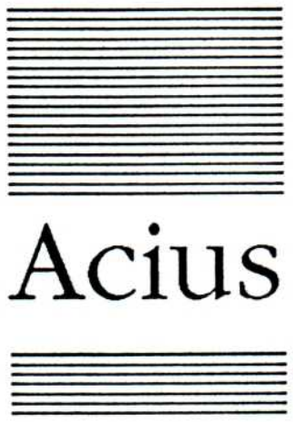
Programmer's Reference

ACIUS

4th DIMENSION™



Acius 4th Dimension™ Programmer's Reference

4th Dimension by Laurent Ribardière

Copyright © 1987 Acius, Inc.
All rights reserved.

This manual was written by Dominique Hermsdorff, Will Mayall, Bruce Barrett, and Bill Kling.

Cover design by Patrick Chédal C & C.

This manual and the software described in it may not be copied, in whole or in part, without written consent of Acius, Inc., except in the normal use of the software or to make a backup copy. It is against the law to copy 4th Dimension on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

Even though Acius has tested and reviewed the software and documentation, **ACIUS MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO SOFTWARE, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS SOFTWARE IS SOLD "AS IS," AND YOU THE PURCHASER ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE. IN NO EVENT WILL ACIUS BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT IN THE SOFTWARE OR ITS DOCUMENTATION,** even if advised of the possibility of such damages. In particular, Acius shall have no liability for any applications developed with, or data stored in or used with, 4th Dimension, including the costs of recovering such programs or data.

Apple, AppleShare, AppleTalk, ImageWriter, LaserWriter, MacDraw, Macintosh, and MacPaint are trademarks of Apple Computer, Inc.

Contents

Figures and tables

Preface

This reference manual gives an overview of the 4th Dimension™ program, describes its language and structures, gives programming tips, and shows how to use its features.

Manual overview

Here's a quick overview of the manual:

- Chapter 1 gives an overview of 4th Dimension.
- Chapter 2 discusses the programming language and use of variables.
- Chapter 3 covers files, records, substructures, layouts, and sorting.
- Chapter 4 gives detailed instructions on working with layouts and their components.
- Chapter 5 looks at the advantages of multi-file databases and the use of links.
- Chapter 6 shows you how to use sets.
- Chapter 7 covers menu setup and use.
- Chapter 8 discusses manipulation of pictures.
- Chapter 9 shows you how to write and use ASCII maps.

For a "how to" approach to 4th Dimension, see *4th Dimension User's Guide.*

Aids to understanding

Look for these visual cues throughout the manual:

- *By the way:* Text set off in this manner presents sidelights or interesting pieces of information.

Important
Text set off in this manner presents important information that you should read before proceeding.

Warning
Warnings like this alert you to situations where you could lose data or damage hardware or software.

This manual uses a special typeface for samples of code and procedure listings:

```
It looks like this.
```

In syntax statements, metasymbols are shown in italic.

Vocabulary

A 4th Dimension *command* always appears in all capital letters. For example, **DEFAULT FILE.** A 4th Dimension *function,* on the other hand, always returns a value and appears with an initial capital letter. For example, **End selection.** The Procedure editor groups 4th Dimension commands and functions together in a window under the term *Routines*. It groups control of flow and assignment terms as *Keywords* in another window.

When referring to user-written code, *routine* means any programming entity you might create. (Developer-created routines appear on the screen in italic type at the end of the list of 4th Dimension routines in the Procedure editor. Externally written and compiled routines appear in bold italic.) When referring specifically to a developer-created procedure or function, the book uses the term *procedure* or *function,* accordingly.

The term *numeric* refers to any data object on which you can perform arithmetic. Thus, *numeric* comprises the data types Real, Integer, and Long Integer.

Chapter 1

Overview

This chapter provides a general overview of 4th Dimension tools, environments, and syntactic definitions.

Tools

4th Dimension offers the application designer powerful tools: database data structures, easy creation of layouts (input and output forms and dialogs), and a powerful programming language for handling input, output, processing, and interfacing. You can also set up complete menu systems and password protection. Figure 1-1 shows 4th Dimension's general architecture.

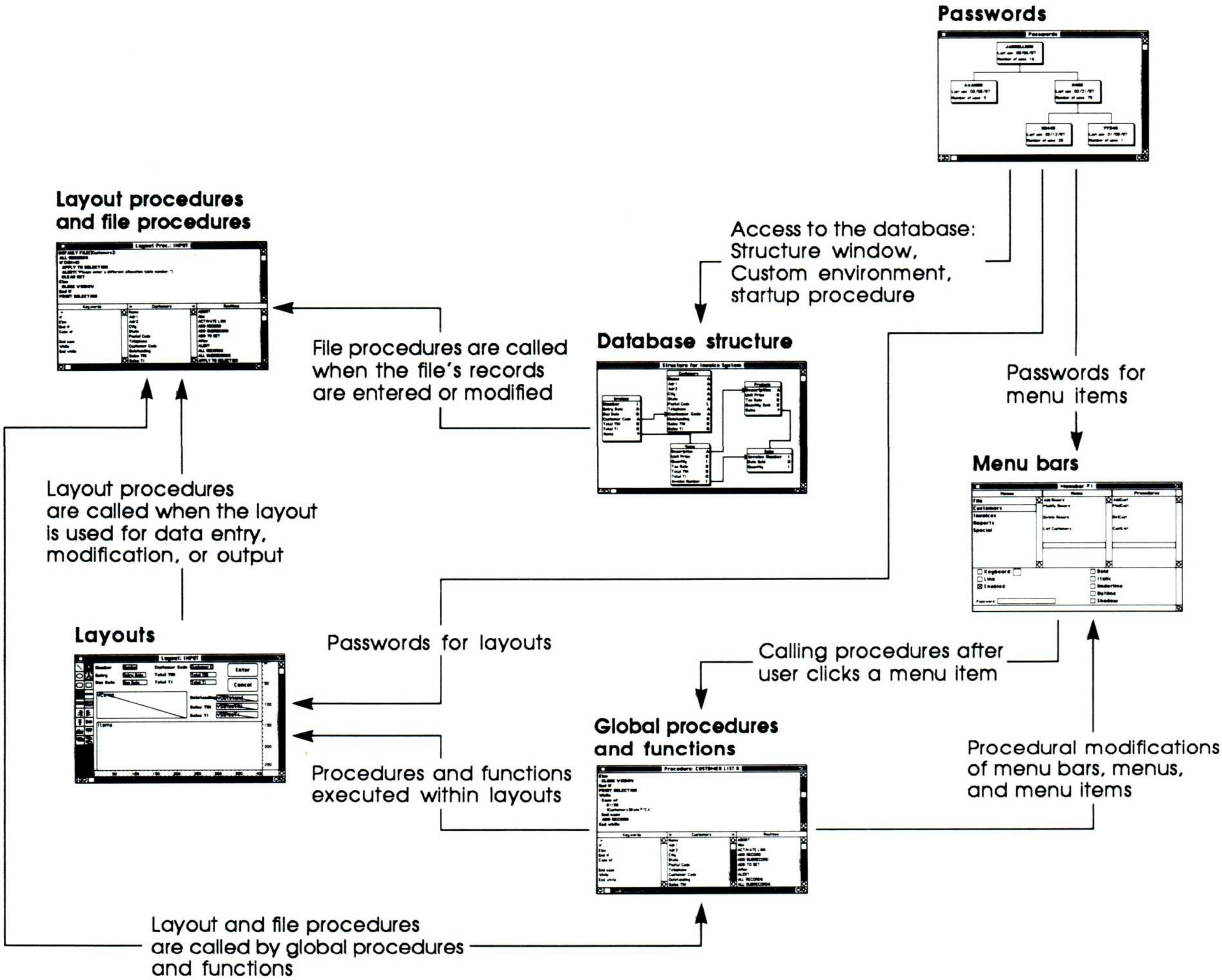


Figure 1-1
4th Dimension general architecture

Here is a quick overview of 4th Dimension's features. The rest of this book details these features and how to use them.

Database structures

A database can contain as many as 99 files; every file can contain up to 511 fields. You can assign any of eight types to a field: Alphanumeric, Text, Real, Integer, Long Integer, Date, Picture, or Subfile. A subfile can in turn contain up to 511 subfields. You can define up to five subfile levels per file. You can also assign attributes to a field; attributes include Enumerated, Indexed, Unique, Enterable, Non-enterable, Modifiable, Can't modify, and Mandatory. You can create relationships between files by drawing lines to link files. Links are activated through procedures.

Layouts

Once you've created a file, you can specify layouts for entering records, for displaying records on the screen or printing them on the printer, and for importing or exporting data. You can also create layouts for custom dialog boxes. You can define up to 32,767 layouts in a single database. 4th Dimension features a layout generator and a highly sophisticated graphics editor; with the editor, you can include all or some of the file's fields in a layout.

You can add variables to display fields from another file, buttons, radio buttons, check boxes, scrollable variable tables, and graph areas for graphing subfile numeric values. You can draw boxes, lines, ovals, create text, and paste in custom MacPaint® or MacDraw® pictures. You can adjust layout details to produce the kind of printed reports you want.

Layout procedures and file procedures

Once you've created a file and its layouts, you can write procedures that 4th Dimension executes every time you enter or modify a record using that layout. With layout and file procedures, you can control new entries, create your own error and range checking, test the validity of values, and access records from other files. You can also change the way a record is printed according to the values it contains.

Procedures and functions

4th Dimension provides you with approximately 200 standard commands and functions. In addition, you can create your own procedures and functions, which you either call directly or execute from a custom menu, a layout procedure, or a file procedure.

Programming language

You can write your procedures and functions in the Flowchart editor, where the various steps of the procedure are graphically displayed and where tests are linked together. You can also use the Procedure editor and type your statements. This editor features automatic statement indentation.

4th Dimension uses a structured language similar to Pascal. The language structures are the sequence, the branch, and the loop. Macintosh developers can add to 4th Dimension's built-in command set by writing external routines written in a compiled language or in assembly language. You can assign external routines to external areas created in your layouts. A procedure can call an unlimited number of standard or custom procedures and functions. Variables can be either globally or locally related to the routines you write.

Menus

You can create custom menu systems. Every menu item you create can have a corresponding keyboard character, and a check mark. You also can choose the typeface you want for your menu item text. Every time the user chooses a menu item, the procedure assigned to that item executes. In addition, you can assign a customized picture to each menu bar which will be displayed in the middle of the screen.

Passwords

You can protect your database with passwords. A password can protect access to the structure of your database, menu items, and layouts.

Environments

4th Dimension has three environments in which to create, test, and run custom applications: the Design environment, the User environment, and the Custom environment.

Design environment

The Design environment lets you create and modify the design of your database, layout and file procedures, global procedures and functions, and menus and passwords. You can change the structure of your database at any time, even if it already contains data.

User environment

In the User environment, you can

- add, modify, and delete records for any file in the database
- add records to the file you're using or to a different file through the use of layout procedures, file procedures, and links
- report (on screen or to the printer) a record or a list of records in any layout
- print multi-file reports showing subtotals with the Quick report generator
- print (on the screen or to the printer) graphs with the Graph Generator (you can also print labels)
- do a multi-criteria search by index
- do a multi-criteria search with logical conditions associated with And, Or, and Except operators
- do a sequential search with the help of test procedures written in the procedure or the Flowchart editor
- sort a selection of records with up to 30 different sort levels
- execute procedures
- import or export data in SYLK, DIF, or Text format with character mapping

Custom environment

Use the Custom environment to execute applications. While still enjoying User environment features, you use your database as if it were an off-the-shelf application for the Macintosh™ computer with its own menus, dialog boxes, and password protection.

Syntactic definitions

This manual uses a consistent set of metasymbols to express command arguments. This section gives you the symbols and syntactical expressions used throughout this guide. Metasymbols appear in italic throughout this book. Table 1-1 summarizes the syntactic symbols; Table 1-2 shows the syntactic metasymbols.

Table 1-1
Syntactic symbols

Symbol	Description	Example
*	Repeat preceding statement up to last required statement.	{*statement*}«{;*}»
{ }	Repeatable any number of times	{*statement*}
« »	Optional argument	**BEEP**«(*posintexpr*)»
|	One (and only one) of two options	*var*|*subfieldname*
. . .	Intervening code	**While...End while**

Note: Both the asterisk (*) and the vertical bar (|) also appear as arguments and operators.

Table 1-2
Syntactic metasymbols

Metasymbol	Description	Example
boolexpr	Boolean expression	**If** (*boolexpr*) . . . **End if**
buttonvar	Button variable	**BUTTON TEXT** (*buttonvar;strexpr*)
date	4th Dimension date	**Day of** (*date*)
docname	Desktop document name	**DELETE DOCUMENT** (*docname*)
expr	An expression of any type	**SEARCH BY INDEX** « (*fieldname*{= | ±}*expr*«{;*}») »
fieldname	Name for field	**CREATE LINKED RECORD** (*fieldname*)
filename	Name for file	**ADD RECORD**« (*filename*) »
intexpr	Integer expression	**TRUNC** (*numexpr;intexpr*)
numexpr	Numeric expression	**Arctan** (*numexpr*)
numvar	Numeric variable	**GET HIGHLIGHTED TEXT** (*var*|*fieldname;numvar1;numvar2*)
picturexpr	Picture expression	*picturexpr* + *numexpr*
posintexpr	Positive integer	**BEEP**« (*posintexpr*) »
strexpr	String expression	**Ascii** (*strexpr*)
strvar	String variable	**GRAPH** (*var,posintexpr;strvarX;numvarY*)
statement	Logical line of code	**APPLY TO SELECTION** («*filename;*» *statement*)
subfieldname	Name for field in a subfile	**Squares sum** (*subfieldname*)
subfilename	Name for a subfile	**End subselection** (*subfilename*)
var	Any variable	**Undefined** (*var*)

Chapter 2

Programming

This chapter discusses the fundamentals of programming in 4th Dimension. It includes

- constants, variables, and arrays
- identifiers
- operators
- basic structures of the language
- procedures, functions, and arguments
- modularizing procedures

People program to automate activities. Programming amounts to writing routines that perform actions (**procedures**) and return values (**functions**). These routines, in turn, are composed of one-line **statements.** You create statements by combining 4th Dimension **keywords** and **commands** with **expressions.** Here are some examples of expressions:

```
42

–5

"Hello"+" "+"World"

Total*1.07
```

The simplest expression is a single **operand** (like 42), although usually an expression contains an operand and an **operator** (like –5) or multiple operators and operands (like the last two examples above). An operand can be a constant or a variable. A **constant** is a fixed value. Program execution does not change it. A **variable** is a name for a place in memory where you can store a value, a value that program execution can change.

Types, constants, and variables

This section looks at data types, constants, and variables.

Data types

Expressions can evaluate to any of five data types:

- Alpha expressions yield a series of Macintosh characters, including alphabetic, numeric, and punctuation characters. (Fields can be typed as Alpha or Text.)
- Numeric expressions yield a numeric object. (Fields can be typed as Real, Integer, or Long Integer.)
- Date expressions yield a calendar date in 4th Dimension format.

- Boolean expressions return either True or False.
- Picture expressions contain Macintosh pictures.

Constants

A constant is an expression which always has a fixed value. There are three types of constants:

- **"4th Dimension"** is an Alpha type constant.
- **!11/28/1990!** is a Date type constant.
- **123.78** is a Numeric type constant.

Variables

A global variable has a name consisting of a maximum of 11 characters. A local variable name begins with a dollar sign ($), followed by up to 11 characters. The first character of a variable name must be alphabetic. Thereafter, you can use alphabetic and numeric characters, the space character, the underscore, and the period character.

In 4th Dimension, a variable is an object whose type and value can change when your procedures execute. Variables are used to keep intermediate results in the Macintosh RAM which routines can access. You create a variable by naming it in a procedure. 4th Dimension keeps a variable table in RAM. Figure 2-1 illustrates such a table.

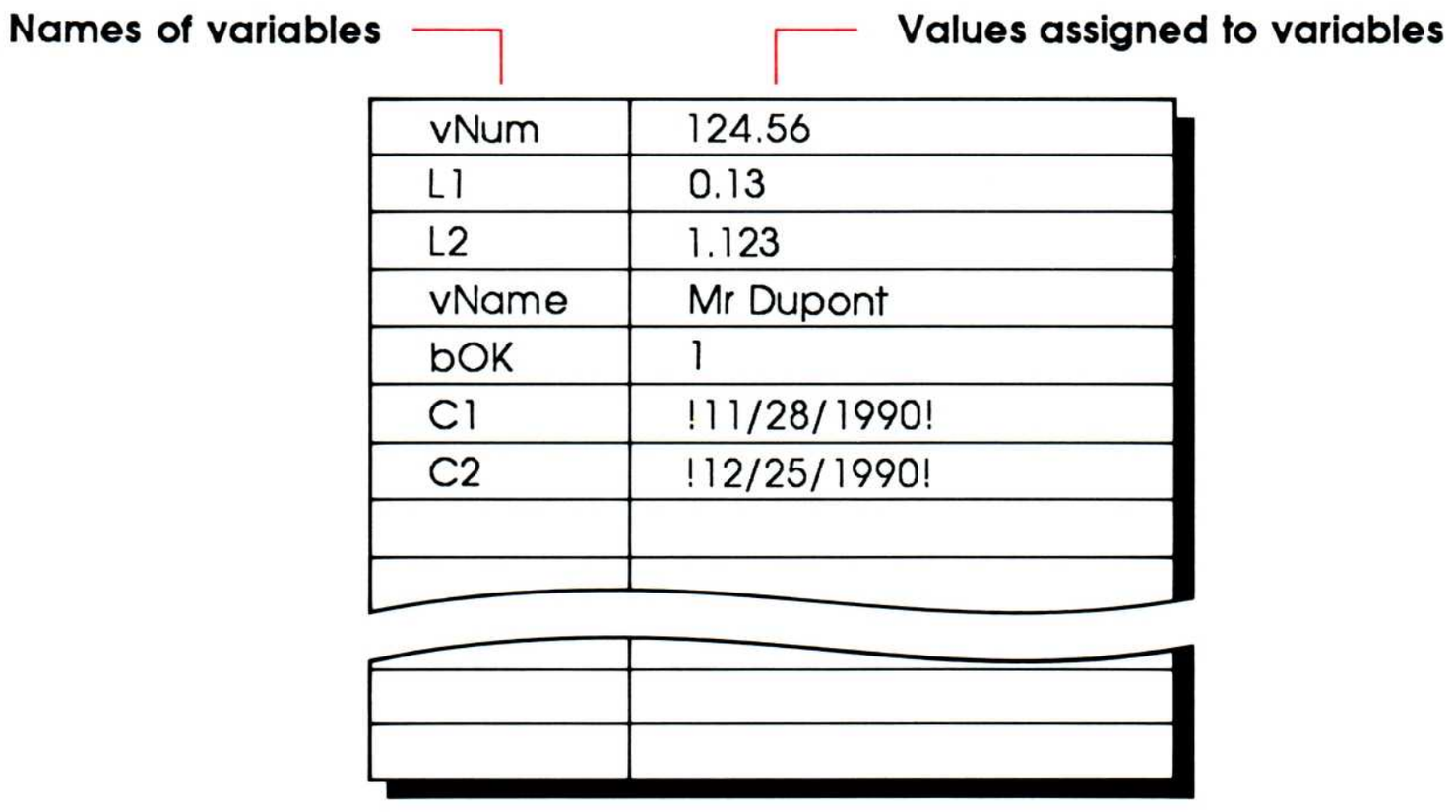


Figure 2-1
Variable table in RAM

If you assign a value to a variable, 4th Dimension puts the value in the variable table and gives the variable the same type as the value. If you assign a value to a non-existing variable, 4th Dimension automatically creates the variable and places it in the variable table. If you try to read a variable to which you haven't yet assigned a value, that variable is considered *undefined.* It contains no value and remains untyped. 4th Dimension provides you with all the necessary programming tools to test whether or not a variable is undefined and to delete variables from RAM to gain memory.

