdxFile: NdxFileType);
```

### 4.2.3. Writing to Files

To write a piece of information to a file, call the routines PutField (or PutFieldAdr) and WriteRecord. You must first open the file with OpenNdxFile or CreateNdxFile. PutField puts information in a record (or updates information already in a record) and WriteRecord writes filled records to the file.

If you use PutField without any subsequent WriteRecord, the information is lost. In fact, since NdxFiles uses single-record buffering, WriteRecord must be called before a different record is accessed with PutField or GetField. Thus, if you call PutField for one record, do no WriteRecord, and call PutField for a second record, the original "put data" is lost. See section 4.4.9 for a discussion of how this restriction might be lifted.

#### 4.2.3.1. PutField:

PutField's parameter list takes the following information: the NdxFile to be used, the record name, the field to be updated, the field type, and the data to be put in that field. PutField is used in exactly the same way whether the record exists or not. If the record exists, it is retrieved, and only the field specified is updated. If the record that you specify does not exist, it is created. Its declaration is:

```
PROCEDURE PutField( VAR NdxFile: NdxFileType; RecName,  
    FileName: ARRAY OF CHAR; TheType: CAR-  
    DINAL; TheField: ARRAY OF BYTE ): BOOLEAN;
```

**Record Names:** These are the "keys" of NdxFiles. Each record has a unique name, which must be an ARRAY OF CHAR the size of which does not exceed the constant RecNameLength in the module NdxTypes. RecNameLength's default value is 20. If the record that you pass to PutField does not exist, it is created (GetField, of course, fails if you pass it a record name that does not exist). Record names are case-sensitive.

**Field Names:** The field names defined in the structure are used to identify the field to be updated. Thus, to update the field 'Name', the ARRAY OF CHAR 'Name' should be passed as the field. Field names are not case-sen-



sitive. Subfields are identified by the parent list name, a period, and the subfield name. For example, to identify a street number from the last example in section 4.2.1 above, pass the field name 'Addr.Street.Num'. You can ignore the field name structure of an NdxFile by using 'RECORD' as a field name. When you do this, you must use a GenList as the data to be written, and you must use GenLists.ListCode as the type code; the GenList will occupy the entire record.

**Field Types:** When data for a field is passed to NdxFiles, you also pass a "type code" for the data. This "type" is a CARDINAL number that is stored with the data and returned to later calling programs that read this field. NdxFiles only cares about two of these type codes--all other types are simply stored and later passed back to reading programs for the programmer's use. The types ListCode and StrCode (both exported from GenLists) are used to tell NdxFiles that a list or string is being passed. A list must be specially identified so that NdxFiles will search for and store all fields and sublists in the list. Strings are special because NdxFiles will only use the amount of storage required for the current text in a string (it does not store the part of the string beyond the null terminator).

An example of a PutField call is:

```
IF PutField( NdxFile, 'J.S.Brown', 'Name', StrCode,  
            'John Stanley Brown' ) THEN
```

In this example, the record 'J.S.Brown' will be retrieved from the file identified by the variable NdxFile, and 'John Stanley Brown' will be stored in the 'Name' field of that record. Remember that the record will not actually be written to the disk until you call WriteRecord.

### 4.2.3.2. WriteRecord:

This routine writes the named record to the DOS file associated with the NdxFile variable named in its parameter list. WriteRecord updates the file and returns TRUE only if either PutField or GetField was previously called for this record, and if the file-write was successful. The routine is exported from NdxFiles as:

```
PROCEDURE WriteRecord( VAR NdxFile: NdxFileType;  
                      RecName: ARRAY OF CHAR ): BOOLEAN;
```

### 4.2.3.3. DeleteRecord:

This routine deletes the named record from the NdxFile named in its parameter list. DeleteRecord updates the file and returns TRUE only if the record name passed exists in the file and the file was previously opened. This is how it is declared:

```
PROCEDURE DeleteRecord( VAR NdxFile: NdxFileType;  
                      RecName: ARRAY OF CHAR ): BOOLEAN;
```

## 4.2.4. Reading Files

*Repertoire* provides a combination of low-level and high-level routines for reading NdxFiles. The high-level routines allow an entire file (or any portion of a file) to be searched with logical operators that operate both between fields and within them. The low-level routines are provided for applications that require more speed and control. In between are the routines GetField, GetFieldAdr, GetRecName, and RecordExists, which provide the most commonly used file-reading functions.

### 4.2.4.1. GetField

You pass GetField the NdxFile to be used, the record name, and the field to be read. GetField's result is TRUE only if the record was found and read. Getfield returns the data found in that field and its stored type. This is how it is declared:

```
PROCEDURE GetField( VAR NdxFile: NdxFileType; RecName,
                   FieldName: ARRAY OF CHAR; VAR TheType:
                   CARDINAL; VAR TheField: ARRAY OF BYTE ):
                   BOOLEAN;
```

You can use GetField to retrieve an entire record as a list of all its fields without specifying a field name by using the word 'RECORD' as the field name and passing a GenList as GetField's last parameter. The list returned will contain one element for every field in the record.

Example:

To retrieve the 'PhoneNum' field from a record named 'J.S.Brown', call GetField like this:

```
IF GetField( NdxFile, 'J.S.Brown', 'PhoneNum', Result-
            Code, ResultStr) THEN
...

```

ResultStr should now contain the phone number stored for 'J.S.Brown', and ResultCode should equal GenLists.StrCode (assuming StrCode was the type code used when the field was written). GetField knows the TypeCode for the data because it is stored in the file. It does not assume that every field of the same name has the same type code. Thus you could, for example, store a string with a single phone number in the 'PhoneNum' field of one record, and store a list of several phone numbers in the 'PhoneNum' field of another record.

When you do this sort of thing (i.e., use one field name for data that may be of different types in different records), you must either do data conversion operations after calling GetField, or you must call GetField twice. The illustration below shows why.



**Example:**

Suppose you have a field in each record named 'coordinates' that stores an array of real numbers. The dimensions of the array, however, are different in each record. The field may contain any of these types:

```
TYPE
  RealArray10 = ARRAY [0..9] OF REAL;
  RealArray256 = ARRAY [0..255] OF REAL;
  TwoDimen = ARRAY [0..15] OF RealArray10;
```

To keep track of the different kinds of arrays, you assign each of them a different type code:

```
CONST
  Array10Code = 1;
  Array256Code = 2;
  ArrayTwoDCode = 3;
```

Before retrieving the 'coordinates' field of any given record, you don't know which kind of array it contains, so you have to do something like this:

```
VAR
  TenReals : RealArray10;
  ManyReals: RealArray256;
  TwoDReals: TwoDimen;
  TypeCode : CARDINAL;
BEGIN
  IF GetField( NdxFile, 'record23',
    'coordinates', TypeCode, TenReals) THEN
    CASE TypeCode OF
      Array10Code:
        (*We don't have to do anything;
          TenReals contains a value of the
          right type. We can process the
          field value here.*)
      | Array256Code:
        (*Whoops. We have to put the field
          value into a bigger variable.*)
        IF GetField( NdxFile, 'record23',
          'coordinates', TypeCode,
          ManyReals) THEN
          (*Process the field value here.*)
        END;
      | ArrayTwoDCode:
        IF GetField( NdxFile, 'record23',
          'coordinates', TypeCode,
          TwoDReals) THEN
          (*Process the field value here.*)
        END;
    ELSE
      (*Error handling code goes here--you've
        gotten an unexpected type code from
        the file.*)
    END;
  END;
END;
```

## Indexed File Subsystem

An alternative way of dealing with situations like this is to use `GetFieldAdr`, which gives you the address of the data it has read; you can use `LowLevel.Move` to transfer the data from that address into a variable of the proper type. You might also use a variable of the largest expected type in the first `GetField`. If the retrieved type code is inconsistent with the type of that variable, you can convert it into the type of variable you need, perhaps using `LowLevel.Move`.

The routine `RecordExists` will tell you if a record exists before you try to read it. This is how it is declared:

```
PROCEDURE RecordExists( NdxFile: NdxFileType; RecName:
                        ARRAY OF CHAR ): BOOLEAN;
```

For example:

```
IF RecordExists( NdxFile, 'J.S.Brown') THEN
    ...
END;
```

`GetRecName` allows you to read the records in a file without knowing their keys. You pass the routine a `CARDINAL` number indicating the position of the record (in terms of record numbers) relative to the beginning of the file, and `GetRecName` will return the name of that record. The name can then be used with any of the above routines. The routine returns `FALSE` if the record does not exist. One could thus use this routine in a `REPEAT` loop to read and process all records in a file, starting from 1, until `GetRecName` returns `FALSE`. This is how it is declared:

```
PROCEDURE GetRecName( NdxFile: NdxFileType; Which:
                     CARDINAL; VAR RecName: ARRAY OF CHAR ):
                     BOOLEAN;
```

The `NdxBones.Retrieve` procedure provides a fast way to access a record. It is exported as:

```
PROCEDURE Retrieve( NdxFile: NdxFileType; RecordName:
                   ARRAY OF CHAR; BufAddr: ADDRESS; VAR
                   RecSize: CARDINAL): BOOLEAN;
```

`Retrieve` transfers the entire record, field separators and all, to the address identified by `BufAddr`. `BufAddr` must already be allocated; passing an uninitialized address as `Retrieve`'s `BufAddr` parameter is an excellent way to crash your machine. It returns `TRUE` if the record was found and retrieved.

`Retrieve` is useful primarily for files that have no field name structure. If you want to read an entire record from a file that does have a field name structure, it is more convenient to use `GetField` and pass it 'RECORD' as the field name. This retrieves the record as a list of all its elements.



#### 4.2.4.2. High-Level File Searching

*Repertoire's* AltKeys module provides a way to search a file for records meeting user-defined requirements that are passed to the procedure LoadAltList as a list. LoadAltList produces another list that can be used as an alternate way of accessing records in an NdxFile.

We begin with an example. Not all of it will be clear until you have read the explanation that follows it, but it will make the explanation easier to understand. For purposes of simplicity, the example uses a hard-coded search. In other words, it shows how to write and compile a module that searches for records meeting a specific set of requirements. For most purposes, however, you will actually want to let users define their search requirements interactively. The PMIDBMS.MOD example program shows how you might allow them to do so.

Suppose you want to search an NdxFile for all records that have the words 'fabric' and 'silk' in their 'ItemsOrdered' fields. These are the steps you would use:

1. Declare a GenList variable with a name suggesting that it contains a list of requirements. We use 'CriteriaList' in the example below. Declare another GenList to hold the list of record names that satisfy your search (we call it 'MatchingRecords' below). Also declare an NdxFileType variable to identify your NdxFile.
2. Initialize the CriteriaList with NewList (imported from GenLists). Then call AddCriteria (imported from AltKeys) with the following parameters:

```
AddCriteria( CriteriaList, or,
              StrSelector, " 'fabric' & 'silk' ",
              'ItemsOrdered' );
```

3. Now call LoadAltList with these parameters:

```
IF LoadAltList( NdxFile, NdxFile^.Ndx,
               'KEY', CriteriaList ) THEN
END;
```

Using the reserved word 'KEY' as the field name tells LoadAltList to create a list of matching record names rather than field values.

4. Now retrieve your list of matching record names by calling GetAltList with these parameters:

```
IF GetAltList( NdxFile, 'KEY',
              MatchingRecords ) THEN
END;
```

5. MatchingRecords now contains a list of record names that have the

## Indexed File Subsystem

words 'fabric' and 'silk' in their 'ItemsOrdered' fields (if it contains nothing, no records satisfied the search).

6. You could now use the Scrn2Ndx module to display the matching records in some format you've defined in a DisplayFile. See section 4.4.1 and the PMIDBMS.MOD example on the distribution disk. You could also use the GetField routine from the NdxFiles module to retrieve and display specific fields of the matching records. For purposes of simplicity, however, the example below merely writes the record names to the screen and terminates.
7. Finally, you may choose either to dispose of the list that has been created, or store it in this or another NdxFile to speed up later searches. The BuildLst module in the PMIDBMS example program on your distribution disk shows how that can be done. The fact that you can store RecNameLists and AltKeyLists as normal lists in an NdxFile is what gives the AltKeys module its name--these lists become alternate ways of accessing NdxFile records. You can store as many of them as your disk space allows.

This is how the steps outlined above would actually be coded:

```
MODULE SearchExample;
```

```
FROM AltKeys IMPORT AddCriteria, Criterion,  
    GetAltList,  
    LoadAltList, LogicOperator, StrSelector;  
FROM ErrorManager IMPORT WARN;  
FROM GenLists IMPORT DisposeList, GenList,  
    GetElmt, ListLength, NewList;  
FROM NdxBones IMPORT OpenNdxFile;  
FROM NdxFiles IMPORT CloseNdxFile;  
FROM NdxTypes IMPORT NdxFileType;  
FROM StringIO IMPORT outp, WriteEol;  
IMPORT InitCompilerMods;
```

```
VAR
```

```
    CriteriaList, MatchingRecords: GenList;  
    requirements: Criterion;  
    NdxFile: NdxFileType;  
    RecName: ARRAY [0..79] OF CHAR;  
    count, TypeCode: CARDINAL;
```