Identifiers

This section describes the various **identifiers** used in the 4th Dimension language. All names for files, fields, and variables follow these rules:

- A name must begin with an alphabetic character.
- Thereafter, the name can include alphabetic characters, numeric characters, the space character, and the underscore character.
- Periods, slashes, and colons are not allowed.
- 4th Dimension will clip any trailing spaces.

Filenames

You designate a **filename** by placing its name between square brackets. The maximum number of characters in a filename is 15.

Examples

[Orders] [Customers] [Letters]

Layout names

A layout name is a string expression. When written as a constant, put a double quotation mark on each side of the layout name. The maximum number of characters in a layout name is 15. You can also write an alphanumeric expression whose value is equal to its name.

Examples

"Input" "Output" "Label" "Dialog" + String (i)

Field names

You indicate a field in one of two ways, depending on the procedure's context. The maximum number of characters in a field name is 15.

In a global, file, or layout procedure, write the field name prefixed by the name of the file to which it belongs.

Examples

[Orders]Total [Customers]Name [Letters]Text

Important

In a global procedure, you must always prefix the field name with the filename.

In a file or layout procedure, you need specify only the field name as long as the field belongs to the current file.

Examples

Total Name Text

Subfields

Subfield naming follows the same principles as field naming.

When writing a file procedure or a layout procedure for the file to which the subfield belongs, you must precede the subfield name with the name of its subfile and an apostrophe.

Examples

Rows'Items Addresses'ZIP Code Keywords'Word

You can write the subfield name without its filename only when working in a subfile procedure or a subfile layout to which subfield belongs.

Examples

Items ZIP Code Word

When writing a global procedure or a layout procedure or a file procedure in another file, you must precede the subfield name with the name of its file, its subfile, and an apostrophe.

Examples

[Orders]Rows'Items [Clients]Addresses'ZIP [Letters]Keywords'Word

A reference to a subfile at any level must include a reference to its parent file(s).

Example

Suppose you have a file named Estimates containing a subfile named Sector which in turn contains a subfile named Row and that Price is a subfield of the Row subfile:

In a Row layout procedure, you access the subfield by typing Price.

In a Sector layout procedure, you access the subfield by typing Row'Price.

In an Estimates layout procedure, you access the subfield by typing Sector'Row'Price.

In a global procedure, you access the subfield by typing [Estimates]Sector'Row'Price.

Variables

A variable is a named area in memory where you store, modify, and retrieve values. A variable is always referred to by its name or, when using indirection, an alphanumeric expression whose value is equal to the variable name. A global variable name can be a maximum of 11 characters. A local variable begins with a dollar sign ($) with the remaining characters (up to 11 allowed) following variable naming rules. The indirection symbols are the section symbol (§) or the curly braces ({}).

Examples

```
GrandTot    A1    §("AA"+String(i))    R{k}
```

In the above examples, using the indirect references § and { }, if i is equal to 23, the name of the variable is AA23, and if k is equal to 52, the name of the variable is equal to R52.

Sets

A set is always indicated by its name placed between double quotation marks (with a maximum of 80 characters) or by an alphanumeric expression whose value is equal to its name.

Examples

```
"Records to be deleted"    "Customer Orders"    "Good"+"Deal"
```

Procedures and functions

A procedure, like a function, is always indicated by its name (with a maximum of 15 characters) or by an alphanumeric expression whose value is equal to its name if you use the **EXECUTE** command.

Examples

```
Add Customer    List Ord    EXECUTE ("PRINT"+String(j))
```

Here, if j is equal to 30, **EXECUTE** ("PRINT"+string(j)) invokes the PRINT30 procedure.

Important

When choosing a name for a field, a variable, a procedure, or a function, make sure you don't use 4th Dimension keywords, such as structure commands **If** or **End case**, procedures or standard functions like **Date**, **Uppercase**, **Time**, or names of system variables like OK or Document.

Constants

An alphanumeric constant is always placed between quotation marks. It is limited to 80 characters.

Examples

```
"Smith"    "Beware! This operation destroys the record!"
```

A numeric constant consists of arabic numerals, a decimal point for non-integer numbers, and a minus sign for negative values.

Examples

```
12732.56    1000    -0.34
```

A date constant is always written as follows: exclamation mark, two digits for the day, slash, two digits for the month, slash, four digits for the year, and an exclamation mark.

Example

```
!6/14/1990!
```

Table 2-1 summarizes 4th Dimension naming conventions.

Table 2-1
4th Dimension naming conventions

Type	Length	Case sensitive
Filename	15	No
Layout name	15	No
Variable name	11	No
Field name	15	No
Procedure name	15	No
Menu title	15	No
Menu item	30	No
Password	13	Yes

Operators

Operators are symbols used to perform calculations. When combined with different expressions, they generate new expressions.

Warning

4th Dimension has a left to right precedence. Only parentheses can override this evaluation. As a result, you must use parentheses to ensure proper evaluation of expressions and statements. Lack of or incorrect use of parentheses can cause either erroneous answers or invalid expressions. For example, 3 + 4 * 5 evaluates to 35. However, if you meant this expression to return 23, you should write 3 + (4 * 5). Likewise, the test expression 50 * 2 = 25 * 4 is invalid. It should be written (50 * 2) = (25 * 4).

Take care to ensure that each left parenthesis character has a matching right parenthesis character.

Numeric expression operators

Table 2-2 shows 4th Dimension's numeric operators.

Table 2-2
Numeric operators

Operation	Symbol	Syntax	Example
Addition	+	*numexpr1* + *numexpr2*	2 + 3 returns 5
Subtraction	–	*numexpr1* – *numexpr2*	3 – 2 returns 1
Multiplication	*	*numexpr1* * *numexpr2*	20 * 2 returns 40
Division	/	*numexpr1* / *numexpr2*	20 / 2 returns 10
Exponentiation	^	*numexpr1* ^ *numexpr2*	2^3 returns 8

String expression operators

Table 2-3 shows 4th Dimension's string expression operators.

Table 2-3
String operators

Operation	Symbol	Syntax	Example
Concatenation	+	*strexpr1* + *strexpr2*	"Pre"+"fix" returns "Prefix"
Repetition	*	*strexpr* * *numexpr*	"AB"*3 returns "ABABAB"

Date expression operators

Table 2-4 shows 4th Dimension's date operators.

Table 2-4
Date operators

Operation	Symbol	Syntax	Example
Difference	–	*datexpr1* – *datexpr2*	!07/10/1990!–!07/17/1990! returns –7, the number of days between the two dates
Addition	+	*datexpr* + *numexpr*	!07/10/1990!+7 returns !07/17/1990!

Comparison operators

Table 2-5 shows 4th Dimension's comparison operators.

Table 2-5
Comparison operators

Operation	Symbol	Syntax	Example
Equality	=	*expr1* = *expr2*	(50 * 2) = (25 * 4) returns TRUE "Good bye" = "Hello" returns FALSE "a"="A" returns TRUE
Inequality	#	*expr1* # *expr2*	(50 * 2) # (25 * 4) returns FALSE "Good bye" # "Hello" returns TRUE

Table 2-5 (continued)
Comparison operators

Operation	Symbol	Syntax	Example
Greater than	>	*expr1* > *expr2*	18 > 10 returns TRUE "B" > "D" returns FALSE
Less than	<	*expr1* < *expr2*	27 < 42 returns TRUE "D" < "B" returns FALSE
Greater than or equal to	>=	*expr1* >= *expr2*	25 >= 25 returns TRUE 25 >= 24 returns TRUE
Less than or equal to	<=	*expr1* <= *expr2*	25 <= 25 returns TRUE 25 <= 24 returns FALSE 25 <= 36 returns TRUE

❖ *Note:* In 4th Dimension, the way to test to see if the case of two characters is different is to compare their ASCII codes. For example, the following statement returns FALSE:

```
Ascii ("A")=Ascii ("a")
```

Logical operators

4th Dimension supports two logical operators: conjunction (AND) and disjunction (OR), both of which work on Boolean expressions. A logical AND returns a TRUE if both expressions are true. A logical OR returns a TRUE if at least one of the expressions is true. See Table 2-6.

Table 2-6
Logical operators

Operation	Symbol	Syntax	Example
Conjunction	&	*boolexpr1* & *boolexpr2*	("A" = "A") & (15 # 3) returns TRUE (5 >= 7) & ("Z" # "ER") returns FALSE
Disjunction	|	*boolexpr1* | *boolexpr2*	("A" = "A") | (15 # 3) returns TRUE (5 >= 7) | ("Z" # "ER") returns TRUE

Picture operators

Table 2-7 summarizes 4th Dimension's picture operators. Picture operators are described in Chapter 8, "Operations on Pictures." Some picture elements are also discussed in Chapter 4, "Layouts."

Table 2-7
Picture operators

Operation	Symbol	Syntax	Description
Horizontal concatenation	+	*expr1* + *expr2*	Move *expr1* to the right
Vertical concatenation	/	*expr1* / *expr2*	Move *expr1* to the top
Horizontal move	+	*expr* + *numexpr*	Move *expr numexpr* pixels to the right
Vertical move	/	*expr* / *numexpr*	Move *expr numexpr* pixels to the bottom
Exclusive superimposition	&	*expr1* & *expr2*	XOR *expr1* and *expr2*
Inclusive superimposition	|	*expr1* | *expr2*	Put *expr2* on top of *expr1*

Three logical functions

4th Dimension offers three logical functions: **True, False,** and **Not.**

Table 2-8
Logical functions

Function	Syntax	Example
Not	**Not**(*boolexpr*)	**While (Not (End selection))** continues the operation until **End selection** returns TRUE
True	**True**	Always returns TRUE
False	**False**	Always returns FALSE

Equality test and assignment operation

It is important to distinguish between the 4th Dimension relational operator, the equal sign (=), and the assignment operator, a colon followed by an equal sign (:=). Some languages, like BASIC, use the equal sign to indicate both equality and assignment.

Use the equal sign to see if two expressions of the same type are equal, that is, contain the identical values (*expr1* and *expr2* must be of the same type). Figure 2-2 shows the equality test, a logical expression that returns either TRUE or FALSE.

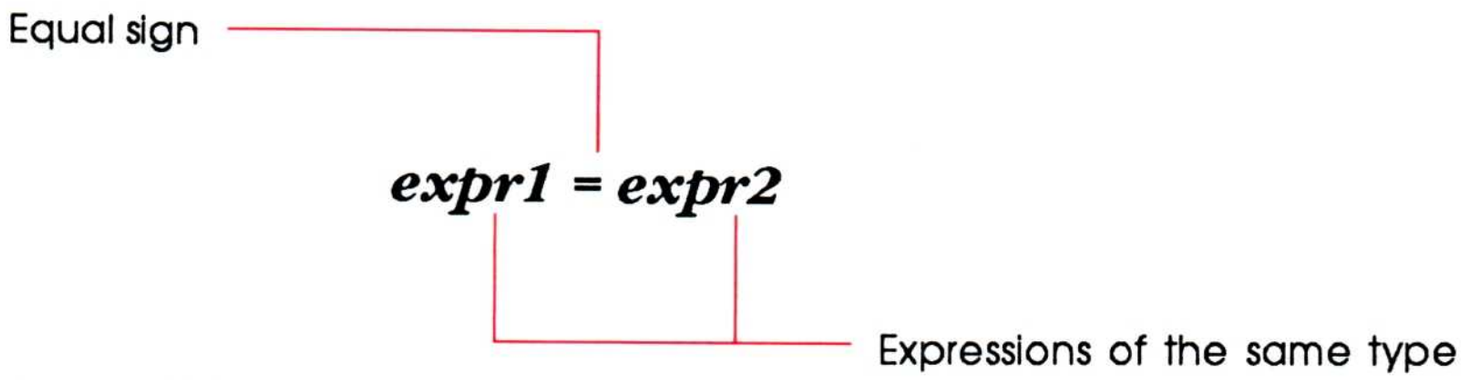


Figure 2-2
Equality test

Examples

[filename]fieldname = "Tax"

This expression returns TRUE if the fieldname field of the filename file contains the string value "Tax".

Substring (var ; 1 ; 1) = "T"

This expression returns TRUE if the alphanumeric type variable var starts with the letter "T" or "t".

Important

4th Dimension evaluates the lowercase and uppercase form of any letter as equal. Thus, "a"="A" returns TRUE.

The assignment operator (:=) assigns or copies the value of the expression to its right into the variable or field to the left of the assignment operator. Figure 2-3 illustrates the assignment operation.

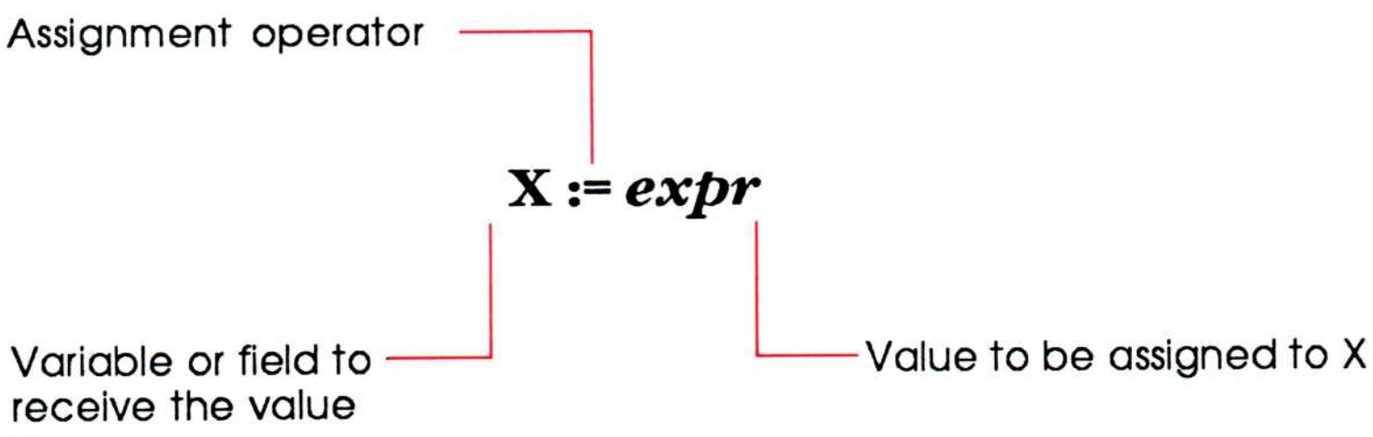


Figure 2-3
Assignment operation

Example

MyVar := **Length** ("Macintosh")

The example places the value 9 (the number of characters in the word *Macintosh*) into the variable named MyVar.

Programming structures

A **routine** is a series of statements which accomplishes a given task. This book uses the term **procedure** to indicate any kind of routine. Strictly speaking, a procedure carries out programmable tasks or actions, whereas a **function** also returns a value. Regardless of the complexity of a programming task, you will always use one or more of three language structures. The three structures are the *sequence* structure, the *branching* structure, and the *loop* structure.

A sequence is a series of statements that 4th Dimension executes one after the other, from top to bottom. See Figure 2-4.

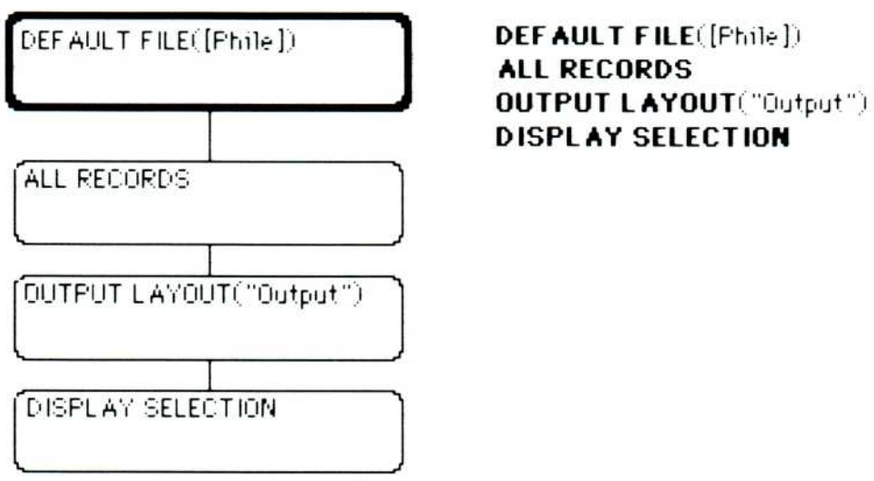


Figure 2-4
Sequence structure

The branching structure tests a condition to determine which of two alternate courses of action to execute. When a logical expression returns a TRUE, 4th Dimension executes the statement(s) following the condition test. If the expression returns a FALSE, 4th Dimension executes any statement(s) following the **Else** keyword. If you omit **Else,** 4th Dimension continues execution with the first statement (if any) following the branch terminating keyword, **End if.** Compare the simple branching structure with the **Case of,** discussed below in the section "Case of: A Structuring Command." See Figure 2-5.

```
DEFAULT FILE([Phile])
ALL RECORDS
If (Records in file=0)
  ALERT("The file contains no records.")
Else
  OUTPUT LAYOUT("Output")
  PRINT SELECTION
End if
ALERT("Operation complete")
```

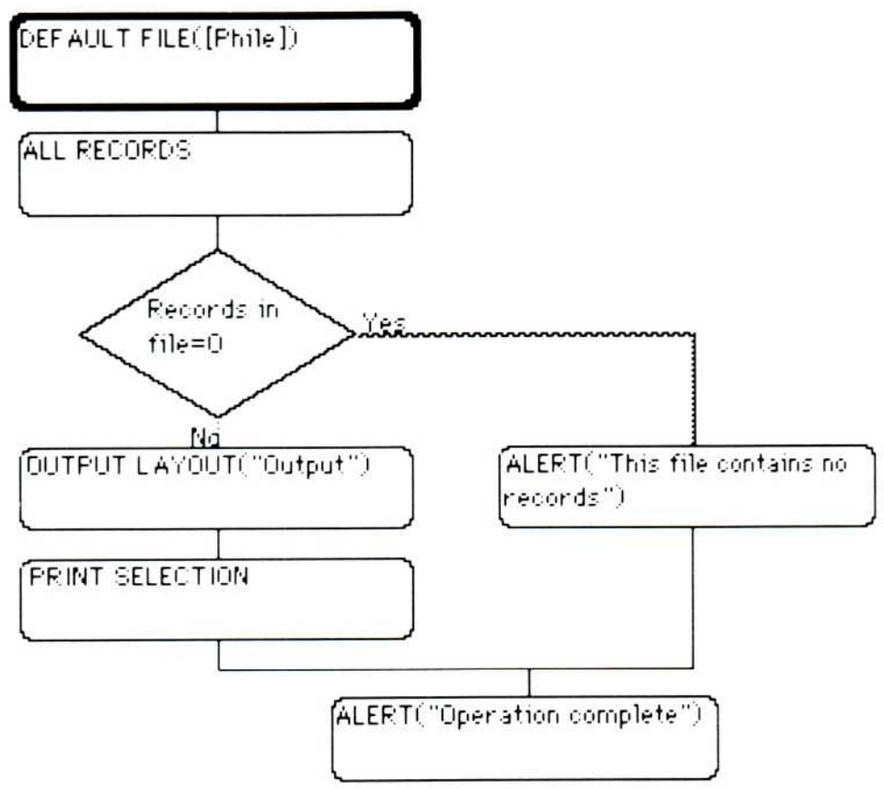


Figure 2-5
Branching structure

The loop executes a sequence as long as a logical expression returns a TRUE. See Figure 2-6.

```
MyStr :="Hello"
n :=1
While (n<=Length(MyStr))
  ALERT("The ASCII code for character n° "+String(n)+" is "+String(Ascii(Substring(Mystr ;n ;1))))
 n :=n+1
End while
ALERT("Loop finished")
```

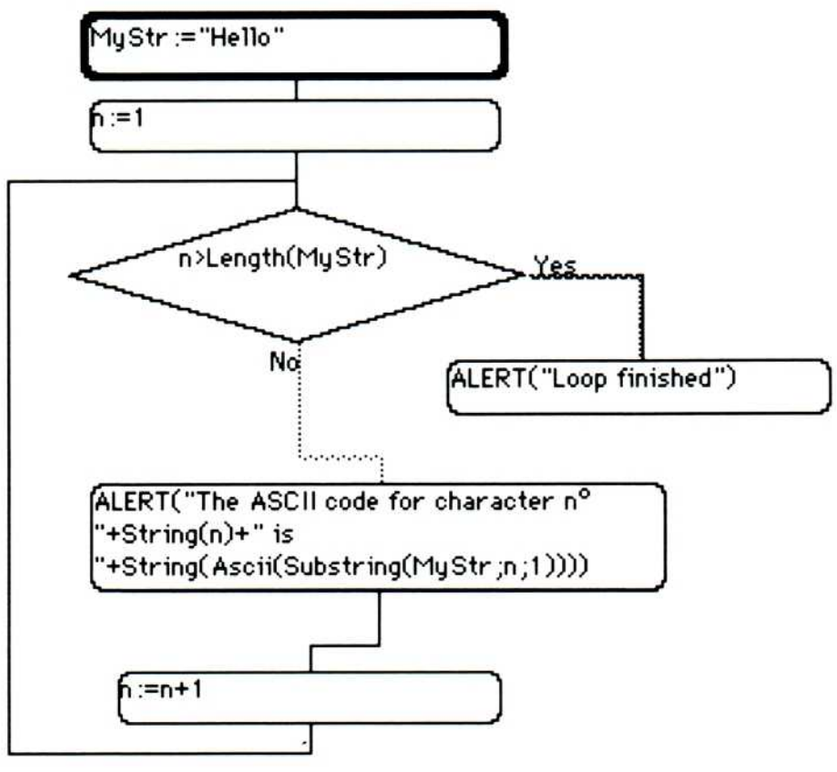


Figure 2-6
Loop structure

Case of: a structuring command

The **Case of** structure works like several **If...Else...End if** commands. It tests a series of Boolean expressions and executes the statement(s) following the first Boolean expression that returns TRUE. If none of the expressions evaluates as TRUE and you have included an **Else** clause, 4th Dimension executes any statement(s) in the **Else** clause. If none of the expressions evaluates as TRUE and you have not included an **Else** clause, 4th Dimension continues execution with the first statement if any, following the **End case** keyword.

You use the **Case of** structure when you want to test several expressions consecutively and only perform an action if one of the expressions is true. **Case of** can be easier to write and it performs more efficiently than a series of **If** statements. The two listings below do exactly the same thing: return the day of the week from a day number. The first uses the **Case of** structure. The second listing uses the **If...Else...End if** command.

```
  `DayCase using Case statement
r:="It must be "
d:=Request("Enter a day number.")
If (OK=1)
   d:=Num(d)
   Case of
     : (d=1)
       r:=r+"Sunday."
     : (d=2)
       r:=r+"Monday."
     : (d=3)
       r:=r+"Tuesday."
     : (d=4)
       r:=r+"Wednesday."
     : (d=5)
       r:=r+"Thursday."
     : (d=6)
       r:=r+"Friday."
     : (d=7)
       r:=r+"Saturday."
     Else
       r:="No such day."
   End case
   ALERT(r)
End if
```

Here's the **If...End if** version:

```
  `DayCase using If statements
r:="It must be "
d:=Request("Enter a day number.")
If (OK=1)
    d:=Num(d)
    If (d=1)
        r:=r+"Sunday."
    Else
        If (d=2)
            r:=r+"Monday."
        Else
            If (d=3)
                r:=r+"Tuesday."
            Else
                If (d=4)
                    r:=r+"Wednesday."
                Else
                    If (d=5)
                        r:=r+"Thursday."
                    Else
                        If (d=6)
                            r:=r+"Friday."
                        Else
                            If (d=7)
                                r:=r+"Saturday."
                            Else
                                r:="No such day."
                            End if
                        End if
                    End if
                End if
            End if
        End if
    End if
    ALERT(r)
End if
```

When 4th Dimension encounters the **Case of** command, it executes only the sequence following the first true case it encounters and then goes directly to **End case.**

❖ *Pascal programmers:* The 4th Dimension **Case of** is more powerful than the Pascal **Case**. While the conditions are related in Pascal, they don't have to be so in 4th Dimension. For example, you can use as test conditions :(A#12) and :(M="aa") in a single **Case of.**

Procedures and arguments

Procedures and functions are two essential parts of a structured language.

Procedures

A **procedure** is one or more statements written to perform one or more tasks. You create a procedure by choosing Procedure from the Design menu while in the Design environment. You can write a procedure in either the Listing (line-by-line) editor or the Flowchart editor. You construct a procedure from flow control keywords, 4th Dimension's built-in commands and functions (routines), assignments, variables, fields, and expressions, and from your own procedures. Once you create a procedure, it becomes part of the language for the database in which you created it.

You can set up a procedure to be called from a menu item by entering the procedure name next to the menu item name in the Menu editor. When the user selects the menu item, 4th Dimension executes the procedure associated with the item. (Selecting an item that has no procedure causes 4th Dimension to leave the Custom environment or runtime program.) You can also call a procedure from another procedure, from a layout procedure, or from a file procedure by typing the procedure name in a flowchart step or in a listing line. Writing modular procedures improves and simplifies the writing of applications.

Structuring and simplifying large procedures

Suppose you need to construct a procedure, called Proc0, that does a number of things. First, study the various aspects of the problem at hand and outline the things you want the procedure to do before coding. Then, build a series of smaller procedures: Proc1, Proc2, and so forth. See Figure 2-7. Writing Proc0 will take less time and its overall structure and legibility will appear clearer if you choose this modular approach. It also makes modification easier.

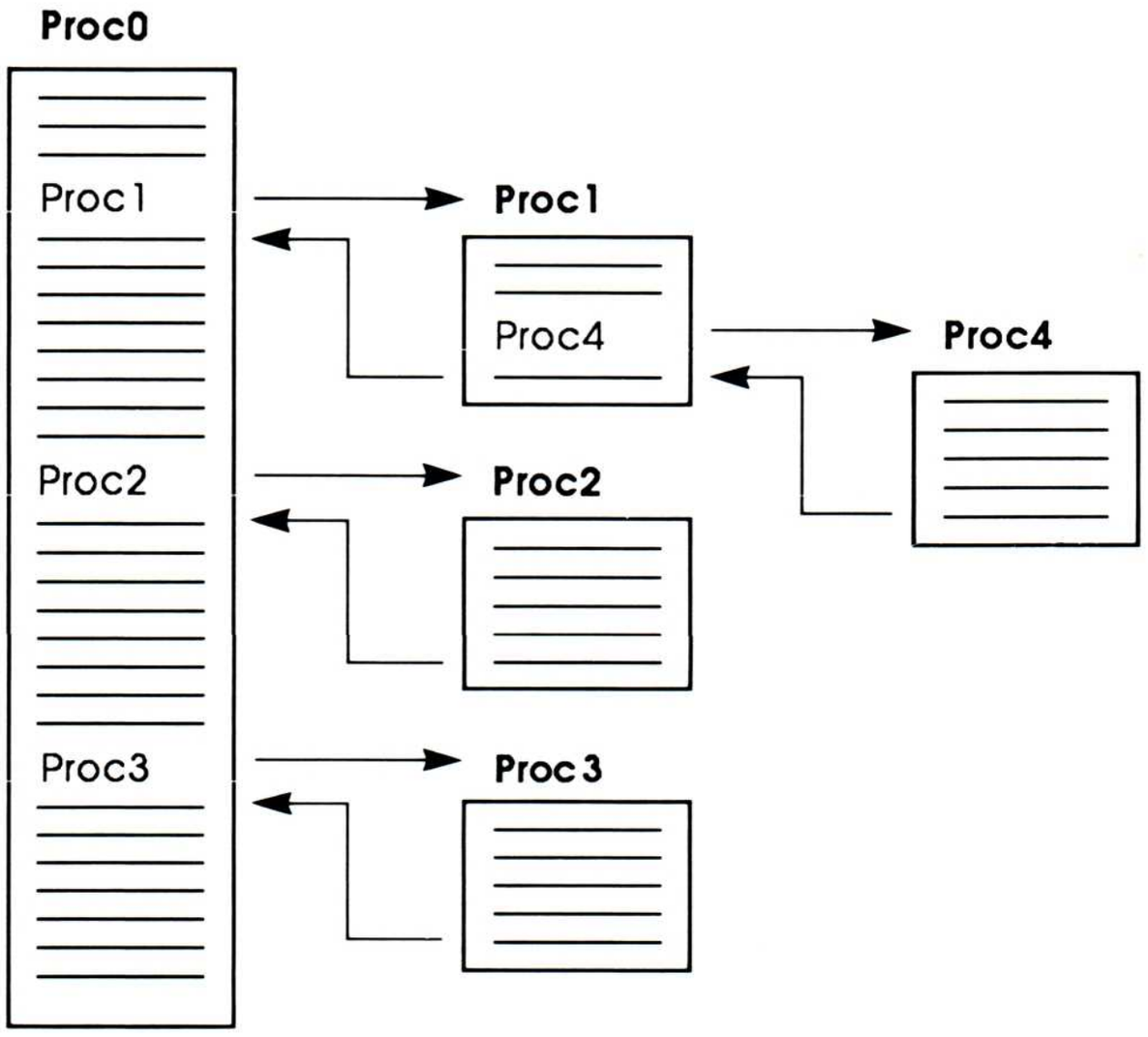


Figure 2-7
Breaking a large procedure into modules

Calling procedures

When a procedure is associated with a menu item, 4th Dimension executes that procedure when the user chooses the menu item or types the corresponding keyboard shortcut.

When you call a procedure from a procedure, 4th Dimension first saves the return statement, which is the statement that follows the calling statement in the calling procedure. 4th Dimension then executes the called procedure. At the end of the called procedure, 4th Dimension returns to the calling procedure and resumes its execution from the return statement.

A procedure called by another procedure can in turn call a procedure and so on. In this way, you can build complex procedures containing many levels. The 4th Dimension language lets you create as many levels as you want. The return statements for the calls are stored in memory. The number of levels you can create is limited by the RAM size.

Modularizing your application

Supposing your database contains 40 files and that you want to implement a special procedure for 20 of those files in order to generate a specific sorted selection and display the selection on the screen or print it on paper. The procedure **filenameX** might look like the one shown in Figure 2-8.

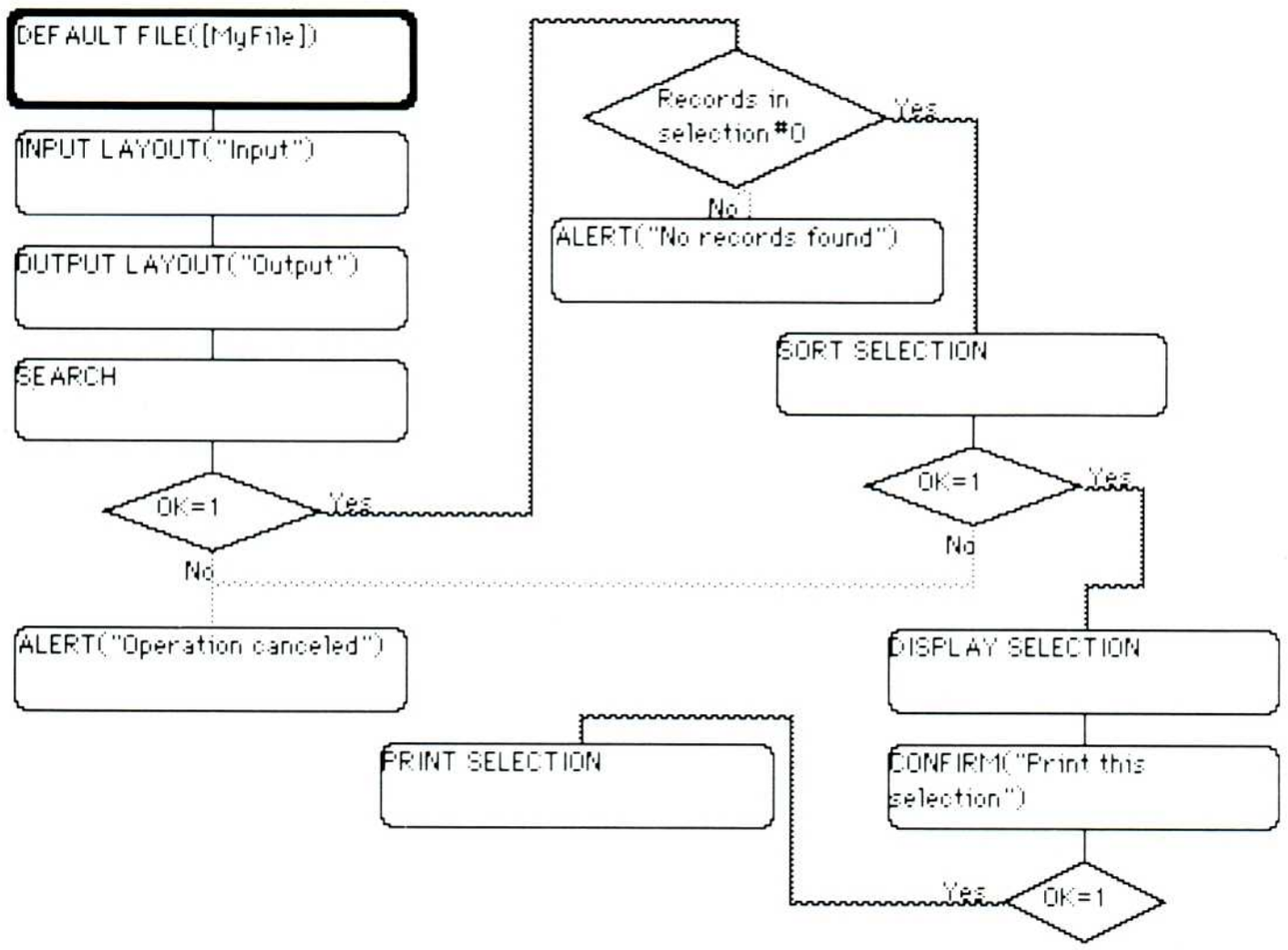


Figure 2-8
Flowchart procedure for sorting and printing

You can see that except for the first three steps, which are different for every file, all the remaining steps are common to the 19 other procedures. You can easily avoid this repetition.

Instead of typing 20 nearly identical procedures, you create a procedure containing the part of the application they all share and then create 20 procedures determining the default file, the input layout, and the output layout of that file, which will call the procedure you can name **MAKE A LIST.**

In the same way, when you plan an application, look for operations that do the same thing and try to reduce them to shared procedures that will be called from other procedures.