```
BEGIN
```

```
    IF NOT OpenNdxFile( NdxFile,  
        'OrderFil.Ndx') THEN  
        WARN('File "OrderFil.Ndx" not found.');
```

```
    END;  
    NewList( CriteriaList);  
    AddCriteria( CriteriaList, or, StrSelector,  
        " 'fabric' & 'silk' ", 'ItemsOrdered');
```

```
    IF NOT LoadAltList( NdxFile, NdxFile^.Ndx,  
        'KEY', CriteriaList) THEN  
        WARN('Error in file search.');
```



```

END;
IF GetAltList( NdxFile, 'KEY',
    MatchingRecords) THEN
    FOR count := 1 TO
        ListLength(MatchingRecords) DO
        GetElmt( MatchingRecords, count, RecName,
            TypeCode);
        WriteEol( outp, RecName);
    END;
END;
DisposeList( CriteriaList);
CloseNdxFile( NdxFile);
END SearchExample.

```

To make a routine like the example above truly useful, however, you must generalize it by using variables instead of constants in the strings passed to LoadAltList and AddCriteria, and you must allow those variables to be passed to the procedure in its parameter list. A few simple modifications of the code in SearchExample turn it into a much more powerful tool:

```

PROCEDURE SearchFile( NdxFile: NdxFileType;
    FileName, rule: ARRAY OF CHAR;
    VAR MatchingRecords: GenList);
VAR
    CriteriaList: GenList;
    RecName: ARRAY [0..79] OF CHAR;
    count, TypeCode: CARDINAL;
BEGIN
    NewList( CriteriaList );
    AddCriteria( CriteriaList, or, StrSelector,
        rule, FileName );
    IF NOT LoadAltList( NdxFile, NdxFile^.Ndx,
        FileName, CriteriaList ) THEN
        WARN('Error in file search.');
```

```

    END;
    IF GetAltList( NdxFile, FileName,
        MatchingRecords ) THEN
    END;
    DisposeList( CriteriaList );
END SearchFile;

```

The central routine in the AltKeys module is LoadAltList. We will explain it after describing the data upon which it operates.

```

PROCEDURE LoadAltList( VAR NdxFile: NdxFileType; In-
    list: GenList; FileName: ARRAY OF CHAR;
    Criteria: GenList): BOOLEAN;

```

Repertoire's AltKeys module distinguishes between four types of lists (all of which are actually GenLists):

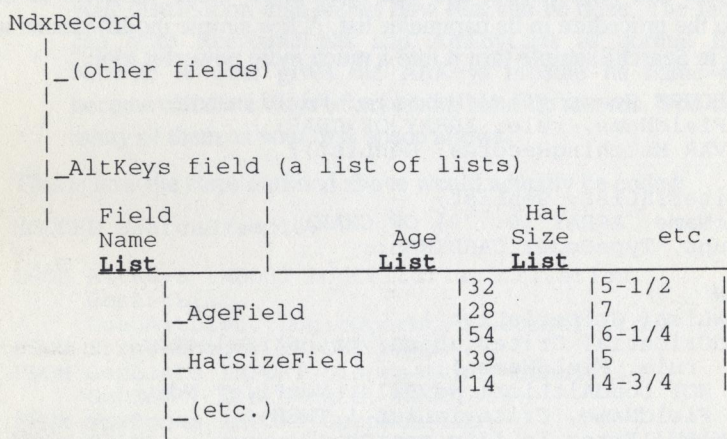
1. NdxLists, which contain elements of type NdxElement (exported from NdxTypes).
2. AltKeyLists, which contain the values in different records of a field

## Indexed File Subsystem

that you specify, as well as the number of the record from which it was read. You can pass this number to the `NdxUtils.GetRecName` procedure to get back the full record name for the field, and you can read the field values without further disk accesses.

3. `CriteriaLists`, which contain the rules that `LoadAltList` uses to determine whether a record matches your requirements.
4. `RecNameLists`, which contain only record names--no record numbers and no field values.

An `AltKeyList` is a member of the `NdxRecord.AltKeys` list. This is a diagram of how the `AltKeys` list is structured:



Each element of a `CriteriaList` is a "Criterion" record. This is what goes into a `Criterion` record:

```
TYPE
  LogicOperator = (or, and, AndNot);

  SelectProc = PROCEDURE(ADDRESS, CARDINAL,
    CARDINAL, ARRAY OF CHAR, VAR BOOLEAN):
    BOOLEAN;

  Criterion = RECORD
    operator: LogicOperator;
    rule, field: ARRAY [0..127] OF CHAR;
    determiner: SelectProc;
  END;
```

`LoadAltList` begins by adding the `FieldName` in its first parameter to the `FieldName` list (it first disposes of any existing identical field names and corresponding lists). Then it retrieves every record in its `InList`, and passes to the first "determiner" procedure variable the value of the field in that re-

106



cord designated by `FieldName`. It also passes to the determiner the "rule" string and the "field" string. The latter contains the field name, and the former generally contains some sort of expression. The determiner tells `LoadAltList` whether this field is true or false. `LoadAltList` then uses the operator field of the criterion record to combine the result of this field evaluation with the results of other field evaluations it has done on this record. When `LoadAltList` reaches the end of the `CriteriaList`, it knows whether the running result for this record is true or false. If false, it throws the record away without adding anything to the `AltList`. If true, it appends the value of the field designated in its third parameter to the corresponding `AltList`. Then it retrieves the next record in its `InList`, starts over at the beginning of its `CriteriaList`, and goes through the same process to determine whether the record meets your requirements.

The procedure `AddCriteria` helps you create `CriteriaLists` conveniently. It is exported as:

```
PROCEDURE AddCriteria( CriteriaList: GenList; op:
                      LogicOperator; TheSelectProc: SelectProc;
                      TheRule, FieldName: ARRAY OF CHAR );
```

Pass it an initialized `GenList`, an operator, a procedure variable of type `SelectProc` (you can use `AltKeys.StrSelector` if you'd rather not define your own), a string that may optionally provide information to the `SelectProc` variable, and the `FieldName` that the `SelectProc` should examine.

We should now return to the example presented at the first of this section and note several points that it glosses over:

1. It initializes `CriteriaList` before passing it to any `AltKey` procedures.
2. It specifies a search of the entire `NdxFile` because it passes the `NdxFile`'s entire `.Ndx` list to `LoadAltList`'s second parameter. It could have done the same thing by passing it a null list (initialized but empty), or by passing it a list of all the record names in the file. It could also have searched a smaller portion of the file by passing either a subset of `NdxFiles^.Ndx` or a shorter list of record names to `LoadAltList`.
3. This example causes `LoadAltList` to search only one field of each record it retrieves because its `CriteriaList` has only one element. Adding elements to the `CriteriaList`, however, allows you to search multiple fields in each record retrieved. The `LogicOperator` passed to `AddCriteria` determines how many of the criteria must be met before `LoadAltList` will add the record name to the 'KEY' `AltList`.

The `AltKeys` module provides additional facilities, not explained here, that allow you to construct much more powerful and flexible file-searching and



## Indexed File Subsystem

analysis routines. Details about how the additional functions work can be found in the Repertoire reference manual.

### 4.2.5. Safety

*Repertoire's* indexed file system keeps a linked "index" list in memory for each open NdxFile. This index lists each record's name, size, and position in the file. When a record is requested, it is located in the index and read into a memory buffer area. When records are added, changed, or deleted, any new record data is immediately written to the file (or at least, to the DOS buffer associated with the file) and the in-memory index is updated. However, the index is not immediately written to the file. The index is normally saved only when the file is closed.

Not saving the index every time the file is changed allows *Repertoire* to handle file updates more rapidly, but it creates a danger: if the machine is turned off before the file is closed, the index is lost. *Repertoire* provides two facilities to protect against this:

1. You may call a routine named *SafetyOn* which will cause *Repertoire* henceforth to save the file index after every file change. When *Safety* is on, *Repertoire* forces DOS to do a physical disk update by duplicating the file handle and immediately closing it. The routine *SafetyOff* can be called subsequently to turn this safety feature off again. Programs can turn safety on and off repeatedly to update the index only occasionally. This is how the routines are declared:

```
PROCEDURE SafetyOn(VAR NdxFile: NdxFileType);
```

```
PROCEDURE SafetyOff(VAR NdxFile: NdxFileType);
```

2. If *Repertoire* opens an index file that was updated and not properly closed, it rebuilds the index from the data record information in the file. Thus, if the index is lost, it will take longer to start the next program that uses the file. However, the file index will be rebuilt without lost data. The routine that rebuilds the index is called *RebuildNdx*, in the *NdxFiles* module.

## 4.3. An Example

The following example is quite basic: it opens a file with three fields (it creates the file if it does not already exist) and allows multiple records to be added, prompting for each field.

```
MODULE AddRec;
```

```
FROM GenLists IMPORT StrCode;  
FROM M2Strings IMPORT Length;  
FROM NdxBones IMPORT OpenNdxFile;
```



```

FROM NdxFiles IMPORT CloseNdxFile, CreateNdxFile,
    GetField, PutField, WriteRecord;
FROM NdxTypes IMPORT NdxFileType;
FROM StrEdit IMPORT AssignStr;
FROM StringIO IMPORT inp, outp, ReadStr,
    WriteEol, WriteStr;
IMPORT InitCompilerMods;

```

```

VAR

```

```

    TheFile: NdxFileType;
    RecName, FileName,
        TheValue: ARRAY [0..80] OF CHAR;
    TheType: CARDINAL;

```

```

PROCEDURE DoRecord( FieldName: ARRAY OF CHAR);

```

```

BEGIN

```

```

    IF GetField( TheFile, RecName,
        FieldName, TheType, TheValue)
        AND ( Length( TheValue) # 0) THEN
        (* if record exists and field not empty,
            display it *)
        WriteStr( outp, "Current ");
        WriteStr( outp, FieldName);
        WriteStr( outp, " is: ");
        WriteEol( outp, TheValue);

```

```

    ELSE

```

```

        (* record does not exist or the field is
            empty *)

```

```

        WriteStr( outp, 'No ');
        WriteStr( outp, FieldName);
        WriteEol( outp, ' currently exists.');
```

```

    END;

```

```

    WriteStr( outp, "Enter New ");

```

```

    WriteStr( outp, FieldName);

```

```

    WriteStr( outp, ": ");

```

```

    ReadStr( inp, TheValue);

```

```

    (* prompt for and read new value for this
        field *)

```

```

    IF NOT PutField( TheFile, RecName,
        FieldName,

```

```

        StrCode, TheValue) THEN

```

```

        WriteEol( outp, 'Cannot put field');
```

```

    END;

```

```

END DoRecord;

```

```

BEGIN

```

```

    AssignStr( 'testfile.ndx', FileName);

```

```

    IF NOT OpenNdxFile( TheFile, FileName) THEN

```

```

        (*If can't open file, create it *)

```

```

        CreateNdxFile( TheFile, FileName,

```

```

            "Name, Age, PctBodyFat");

```

```

    END;

```

```

    REPEAT

```

```

        (*repeat the following for multiple records*)

```

```

        WriteStr( outp, "Enter ID Number: ");

```

```

        ReadStr( inp, RecName);

```

```

        (* prompt for and read record name *)

```

```

        DoRecord( 'Name');
```

## Indexed File Subsystem

```
DoRecord( 'Age');
DoRecord( 'PctBodyFat');
(* accept input for the three fields *)
IF NOT WriteRecord( TheFile, RecName) THEN
  WriteEol( outp, 'Cannot Write Record');
END;
WriteStr( outp, "Another Record? (y,n): ");
ReadStr( inp, RecName);
(* prompt for desire to continue *)
UNTIL (Length(RecName) = 0) OR
      (RecName[0] # 'y');
CloseNdxFile( TheFile);
END AddRec.
```

You may compile and link the program to see how it appears to the user. The program will prompt for an ID to use as the next record name. Then, for each of the three fields in the record, any existing value is displayed and a new value is requested. This process is repeated for as many records as the user wishes to add. Using the various NdxFiles procedures, many embellishments could be added. A special message could be printed if a new record is being added, the file structure could be read first and parsed so that the program prompts for all of the fields in any file, record display and deletion could be added, etc.

## 4.4. Advanced Topics

### 4.4.1. Using NdxFiles With the Screen System

In the module *Scrn2Ndx*, *Repertoire* provides several simple routines that facilitate communication between the screen system and the file system. They provide a way to write a record from a screen to an NdxFile with a single procedure call, and to fill a screen with data from a record using a single procedure call. The routines are named *PutScrRec* and *FillScr*. They are included primarily as examples of how you might want to establish communications between the two subsystems. Complicated applications will need to expand on these procedures considerably. Using the routines involves three major steps:

#### 4.4.1.1. Creating the Index File

To use a screen with an Ndx file, the screen and Ndx file must be created with related structures. First, the data entry screen you will use should be created with your editor and laid out as explained in the normal screen system instructions in Chapter 3. Each field that you want to save must be given a name in the "Fields" section of the screen's command area. Then you should write a simple program that calls the routine *CreateNdxFile* to create the Ndx File. The structure you pass to *CreateNdxFile* must be a string of all the names of the fields that you want to save. You need not worry about using consistent capitalization between field names in your



NdxFile and field names in your screens because *Repertoire's* field-name searches are not case-sensitive. (Its record name searches are.)

The screen may contain named fields not present in the file structure, and the file structure can contain fields not on the screen; only fields with matching names will be transferred between the screen and the NdxFile. This allows you to use the same Ndx file with many different screens, as long as all of the screens use the same names for the same fields. For example, the data entry screen may be formatted differently from the data display screen, but the NdxFile data can still be taken from the data entry screen, stored in the file, and later retrieved and displayed on the display screen.

#### 4.4.1.2. Saving data from a Screen

The program that stores data entered on a screen in an Ndx file should:

1. Open both the screen and Ndx files;
2. Display the data entry screen (possibly as part of a much larger set of screen interactions);
3. Call ControlFrame after displaying the screen;
4. Call PutScrRec before calling Display again for any screen in the screen file. PutScrRec will transfer the data from the screen fields into the file fields with the same names. PutScrRec is declared as follows:

```
PROCEDURE PutScrRec( VAR NdxFile: NdxFileType; VAR
    ScrFrame: DisplayFrame; VAR RecName: ARRAY
    OF CHAR; KeyField: ARRAY OF CHAR): BOOLEAN;
```

You can indicate the Record name to use for this data in one of two ways:

1. If the record name is not in a single field on the data entry screen, you may pass a string containing the record name to the parameter RecName;
2. If the record name is in a field on the screen, pass that field name in the string KeyField and pass a variable containing a NULL string for RecName.

PutScrRec will always return the record name used in the RecName variable. The function result is TRUE if at least one field was found in common between the Ndx file and the screen, and if the write was successful. If the record already existed, the data in the screen fields overwrites any existing file data. PutScrRec functions like PutField in that it WILL NOT WRITE THE DATA TO THE FILE. The routine WriteRecord must be called for this RecName--after this PutScrRec call, and before any subsequent PutScrRec.

## Indexed File Subsystem

### 4.4.1.3. Filling a Screen with NdxFile Data

A similar routine, FillScr, retrieves data from an NdxFile and displays it on the screen. The calling program should:

1. Open both the screen and Ndx files;
2. Read the data display screen (possibly as part of a much larger set of screen interactions);
3. Call the routine FillScr to transfer the data from the Ndx file into the screen fields with the same names. It is declared as:

```
PROCEDURE FillScr( VAR NdxFile: NdxFileType; VAR Scr-
                  Frame: DisplayFrame; VAR RecName: ARRAY OF
                  CHAR; KeyField: ARRAY OF CHAR): BOOLEAN;
```

As with PutScrRec, RecName must either contain the record name to be retrieved and displayed on this screen, or it must be NULL. If RecName is NULL, the parameter KeyField must contain the name of the field on the screen that is to be used as the record Name. FillScr will always return the record name used in RecName. FillScr returns TRUE if the record was found and at least one field was found in common between the Ndx file and the screen.

4. Either display the frame that you've read and loaded with FillScr, or (if it is a frame you've already displayed), call ControlFrame, WITH ITS LAST PARAMETER SET TO TRUE. This will display on the screen the data that has been retrieved from the file.

Of course, both files should be closed before the program exits.

### 4.4.1.4. Example

You may find the example provided on the distribution diskette (PMIDBMS.MOD) too advanced to serve as an introduction. The material below glosses over most of the fine points, but should help you gain an initial understanding of how to set up programs that use both the screen system and the file system.

Suppose your program uses only two screens:

```
(* Top of file DEMO1.EDT *)
=====
:NameScrn
  ParentFrame: 0
    (*The EndCode; means backing up from
      this screen ends the frame sequence.*)
  NormalNext: EntryScreen
  Fields:{
    () 'RecName' string; required
  }
-----
```



Enter a record name:

#

#

```
=====
:EntryScreen
  ParentFrame: NameScrn
  NormalNext: none
  Fields:{
    () 'RecName' Display
    () 'age' integer [0..149]
    () 'job' string; required
  }
-----
```

Record Name: #

#

Age: # #

Occupation: #

#

(\* Bottom of file DEMO1.EDT \*)

As these screens suggest, our example program will only store, retrieve, and display a very simple kind of record. Now, assuming that you have used ScrnComp to turn DEMO1.EDT (above) into a displayable file named DEMO1.DSP, you can examine the demonstration by compiling, linking, and executing the program below:

MODULE Demo1;

```
IMPORT ControlUtils;
IMPORT DspFiles;
IMPORT ErrorManager;
IMPORT FramePainter;
IMPORT InputManager;
IMPORT NdxBones;
IMPORT NdxFiles;
IMPORT NdxTypes;
IMPORT PosUtils;
IMPORT Scrn2Ndx;
IMPORT ScrnTypes;
IMPORT ScrnUtl1;
IMPORT ScrnUtl2;
IMPORT VWindows;
IMPORT WindowPrims;
IMPORT InitCompilerMods;
```

VAR

```
  NdxFile: NdxTypes.NdxFileType;
  FrameVar: ScrnTypes.DisplayFrame;
  ScreenFile: ScrnTypes.DisplayFile;
  NextFrame: ScrnTypes.AFrameName;
  RecName: ARRAY [0..19] OF CHAR;
  SavedHeight: CARDINAL;
```

## Indexed File Subsystem

```
BEGIN
  SavedHeight := VWindows.GetCursorHeight(
    VWindows.CurrentWindow );
  IF NOT NdxBones.OpenNdxFile( NdxFile, 'demo1.ndx' )
    THEN
    (*If the file doesn't exist, create it.*)
    NdxFiles.CreateNdxFile( NdxFile, 'demo1.ndx',
      'RecName, age, job' );
  END;
  IF NOT DspFiles.OpenDisplayFile( ScreenFile,
    'demo1.dsp' ) THEN
    ErrorManager.WARN( 'Could not open DEMO1.DSP' );
    RETURN;
  END;
  ScrnTypes.InitDisplayFrame( FrameVar,
    VWindows.CurrentWindow );
  REPEAT
    NextFrame := 'NameScrn';
    ControlUtils.ReadShowAndCntrl( ScreenFile, FrameVar,
      NextFrame );
    (*This will always display the screen that says
      "Enter a record name."*)
    IF NOT PosUtils.Equal( NextFrame,
      ScrnTypes.ProgramExit ) THEN
      ScrnUtl1.GetFieldText( FrameVar,
        FrameVar^.CurrentField, RecName );
      DspFiles.ReadDisplayFrame( ScreenFile, FrameVar,
        'EntryScreen' );
      IF NOT Scrn2Ndx.FillScr( NdxFile, FrameVar,
        RecName, '' ) THEN
        ControlUtils.ChangeField( FrameVar, RecName,
          'RecName', TRUE );
      END;
      FramePainter.ShowDisplayFrame( FrameVar, 0, 0, 0,
        0 );
      InputManager.ControlFrame( FrameVar, 0, '', FALSE,
        NextFrame );
      IF PosUtils.Equal( NextFrame, FrameVar^.normlnext
        ) THEN
        IF NOT Scrn2Ndx.PutScrRec( NdxFile, FrameVar,
          RecName, '' )
          THEN HALT(); END;
        IF NOT NdxFiles.WriteRecord( NdxFile, RecName )
          THEN HALT(); END;
      END;
    END;
  UNTIL PosUtils.Equal( NextFrame, ScrnTypes.ProgramExit
    );
  ScrnUtl2.CloseDisplayFrame( FrameVar );
  NdxFiles.CloseNdxFile( NdxFile );
  VWindows.ClearPart( VWindows.CurrentWindow, 0, 0,
    VWindows.EndCol(VWindows.CurrentWindow),
    VWindows.EndRow(VWindows.CurrentWindow) );
  VWindows.SetCursorHeight( VWindows.CurrentWindow,
    SavedHeight );
END Demo1.
```



## Beginner's Note:

The most common problem people first encounter in using the `CreateNdxFile` procedure is that very long Structure strings (i.e., the string with all the field names--the procedure's last parameter) will not fit on a single line, because most compilers have some limit on the length of lines in the source text. If you have exceeded the limit, the compiler will complain that your "String constant is not terminated on this line," or something to that effect. If you're not sure what to do about this, you need to study the *Repertoire Reference Manual's* explanation of the procedures in the `StrEdit` module. The quick explanation, however, is that you must declare a string variable for the Structure string and then use `StrEdit's Append` procedure enough times to add all necessary field names to that variable. For example:

```
VAR
  Str1 : ARRAY [0..511] OF CHAR;
BEGIN
  ...
  IF NOT OpenNdxFile( NdxFile, 'demo1.ndx' ) THEN
    SetLength( Str1, 0 );
    Append(Str1, 'name, age, hatsize, job, ');
    Append(Str1, 'shoesize, height, weight, ' );
    Append(Str1, 'address {street, city, ' );
    Append(Str1, 'state, zip}, AcctNum, ' );
    Append(Str1, 'AcctBalance, StartDate' );
    CreateNdxFile(NdxFile, 'demo1.ndx', Str1 );
  END;
  ...
```

#### 4.4.2. Space Usage: Crunching Files

When a record is deleted from an `NdxFile`, a "hole" is created in the file. Holes may also be caused by records that grow larger than the space allocated to them; they must be moved to some other position in the file--most likely, the end. When a new record is created, it is written into one of these holes, if any of them are sufficiently large. Since an `NdxFile` stores each of its records in only as much space as it requires, the new record will not completely fill the hole, and a smaller amount of unused space will be left. Thus, a much-used file will normally be less than 100% used. Files that have had a great deal of deletion and change activity will show the lowest "percent used" measures. *Repertoire* provides two routines to manage this garbage space. One measures the amount of garbage in the file, and the second `Crunches` the file to remove the garbage.

The routine `SizeInfo` returns the following information for the `NdxFileType` variable passed in its parameter list:

1. the total file size in bytes;

## Indexed File Subsystem

2. the number of bytes that could be saved with a Crunch;
3. the percent of the file that is not garbage;
4. the total number of current data records, not including garbage records; and
5. the current number of garbage records.

Thus, the routine is useful for more than determining whether a file needs to be Crunched; it reports the size of the file for any purpose that the programmer requires. (The file size returned by this routine reflects the file size that would result if the file were closed at this point. If safety is not on, this may not match the actual size of the file in DOS at this time because *Repertoire's* index in memory may not match what was last saved in the file.) The file must be open to use `SizeInfo`. This is the routine's declaration:

```
PROCEDURE SizeInfo( VAR NdxFile: NdxFileType; VAR
    TotlSize, GbgSize: LongInt; VAR PctUsed,
    NumDataRecs, NumGbgRecs: CARDINAL );
```

The procedure `CrunchNdxFile` compacts a file so that no garbage space remains. Normally, a program will call `SizeInfo` only rarely and will call `CrunchNdxFile` only if the amount of garbage is significant. The exact point at which a Crunch is needed depends on your performance requirements and on the storage capacity of the disk. You can determine the amount of free space on the disk with the routine `Directory.FreeDiskSpace`. See the `Directory` section of the *Repertoire Reference Manual*.

This is how the routine is declared:

```
PROCEDURE CrunchNdxFile( VAR NdxFile: NdxFileType );
```

`CrunchNdxFile` crunches in place--it moves records around within the file and then truncates--so that you no longer need any free disk space to crunch an `NdxFile`.

### 4.4.3. Rebuilding Damaged Files

*Repertoire* uses redundant systems to determine whether an `NdxFile` has been damaged. When it opens the file, it reads a four-byte value from the very end of the file. That value is supposed to tell it the location of a unique set of characters in the file that marks the beginning of the `Ndx`. *Repertoire* calls this pattern of characters the `NdxMarker`. *Repertoire* then moves the file pointer to that location and determines whether the `NdxMarker` is actually there. If it is not, *Repertoire* concludes that the file has been damaged.

Even if the `NdxMarker` is in the right place, the `ndx` could still have been



damaged by a power failure between a physical record-write and a physical Ndx-write. Thus, before *Repertoire* writes a record to a file that has its safety feature turned on, it marks the file's zero-record (which contains the file's structure) as garbage. When *Repertoire* updates the ndx, it marks the zero-record as "not garbage." *Repertoire* forces DOS to update the disk physically by making a duplicate handle for the file and then closing it.

If an NdxFile's safety feature has been turned off, *Repertoire* does not force physical writes, and physical writes will therefore occur later than logical writes because DOS buffers its file I/O.

#### 4.4.4. Error Checking

*Repertoire*'s indexed file system and its generic list handling system are closely related in their handling of errors. A large percentage of the errors possible in handling an NdxFile will actually take place in the GenLists module. Both modules describe their errors to the ErrorManager.WARN procedure variable with English sentences. These are examples of the strings passed:

```
'Empty name found in NdxFile structure
  definition.'
```

```
'Closing brace missing in NdxFile structure
  definition.'
```

```
'Comma missing after "]" in NdxFile structure
  definition.'
```

```
'Not enough room to make the crunched copy.'
```

*Repertoire* reports DOS errors through the PrintMessage routine in StringIO, which calls Abort. The reason is that DOS errors tend to be too serious for reasonable attempts at recovery, but you may want to modify the indexed file system to make it try.

#### 4.4.5. Type checking

It sometimes appears that using NdxFiles requires you to sacrifice much of Modula-2's compile-time type checking. That is true only in the sense that the language's compile-time type checking can't help you with some of the functions that *Repertoire*'s NdxFile system adds to Modula-2. It does not, however, reduce your type-checking support when it is used for more mundane tasks (storing and retrieving fixed-length records of a single type, for example). No language guarantees when you open a file that the type of data stored in the file matches the type of variables into which you read the data. NdxFiles does nothing to make that problem worse, and does one thing that makes it easier to manage.

*Repertoire* allows run-time type checking with a system of "type codes"



## Indexed File Subsystem

used in coordination with the type codes used in *GenLists*. You could designate the number 1, for example, as the type code for real numbers. Pass that number to *PutField* when you insert a real number into a record, and you will get that number back as the type code for the field when the record is read. You can check it before moving the field value into a *REAL* variable. As noted above, *Repertoire* cares only about a few type codes--*NdxTypeCode*, *StrCode*, *ListCode*, and a few others exported from the *AltKeys* module--which leaves about 65500 type codes for your own use. *Repertoire* numbers its own type codes backwards from 65535. We suggest that you consider type codes above 65000 reserved for *Repertoire's* use. Although for purposes of readability and security we would prefer to use enumerated types, *Repertoire* can't do that and still allow you to define your own.

### 4.4.6. Lists and Sublists

Remember that *Repertoire* will check to make sure that data passed to a list field is indeed a list. Because any element can actually be a list, however, you need not designate lists in the structure record if you find *Repertoire's* type checking inconvenient. For more explanation of *Repertoire's* list-handling system, please refer to Chapter 2.

*Repertoire* gives you a pointer directly into the *NdxFile's* *ListBuf* when you use *GetField* on a field that is a list. Conversely, when you put a list into a record using *PutField*, *Repertoire* does not make a copy of the list before putting it into the *ListBuf*--it absorbs that list directly into the *ListBuf*. (The same rule holds in the screen system, where lists passed into and out of editor fields are absorbed, not copied.) This makes the system a great deal faster, but it requires attention to the problem of which subsystem is in charge of deallocating any given list. In general, when you get a list from either the file system or the screen system (using *GetField* or *ListFromEdField*, respectively) you should use the *GenLists.CopyList* routine to make a true copy of that list before passing it to any other *Repertoire* subsystem and before doing anything (such as retrieving another record or displaying another window) that will cause the screen system or the file system to dispose of the list buffers which actually contain the list you have retrieved.

Failure to use *CopyList* when transferring lists between the screen system and the file system will cause one system or the other to halt with an "Uninitialized list" message or a "Premature disposal of child list" message if the list transferred has any sublists. If the list does not have sublists, it will corrupt the *Storage* module and cause your program to behave in unpredictable (but certainly unpleasant) ways.

The two most commonly made errors in using a *Repertoire NdxFile* to store lists and sublists are both related to the fact that lists are absorbed into the *NdxFile's* *ListBuffer* after being passed to *PutField*. New users tend to for-



get that a list is no longer valid after it is passed to PutField, and they tend (after gaining some experience with GenLists) to want to dispose of lists after passing them to PutField. Both will cause your program to misbehave.

Consider the following incorrect example:

```
MODULE Wrong1;

FROM GenLists IMPORT NewList, ListInsert,
    StrCode, ListCode, GenList;
FROM NdxFiles IMPORT CreateNdxFile, WriteRecord,
    PutField, CloseNdxFile;
FROM NdxBones IMPORT OpenNdxFile;
FROM NdxTypes IMPORT NdxFileType;
IMPORT InitCompilerMods;

VAR
    NdxFile : NdxFileType;
    MyList   : GenList;
    s        : ARRAY [0..10] OF CHAR;
    e        : BOOLEAN;
BEGIN
    s := 'Dummy Data';
    NewList(MyList);
    ListInsert(s, StrCode, MyList, 1);

    CreateNdxFile (NdxFile, 'dummy.ndx', '');
    e := PutField(NdxFile, 'A', 'RECORD', ListCode,
        MyList);
    e := WriteRecord (NdxFile, 'A');
    e := PutField(NdxFile, 'B', 'RECORD', ListCode,
        MyList);
    e := WriteRecord (NdxFile, 'B');
    e := PutField(NdxFile, 'C', 'RECORD', ListCode,
        MyList);
    e := WriteRecord (NdxFile, 'C');
    CloseNdxFile (NdxFile);

    NewList(MyList);
    ListInsert(s, StrCode, MyList, 1);
    ListInsert(s, StrCode, MyList, 1);
    (* make the list longer *)
    e := OpenNdxFile(NdxFile, 'dummy.ndx');
    e := PutField(NdxFile, 'A', 'RECORD', ListCode,
        MyList);
    e := WriteRecord (NdxFile, 'A');
    e := PutField(NdxFile, 'B', 'RECORD', ListCode,
        MyList);
    e := WriteRecord (NdxFile, 'B');
    e := PutField(NdxFile, 'C', 'RECORD', ListCode,
        MyList);
    e := WriteRecord (NdxFile, 'C');
    CloseNdxFile (NdxFile);
END Wrong1.
```

This program contains the first sort of error discussed above--it forgets that MyList is no longer valid after the calls to PutField. After you pass MyList

## Indexed File Subsystem

to PutField the first time after initializing MyList with NewList, MyList is absorbed into the NdxFile's ListBuffer. It's no longer safe to do anything with a list that's been passed to PutField. In this case, after the program writes record 'A' to the NdxFile, it passes MyList to PutField twice more before passing it back to NewList. MyList is uninitialized in the second and third calls to PutField, so the GenLists module complains about uninitialized lists in various places.

To be more precise, MyList becomes a sublist of NdxFile's ListBuf the first time you pass it to PutField. In the second call to PutField, PutField (actually InitBuffer) notices that you're changing records, so it disposes of the old ListBuf and creates a new one containing what is or will be the contents of record 'B'. In the process of disposing of the old ListBuf, however, it disposes of MyList.

There is a correct way to do this. Just declare two lists--in this example, MyList and, say, TmpList. Do whatever you want with MyList, but don't pass it to PutField. Instead, use the GenLists.CopyList procedure to copy MyList into TmpList before each PutField. Then pass TmpList to PutField. And remember not to dispose of TmpList; once you've passed it to PutField, the NdxFile system is responsible for disposing of it.

### 4.4.7. Changing File Structures

*Repertoire* does not presently provide a convenient way to change an NdxFile's structure list (i.e., the list of field names that it uses to allow retrieval of fields by name) without recopying the file's data. In theory, you need only retrieve the NdxFile's zero-record, change it, store it back in the first record of the file, update the first two bytes of the file to reflect the new length of the structure record, close the file, and then reopen it. In practice, that approach would require you to worry about whether the structure record could be lengthened (i.e., whether the next record was a garbage record), moving the next record to some other location if the next record was not garbage, and ensuring that the structure record was restored correctly.

*Repertoire* will eventually provide a routine that will conveniently accomplish all that, but for now we suggest that you create a new file with the desired structure, and then retrieve records and fields sequentially from the old file and write them into the appropriate records and fields of the new one. This shouldn't be too much of a burden, since an NdxFile's structure should very seldom need to be changed.

### 4.4.8. Adding NdxFile Features

There are several things you may want to do with *Repertoire*'s NdxFile system if you have the full-source version. You might, for example, want to



implement multiple-record buffering to make it possible to access several records simultaneously without intervening calls to `WriteRecord` or physical record-retrievals. This shouldn't be hard to do if you find single-record buffering restrictive. The `ListBuf` subfield of the `NdxFileType` record could be treated as a list of record buffers, each identified by a record name in the first sublist of the list. (*Repertoire's* `AltKeys` module uses a similar technique to identify fields for which alternate key lists have been loaded; see section 4.2.4.1.) `PutField`, `GetField`, and `WriteRecord` could then be modified to make them (1) look through the list of record names presently associated with one of the list buffers, and (2) use the corresponding child-list of the `ListBuf` in the same way that these routines now use the entire `ListBuf`. `GetField` and `PutField` could allocate new children of the `ListBuf` when they received record names not presently in a list buffer; `WriteRecord` could deallocate children of the `ListBuf` and delete record names from the first child-list of the `ListBuf`.

*Repertoire* does not implement multiple-record buffering because it involves at least two-tradeoffs: (1) it adds another level of indirection to every record-access, thus reducing performance and increasing the complexity of the system; and (2) it makes buffer-deallocation more complicated; the system must either have a fixed number of buffers or it must impose on the user responsibility for deallocating buffers explicitly--deallocating them with `WriteRecord` alone would not be sufficient because not all records accessed will eventually be written.

#### 4.4.9. Record and File Size Limits

The maximum number of records one could theoretically put in an `NdxFile` is 65,535. The reason is that the `NdxBones` module stores the file's `NdxList` in a `GenList`, and the `GenLists` module numbers the elements of a `GenList` with `CARDINALS`. That might not be hard to change--you could make `GenLists` number its elements with `LONGINTs`--but other more practical limits on the size of an `NdxFile` would have to be dealt with before changing the maximum length of a `GenList` would help.

The most important limit is that most MS-DOS machines still do not have enough memory to allow more than about 10,000 records per `NdxFile`. To compute the actual maximum number of records that can be handled on a given machine, divide the amount of memory available to your program (i.e., total system memory minus DOS and the size of your program and any resident programs) by the value you've chosen for `RecNameLength` (a constant in the `NdxLists` module) plus 24 bytes per record consumed by `GenList` records and the `LongInts` and `CARDINALS` that identify each record's size, position, allocated space, etc.

To use the file system in situations where more records are needed, you



## Indexed File Subsystem

should split your data up into multiple NdxFiles and then open and close them as needed. Even large NdxFiles can be opened and closed very quickly.

You might also devise a way to treat records as fields within records. Record 1, for example, could contain all records with names beginning "ABB..." through "ACU..." Contents of those "records" (now fields within records) could be stored as structured lists associated with the record names. Because each record in an NdxFile may contain up to 64K bytes and up to about 16,000 fields and subfields, a strategy like this would allow storage of an essentially unlimited amount of data, but it depends on your ability to accept a limitation of about 10,000 top-level entry points into the database.

Finally, you might consider using Expanded Memory (Lotus/Intel/Microsoft EMS memory) in conjunction with EmsStorage. When EmsStorage is linked in, all GenLists are automatically allocated from expanded memory. NdxFiles with 65,535 records might be manageable with a sufficient amount of expanded memory, but we have never encountered NdxFiles with more than 25,000 records. *Repertoire's* NdxFiles really aren't designed for applications that will need more records than that.

## 4.5. How the File System Works

When a record or a field is requested, the NdxBones module determines the position and size of the record in the file, and NdxBones (which provides the most skeletal of the indexed-file system's services) reads the entire record in a single operation. Then it does a fast byte scan to find a special set of characters that delimit every element of the file, and uses those characters to transform the record into a list (if the record has been given a list structure). One set of the special characters marks the beginning of an element, and the other set marks the end; two BeginMarkers together indicate the beginning of a deeper level of nesting, and two EndMarkers together indicate a return to a higher level. When a record is written, *Repertoire* encodes (and decodes when the data is read) any occurrences of the NdxFile code character in the user's file data to guarantee the uniqueness of these special delimiters.

The following description of the record to which an NdxFileType variable points should fill in the details.

```
NdxRecord =  
  RECORD  
    name : ARRAY [0..75] OF CHAR;
```

This field allows your programs to determine the file's name and its loca-



tion in the disk's directory structure. It allows your programs to be intelligent about disk management problems. When you run out of space, for example, you can tell the user which file needs to be copied onto a larger device, or you can close the file (to save memory), reopen it with the `HandleIO.OpenFile` procedure (to avoid building an `Ndx`), and copy the file from within the program using the `Drectory.CopyFile` procedure.

```
handle: CARDINAL;
```

This is how DOS refers to your `NdxFile`. You can use the handle to manipulate an `NdxFile` directly using routines in `StringIO`, `HandleIO`, `Drectory`, etc.

```
SafetyOn: BOOLEAN;
```

Indicates whether a new copy of the `ndx` is to be written at the end of the file after every change.

```
InitCheck: CARDINAL;
```

All operations on an `NdxFile` check this first to make sure it contains the private value given it by `OpenNdxFile`.

```
NdxFilePtr: LongInt;
```

`NdxFilePtr` is the position in the file of the `NdxMarker` after the last data record. The first byte in the file is at offset 0, not offset 1.

```
UpdateNeeded: BOOLEAN;
```

*Repertoire* looks at this variable when it is asked to close the file. If it is true, it writes the `Ndx`. If it is false, it knows the `Ndx` is current.

```
BufRecName: ARRAY [0..RecNameLength] OF CHAR;
```

`BufRecName` is the name of the record presently in this file's list buffer.