This approach has many advantages:

- You save time when you write your applications.
- You gain disk space. In the above example, 20 identical procedures would have taken up 280 steps and tests. By calling a common procedure, you have 11 (common procedure) + (3 + 1) * 20 (three different steps, plus the call for the shared procedure for the 20 different procedures), which comes to 91 steps and tests. This means that you have three times fewer steps to write.
- Debugging takes less time.
- Your code is more legible and easier to maintain.

Suppose that in the above example, you write = instead of # in the (Records found # 0) test. You've already created 20 identical procedures when you realize while in the Custom environment that you made a mistake. You now have to correct 20 procedures. If you had created a common procedure, you would have had to correct only one procedure.

Once a procedure is corrected and works smoothly, it can later be called by other procedures and you know for sure that it will work. You won't have to go into debugging at all. See Figure 2-9.

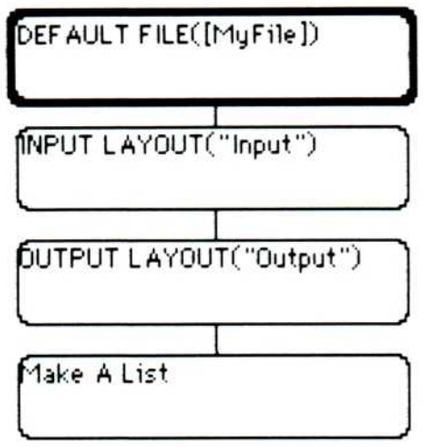


Figure 2-9a
Procedure that calls MAKE A LIST

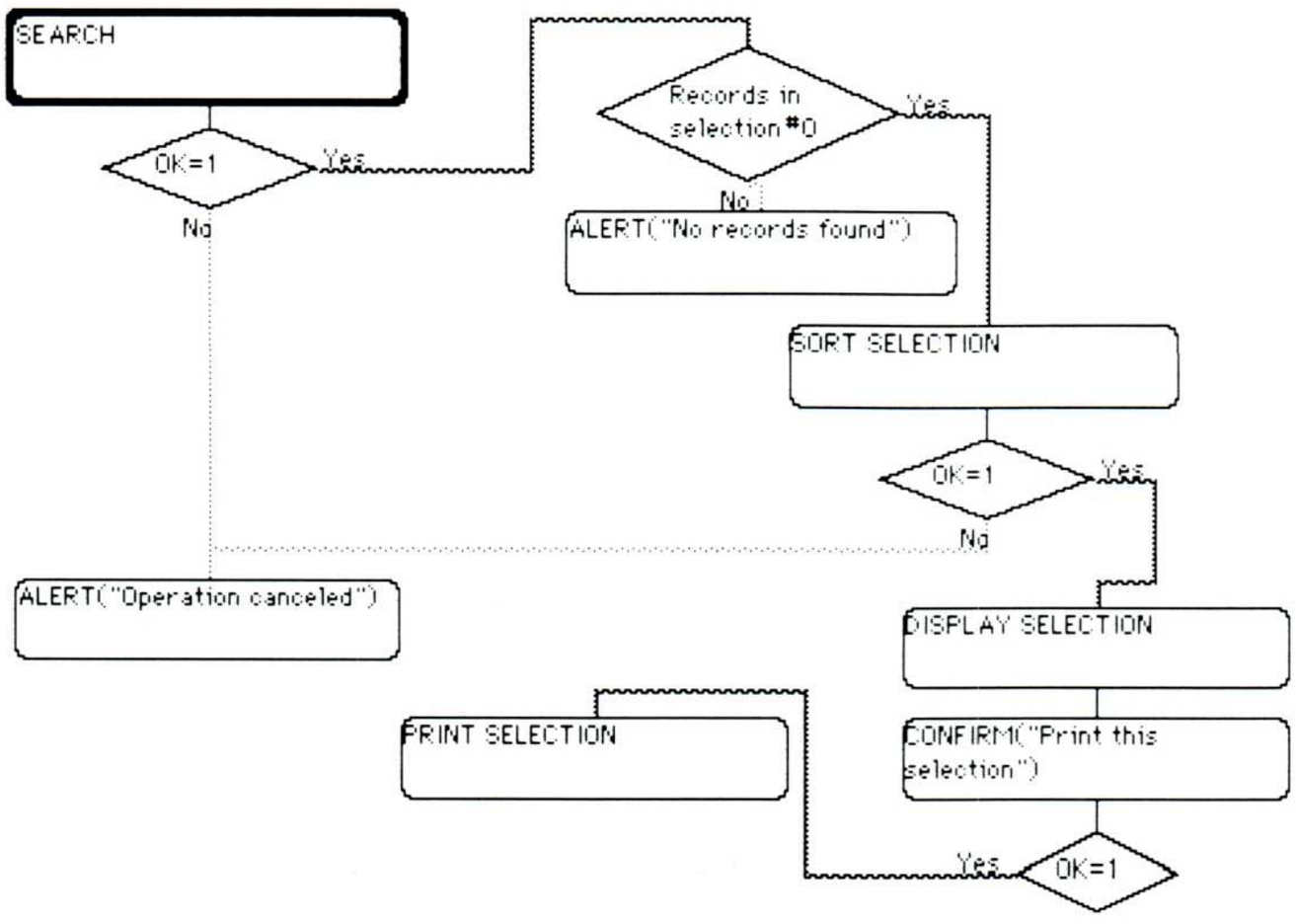


Figure 2-9b
Flowchart for MAKE A LIST procedure

Parameter passing

You can further improve the legibility and simplicity of your application by passing arguments.

To pass arguments to a procedure, specify the procedure name followed by the list of arguments in parentheses, separating each argument with a semicolon:

```
Proc (Expr1 ; Expr2 ; ... ; Exprn)
```

When you pass arguments to a procedure, 4th Dimension stores the arguments in memory. The number of arguments you can pass to a procedure is only limited by the RAM size.

In a called procedure, the passed arguments become parameters with local variable names $1, $2, and so on through $n. $1 stands for the first argument, $2 for the second argument, and so on.

If in the MAKE A LIST example you explicitly indicate that the input layout name will be passed as first argument and the output layout name as second argument, the procedures will look like the one shown in Figure 2-10.

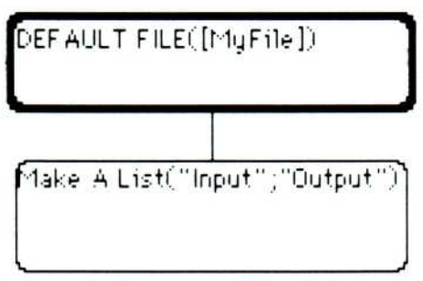


Figure 2-10a
Procedure that calls MAKE A LIST

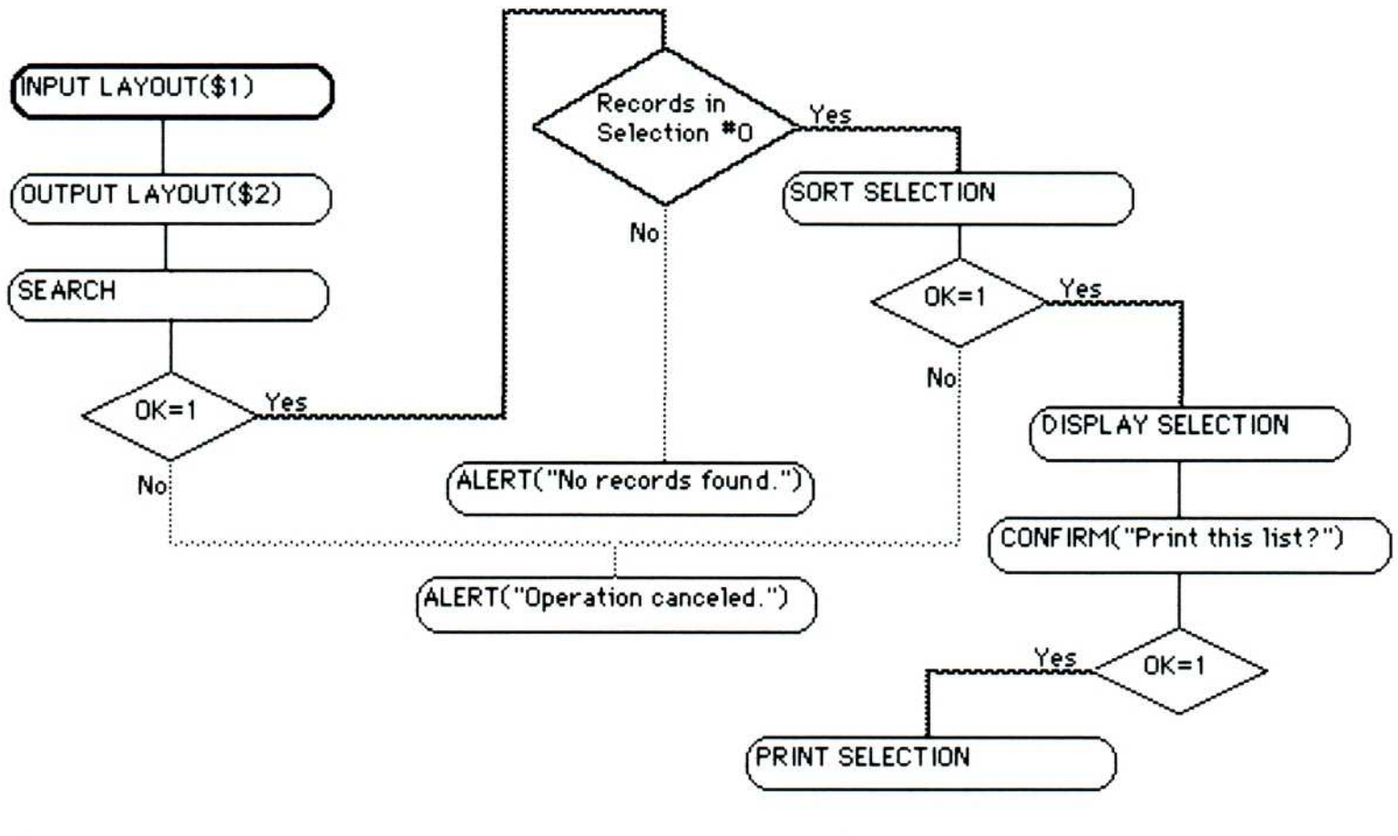


Figure 2-10b
Flowchart for new MAKE A LIST procedure

Scope of arguments, global variables, and local variables

Because 4th Dimension supports both global and local variables, it becomes important to understand the difference between the two and when you should use each type for best programming results. It is also important to understand that 4th Dimension manufactures its own local variables, while allowing you to create local variables. As an example, suppose a procedure contains a call to a procedure:

```
If(MyVar >1000)
  START ALERT (MyVar)
End if
```

With the START ALERT procedure written as follows:

```
ALERT (String ($1) + " is too large! ")
```

When you pass the argument to the START ALERT procedure, 4th Dimension automatically creates a new variable to which it copies the value of the argument and specifies the argument type. This new variable is only known to the START ALERT procedure and referenced as $1. It is a local variable.

If you were to change the $1 variable in the START ALERT procedure by assigning it a new value, the variable alone would be affected, and not the passed argument. The $1 local variable only exists in the START ALERT procedure. At the end of the procedure, 4th Dimension automatically clears the $1 variable from memory before returning to the calling procedure.

Conversely, the MyVar variable is a global variable. It is known to all procedures for as long as it exists in memory. (The **CLEAR VARIABLE** command and quitting a database clear global variables.) If START ALERT were to assign it a new value upon resuming the calling procedure, the value of the MyVar variable would be changed and might even be of a different type.

Generating a local variable when passing an argument is an overall 4th Dimension concept: you explicitly create local variables for a procedure or a function by placing a dollar sign ($) before a variable name. (The variable name must still begin with an alphabetic character.) If you create a local variable in a procedure, it will only exist in that procedure and will be cleared from memory at the end of the procedure.

Figure 2-11 illustrates a procedure calling a second procedure. Global and local variables are involved.

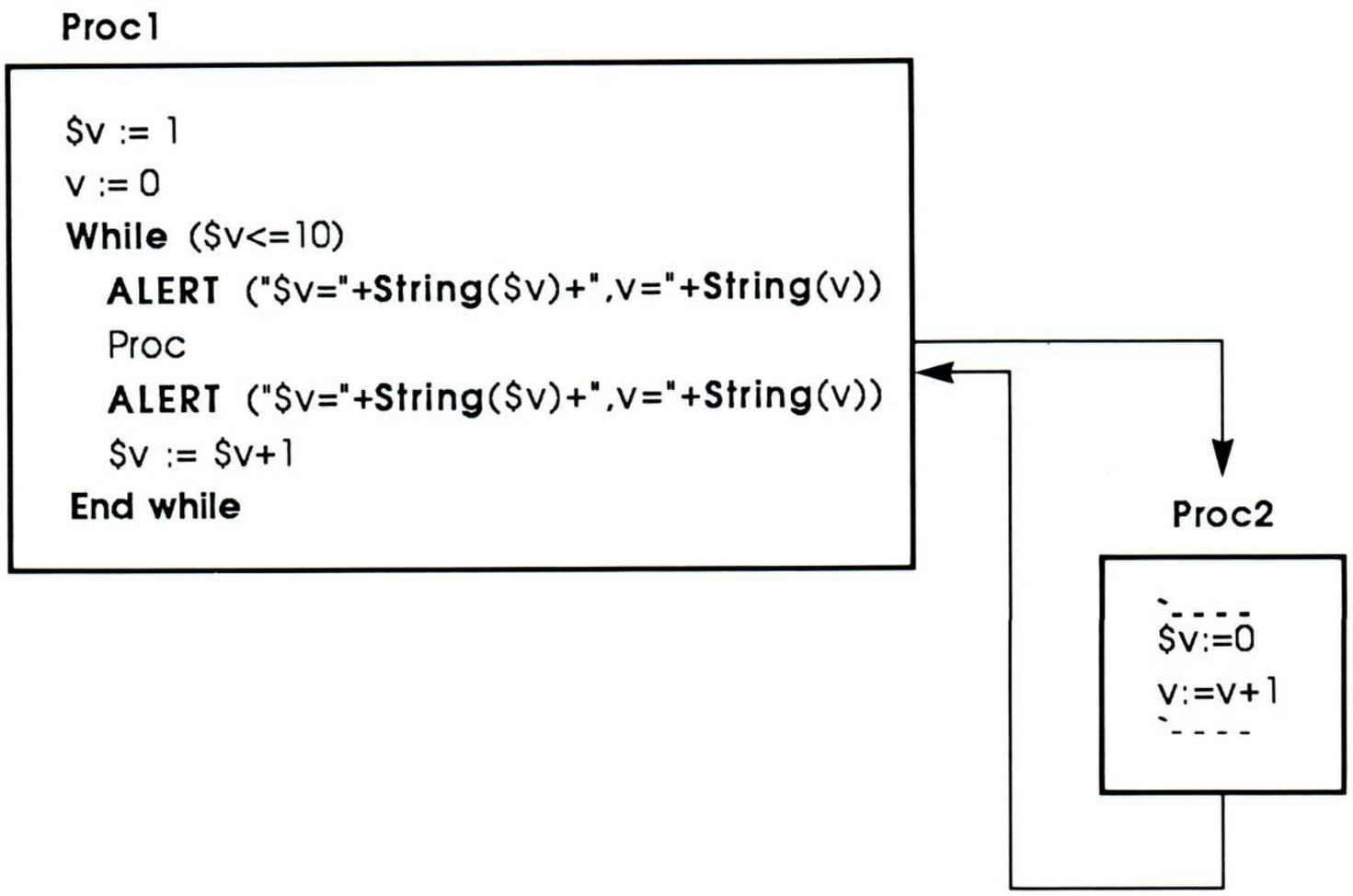


Figure 2-11
Global and local variables in two procedures

In the **Proc1** procedure, you use two variables: **$v** and **v**. The variable **$v** is a **Proc1** local variable, while **v** is a global variable. In the **Proc2** procedure, you use two variables: **$v** and **v**. The variable **$v** is a **Proc2** local variable, while **v** is a global variable. The **$v** variable in **Proc1** is different from the **$v** variable in **Proc2**. When **Proc1** is called, **$v** is created and it will only be deleted from memory at the end of the procedure. When **Proc2** is called, a new copy of **$v** is created and will be deleted from memory when returning to **Proc1**. The **v** variable is a global variable common to both procedures. When **$v** is reset in **Proc2**, **$v** in **Proc1** remains unchanged. Upon execution, subsequent alerts will display the values listed below:

1st passage in the loop:

$v = 1, v = 0

$v = 1, v = 1

2nd passage in the loop:

$v = 2, v = 1

$v = 2, v = 2

10th passage in the loop:

$v = 10, v = 9

$v = 10, v = 10

Thus, local variables and global variables are different kinds of variables, and any confusion between the two may lead to programming errors. If in a procedure called up in certain cases (following tests in the calling procedure), you change the value or the type of a global variable that will be used when returning to the calling procedure, you'll get error messages from time to time which may seem random but which are quite logical.

You'll then have to find out in which cases the procedure is called upon and in which cases it generates an error message only to realize that you are modifying a variable. However, if you want a procedure to modify or return certain values and avoid the above-mentioned problem, you should use a function.

As a general rule, use local variables whenever possible. This practice greatly simplifies modularization and program logic. Local variables reduce the unnecessary duplication of variables that makes complex procedures difficult to read and prone to errors. Use global variables primarily when calling a series of procedures between which parameters will be passed.

Functions

A **function** is a routine that returns a value. You create a function by choosing Procedure from the Design menu while in the Design environment. Built-in 4th Dimension functions begin with an initial capital letter with the rest of the name in lowercase. For example, **Current date.**

A function is always called by a global procedure, a layout procedure, a file procedure, or another function. Because a function returns a result, it can only exist in an expression or to the right of the assignment symbol.

Just as with procedures, you can pass arguments to a function. The arguments are named $1, $2, through $n. The result returned by the function is named $0 inside the function. A function can call a procedure or another function. A user-defined function acts just like a built-in function. That is, you write a statement containing the name of the function and giving its necessary arguments, if any.

Example 1

You are going to create Initial, a function returning the first character of an alphanumeric expression. For example, the statement Initial ("Foo") returns "F".

```
$0 := Substring ($1 ; 1 ; 1)
```

Thus, you may write in your procedures

```
var := Initial ([MyFile]MyField)
```

and write a test like

If (Initial ([MyFile]MyField) # Initial ([YourFile]YourField))

Example 2

You've retrieved data from an application in which all records have a fixed size. "Whiter", for instance, is a 70-character alphanumeric field, which takes up 70 characters: 6 letters followed by 64 spaces. In 4th Dimension, fields take up only the space they need, and "Whiter" in an alphanumeric field would take up 7 characters: 6 letters plus 1 byte for the length.

To recover the space taken up by the 64 trailing blanks, you need to delete spaces at the end of the string. To do this, create a function named Clean up:

```
`Clean up function to delete trailing spaces
vChar := " "
n := Length ($1)
While ((vChar = " ") & (n > 0))
    vChar := Substring ($1 ; n ; 1)
    If (vChar = " ")
        n := n - 1
    End if
End while
If (n = 0)
    $0 := ""
Else
    $0 := Substring ($1 ; 1 ; n)
End if
```

If you invoke Clean up with the argument "spaces5 ", it returns "spaces5".

Tips for writing applications

You can edit procedures either in the Flowchart editor or in procedure (listing) mode. Flowcharts are helpful for beginners, because the graphic depiction of structures helps them rapidly master test and loop structures. For large procedures, however, choose the listing mode; automatic indentation in the Procedure editor lets you see the different levels of the application clearly.

❖ *Incompatibility:* A procedure written in one editor cannot be read or translated by the other editor.

In addition, it's easier to change a large procedure in listing mode, because you can delete or add lines. Changing steps and tests graphically can be quite time-consuming in a flowchart. When writing in the listing form, the results of complex applications are more structured, thus more legible. You can use both forms in a single database.

Whatever the form you choose to write applications, you call a 4th Dimension standard procedure or function by typing its name at the keyboard or by clicking its name in the list displayed on the screen. The same goes for keywords.

The names of 4th Dimension commands are always in uppercase. For example: **DISPLAY SELECTION.**

The names of 4th Dimension functions begin with an uppercase character with the rest of the function name in lowercase. For example: **Current date.**

You call a procedure or function you created by typing its name on the keyboard or by clicking its name in the list displayed on the screen. Your global procedures appears in italic toward the bottom of the Routines list.

If you type the name of a 4th Dimension command or function on the keyboard, you can use the "at" sign (@) as a wildcard. For instance, if you type DEF@, 4th Dimension will look for a procedure or a function beginning with DEF. When the procedure or function is found, it's added to your application. Here you'll get **DEFAULT FILE.**

You can insert comments into your procedures, regardless of the form you're working in. Whether in the Listing or Flowchart editor, you begin every comment with a grave accent (`).

❖ *Editor tip:* If you write a statement that creates a syntax error, the editor surrounds the statement with bullets. Fixing the error removes the bullets.

Variable types

You've seen earlier that a variable takes on the type of its assigned value. An alphanumeric variable accepts up to 32,000 characters. For example:

```
Today := "Today is " + String (Current date)
```

A numeric variable accepts any numeric value with up to 19 significant digits. For example:

```
StateTax := [Invoices]Total * .065
```

A date variable accepts 4th Dimension dates between !00/00/00! and !12/31/32000!. For example:

```
GetDate := Date ("01/01/" + String (Year of (Current date) + 1))
```

A Boolean variable accepts any logical argument. For example:

```
v := End selection & ([Invoices]Paid # 1)
```

A picture variable accepts a Macintosh picture. For example:

```
vNewHome := [Houses]Pict1
```

The 4th Dimension language introduces arithmetic operations on Macintosh pictures. This aspect is discussed in detail in Chapter 8, "Operations on Pictures."

Variable tables: indirection and index notation

When executing procedures, you may create several variables in memory, some of which pertain to a set of variables such as v1, v2, through v100. To process these variables, you could write 100 lines of code, like this:

```
v1 := expr1
v2 := expr2
.
.
.
v100 := expr100
```

To simplify programming, you can use the **array** or **variable table** constructs instead. Both are forms of **indirection.** You can choose either of two symbolic notations:

§(*strexpr*)

var{*numexpr*}

The §(...) notation introduces the **indirection** concept on variables. The *var*{...} notation introduces the index concept on variables.

When you specify a variable with indirection, 4th Dimension calculates the alphanumeric expression you typed in parentheses to create the name of the variable. The example assigns the four variables to zero.

Example

```
type:="AA.BB.BA.CC"
i:=1
While(i<Length(type))
    §("Part"+Substring(type;i;2)):=0
    i:=i+3
End while
```

Only the § operator allows use of alphabetic operators, whereas {...} takes numeric operators.

When you specify a variable with an index (the curly braces form) in a procedure, 4th Dimension calculates the numeric expression you typed in braces, and the returned value is concatenated with the name of the variable to create the final name.

Example

The instruction Month {**Month of** (**Current date**)} # 0 is valid. **Current date** is a 4th Dimension function returning the date shown on the Macintosh clock. **Month of** is a 4th Dimension function returning the number of the month of the date passed as parameter. Suppose the date is September 11, 1990. **Current date** returns !09/11/90!. **Month of** returns 9. Thus, the instruction is equivalent to Month 9 # 0.

The advantage of indirection or of index notation is that the content of the given expression placed between parentheses or braces may vary.

Example

You want to process the variables v1, v2, through v100. You write a loop:

```
i := 1
While (i < 101)
   v{i} := expr
   i := i + 1
End while
```

Chapter 3

Files

This chapter discusses files and their components, records and fields. These essential database elements are defined as follows:

- A file is a finite set of records.
- A record is a finite set of fields.
- A field is a category of data, in which the data has a specific type.

As an example, think of a teacher's list of students. The list is a finite set of records, with each record containing information about one student. A student's record is, in turn, a finite set of fields with one field each for the student's last name, first name, date of birth, average grade, and so on. Each field contains a specific value that reflects the field's type. For instance, a student's name consists of alphanumeric characters, whereas the average grade field contains numeric values.

Creating a file

In 4th Dimension, you create a new file in the Design environment. Choose the New File command from the Structure menu. The file box represents the file you are creating. You can change the name of a file. Select it by clicking anywhere within the file box; then choose the Rename File command from the Structure menu. To add a field to the file, double-click within the file box or choose the New Field command from the Structure menu. You can also change the type or attributes of a field once you've selected it. Click within a given field to select it; then choose the Change Field command from the Structure menu or simply double-click it. Figure 3-1 identifies the different parts of the file box.

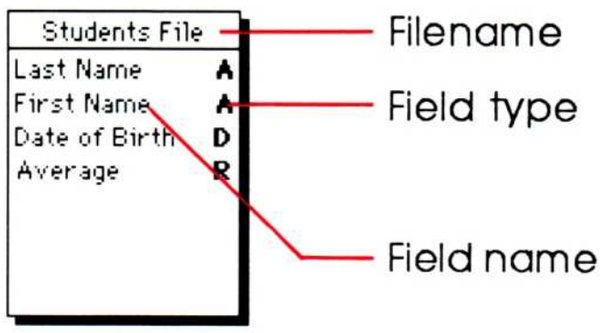


Figure 3-1
Parts of the file box

To delete a field, select it and choose the Delete Field command from the Structure menu. To delete a file, select it and choose the Delete File command from the Structure menu.

Warning

Deleting fields or files considerably alters the structure of your database. Should power fail during this time-consuming operation, the entire database will be lost. Be sure to back up your database before choosing either of these commands.

A file can have as many as 511 fields, including subfile type fields. In turn, each subfile field can have 511 subfields. You can nest subfile fields to a depth of five. See the *4th Dimension User's Guide* for information on nesting subfiles.

Specifying field types

A field must be one of eight types (all shown in Figure 3-2): Alpha, Text, Real, Integer, Long Integer, Date, Picture, and Subfile.

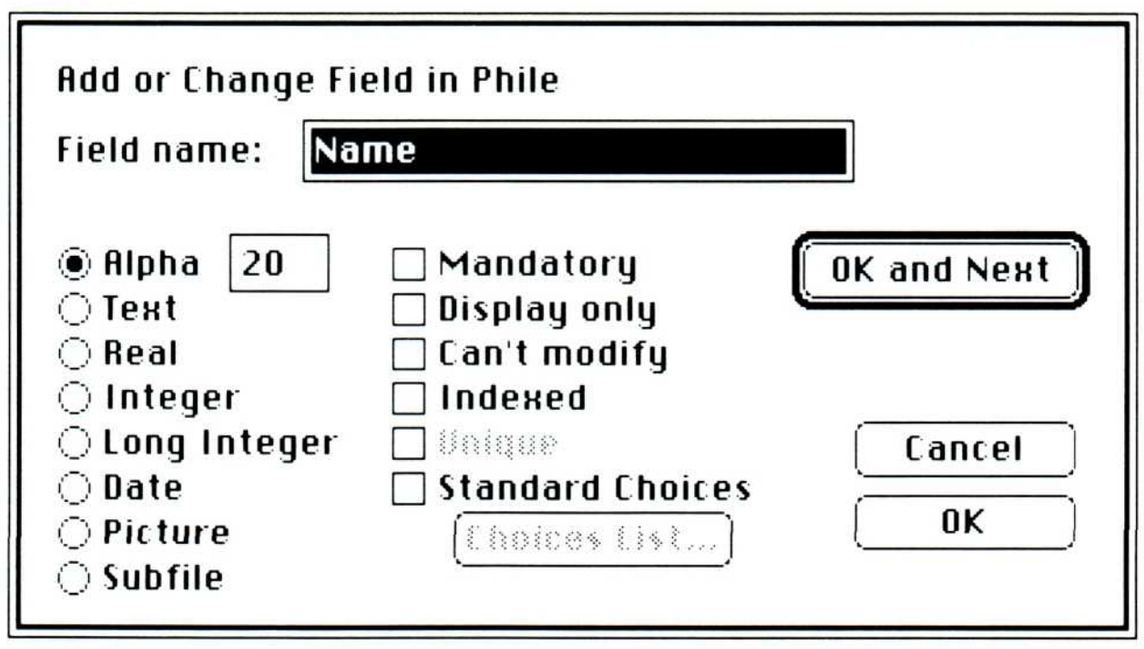


Figure 3-2
Add or Change Field dialog box

Alpha

The user can enter any keyboard character except for Tab, Return, Backspace, or any control character into an Alpha. Maximum length is 80 characters. An Alpha field is not a fixed length field and takes 1 byte per character plus 1 byte for the data length. In the file box, the letter *A* to the right of a field name indicates that the field is Alpha.

Text

The user can enter any character from a Macintosh keyboard into a Text field, except for control characters, Tab, and Backspace. (Return characters can be entered into Text fields, but not into Alpha fields.) A Text field has a maximum capacity of 32,767 characters. A Text field takes 2 bytes plus its length in characters for storage space. *T* is the field type symbol for Text fields.

Important

Procedural calculations convert numbers to reals, regardless of type. When calculated numbers are assigned to fields, the numbers take on the type of the field.

Real

Enter any numeric value in this field up to 19 significant digits. The calculable range for reals is ±1E1022. Reals follow specifications for SANE extended reals. 4th Dimension allots 10 bytes to each Real field. *R* is the symbol for such fields.

Integer

Enter any numeric value in this field in the range of ± 32,767. An Integer field takes 2 bytes. The letter *I* marks Integer fields.

Long Integer

A Long Integer field handles any number in the range of ± 2 billion. A Long Integer uses 4 bytes. *L* is the symbol for Long Integer fields.

Date

A Date field consists of a month (1–12), a day (1–31), and a year (to 32,767). Dates are sorted chronologically. A Date field can manage dates exceeding the Macintosh clock limits. A Date field requires 6 bytes and is marked by the letter *D*. 4th Dimension does not take Gregorian calendar changes (circa 1600 CE) into account.

Picture

A Picture field handles any Macintosh picture, for example, MacPaint bit-map images or MacDraw picture format graphics. *P* is the symbol for Picture fields. A Picture field takes 8 bytes plus the size of its contents on the Clipboard.

4th Dimension retains all of a picture's information. If you copy a picture back into its application, all the information will be there.

Subfile

A Subfile field lets you generate a substructure with subfields (see the "Subfiles" section near the end of this chapter) and is indicated by the asterisk character. You can calculate the storage requirements of a subfile by multiplying the bytes used by fields times the number of subrecords.

Specifying field attributes

You can set any of the following attributes to a field: Indexed, Unique, Mandatory, Non-enterable, Can't modify, and Standard Choices..

Indexed

When you assign a field the Indexed attribute, 4th Dimension creates and automatically updates an index table as records are added, modified, or deleted. Text fields, Picture fields, and Subfile fields cannot be indexed, whereas Alpha, Real, Integer, Long Integer, Date, and fields within subfiles can. Indexing a field creates a 4th Dimension document called *database*.Index*n* on disk, where *n* is the index number.

When calculating the size of a database, include the amount of storage space taken by index tables. Storage space for Indexed fields is as follows:

- Alpha: (Maximum length + 15 or 16)
- Real: 24 (Length + 14)
- Integer: 16 (Length + 14)
- Long Integer: 18 (Length + 14)
- Date: 20 (Length + 14)

- ❖ *Alpha length:* The storage space for an indexed Alpha field is **Maxlength+15**, unless the result is an odd number. In the case of an odd number, the storage requirement is **Maxlength+16**. **Maxlength** is the field length specified in the Design environment, not the number of characters typed into each field.

Indexed field names appear in bold characters in the file box. An Indexed field lets you search for a specific record among any number of records significantly faster than a non-indexed search would. If the search by index activates more than one record, search speed increases logarithmically. That is, on a per-record basis, 4th Dimension finds ten records quicker than just one. You can sort Indexed fields far faster than non-indexed fields. Index the fields that you will frequently search or sort to save time.

Unique

Unique fields must be indexed. Giving a field this attribute makes it impossible to enter a duplicate value for the field within the file. 4th Dimension will alert the user if the user tries to enter a duplicate value.

Mandatory

4th Dimension will not save a record if the user has failed to make an entry in this field. Use the Mandatory attribute for required data. 4th Dimension considers a field empty if it contains no character at all or if a field contains a "non-value value," as follows:

- □ Alpha: "" (null string)
- □ Real: 0
- □ Integer: 0
- □ Long Integer: 0
- □ Date: !00/00/00!

Non-enterable

The user cannot enter values into Non-enterable fields. Only 4th Dimension procedures can write values to such fields. Choose Non-enterable to protect fields which will contain values resulting from calculations, like a line total in an invoice.

Can't modify

When a field is Can't modify, 4th Dimension accepts an initial entry into the field, but does not allow keyboard modification of the initial entry after you save it. Only a 4th Dimension procedure can modify a Can't modify field.

Standard Choices

When working in the Design environment, you can create a list that will become active when, during data entry, the user moves into a field bearing this attribute. Figure 3-3 shows the Standard Choices dialog box. The user can choose only one item from among the choices listed for that field for a given record. The user chooses the item by clicking its name, by using the arrow keys, or by typing one of the entries and pressing Return. This attribute does not apply to Picture and Subfile fields. You could use this attribute

- for fields that are restricted to a small group of values, like freight carriers a company deals with (you won't have to type the field value every time you enter the record)
- to automatically control the spelling of complex names

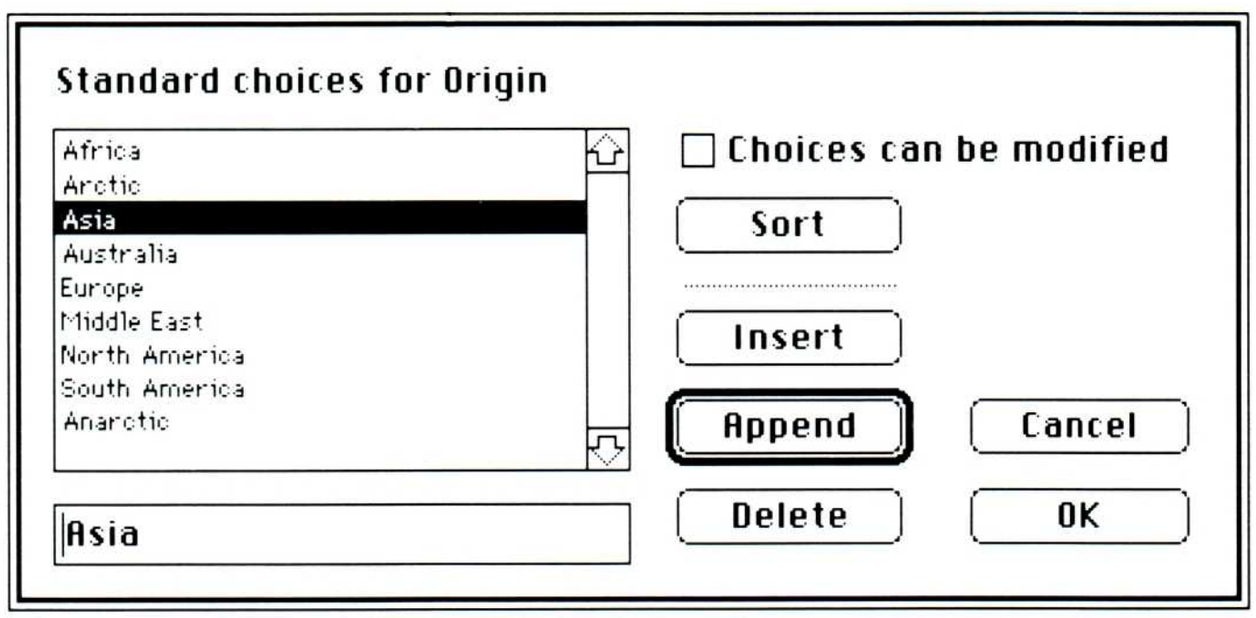


Figure 3-3
Standard Choices dialog box

By selecting the Modify option in the Standard Choices box, users can modify the list of standard choices whenever they enter values in a record.

❖ *Note:* Modify, in this case, is an option, not an attribute.

Keep these things in mind:

- If the list for a field contains several hundred items, or if a single item has to be entered in different fields, use files and links to emulate the list of choices. This procedure is explained later on in this chapter.
- No item on the list can be more than 30 characters long.