```
Ndx, AltKeys, ListBuf, StructLst: GenList;
```

These four lists have the following functions. `Ndx` tells the `NdxBones` module where each record in the file is located, and how long it is. `AltKeys` is a list of lists. Its first list contains field names. For each field name in that first list, the `AltKeys` list contains another list of field values. Each element of the list of field values actually contains in its first two bytes the number of the record from which it came. `ListBuf` contains, in list form, data being transferred to or from an `NdxFile`. `StructLst` tells *Repertoire* where to look in the `ListBuf` for fields referred to by name.

```
FileBufPtr : ADDRESS;
```

An `NdxFile` actually has two buffers: one in list form, and another that is an exact image of the record most recently read. `FileBufPtr` tells *Repertoire*

## Indexed File Subsystem

where to find the exact-image buffer. We sometimes call this the "flat" buffer.

BufSizeNow: CARDINAL;

This field tells *Repertoire* the present size of the flat buffer.

### 4.5.1. Byte Map of an NdxFile

We conclude this explanation with a rather cryptic description of the file structure within an NdxFile. This information should very seldom be needed, but is provided for situations in which an NdxFile must be accessed without the support of the modules making up *Repertoire's* NdxFile system.

File has structure:

Zero record has structure:

```
Size of ZeroRec (2 byte CARDINAL)
Startsep (2 bytes: CodeChar + StartChar)
    + RecType(2 bytes)
    Startsep + RecType(2 bytes)
        + File GarbageByte (1 char)
        + StructName + EndSep (2 bytes)
    StartSep + RecType(2 bytes) + RecName + EndSep
    StartSep + RecType(2 bytes) + RecName + EndSep
    StartSep + RecType(2 bytes) + RecName + EndSep
    .
    .
    .
EndSep
```

Then the data records appear.

Each data record is stored as:

```
Startsep + RecType
    Startsep + RecType
        + GarbageByte (1 char)
        + RecName +
    EndSep
    StartSep + RecType + First Field + EndSep
    StartSep + RecType + Second Field + EndSep
    .
    .
    .
EndSep
```

Any data in this area is garbage.

```
Startsep + RecType
Startsep + RecType + GarbageByte + RecName + EndSep
StartSep + RecType + First Field + EndSep
StartSep + RecType + Second Field + EndSep
.
.
.
```



EndSep

If a field is a linked list, it is stored:

```

StartSep  + RecType +
           StartSep + RecType
           + FirstField +
           EndSep
           StartSep + RecType
           + SecondField +
           EndSep
           StartSep + RecType
           + ThirdField +
           EndSep
           .
           .
EndSep
StartSep
.
.
.
EndSep

```

After the last data record, comes the NdxMarker and  
then the NDX:

```

NdxMarker (2 bytes)
INDEX = { RecordName
          [fixed length]
          RecOffset
          [LONGINT]
          RecordSizeAllocated
          [CARDINAL]
          RecordSizeUsed
          [CARDINAL; if 0 , record is garbage]
        }
each record is same length
.
.
.

```

At the end of the file, 4 bytes represent the number of bytes between the NdxMarker and the start of the file.

EOF

An empty, newly created file looks like this:

```

0006H
StartSep
RecType
EndSep
(No data)
NdxMarker
(No index)
last 4 bytes point to NdxMarker

```

## Indexed File Subsystem

*Repertoire* uses a flat read buffer and a linked

list buffer for file I/O. The flat buffer contains

straight record contents as read from the file:

```
Startsep + TypeCode
Startsep + TypeCode + GarbageByte+RecName+EndSep
Startsep + TypeCode + First Field + EndSep
Startsep + TypeCode + Second Field + EndSep
.
.
EndSep
```

*Repertoire* also maintains certain data structures

in memory that are related to the file structure.

ListBuf is a linked GenList of the above without the

Start and End Separators. Contents are:

- \* encoded for CodeChars that occur, and
- \* include rec name

Structure is Genlist of form:

```
Recname1
Recname2
Sublist of Recname2
Recname3
```

More information on using the file system can be gotten from explanations of individual routines in the *Repertoire Reference Manual*.



## 5. The Editor

### 5.1. Overview

*Repertoire's* screen system relies heavily on `TheEditor`, a procedure variable exported from the `VEditor` module. The initialization block of the `UserOps` module assigns to it the procedure `DefaultEditor`, from the `Editor` module. Most of the information necessary to use `TheEditor` as it is employed in the screen system (i.e., as the heart of an editor field) is provided in Chapter 3. This chapter explains how to use `TheEditor` in other contexts.

`TheEditor` allows you to write programs that include simple text entry and editing functions. Your programs can send text to `TheEditor` to be displayed, and can receive from `TheEditor` altered or entered text. You can specify the window in which `TheEditor` will appear, as well as the exact point in the file (and in the window) where the cursor will initially be placed.

These functions are demonstrated, of course, in every editor field in the demonstration programs provided on *Repertoire's* distribution disks. Those programs also contain demonstrations of `TheEditor` being called in a more conventional way. The `PMIdbms` program, for example, contains an "Edit File" option under the top-level "Manage Files" menu option. The Edit File option calls `TheEditor` with most of its defaults in place. A similar demonstration is also included in the `Demo.exe` file on your *Repertoire* demonstration diskette. The editor portion of `Demo` reads the contents of two other files, `FileA` and `FileB`, and edits each text in a separate window.

As you will quickly learn from experimenting with the demonstrations, `DefaultEditor` has very little power in comparison with most commercial text editors. It doesn't have block-move or search-and-replace functions, for example. We would not, and do not, use it for program development. On the other hand, `DefaultEditor` is simple, flexible, and easily integrated into your programs. It is uniquely compatible with *REPETOIRE's* screen system and `NdxFile` system. Its simplicity makes it easy for your users to learn, and easy for you to modify. We think it fills a certain class of needs that more ambitious editor toolboxes do not.

### 5.2. How to Call It

In releases of *Repertoire* preceding 1.5, the only way to call the editor was through the `ScrnEdit` procedure in the `Editor` module. Its parameter list is long and confusing, and because it is not a procedure variable, programs that needed editors with less or more sophistication had to modify the screen system's source code to use different versions of the editor at differ-



## The Editor

ent times. ScrnEdit also relies on global variables, which increasingly cause problems as multitasking and multiuser environments become more common.

ScrnEdit and all the old global variables are still exported from Editor, but only for compatibility with old code; the rest of *Repertoire* does not use them. As soon as possible, you should begin using either VEditor's TheEditor procedure variable. You may also call Editor's DefaultEditor procedure directly if you do not need the flexibility that the procedure variable provides.

To call TheEditor, import TheEditor, InitEdRec, and the type AnEdControlRec from *Repertoire*'s VEditor Module. Declare a variable of type AnEdControlRec, initialize it with InitEdRec, and then pass it and a GenList to TheEditor -- the text you send and receive from TheEditor is contained in a GenList (see Chapter 2). This is how TheEditor is declared:

```
TYPE
  AnEdControlRec = RECORD
    Col1, Row1, Col2, Row2 : CARDINAL;
    TextRow1, CursorCol, CursorRow: CARDINAL;
    Wrapping, ReadOnly, overwriting, DoCounting,
      ChangeMade: BOOLEAN;
    FrameRec: DisplayFrame;
    WindowHandle: AWindowHandle;
    NextFrame: AFrameName;
    MaxLines, EdLastKey, EntryBox, ExitBox,
      CountRow: CARDINAL;
    EntryForeColor, EntryBackColor,
      ExitForeColor, ExitBackColor,
      EntryForeBox, EntryBackBox,
      ExitForeBox, ExitBackBox, EdPromptFore,
      EdPromptBack: AColor;
    EntryAttrib, ExitAttrib, EdPromptAttrib:
      AMonoAttribute;
  END;

  AnEditor = PROCEDURE(VAR GenList, VAR
    AnEdControlRec);

VAR
  TheEditor : AnEditor;
```

As you can see, AnEdControlRec allows *Repertoire* to provide convenient defaults without global variables, and without passing inordinate amounts of data on the stack. It means that AnEditor type can get by with only two parameters, the second of which is actually a pointer to the large record type.

The text is passed in the first (GenList) parameter. If this list is empty, (i.e. initialized by GenLists.NewList) the edit window is initially empty. The cursor is placed in the first position within the edit window and, if AnEd-



ControlRec's `ReadOnly` field is `FALSE`, the user is allowed to add text. If `TheList` is not empty, `DefaultEditor` positions the text and the cursor according to the values in `AnEdControlRec's` `TextRow1`, `CursorCol`, and `CursorRow` fields. `TextRow1` is the number of the line of text that will appear on the first line of the edit window.

The first four fields in `AnEdControlRec` define the window inside which editing will take place. `Col1` and `Row1` are the coordinates of the upper left corner of the window, and `Col2` and `Row2` are the lower right corner. `DefaultEditor` draws a double-lined box around this area, using `Col1`, `Row1`, `Col2`, and `Row2`, if `AnEdControlRec's` `EntryBox` parameter equals 2; it uses a single lined box if it equals 1, and no box at all if it equals 0. The text is actually displayed inside this border. `InitEdRec` by default sets these values to the coordinates of the top left and lower right corners of the screen.

`CursorRow` and `CursorCol` specify the row and column, relative to the top of the window, where the cursor is initially placed. When `TheEditor` exits, it reports the exiting text and cursor position in `AnEdControlRec's` `TextRow1`, `CursorCol`, and `CursorRow` fields.

The `Wrapping` field indicates whether the cursor will wrap back to the left edge of the window when it reaches the right edge, and thus, whether horizontal scrolling will be in effect.

`AnEdControlRec's` `ReadOnly` field allows the editor to be used as a large, scrolling display rather than as a true editor. If `ReadOnly` is `TRUE`, all the cursor movement keys (up, down, left, right, `PgUp`, `PgDn`, etc.; see below) work as they do normally, but keys that can change the text are disabled.

`AnEdControlRec's` `ChangeMade` field tells the caller whether the user edited the text while in `TheEditor`. Although the name of this field may suggest that it will be `TRUE` if the text going into `TheEditor` differs from the text coming out, it actually indicates only whether the user did anything that could have changed the text. If, for example, the user inserted three spaces and then immediately deleted them, `ChangeMade` will be `TRUE`.

`AnEdControlRec's` `FrameRec` field makes `TheEditor` easier to use with *Repertoire's* screen system. For example, it gives `UserOps.TheKeyHandler` the information it needs to provide context-sensitive help. If your program does not use *Repertoire's* screen system, you can leave this field alone; `InitEdRec` sets it to `NIL`. If you are calling `TheEditor` from within the screen system, however, you should assign the current `DisplayFrame` variable to the `FrameRec` field.

The `EdLastKey` and `NextFrame` fields indicate how the user exited `TheEditor`. `NextFrame's` value is determined by `UserOps.TheKeyHandler`, and may depend on values in the `FrameRec` passed in `TheEditor's` param-



## The Editor

ter list. TheKeyHandler is meant to be customized, of course, so you can easily change what happens when the user presses particular function keys. By default, however, when the user presses the backup key, TheKeyHandler tells TheEditor to terminate with the NextFrame field equal to FrameRec^.parent. When the user presses the submit key, TheKeyHandler tells TheEditor to exit with the NextFrame field equal to FrameRec^.normlnext.

You can control the row on which TheEditor writes status messages (i.e., whether it is in insert or overwrite mode, the present position of the cursor in the text, the present size of the text, etc.) by setting the CountRow field variable. Its default value is the first line up from wherever the bottom of the physical screen happens to be.

You can determine whether TheEditor will write out any status messages at all. If you set the BOOLEAN DoCounting field to FALSE, it will not. In the screen system's editor fields, for example, we prevent TheEditor from displaying status messages by setting DoCounting to FALSE. Its default is FALSE.

You can determine whether the DefaultEditor will draw a box around its window upon entry, or upon exit, or upon both or neither. These functions are controlled through the CARDINAL variables EntryBox and ExitBox. Each has only three legal values: 0, 1, and 2. Zero means no box will be drawn. One means a single-lined box will be drawn. Two means a double-lined box will be drawn. Both default to 2.

You can control the colors and/or the monochrome attributes of every part of an editor window. The EntryForeColor field controls the foreground color of text in the DefaultEditor upon entry; ExitForeColor controls it upon exit (the DefaultEditor rewrites the window to make the change as it exits if the colors are different). EntryBackColor and ExitBackColor do the same things for the background of the DefaultEditor window. EntryForeBox, EntryBackBox, ExitForeBox, ExitBackBox, control the entry and exit foreground and background colors of the box that is drawn around the editor window. EdPromptFore and EdPromptBack control the foreground and background colors of the status line at the bottom of the screen (if it is displayed). EntryAttrib, ExitAttrib, and EdPromptAttrib control corresponding parts of the DefaultEditor window on monochrome screens.

You can determine whether the DefaultEditor begins in insert or overwrite mode by setting the BOOLEAN field named overwriting.

## 5.3. The User Interface

*Repertoire* has horizontal scrolling and a multiline delete/undelete buffer to the simple editing functions in DefaultEditor. The arrow keys control



cursor positioning; Ctrl-Left and Ctrl-Right move between words; Del and BACKSPACE delete characters; Ins toggles between Insert and Overwrite modes; Alt-D deletes the current line; Alt-I and Alt-U copy the last line or block of lines deleted back into the text at the current cursor position; typing past the end of the window either scrolls the window horizontally or automatically wraps text to the next line, depending on the value of the Wrapping field; current character and line counts may be shown on the screen, but by default are not. DefaultEditor also reformats paragraphs in response to the F7 function key, and toggles wrapping on and off in response to Shift-F7. Instructions for performing specific functions are given below.

### 5.3.1. User Instructions.

The information below should allow users of your programs to make full use of any editor windows you define:

Erase a character:

Press the Del key to erase the current character, or backspace to move it over the previous one.

Move to another character on this line:

Use the left and right arrows to move around on the current line. Ctrl-Left-Arrow and Ctrl-Right-Arrow move you to the next word and prior word, respectively, even if that word is not on the current line. The Home and End keys move the cursor to the beginning and end of the current line.

Go to another line:

As you type, the DefaultEditor will automatically move the cursor to the next line when necessary. You can insert a line by pressing Return at the end of the line above it. If you want to move the cursor to a line below its present position, press the down arrow. Pressing the up arrow will move the cursor to the line above.

Insert letters:

Place the cursor under the letter before which your text is to be inserted, and begin typing. If you are in "Insert mode" the text is inserted. The lower right corner of the screen tells you what your mode is: insert or overwrite. To change modes, press the Ins key. Note: If you are inserting in the middle of a line, DefaultEditor will push characters to the next line when the current one is full. And it will push characters off the next one when it gets full, etc. If you are inserting a lot of information many lines from the bottom, you may be able to type faster than the computer can move these letters around, so type slower if it beeps at you. It doesn't matter which mode you use at the end of your text.

## The Editor

### Overwrite letters:

Move the cursor to the text you want to overwrite, choose "Overwrite" with the Ins key (see above), and begin typing. You will overwrite letters as you come to them.

### Break one line into two:

Move the cursor under the letter that you want to be the first character of the new line below, and press Return.

### Join two lines together:

Move the cursor under the first letter on a line and press the backspace key. The line will be joined to the one above it. You can also move the cursor to the end of a line and press the <Del> key. The line will be joined to the one below it.

### Delete a line:

Put the cursor any place in the line, hold down the Alt key and press d.

### Move to other parts of the Text:

You can always move between lines of text by pressing the up and down arrows—even if the next line is not shown on the screen. When a new line is needed, the text is scrolled to make room for it. If you press the down arrow when the cursor is on the bottom line of the window, the text above is moved up one line, making the old top line disappear. The next line of your text appears on the bottom line.

### Reformat Paragraphs:

If you press the F7 key, the text from the current cursor position down to the end of the current paragraph is reformatted. *REPertoire* reformats by packing the text into as few lines as possible without breaking words. For example, in a window 70 characters wide, four lines of about 30 characters each may be packed into two lines: one line of about 70 characters, and one of about 50.

### Paging up and down:

The PgUp and PgDn keys cause the editing window to move up and down, respectively, by the full length of the window (i.e., by a page).

### Moving to the top and bottom of the file:

Ctrl-PgUp and Ctrl-PgDn move the cursor and the editing window to the top and bottom (respectively) of the file.

### Copy a line or block of lines to another location



Press Alt-D to delete the lines you want to copy, and then immediately press Alt-I to reinsert them at the present cursor location. Now you can move to the location where you want to copy the lines, and press Alt-I again to insert the deleted lines there. The deleted lines remain in the delete buffer until you press Alt-D to delete a different set of lines. The delete buffer remains intact between windows, so if you want to move or copy text from one editor window into another, use exactly the same steps.

## 5.4. Line and Character Counters

DefaultEditor can display on the screen the total number of lines and characters in the text and the number of the text line and character where the cursor is sitting. A word at the end of this status line indicates whether the DefaultEditor is in Insert or Overwrite mode. These counters and indicator are shown in a status line that extends across the screen, beginning in column 2 of the row indicated by CountRow. You can turn the status line completely off by setting DoCounting to FALSE. The DefaultEditor responds to keys more quickly if DoCounting is FALSE because it does not have to convert its LONGINT character-count to a string and write it at the bottom of the file with every keystroke, as it does when DoCounting is TRUE. You may set CountRow to any value, but it normally is set to a row number at or near the top or bottom of the screen.

## 5.5. Editor Function keys

DefaultEditor and *Repertoire's* screen display system share a single system for handling function keys. DefaultEditor calls the UseOps.TheKeyHandler procedure variable after every keystroke, just as the ControlFrame procedure in InputManager does. Function keys in DefaultEditor have the same functions as function keys in the rest of the screen system; when DefaultEditor is used outside the screen system (i.e., when it is called outside of ControlFrame) function keys work as they would in a completely blank frame.

## 5.6. Editing To/From Files

Your program may require the ability to edit a DOS file, and not simply text passed from your program. The following example shows how a program could edit a file:

```
MODULE EditFile;

IMPORT Editor;
IMPORT GenLists;
IMPORT ListUtils;
IMPORT ScrnTypes;
IMPORT SmartScreen;
```

## The Editor

```
IMPORT StrEdit;
IMPORT StringIO;
IMPORT VWindows;
IMPORT InitCompilerMods;

VAR
  ChangeMade: BOOLEAN;
  thelist: GenLists.GenList;
  result : INTEGER;
  Trow, Ccol, Crow: CARDINAL;
  fname: ARRAY [0..79] OF CHAR;
  DummyFrame: ScrnTypes.DisplayFrame;

BEGIN
  GenLists.NewList( thelist);
  ScrnTypes.InitDisplayFrame( DummyFrame,
    VWindows.CurrentWindow );
  StrEdit.AssignStr( "filename.in", fname);
  StringIO.PrintMessage(
    ListUtils.TextFileToList(
      fname, thelist));
  SmartScreen.ClearScreen();
  Trow := 1;
  Ccol := 1;
  Crow := 1;
  result := Editor.ScrnEdit(1,1, 80,25, thelist,
    Trow, Ccol, Crow, ChangeMade, FALSE,
    DummyFrame);
  SmartScreen.ClearScreen();
  IF ChangeMade THEN
    StringIO.PrintMessage(
      ListUtils.TextListToFile(
        thelist, fname));
  END;
END EditFile.
```

## 5.7. Editing in Windows

Normally, you will not use the entire screen for editing. Even when the entire screen is devoted to an editing task, the actual text is not usually written on the entire screen. You may have one or more title lines at the top of the screen, the DefaultEditor status line and a line explaining the function keys at the bottom, and a box drawn around the edit window. You may want to write this border information using *Repertoire's* screen display system. Alternatively, you could directly call screen I/O routines such as those provided by *Repertoire's* SmartScreen module.

## 5.8. Multiwindow Editing

You may require that two or more edit areas be simultaneously displayed on the screen. These could be shown in separate or overlapping windows. One of your function keys could allow the user to move between the windows and edit the texts. TheEditor provides these features in two major



ways:

1. You can execute two or more DefaultEditors in a LOOP until a designated exit code is found. By using the last three AnEdControlRec fields, you can always place the text and cursor in each window where they were when the user last left that window.

2. You can create a separate Modula-2 process for each editing window. Then, a special function key can directly TRANSFER between the processes. This is certainly the fastest and most elegant approach, but it requires a little more RAM and a little more understanding of Modula-2. You can easily crash your machine if the process-handling code contains any mistakes. We provide equivalent examples of each approach below.

### 5.8.1. Simple Multiple Windows

The following program reads the text in DOS file fileA and puts this text in an edit window in the upper left corner of the screen. Another window is opened in the lower right corner for editing. The user moves between these windows by leaving either edit session with any key except the "exit" key. The user moves between windows until the exit key is pressed. Then the text from fileA is written back to fileA, and the other window's text is written to fileB. Your distribution diskette contains the .MOD and .EXE files of a modified version of this program, named EdDemo, which is used by *Repertoire's Demo* program. EdDemo differs from the program below in that EdDemo uses colors, returns a result to another program, and reads the text for each window from a file.

```
MODULE TwoFile1;

IMPORT Editor;
IMPORT GenLists;
IMPORT ListUtils;
IMPORT ScrnTypes;
IMPORT SmartScreen;
IMPORT StrEdit;
IMPORT StringIO;
IMPORT UserOps;
IMPORT VWindows;
IMPORT InitCompilerMods;

VAR
  TextA, TextB : GenLists.GenList;
  fname: ARRAY [0..79] OF CHAR;
  ChangeMade : BOOLEAN;
  DummyFrame: ScrnTypes.DisplayFrame;
  TextRowA, CursRowA, CursColA, TextRowB,
    CursRowB, CursColB : CARDINAL;

BEGIN
  SmartScreen.ClearScreen();
  GenLists.NewList(TextA);
```

## The Editor

```
GenLists.NewList(TextB);
StrEdit.AssignStr("fileA", fname);
StringIO.PrintMessage(
    ListUtils.TextFileToList(
        fname, TextA) );
ScrnTypes.InitDisplayFrame( DummyFrame,
    VWindows.CurrentWindow );
DummyFrame^.parent := '1';
(*DummyFrame^.parent will be returned from
    TheFuncProc when Esc is pressed; we don't
    want it to be the EndCode.*)
GenLists.NewList( TextB);
Editor.EntryBox := 2;
Editor.ExitBox := 1;
TextRowA := 1;
CursRowA := 1;
CursColA := 1;
TextRowB := 1;
CursRowB := 1;
CursColB := 1;
Editor.CountRow := 25;
LOOP
    IF UserOps.EndCode=Editor.ScrnEdit( 3, 1, 48,
        17, TextA, TextRowA, CursRowA, CursColA,
        ChangeMade, FALSE, DummyFrame) THEN
        EXIT;
    END;
    IF UserOps.EndCode=Editor.ScrnEdit( 30, 8, 75,
        24, TextB, TextRowB, CursRowB, CursColB,
        ChangeMade, FALSE, DummyFrame) THEN
        EXIT;
    END;
END;
IF ChangeMade THEN
    StringIO.PrintMessage(
        ListUtils.TextListToFile(
            TextA, "fileA" ) );
    StringIO.PrintMessage(
        ListUtils.TextListToFile(
            TextB, "fileB" ) );
END;
SmartScreen.ClearScreen();
END TwoFile1.
```

### 5.8.2. Separate Processes

We can exploit the power of Modula-2 to implement multiple windows in a more elegant fashion: use separate Modula-2 processes to edit TextA and TextB. We must caution again that this technique is both powerful and dangerous. The following program functions equivalently to the simple program above, except we do not write the results to a file. The routine SpecialKeys is our DoEdFunctions routine from EdUserOps, modified to support switching between windows. The user moves between windows by pressing Esc.

```
MODULE TranEd;
```



```

IMPORT Editor;
IMPORT GenLists;
IMPORT ListUtils;
IMPORT ScrnTypes;
IMPORT SmartScreen;
IMPORT Storage;
IMPORT StringIO;
IMPORT SYSTEM;
IMPORT UserOps;
IMPORT VWindows;
IMPORT InitCompilerMods;

```

```
CONST
```

```
    EditWA = 5000;
```

```
VAR
```

```

    TextA, TextB : GenLists.GenList;
    fname: ARRAY [0..20] OF CHAR;
    TextRow, CursRow, CursCol : CARDINAL;
    ProcA, ProcB, ProcMain: SYSTEM.PROCESS;
    WkAreaA, WkAreaB : SYSTEM.ADDRESS;
    which : CHAR;
    ChangeMade : BOOLEAN;
    DefaultKeyProc: UserOps.AKeyHandler;
    DummyFrame: ScrnTypes.DisplayFrame;

```

```
PROCEDURE TmpKeyHandler( FrameRec :
```

```
    ScrnTypes.DisplayFrame;
```

```
    VAR LastKey: CARDINAL;
```

```
    VAR NextFrame: ScrnTypes.AFrameName );
```

```
VAR SavedCol, SavedRow: CARDINAL;
```

```
BEGIN
```

```
    IF LastKey = 27 THEN
```

```
        (*Esc*)
```

```
        IF which='A' THEN
```

```
            which := 'B';
```

```
            SYSTEM.TRANSFER(ProcA, ProcB);
```

```
        ELSE
```

```
            which := 'A';
```

```
            SYSTEM.TRANSFER(ProcB, ProcA);
```

```
        END;
```

```
    ELSE
```

```
        DefaultKeyProc(FrameRec, LastKey, NextFrame);
```

```
    END;
```

```
END TmpKeyHandler;
```

```
PROCEDURE EditProc1();
```

```
    VAR DumInt : INTEGER;
```

```
BEGIN
```

```
    REPEAT
```

```

        DumInt := Editor.ScrnEdit(3,1, 38,12,TextA,
            TextRow, CursRow, CursCol, ChangeMade,
            FALSE, DummyFrame);
    UNTIL FALSE;
```

```
END EditProc1;
```

```
PROCEDURE EditProc2();
```

```
    VAR DumInt : INTEGER;
```

## The Editor

```
BEGIN
  REPEAT
    DumInt := Editor.ScrnEdit(40, 13, 75, 24,
      TextA, TextRow, CursRow, CursCol,
      ChangeMade, FALSE, DummyFrame);
  UNTIL FALSE;
END EditProc2;