Layouts

Layouts determine the way the information contained in the records of a file will be displayed. You can create up to 32,767 different layouts for a single file in a database. You use layouts to enter data, to display data on the screen, to print data on paper, and to create custom dialogs. Layouts are the interface between the user and 4th Dimension's data structures.

In a layout you can display only some of a file's fields and use variables to display fields belonging to another file. Giving a file several layouts enables you to display all or part of its fields in ways that best suit your needs. Figure 3-4 shows an input layout (single record) and an output layout (multiple records). Figure 3-5 shows how these layouts look when in use: an input form with data entered into it, and a printout of three records corresponding to the input.

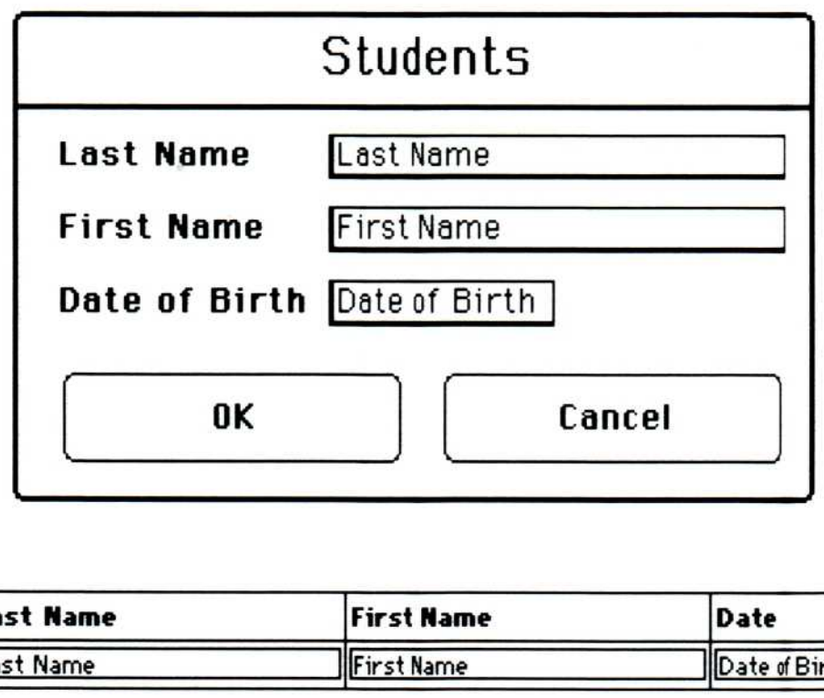
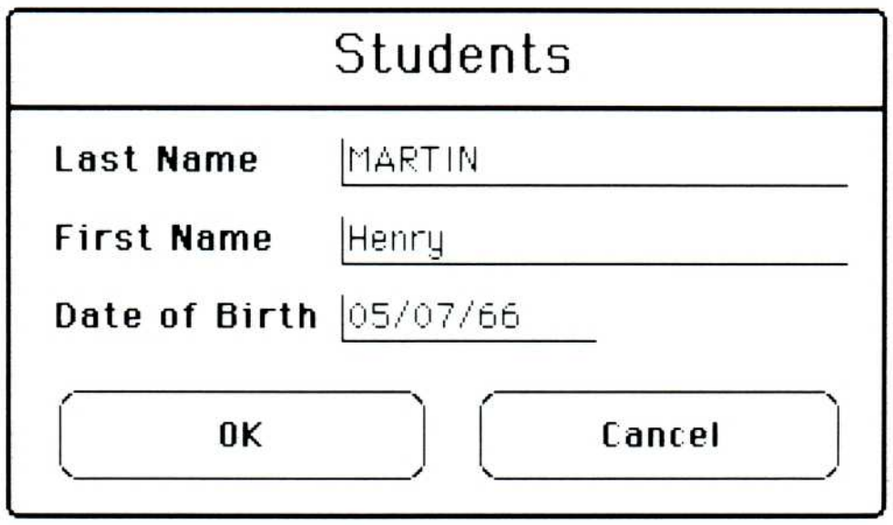


Figure 3-4
Input and output layouts



Last Name	First Name	Date
DAVIS	Jean	03/12/66
MARTIN	Henry	05/07/66
PRICE	Andy	11/05/66

Figure 3-5
Input template and output printout

❖ *Note:* Records and layouts are two different concepts. A record contains data and a layout is the means of displaying that data. Adding, deleting, or changing a field in the Structure window changes the file structure, whereas adding, deleting, or changing a field in a layout only changes its appearance and does not change the file structure or its contents.

Layouts are a powerful 4th Dimension feature. You can create extremely sophisticated layouts and adapt them to printed forms if you wish.

Output layouts and input layouts

You can designate any layout as an input layout or an output layout. Normally, input layouts take data entry and modifications, and output layouts display or report data.

Two rules control the use of layouts in 4th Dimension:

- **Data output:** 4th Dimension uses the current output layout of a file to display or print its contents or to export a SYLK, DIF, or Text file. You must explicitly specify one of the file layouts as the output layout of that file. Use **OUTPUT LAYOUT** in a procedure, or choose the Choose File Layout from the File menu in the User environment, or designate them in the Design environment Layout dialog box.
- **Data input:** 4th Dimension uses the current input layout of the file to enter records or to import a SYLK, DIF, or Text file. You must explicitly specify one of the file layouts as the input layout of that file. Use **INPUT LAYOUT** in a procedure or choose the Choose File Layouts from the File menu in the User environment or designate them in the Design environment Layout dialog box.

  You can specify any layout as the file output layout and/or input layout at any time, according to your specific needs. Each file has its own input and/or output layouts.

Layout procedures and file procedures

You can write procedures that work only when a specific layout is active (**layout procedures**) or when a specific file is active (**file procedures**). A layout procedure becomes active any time the layout (whether an input, output or dialog layout) is called. A file procedure is executed for data entry and modification. Layout procedures are frequently used. File procedures are rarely used.

File procedures

You can have at most one file procedure per file or subfile. You create or open a file procedure by selecting the file in the Design environment's Procedure dialog box and clicking Open. 4th Dimension only executes file procedures for input layouts. Data output operations and dialog layouts do not execute file layout procedures.

4th Dimension always executes a file procedure before each execution of an input layout procedure. The order of execution is

1. Set a particular execution cycle function (**Before, During,** or **After**) to TRUE.
2. Execute the file procedure.
3. Execute the input layout procedure.

When you are reading a description of an input layout procedure, you can assume that if a file or subfile procedure exists, 4th Dimension will execute it before executing the input layout procedure. The two reference manuals assume that you understand the place of file procedures in the execution cycle.

Layout procedures

Write a layout procedure any time you want 4th Dimension to do some kind of processing whenever a particular layout is used. Consider the example of a file of students records. The teacher wants

- last names to be in uppercase even when entered in lowercase
- first name initials to be in uppercase and the remaining characters in lowercase
- to automatically display the corresponding age whenever a date of birth is entered in the Students file, and therefore needs to add an Age variable to the layout

The following layout procedure covers all three requirements. Figure 3-6 shows a filled-in input template with the **Age** variable in place.

```
Last Name := Uppercase(Last Name)
First Name := Uppercase (Substring (First Name ; 1; 1) + Lowercase (Substring (First Name ; 2 ; Length
   (First Name)-1))
Age := Int ((Current date - Date of birth) / 365.25)
```

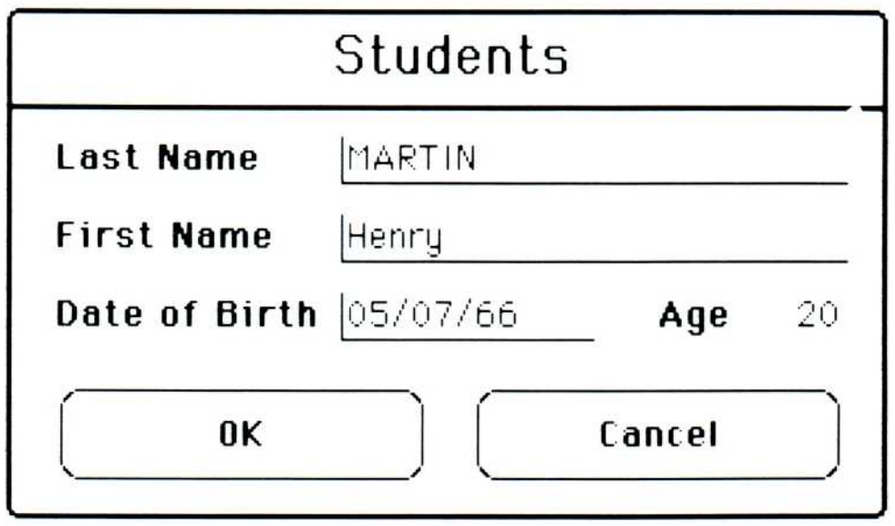


Figure 3-6
Student data input layout with Age variable

Layout procedures and the execution cycle

The 4th Dimension execution cycle consists of phases, phases intiated both by internal processing and user actions. A phase comes into being when 4th Dimension sets a particular function to **TRUE**. Each phase executes the layout procedure. Within the procedure, you can determine which phase is active and take appropriate actions. There are separate execution cycles for input and for output.

The phase functions for input are

- **Before**
- **During**
- **After**

The phase functions for output are

- **In header**
- **Before**
- **During**
- **In break**
- **In footer**

The next two sections show how the execution cycle works for records having no subrecords. The "Subfiles" section below discusses how the execution cycle and subfiles work together.

Execution cycle for input to a record with no subfiles

The user events that can trigger a change of phase during input include

- completing an entry to or a modification of a field by pressing Tab or Return or clicking another field
- clicking a button
- clicking a scrollable or external area
- selecting a menu
- forcing a redraw action by resizing a window, scrolling, or completing a Full Page (included subfile) layout edit

There are two input layouts that do not include subrecords: the simple input layout with no subrecords and the input layout with an included file. The order of phase execution for a simple input layout with no subrecords is as follows:

1. Before phase (before 4th Dimension displays the layout)
2. During phase (for each user event until a validation or a cancel)
3. After phase (only when the user validates the current record)

The order of phase execution for an input layout with an included file is as follows:

1. Before phase (before 4th Dimension displays the layout for the main file)
2. During phase executes the included layout procedure for each included record displayed. 4th Dimension sets **During** TRUE and then draws each displayed record for the included layout procedure.
3. During phase (for each user event)
4. After phase (for the main file only when the user validates the current record)

Here is an example of a simple input layout procedure:

```
Case of
   :(Before)
      If(Entry date = !00/00/00!) `If date is 0 this is an ADD RECORD
         Entry date := Current date
      End if
   :(During)
      If(Modified(Company))
         Company:=Uppercase(Company)
      End if
   :(After)
      Month := Month of (Current date)
End case
```

When **Before** is TRUE and the date is empty, the procedure stores the current date in the **Entry date** field. The user will see the date already filled in when the layout appears on the screen. When **During** is TRUE, the procedure puts the company name in uppercase. When **After** is TRUE, the procedure parses the month number from the current date and stores it in the **Month** field (for sorting by months). When entering data, the user never sees the **Month** field.

Execution cycle for output with no subfiles

There are three basic execution cycles when dealing with output layouts having no subrecords: screen display only, printed output lacking either sorted fields or subtotals, and printed output with sorted fields and subtotals.

With the exception of canceling a printing report, user actions cannot change phases in output layouts.

A display-only output layout has only one phase: 4th Dimension sets **Before** and **During** to **TRUE** simultaneously for each record, before the record is displayed.

The order of phase execution for an output layout with no sorted fields, no subtotals, and no subrecords is as follows:

1. Header phase once for each page (**Before selection** returns TRUE the first time only)
2. Before phase for each record
3. During phase for each record before the record is printed
4. In Break phase once for each page (for level 0 only)
5. Footer phase once for each page (**End selection** returns TRUE for the last footer only)

The order of phase execution for an output layout with sorted fields, subtotals, and no subrecords is as follows:

1. Header phase once for each page (**Before selection** returns TRUE the first time only)
2. Before phase once for each record
3. Test: If there is not enough room on the page to print the record, do both a Footer and Header phase
4. During phase once for each record
5. Test: If the next record will not cause a break, return to step 2

6a. Break (least significant break level)

6b. Test: If no change in the next most significant break level, return to step 2

7a. Break (next most significant break level)

7b. Test: If not at end of selection, return to step 2

8. Break (for level 0 only)
9. Footer (**End selection** returns TRUE for the last footer only)

Here is an example of an output layout procedure with sorted fields:

```
If (Before)
   If (Before selection)
     vPage:=0
   End if
   gQuarter:=[Income]Quarter
   vStr1:=""
   vStr2:=""
End if
If (In break)
   Case of
    : (Level=1)
      vStr1:="Subtotal for Quarter "+String(gQuarter)+": $"+String(Subtotal(Sale))
    : (Level=0)
      vStr1:="Final figures to date....        Maximum: $"+String(Max([Income]Sale))
      vStr2:="                          Total: $"+String(Sum(Sale))
   End case
End if
If (In footer)
   vPage:=vPage+1
End if
```

Current selection and current record

The teacher in the example may want to search student records on the basis of one or more criteria; for instance, all students born after a given date. Such an operation would yield a subset of all records in the file. This subset is called the **current selection.** Unless the current selection is empty, it will have a **current record.**

Selecting a file

You may define a subset of all records in a file as a current selection. Conducting a search results in creating a current selection for the file. Figure 3-7 illustrates the change from all records in a file to the current selection.

When in the User environment, choose the Search, Search and Modify, and Search by Formula commands. When writing procedures, use **SEARCH, SEARCH SELECTION,** or **SEARCH BY INDEX.** If no matching record is found, 4th Dimension returns an empty selection. The current selection can contain all the records of the file. Choose Show All in the User environment or use the **ALL RECORDS** command in a procedure.

Important
Making a selection destroys any sort order. If you want to see records in a particular order, always sort after a search.

The current selection enables you to manipulate a group of records in a file, whether to print them, modify them, perform calculations, or take other actions.

- *Note:* A current selection does not actually contain the matching file records, but only points to them. It is, in fact, a table in memory. The number of records that a selection can contain therefore depends on the RAM size of your Macintosh. A current selection takes up 4 bytes for each selected record.

Students File: 12 of 12

Last Name	First Name	Date	Average
MARTIN	Henry	05/07/66	14
DAVIS	Jean	03/12/66	16
PRICE	Andy	11/05/66	17
DRUMMOND	Jack	02/11/65	16
ELLIS	Patricia	09/07/63	12
SMITH	Julia	09/08/64	15
BAILEY	Henry	09/05/65	13
ANDERSON	Peter	03/09/64	11
RAND	Larry	06/09/63	10
PASCAL	Phillip	07/14/63	13
ALINE	Toni	07/16/64	14
PETERS	Jacquiline	03/19/63	17

ALL RECORDS ((Students File))

Students File: 8 of 12

Last Name	First Name	Date	Average
MARTIN	Henry	05/07/66	14
DAVIS	Jean	03/12/66	16
PRICE	Andy	11/05/66	17
DRUMMOND	Jack	02/11/65	16
SMITH	Julia	09/08/64	15
BAILEY	Henry	09/05/65	13
ANDERSON	Peter	03/09/64	11
ALINE	Toni	07/16/64	14

SEARCH ((Students File);(Students File)Date>=!1/1/64!)

Students File: 5 of 12

Last Name	First Name	Date	Average
ELLIS	Patricia	09/07/63	12
BAILEY	Henry	09/05/65	13
ANDERSON	Peter	03/09/64	11
RAND	Larry	06/09/63	10
PASCAL	Phillip	07/14/63	13

SEARCH ((Students File);(Students File) Average <14)

Figure 3-7
Two search criteria applied to the same file

Current record

When you perform an action on a current selection, 4th Dimension by default starts with the first record in the current selection, which becomes the current record, then goes on to the next, which in turn becomes the current record, and so on to the last record in the selection.

When the search operation results in a current selection containing at least one record, the first record becomes the current record in the file. When the selection is empty, there is no current record. When you add a record to a file, the new record becomes the current record when the user validates it.

The 4th Dimension programming language provides you with commands such as **PREVIOUS RECORD** and **NEXT RECORD** to move the record pointer through a current selection, so that you can select records one by one as the current record.

Selecting a record as the current record automatically loads it into memory. Running a procedure to modify a field amounts to changing the value contained in the current record. To save the modification, you must tell 4th Dimension to save the record, with **SAVE RECORD.** When values are modified during data entry, these modifications cannot be saved unless the user validates the entry.

Sorting

Once you have searched a current selection, you may want to change the order in which records are displayed or printed. This means sorting the current selection. Sorting a selection does not modify its contents. It only changes the way records are arranged within the selection. Once the selection is sorted, the first record becomes the current record.

4th Dimension lets you sort all the records in a current selection using any field type, except Picture and Subfile type fields. You can sort to a maximum depth of 30 fields. You can sort fields in ascending or descending order. When sorting a selection on multiple fields, you may choose either sort order on each field.

Any non-indexed sort operates on the current selection table, ignoring any previous sort order. Within a given sort instruction, 4th Dimension gives priority to the first field to be sorted, then to the second, and so on. The first field sorted is considered a level 1 sort, the second field sorted a level 2 sort, and so on.

Figure 3-8 shows student records in the order entered by a teacher.

Students File: 12 of 12

Last Name	First Name	Date	Average
MARTIN	Henry	05/07/66	14
DAVIS	Jean	03/12/66	16
PRICE	Andy	11/05/66	17
DRUMMOND	Jack	02/11/65	16
ELLIS	Patricia	09/07/63	12
SMITH	Julia	09/08/64	15
BAILEY	Henry	09/05/65	13
ANDERSON	Peter	03/09/64	11
RAND	Larry	06/09/63	10
PASCAL	Phillip	07/14/63	13
ALINE	Toni	07/16/64	14
PETERS	Jacquiline	03/19/63	17

Figure 3-8
Records in the order entered

Figure 3-9 shows the User environment Sort dialog box set to sort the Last Name field in descending order.

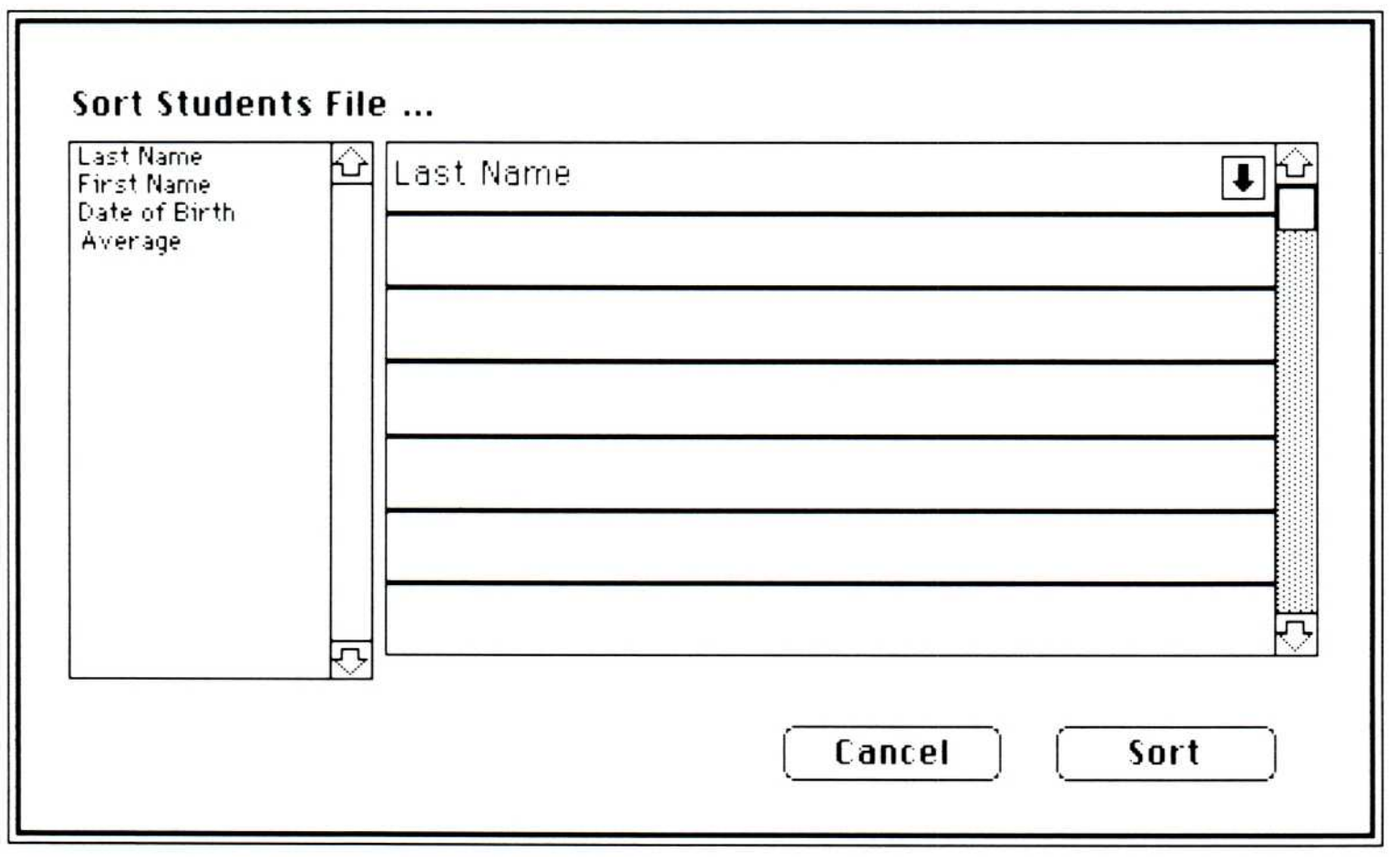
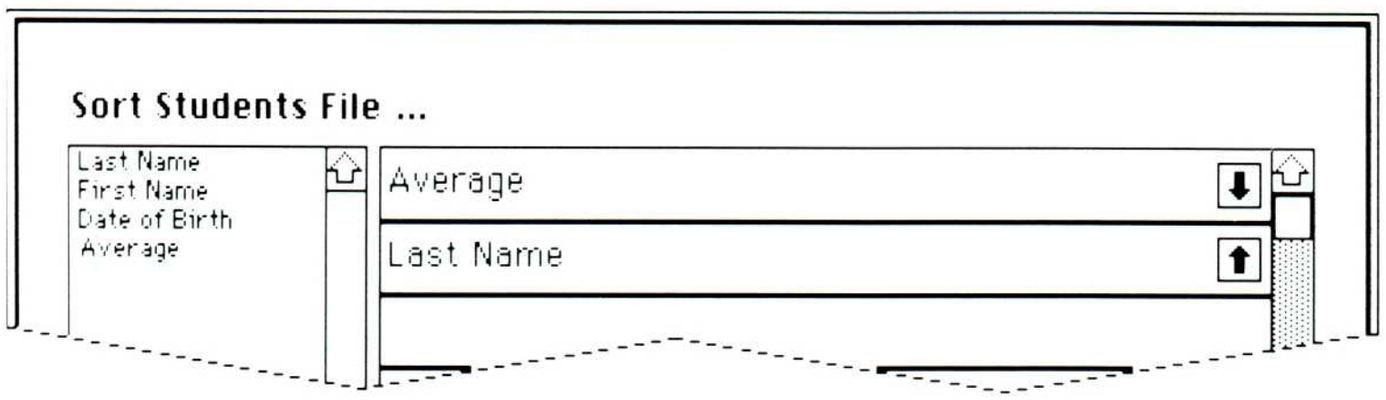


Figure 3-9
Sort dialog box: Sort descending on Last Name field

Figure 3-10 shows a Sort dialog set to sort the Average field in descending order, the Last Name field in ascending order, and the records that appear as a result.



Students File: 12 of 12

Last Name	First Name	Date	Average
PETERS	Jacquiline	03/19/63	17
PRICE	Andy	11/05/66	17
DAVIS	Jean	03/12/66	16
DRUMMOND	Jack	02/11/65	16
SMITH	Julia	09/08/64	15
ALINE	Toni	07/16/64	14
MARTIN	Henry	05/07/66	14
BAILEY	Henry	09/05/65	13
PASCAL	Phillip	07/14/63	13
ELLIS	Patricia	09/07/63	12
ANDERSON	Peter	03/09/64	11
RAND	Larry	06/09/63	10

Figure 3-10
Students sorted with average in descending order and last name in ascending order

Sorting is a dynamic operation. It acts on the current selection. This is the reason any of the following actions may result in a loss of the current sort order:

- changing the current selection
- performing a search
- sorting again
- using a set
- quitting the application

Subfiles

This section describes subfiles, how and where to place them, and how the execution cycle affects them.

Subfiles defined

A **subfile** is, in reality, a field containing a varying number of records. These records are referred to as **subrecords.** Each subrecord contains one or more **subfields.** The maximum number of subfile records per parent record is 32,767 (limited by RAM). This book refers to subfiles as **substructures** when referring to the structure of subfiles and subfields as seen in the Structure window.

The maximum number of subfile levels is five. Whether referring to one field and its subfile or a subfile field containing a second subfile, the book uses the terms **parent** and **offspring.** The parent is the data structure containing the offspring structure. To see how subfiles work, study the following examples.

Subfile example 1

Figure 3-11 shows a file named Items. This file contains the elements (items) of an invoice detail.

Figure 3-11
Detail file for an invoice

Suppose you wanted to store data for a sales report containing detailed information about each item sold: date of sale, quantity sold, and invoice number. You also want to know the total sales per item. Here are the three alternatives for accomplishing this goal:

1. In the first alternative, you could add three fields named [Items]Date, [Items]Quantity, and [Items]Invoice Number to the file. (See Figure 3-12.) In this case, you would have to add a new record to the Items file every time an item is sold and repeat the stock number, description, price, and tax. This solution would be time-consuming and wasteful of storage space, because you would have to enter the same four values every time a particular item is sold. An additional disadvantage is that you would only get the total amount of sales per item by going through every record and adding the sales values.

2. As an alternative, you could create a new file named **Sales** containing three fields: **[Sales]Date**, **[Sales]Quantity**, and **[Sales]Invoice Number**. (See Figure 3-12.) You would still have to enter the item reference number for each item sold and go through all the records referring to the same item to calculate the total quantity sold.

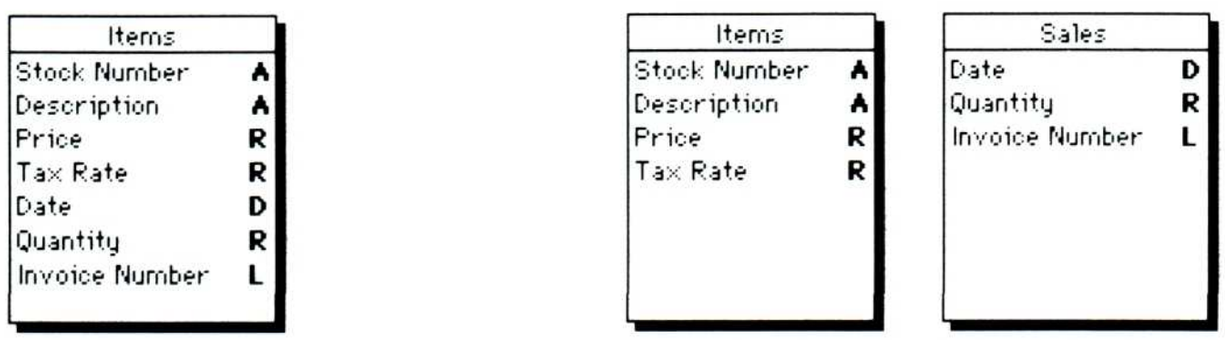


Figure 3-12
Alternatives 1 and 2

3. The third alternative is to insert the list of sales per item within each Part record. The list would have the capacity to insert a number for quantities sold. In other words, this would amount to adding a smaller file within each item record—a subfile.

   To solve the invoice problem, add a field of type Subfile named **[Items]Sales**. You create the actual subfile by adding subfields to **[Items]Sales**.

   These subrecords belong to a given record and will be loaded into memory at the same time as the record to which they belong. A subrecord pointer points to the first subrecord, defining it as the current subrecord.

   This makes it possible to view the quantity sold per item. 4th Dimension has a number of routines to manipulate subrecords. Thus, you can search, sort, create, modify, delete, graph, and perform statistical and arithmetical calculations on subrecords. For example, you can find the quantity sold per item by using the **Sum** function. Figure 3-13 shows the subrecord solution.

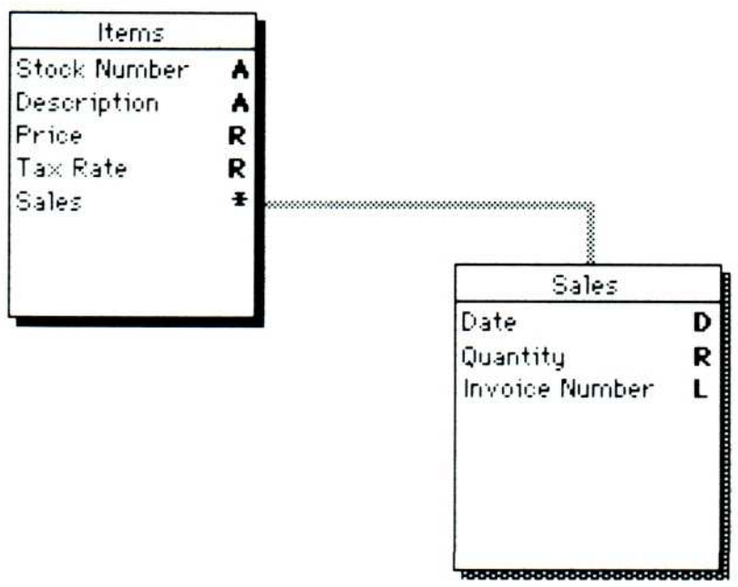


Figure 3-13
Subfile alternative

Subfile example 2

Suppose you want to create a specific file to invoice the above-mentioned items.

This file would handle all necessary information such as the invoice number, date, customer's address, price (tax included, tax not included), and the tax total. You will have to create at least six fields. The next step would be to list all invoiced items, specifying the stock number, description, quantity sold, price, tax, total tax not included, total tax included, and the total tax for each item. You will need to create at least seven fields per item.

Assuming that 20 items are listed on one invoice, you would need to create 166 fields (6 + 8 * 20) to specify the invoice accurately. Whether your invoice contains 2 items (with more than 100 fields the majority of which won't be used) or 21 (you would need another record to add 8 fields and so on), invoicing is quite a chore.

This file structure does not allow easy data management. A solution would be to create two files: one, named **Invoices**, contains invoicing information. Another, named **Items**, containing details of every invoice plus a field in which the number referring to the appropriate invoice is stored. This brings you back to the problem encountered in Example 1. You would also have to go through every detail item in the second file to display or print an invoice.

Typically, a group of items detailed in one invoice should function as a smaller file within an invoice record. Therefore, it's best to create a subfile field named **[Invoices]Items**, which generates a subfile composed of subfields that detail every item on the invoice. The group of items on an invoice is similar to a subfile. Thus, you can create a number of detail items and calculate prices with and without including taxes. See Figure 3-14.

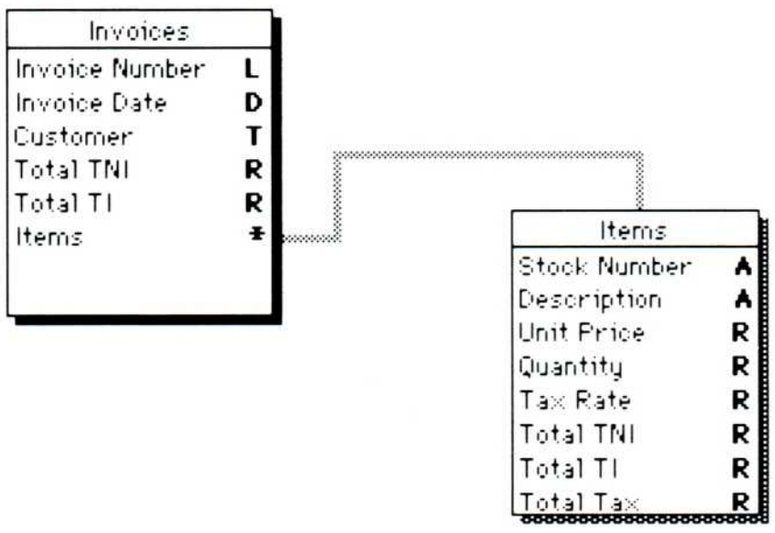


Figure 3-14
Invoice file with its detail subfile

Once you have specified a file and its subfile, you can add a varying number of subrecords within any record of the file. Figure 3-15 shows the structural relationship between records and their subfiles. A given group of subrecords that belong to one file record constitute a subfile.

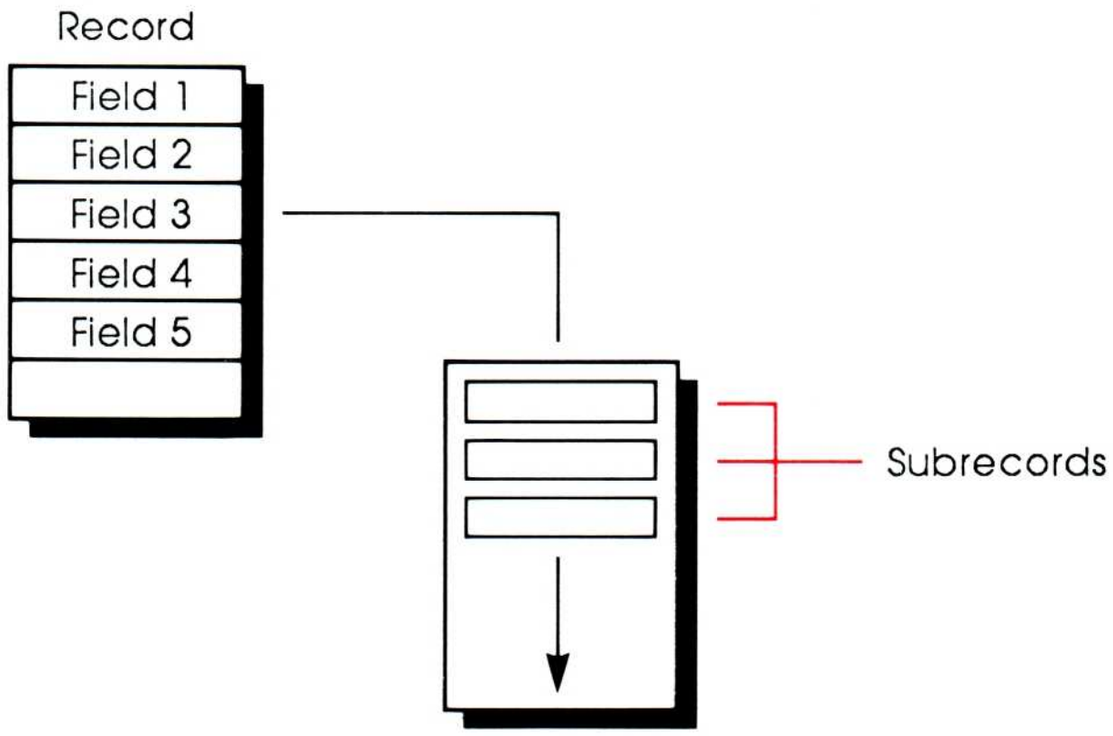


Figure 3-15
Structure of records and their subfiles

Only the subrecords belonging to the current record are in memory at a given moment, rather than all the subrecords of all records in the file or selection. Thus, when you add, delete, modify, search, sort, or print subrecords, you are acting only on a selection of subrecords belonging to the current record. To work on more than one record's subrecords, you would have to work within a loop that would move through a selection of records.

You can search both in the User environment and through a procedure called from a custom menu. In either case, you can search subrecords in two ways: you perform a search within the file with

```
[filename]subfile'subfield = Value.
```

A statement following this template could look like this:

```
SEARCH([Invoice]Item'Part No = "A5430B")
```

In this case, 4th Dimension selects the records in the file containing at least one matching subrecord. This returns a record selection and not a set of subrecords. Figure 3-16 illustrates a search that returns a selection (the current selection) of records based on values in a subfile.