BEGIN
  SmartScreen.ClearScreen();
  GenLists.NewList(TextA);
  GenLists.NewList(TextB);
  ScrnTypes.InitDisplayFrame( DummyFrame,
    VWindows.CurrentWindow );
  Editor.EntryBox := 2;
  Editor.ExitBox := 1;
  Editor.DoCounting := FALSE;
  DefaultKeyProc := UserOps.TheKeyHandler;
  UserOps.TheKeyHandler := TmpKeyHandler;
  fname := 'demol.edt';
  StringIO.PrintMessage(
    ListUtils.TextFileToList(
      fname, TextA ) );
  TextRow := 1;
  CursRow := 1;
  CursCol := 1;
  which := 'A';
  Storage.ALLOCATE( WkAreaA, EditWA);
  SYSTEM.NEWPROCESS( EditProc1, WkAreaA, EditWA,
    ProcA);
  Storage.ALLOCATE( WkAreaB, EditWA);
  SYSTEM.NEWPROCESS( EditProc2, WkAreaB, EditWA,
    ProcB);
  SYSTEM.TRANSFER( ProcMain, ProcA);
  SmartScreen.ClearScreen();
  SmartScreen.SetCursorHeight(2);
END TranEd.
```



## 6. Natural-Language Analysis: StrLogic

The StrLogic section of the Repertoire Reference Manual explains the string-analysis functions provided by the StrLogic module. The central routine in StrLogic is the function procedure ExpressionTest, which analyzes strings to determine whether they satisfy conditions expressed in boolean logic. To make sense of that last sentence, you should now familiarize yourself with the ExpressionTest routine by reading the StrLogic section of the Repertoire Reference Manual.

### 6.1. Suggested Applications

The ExpressionTest routine can be used to great advantage in many ways, but *Repertoire* provides special support for using it in the *Repertoire* screen system. The basic theory is that the screen system allows you to determine what the user said in a phrase or sentence typed into a string field. On the basis of that determination, the screen system can automatically branch to different screens.

Natural-language analysis within the screen system is controlled by a section of the command area in each screen. The section is called the Data section, and it resembles the Fields and Help sections in that the word Data must be followed by a colon and a left brace, and the section is terminated by a right brace. For example:

```
Data:{
  (22) ['red' & 'green'] & ~['blue' | 'gold']
  \ (5) 'orange' | 'pink'
  \ (23) ['my' < 'dress'] & ['dress' < 'blue']
}
```

In the example above, the screen system could be made automatically to display frame 22 if the user types "my favorite colors are red and green." If he typed, "my favorite color is pink," screen 5 would be displayed next. If he typed "my dress is blue," screen 23 would be displayed. The less-than symbols in the expression on the third line require the words 'my,' 'dress,' and 'blue' to appear in that order. The sentence "blue dresses make my stomach turn" would not satisfy the expression on the third line.

With a little creativity, these features can make simple programs stunningly powerful. The following suggestions may bring others to mind:

1. Suppose you're designing a complicated set of forms that can be completed in many different ways. The sequence in which the user is to complete the forms (and which of the forms he has to complete) depends on how he has answered previous questions. In a health and fitness questionnaire, for example, you might want to begin by asking



the user to list any diseases for which he has been treated in the last five years. *Repertoire's* language-analysis facilities can automatically examine the answer and branch immediately to a subsidiary form for, say, hepatitis, if that word appears in the user's answer. The user doesn't have to step through hundreds of yes-or-no questions about specific diseases, and your program doesn't have to ask the user whether he listed 'hepatitis' before showing him the hepatitis form. *Repertoire* handles the operation automatically.

2. You could also use *Repertoire's* language analysis facilities to write interactive-fiction games, like ZORK (from Infocom Corp., Cambridge, MA) or the public domain classic ADVENTURE. Because the basic structure of such a game can be defined entirely in a *Repertoire* EDT file ("EDT files" are the readable, editable versions of the screen files used in *Repertoire's* screen system; they are compiled into "DSP files"), *Repertoire* lends itself to the construction of easily understandable, streamlined game text. The game's hard code (i.e., its Modula-2 code) still has to handle functions such as tracking possessions and the locations of objects, but the hard code does not have to be complicated by the details of analyzing natural-language input.
3. Perhaps the most important use for *Repertoire's* language-analysis facilities is in the construction of highly interactive educational software. The Data section of each screen's command area allows such programs to ask and answer free-form questions intelligently and conveniently.

## 6.2. How to Use It

In the Data sections of each command area in your editable screen files (EDT files), you may construct any string that could legally be passed to ExpressionTest in its second parameter (FormulaList). See the StrLogic section of the *Repertoire* Reference Manual. You need not worry about line breaks in the Data section.

For normal applications, the best strategy is to precede each of the expressions in the Data section with a screen number in parentheses. Your code will then be able to control the succession of frames by asking which of the expressions is satisfied by the user's input.

This is how. After your program displays a screen containing a Data section, and after you call Control, Input, or ControlFrame to obtain the user's input, call the function procedure WhatTheySaid from the LogicInput module. WhatTheySaid has two parameters: the name of the DisplayFrame, and the name of the field containing the user's input. This is how it is de-



clared:

```
PROCEDURE WhatTheySaid(VAR TheFrame : Display-
    Frame;FieldName : ARRAY OF CHAR) : CAR-
    DINAL;
```

**WhatTheySaid** returns a **CARDINAL** value indicating the result of applying **ExpressionTest** to the string in the named field using the expression from the **Data** section for the most recently displayed screen. That value will represent the next frame to be displayed, unless you've structured your application in some other fashion. Use it as the next screen to display, and the program will proceed properly.

This is an example of the critical loop in a program using *Repertoire's* language-analysis facilities:

```
REPEAT
    Input( TheScrnNum );
    IF TheScrnNum = ReturnedInt(
        MainFrame^.normlnext) THEN
        TheScrnNum := WhatTheySaid(aInFrame, '' );
    IF TheScrnNum = 0 THEN
        (*No patterns matched.*)
        TheScrnNum := ReturnedInt(
            MainFrame^.normlnext);
    END;
END;
UNTIL TheScrnNum = EndCode;
```

The third line of the **REPEAT** loop in the above example tests the result of **Input** to determine whether it has returned the **normlnext** value (see section 3.7.5), because that is what *Repertoire* always returns if the user submits the screen while a string field (actually, any field except a **Goto** field) is selected. Although there may be exceptions, we assume that most programs will have users exiting from the fields in which the strings to be analyzed were entered.

The next few lines determine whether **WhatTheySaid** returns a zero. If so, we know that the user's input has failed to satisfy any of the expressions in the **Data** section for **TheScrnNum** (in other words, the expressions in the screen have failed to decipher what the user said). The most proper response in that case is probably to display the **normlnext** screen. You may, however, want to control this situation explicitly from your **Modula-2** code; if so, you would assign **TheScrn** the number of some screen that indicates your failure to understand the last answer, or you would call some more complicated routine designed specifically for the unpredicted-answer situation.

Notice that the example above is almost all the **Modula-2** code you need to control many applications resembling the three suggested above.

# Appendix A: Extended Scan Codes

Using various Repertoire keyboard-reading routines (e.g., KeyHit), you have the ability to detect any key pressed. On occasion, you may also want to specify the valid input characters with KeyNumSets, as explained in the KbdInput section of the Repertoire Reference Manual. Normally, the ASCII character corresponding to the key the user pressed represents that key in Repertoire. Function keys, arrow keys, and a few others, however, do not have ASCII representations. To provide for them, IBM defined "extended" codes to represent such keys. Repertoire represents an extended character by adding 256 to its ordinal value. Below, we tabulate the ordinal values of various keys in the extended code set.

| <u>Extended</u><br><u>Code</u> | <u>Plus</u><br><u>256</u> | <u>Key Pressed</u>            |
|--------------------------------|---------------------------|-------------------------------|
| 3                              |                           | null: ASCII(0)                |
| 15                             | 271                       | backtab                       |
| 16..25                         | 272..281                  | Alt Q, Alt W, Alt E...Alt P   |
| 30..38                         | 286..294                  | Alt A, Alt S, Alt D...Alt L   |
| 44..50                         | 300..306                  | Alt Z, Alt X, Alt C...Alt M   |
| 59..68                         | 315..324                  | F1, F2, F3...F10              |
| 71                             | 327                       | Home                          |
| 72                             | 328                       | Up Arrow                      |
| 73                             | 329                       | PgUp                          |
| 75                             | 331                       | Left Arrow                    |
| 77                             | 333                       | Right Arrow                   |
| 79                             | 335                       | End                           |
| 80                             | 336                       | Down Arrow                    |
| 81                             | 337                       | PgDn                          |
| 82                             | 338                       | Ins                           |
| 83                             | 339                       | Del                           |
| 84..93                         | 340..349                  | Shf F1, Shf F2...Shf F10      |
| 94..103                        | 350..359                  | Ctrl F1, Ctrl F2...Ctrl F10   |
| 104..113                       | 360..369                  | Alt F1, Alt F2...Alt F10      |
| 114                            | 370                       | Ctrl PrtSc                    |
| 115                            | 371                       | Ctrl Left Arrow               |
| 116                            | 372                       | Ctrl Right Arrow              |
| 117                            | 373                       | Ctrl End                      |
| 118                            | 374                       | Ctrl PgDn                     |
| 119                            | 375                       | Ctrl Home                     |
| 120..131                       | 376..387                  | Alt1, Alt2...Alt0, Alt-, Alt= |
| 132                            | 388                       | Ctrl PgUp                     |



# Index

- <CR>, 58
- action, 70
- AddCriteria, 107
- Adding DisplayFrames, 90
- Adding To Frames, 81
- AddMenuItem, 82
- AddRec example, 108
- address arithmetic, 16
- AddStrField, 81, 82
- AddToMenu, 56
- AFunctProc, 85
- Alt-D, 58
- Alt-E, 58
- AltKeyList, 106
- AltKeyLists, 105
- analstr, 71
- append, 115
- appending list elements, 23
- ARRAY OF BYTE, 18
- ARRAY OF REAL, 101
- attributes in editor, 130
- Back-Up key, 37
- background colors, 43
- backing up, 60
- BACKTAB, 58
- Backup Key, 86
- backup key, 55
- beeps in editor, 131
- BeginCmdLine, 36
- BeginMarkers, 122
- BeginTimer, 15
- binary file data, 101
- blanks in fields, 62
- Blinking, 43
- blinking text, 43, 74
- BlockRead, 120
- BlockWrite, 120
- Bold, 43
- boldfacing, 74
- Boolean fields, 34
- bordatrb, 71
- bordbak, 71
- Border Colors, 45, 71
- bordfor, 71
- Box, 54
- box, 70
- box around editor, 130
- box drawing, 29
- Building Frames in Programs, 81
- BuildLst, 104
- BW640, 46
- C, using Repertoire from, 16
- calculating abs screen positions, 87
- case in frame names, 27
- Case Sensitivity, 98, 110
- case sensitivity, 30
- centering of prompts, 89
- ChangeField, 80
- ChangeField, example, 57
- ChangeMade, 129
- Changing Keys, 84
- changing screen data, 57
- Changing Screens in Programs, 80
- Changing Special keys, 85
- Choice fields, 34
- ChoicePrompt, 89
- clear, 70
- ClearAfter, 41
- ClearFirst, 41
- CloseDisplayFile, 49, 67
- CloseDisplayFrame, 91
- CloseNdxFile, 96, 98
- CodeChars, 122
- col, 73
- Color Code Table, 43
- Color Commands, 42
- Color commands
  - example, 45
- Color delimiters, 44
- Coloring Frame Parts, 44
- Colors, 28, 71
- colors in editor, 130

## Index

- colors, determining a frame's, 71
- column alignment, 35
- Command Area, 25, 28, 54
- Command Area delimiters, 36
- command area delimiters, 27
- CommandBegin, 28
- CommandEnd, 28
- comments, 65
- ContinueCode, 85
- ContinueInput, 87
- Control, 51
- ControlFrame, 51, 54, 57, 60, 112
- ControlFrame, result table, 55
- ControlFrameGroup, 37
- CopyList, 107, 118
- corruption, 23
- CreateNdxFile, 96, 115
- Creating DisplayFrames, 90
- Creating Frames in Programs, 81
- Creating records, 98
- CriteriaList, 107
- CriteriaLists, 106
- Criterion records, 106
- CrunchNdxFile, 96, 115
- Ctrl-Left, 58
- Ctrl-Right, 58
- Current .DSP file, 48
- current field, 76
- Current frame, 48
- CurrentField, 87
- Cursor Movement Keys, 86
- cursor size, 49
- Custom key handling, 85
- cut and paste, 127
- Cutting off lines, 90
- damaged NdxFiles, 116
- data display screens, 111
- data entry screens, 110
- Data in frames, 28
- Data section, 39, 139
- DataList, 71
- Date Fields, 32
- DBMS, 112
- default colors, 42
- default field values, 35
- default screen coordinates, 40
- DefaultFieldHandler, 87
- DefaultKeyHandler, 86
- Del, 58
- Delete line, 58
- DeleteRecord, 96, 99
- deleting lines, 132
- delimiter length, 27
- demo.exe, 127
- Detecting screen modes, 46
- determiner procs, 106
- DiagMode, 24
- directory listings, 77
- DirManager, 77
- disappearing frame variables, 91
- Disk Transfer Area, 14
- Display, 51
- Display fields, 34, 62, 82
- displayable screens, 25
- DisplayFile, 67, 69
- DisplayFiles, location of, 53
- DisplayFrame, 51, 69, 77, 81, 90
- DisplayFrames
  - Example, 91
  - Multiple, 90
- DisplayOnly Frames, 29
- Disposing AltKeyLists, 107
- DoFieldInput, 57
- DoFunctions, 92
- DoPrompting, 89
- DOS errors, 117
- Double Box, 40
- DownArrow, 58
- Drectory, 14
- DSP files, 26, 48
- Duplicate handles, 108
- duplicate handles, 116
- dynamic memory allocation, 16



- editable screen file, 64
- Editor fields, 30, 32, 62
- editor fields, 58
- editor fields, MaxLines, 32
- editor fields, ReadOnly, 32
- editor fields, width of, 32
- editor instructions, 131
- Editor, detecting changes, 129
- Editor, initial cursor position, 129
- Editor, ReadOnly, 129
- Editor, speed, 133
- EDT files, 26, 47, 64
- educational software, 140
- EmsStorage, 18
- EncodeErr, 74
- End, 59
- EndCmdLine, 36, 47
- EndCode, 31, 79, 85
- endcol, 70
- ending frame, 33
- EndMarkers, 122
- endrow, 70
- EntirelyVisible, 88
- EntryBox, 40
- EnvironUtils, 15
- Erase to eol, 58
- erasing frames, 55
- Error Checking, 54
- Error checking in frames, 52
- error feedback, 56
- Esc, 60
- Example Frame, 82
- Example Program, 83, 91
- Example Screen, 64
- Example Screen Program, 49, 51, 52, 67
- Examples of 1.5 features, 49
- Exit key, 86
- ExitBox, 40
- Expanded Memory, 18
- ExpressionTest, 140
- extended character set, 46
- F1, 59
- F10, 60
- F2, 60
- F3, 60
- FAPI, 15
- FeildNum, 57
- Field commands, 29
- Field Editing, 58
- Field entry control, 87
- Field Entry Keys, 86
- Field Mark, 28, 30, 47
- field mark replacement, 35
- Field Marks, 35
- field name, 30
- Field Names, 98
- Field names, 97, 111
- field names, non-unique, 57
- Field numbers, 57
- Field Prompts, 89
- Field Types, 99
- field types, 75
- field-by-field error checking, 57
- Field-specific prompts, 31
- FieldList, 74, 87
- FieldPtr, 73
- Fields, 65
- fields, 76
- Fields as Lists, 97
- Fields section, 110
- fields, default values, 35
- fields, position of, 73
- File Structures, 120
- FileIO, 14
- Files for Screen Input, 78
- FileStructure, 97
- FillScr, 79, 110, 112
- FindField, 81
- FindFieldNum, 81
- FindFile, 53
- formatting characters, 26
- Frame Names, 27
- frame parts, 44

## Index

- Frame Scrolling, 84
- Frame Stacks, 77
- frame text, 47
- frame, definition of, 26, 69
- FrameSequence, 48, 49, 67, 78, 86
- FreeDiskSpace, 116
- FromFile, 71
- function keys, 130
- FunctKeySet, 85
- game programs, 140
- Garbage collection, 115
- generic utilities, 18
- GetAltList, 103
- GetField, 96, 100
- GetFieldImagePtr, 88
- GetFieldPtr, 74
- GetMatchingKeys, 107
- GetProgParams, 15
- GetRecName, 102
- Ghost Application phenomenon, 38
- GoTo field, 55
- goto field value, 76
- GoTo fields, 62
- GoTo values, 83
- Gough, K. John, 18
- graphics characters, 46
- Graphics Modes, 46
- group field, 58
- Group fields, 34
- group fields, 76
- HALT, 60
- Heading Line, 41, 84
- headline, 70
- Help (field-specific), 30
- Help Frame, 31
- help frames, parent of, 31
- Help in display file system, 65
- Help Key, 86
- Help key, 59
- help sequences, 38, 77
- help sequences, terminating, 31
- Help system, 28
- HelpFrame, 75
- helpframe, 71
- highlighted menu characters, 30
- highlighted selection chars, 34
- HitAnyKey, 28, 29
- HOME, 59
- Horizontal scrolling, 129
- horizontal scrolling, 90
- ImageElement records, 82
- InitCompilerMods, 16
- InitDisplayFrame, 69, 91
- Initializing GenLists, 107
- Input, 50, 141
- Input Prompts, 89
- InputFieldRecord, 74, 82
- Ins key, 58
- Insert Mode, 58
- Insert mode, 131
- insert mode, 130
- InsuffMem, 23
- INTEGER field, 59
- Integer fields, 31
- integer fields, 35
- IntegerPrompt, 89
- Joining lines, 132
- JumpPrompt, 89
- KEY field, 103, 107
- Key fields, 111
- KeyNumSets, 85
- Keys, 98
- LandingField, 56
- Large Frames, 84
- LeftArrow, 58
- length of names, 27
- line length in editor, 134
- LISP, 16
- List fields, 97
- ListCode, 23, 99
- ListFromEdField, 62, 118
- ListToEdField, 118
- LoadAltList, 105



- Locating Frame Fields, 81
- LogicInput, 63, 140
- LogicOperator, 106
- LOGO, 16
- long structure strings, 115
- MainFrame, 51, 69, 77, 93
- MainFrame, associated file, 53
- Max records, 121
- MaxLines in editor fields, 32
- memory leakage, 91
- menu items, 76
- menu options, multi-line, 33
- Message Box, 56
- Message Colors, 45
- Microsoft Windows, 46
- mixed data types, 101
- mixing ReadFrame &  
    ReadDisplayFrame, 91
- modifying fields, 87
- Modifying Screens in Programs,  
    80
- monochrome attributes, 43, 74
- monochrome screens, 42
- msgatrb, 71
- msgbak, 71
- msgfor, 71
- multi-line menu options, 33
- Multiple DisplayFrames, 90
- multiple processes, 93
- Multiple Windows, 90
- Named fields, 97
- Names, frame, 27
- Natural-Language Analysis, 63
- NdxBones, 122
- NdxFile size, 115
- NdxFiles, 78
- NdxFileType, 95, 96
- NdxLists, 105
- NdxMarker, 116
- negative bounds, 32
- NewList, 129
- Next Frame, 55
- NextFrame of 0, 33
- No Box, 40
- NoInit, 23
- Noise, 15
- NoListError, 22
- non-document modes, 26
- nonprintable characters, 15
- Normal Frame Submission, 79
- Normal Next, 28, 55
- Normal Next Frame, 36, 79, 83
- Normal Text Colors, 45
- NormalNext, 60, 65
- normatrb, 71
- normbak, 71
- normfor, 71
- normlnext, 55, 71, 141
- NoSuchList, 23
- NULL record names, 111, 112
- NumConv, 15
- object module independence, 16
- object-only version, 17
- OpenDisplayFile, 49, 67
- OpenNdxFile, 97
- overflow, 22
- overhead, GenLists, 18
- Overlaying Frames, 41
- OverWrite mode, 58
- overwrite mode, 130
- parent, 71, 79
- Parent Frame, 28, 36
- ParentFrame, 65
- pbatrb, 71
- pbbak, 71
- pbfor, 71
- PgDn, 132
- PgUp, 132
- Plain, 43
- PMIdbms, 127
- PMIDBMS.MOD, 104, 112
- Pointer Bar Colors, 45, 71
- pointer bar in graphics mode, 47
- PointerBar, width of, 80

## Index

- PopFrame, 77
- Popping Frames, 77
- PosUtils, 15
- Premature disposal, 118
- premature list disposal, 24
- PreserveLength, 80
- Press any key, 28
- PrintMessage, 117
- Procedure Variables, 117
- procedure variables, 85
- processes, 135
- Program Segment Prefix, 15
- program termination, 34
- ProgramExit, 31, 50, 55, 60
- promatrb, 71, 90
- prombak, 71, 90
- promfor, 71, 90
- prompt, 30
- Prompt Colors, 45
- Prompt Line Colors, 71
- Prompt Positioning, 89
- PromptCol, 89
- PromptLength, 89
- PromptLine, changing it, 130
- PromptList, 72
- PromptNum, 75
- PromptRow, 89
- Prompts, 89
  - Field-specific, 31
- Prompts (in Screen File), 89
- Prompts, centering of, 89
- prompts, field specific, 75
- pull-down windows, 76
- PushedFrame, 70
- PushFrame, 77
- Pushing Frames, 77
- PutField, 68, 96, 98
- PutFieldRec, 82
- PutScrRec, 79, 110, 111
- qualified identifiers, 16
- range validation, 76
- ReadDisplayFrame, 51
- ReadDisplayFrame, example, 57
- ReadEnvironment, 15
- ReadFrame, 91
- Reading file data into Screens, 78
- reading strings, 15
- ReadInput, 61
- ReadNextInput, 62
- ReadOnly editor, 129
- ReadOnly editor fields, 32, 33
- ReadShowAndCntrl, 67
- ReadTimer, 15
- REAL field, 59
- Real fields, 32
- real fields, 35
- RealConv, 15
- RealPrompt, 89
- RebuildNdx, 108
- RecNameLength, 98
- RecNameLists, 106, 107
- ReconstructFrame, 77
- RECORD as field name, 98
- Record buffering, 98
- record counts, 115
- Record Limits, 121
- record name, finding, 102
- Record Names, 98
- RecordExists, 102
- Redraw parameter, 57
- RedrawField, 88
- reformatting paragraphs, 132
- RefPastEnd, 23
- RefToZero, 23
- replace option, 35
- required designation, 30
- required fields, 31, 54, 75, 87
- reserved type codes, 117
- ResetFieldPtr, 77
- restoring underlying frames, 55
- Retrieve, 102
- return key, 55
- Reverse Video, 43



- reverse video, 74
- RightArrow, 58
- row, 73
- Run command, 48
- SafetyOff, 117
- SafetyOn, 99, 108
- Saving screen data, 111
- Saving Screen Input, 78
- Saving Screens, 77
- Screen Colors, 71
- screen compiler, 25, 47
- Screen Example Program, 49, 51, 52
- screen names, 27
- Screen Prompts, 89
- Screen System Example, 64
- screen, definition of, 26
- Scrn2Ndx, 104, 110
- ScrnComp, 140
- ScrnUtil, 60
- Scrolling, 90
- scrolling, 41
- Scrolling Frames, 84
- scrolling windows, 29
- SearchFile, 105
- selatrb, 71
- selbak, 71
- selected fields, 76
- Selected Text Colors, 45
- selecting listed items, 56
- selection character, 30, 34
- selection characters, 34
- SelectProc, 106
- selfor, 71
- Shammas, Namir, 18
- ShowDisplayFrame, 51
- showfields parameter, 57
- ShowUsage, 15
- Single Box, 40
- SIZE, 16
- SizeInfo, 96, 115
- skip, 62
- Small Windows, 90
- SmartScreen, 14
- Space Usage, 115
- spaces in names, 27
- Speaker, 15
- Special Keys, 59
- Special keys, 85
- special keys, 59
- Stacks of Frames, 77
- standard frame parts, 44
- startcol, 70
- startrow, 70
- status messages in editor, 130
- Storing AltKeyLists, 104
- Storing Lists, 99
- Storing Screen Input, 78
- StrAnalysis, 39, 139
- StrCode, 23, 99
- StrConv, 15
- StrEdit, 15
- String entry editing, 87
- STRING field, 59
- String fields, 31
- string fields, 35
- StringIO, 14
- StringPrompt, 89
- strings too long, 115
- StrSelector, 107
- structure record, 116
- Structure string, 96
- Subfields, 97, 98
- SubLists, 97
- Submit Key, 86
- submit key, 37, 54, 55
- submitting frames, 60
- TAB, 58
- terminating help sequences, 31
- text attributes, 74
- TextRow1, 129
- TheFieldHandler, 57, 87, 88
- TheKeyHandler, 85, 88
- ThisFrame, 69

## Index

- time key, 60
- Trailing Spaces, 98
- TRANSFER, 135
- TSIZE, 16
- Type checking, 117
- type checking, run time, 23
- type code 0, 23
- Type codes, 99
- type codes, 117
- TypeCheck, 24
- Under Scored, 43
- underflow, 22
- underlining, 74
- Uninitialized List, 118
- unstructured records, 98
- Unused file space, 115
- UpArrow, 58
- UserOps, 57
- VStorage, 16
- WARN, 117
- WhatTheySaid, 63, 140, 141
- Window Commands, 39
- window commands, 28
- window headings, 29
- window position, 54
- WindowPrims, 14
- word processors, 26
- WriteRecord, 96, 98, 99, 111
- WriteToFrame, 82
- writing entire record, 98
- zero as frame name, 31
- zero record, 116, 120







PMI Software

3279 N Nicolet Drive • Green Bay, WI 54311  
(414) 468-6040 FAX: • (414) 465-0464 • BBS: (414) 465-1656