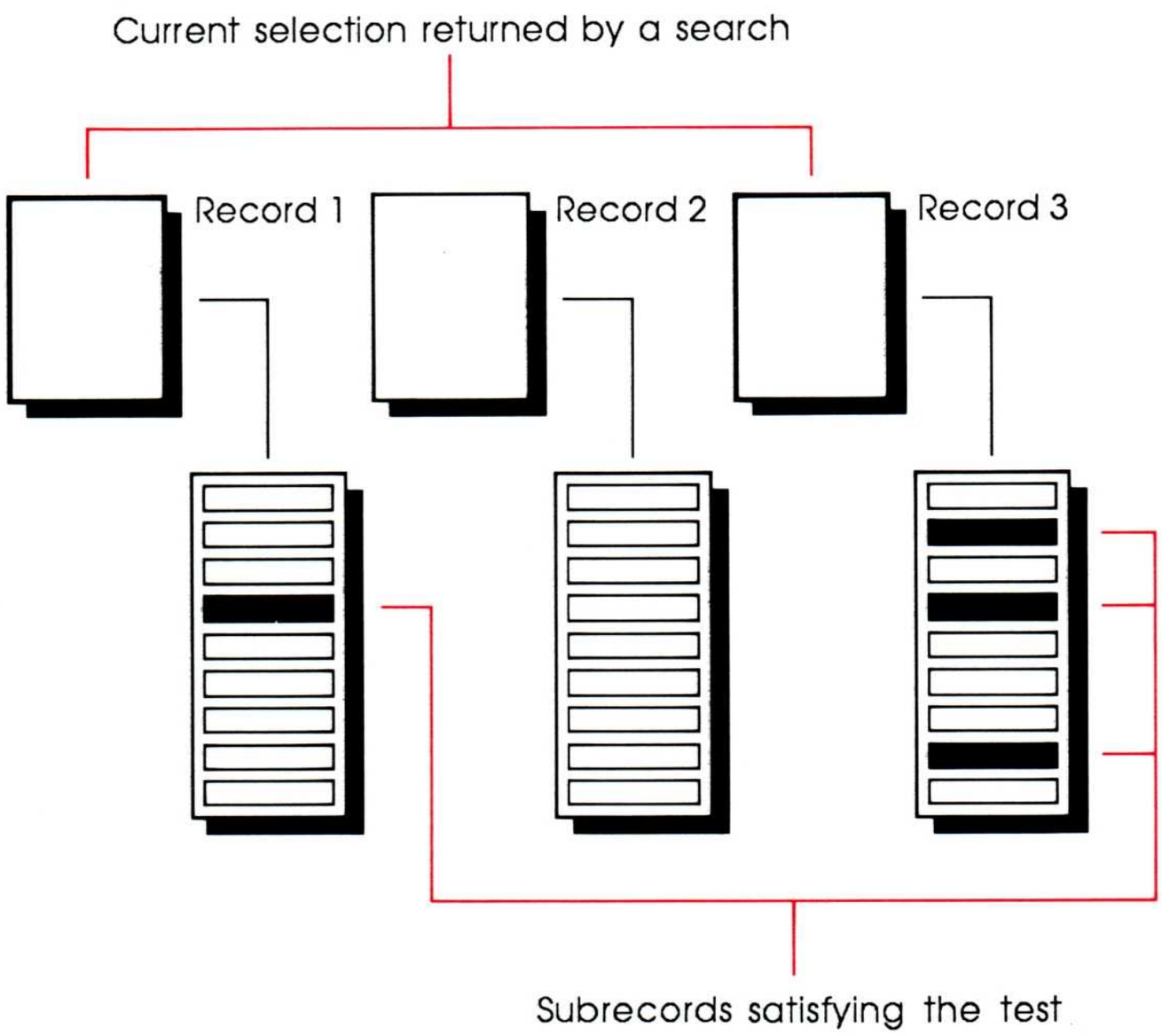


Figure 3-16
Subrecord search returning current selection of records

You can also perform a search only on the subrecords that belong to the current file record. A procedural version of this might look like the following:

```
SEARCH SUBSELECTION([Invoice]Item'Part No = "A5430B")
```

4th Dimension selects matching subrecords, creating a subselection within the subfile. When this subselection is not an empty one, the first subrecord becomes the current subfile subrecord. 4th Dimension provides you with commands like **PREVIOUS SUBRECORD** and **NEXT SUBRECORD** to move through a subselection. Figure 3-17 illustrates a search within a subfile.

Important

Unlike a record selection, a subselection does not automatically have a current subrecord. Always select the current subrecord with **ALL SUBRECORDS** or **FIRST SUBRECORD.**

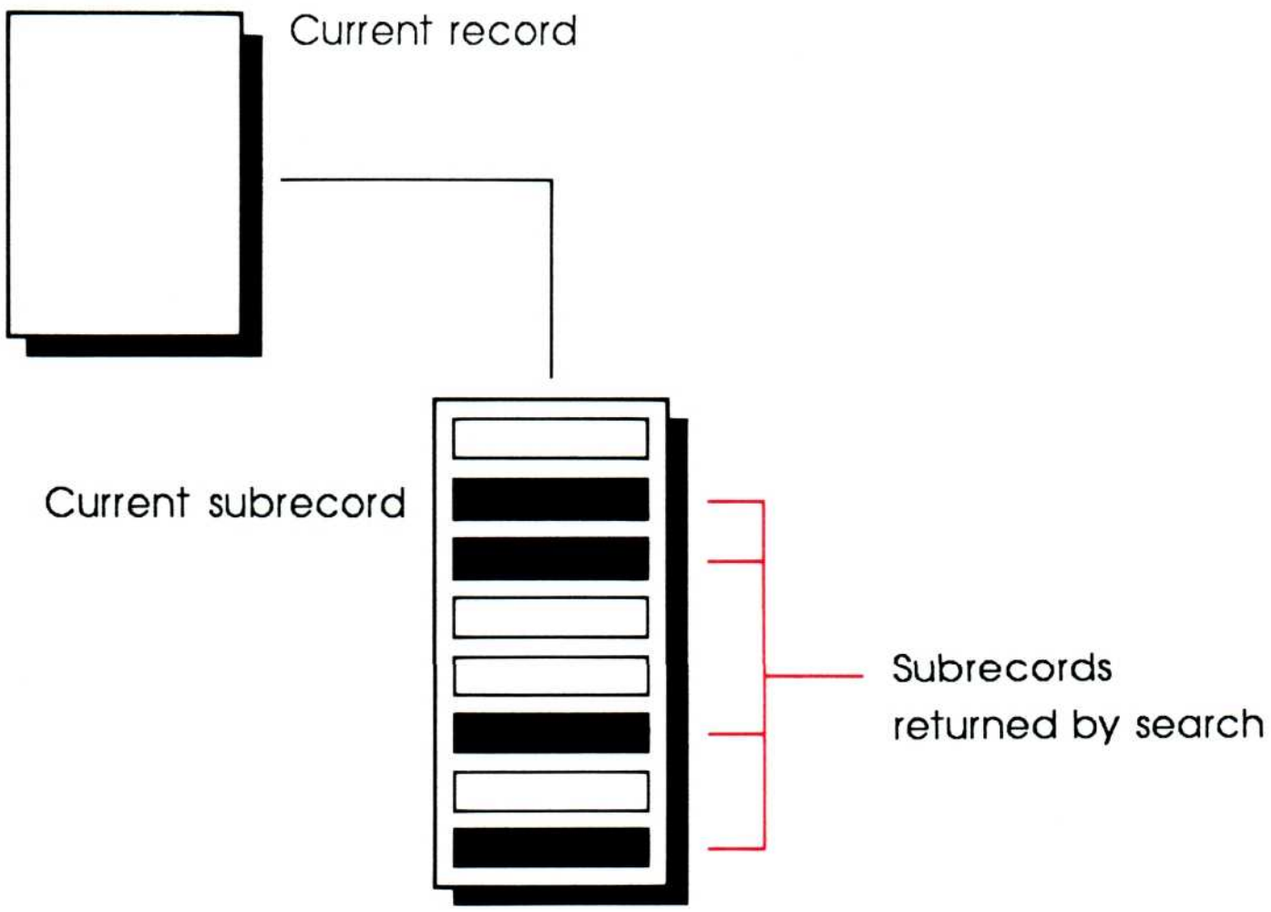


Figure 3-17
Search within a subfile

Subfile example 3

Suppose you want to create a documentation database. You'll have a file named Documents, where each record will contain a particular text stored in a Text field. You then want to search for texts containing one or more specific words. When records contain long texts and the file contains many records, a sequential search such as

```
SEARCH([Documents]Text = "@Value@")
```

would take too long. Therefore, a solution would be to work with a subfile. You create a field of type Subfield, named [Documents]Keywords.

In the subfile, you create a subfield named [Documents]Keywords'Word. Several words can be saved as keywords for a single text. Knowing that a subfield can be indexed, index the Word field of the [Documents]Keywords subfile. You then can perform an indexed search with

```
SEARCH BY INDEX ([Documents]Keywords'Word = "Value")
```

This returns a current selection representing all the records containing at least one matching subrecord. Thus, you have a set of text records containing the specified keyword. Don't forget that you must be able to specify keywords related to the text while entering a record.

If you want, you could write a procedure that would do the time-consuming search just once, adding a subrecord to each record where the value is found.

```
SEARCH([Documents]Text="@Value@")
```

A subfile can contain one or more fields of type Subfile. A field of type Subfile will generate a lower-level subfile. Hence the concept of *subfile levels*.

Working on a level deeper than two can be quite a complex matter. In other words, to access a given subrecord within an *n* level subfile, you must first access the parent record and through it the first-level subfile and the desired first-level subrecord, and through it the second-level subfile and the desired second-level subrecord, and so on. See Figure 3-18.

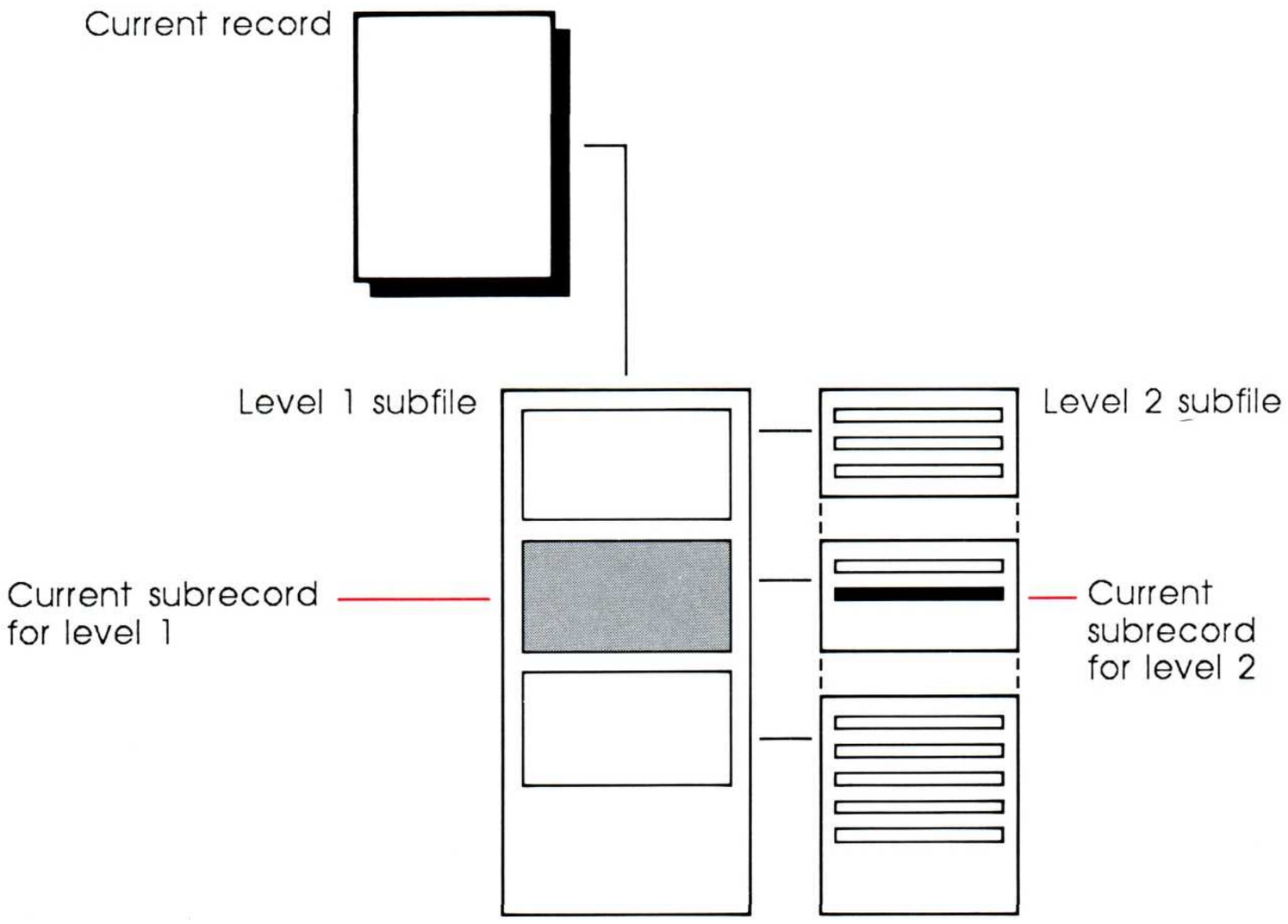


Figure 3-18
Multiple-level subfile access

Subfiles and layouts

To enter data into subrecords, or to display or print subrecord data, you need to create subfile layouts in the same way as you create layouts for a file. Once you have created your subfile layout(s), you create an included layout area to include these layouts in an appropriate file or parent subfile layouts. The output layout and input layout concepts (as defined earlier) also apply to subfile layouts. Figure 3-19 shows a layout dialog box for a subfile.

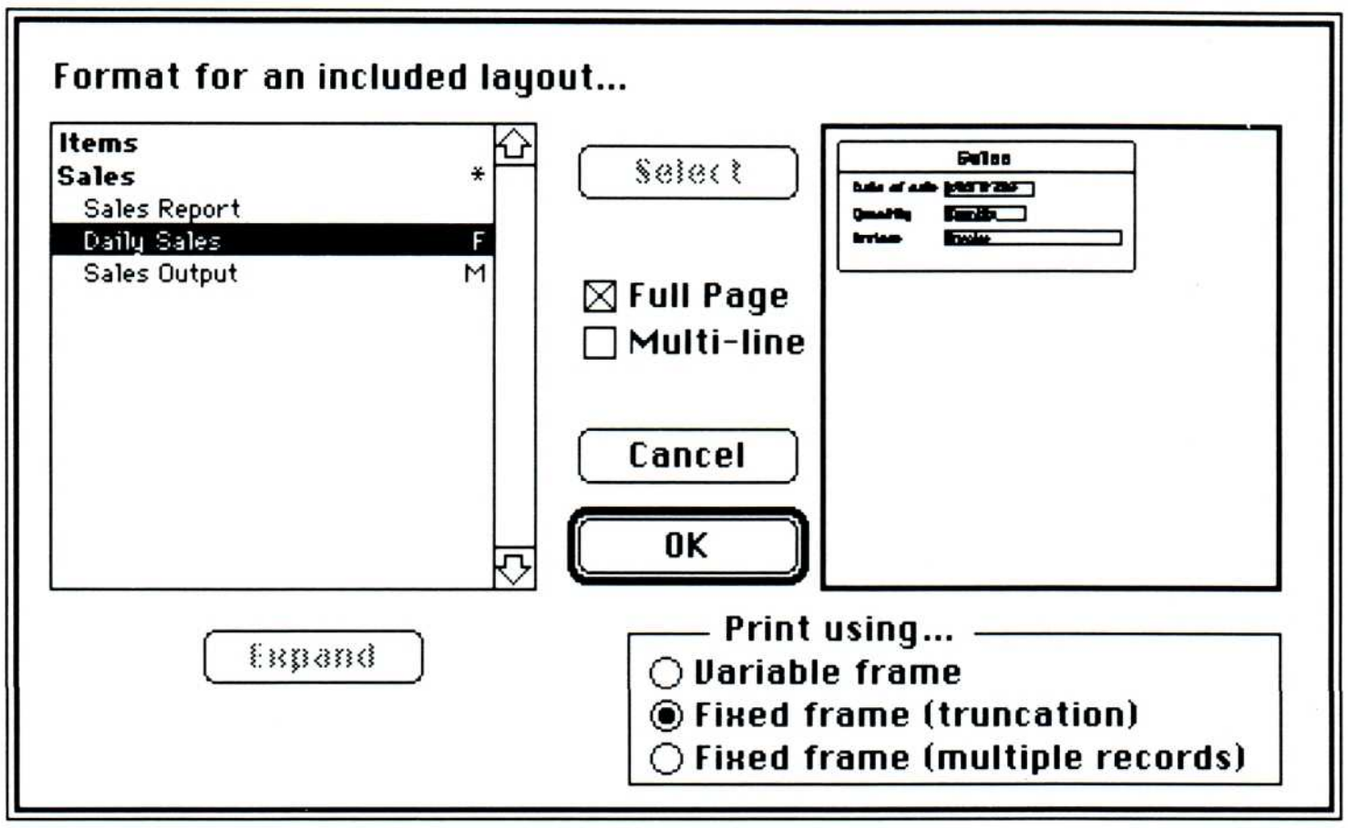


Figure 3-19
Layout dialog box for a subfile

To enter, display, and report data in subfiles, you can follow five steps when creating layouts:

1. Create a Multi-line layout for each subfile for subrecord display (output).
2. Create a Full Page layout for each subfile for subrecord entry (input).
3. Create an input layout for the parent file.
4. Within the parent-file input layout, create an included layout area.
5. Indicate that the subfile layout for the area is the Multi-line layout created in step 1.

You can also enter directly into subrecords with **ADD SUBRECORD** and **MODIFY SUBRECORD.** These take a subfile layout name as an argument.

See *4th Dimension User's Guide* for details on how to create layouts and include subfile layouts in file layouts. Figure 3-20 shows a file structure with its related record and subfile layouts.

❖ *Note:* When you create subfile layouts, you can select the subfile default Multi-line and Full Page layouts. When you create a subfile area in a file layout, you can select the subfile and layouts you want. These layouts can be identical to or different from the subfile default layouts. Default layouts are activated if you do not specify any other layouts for the subfile.

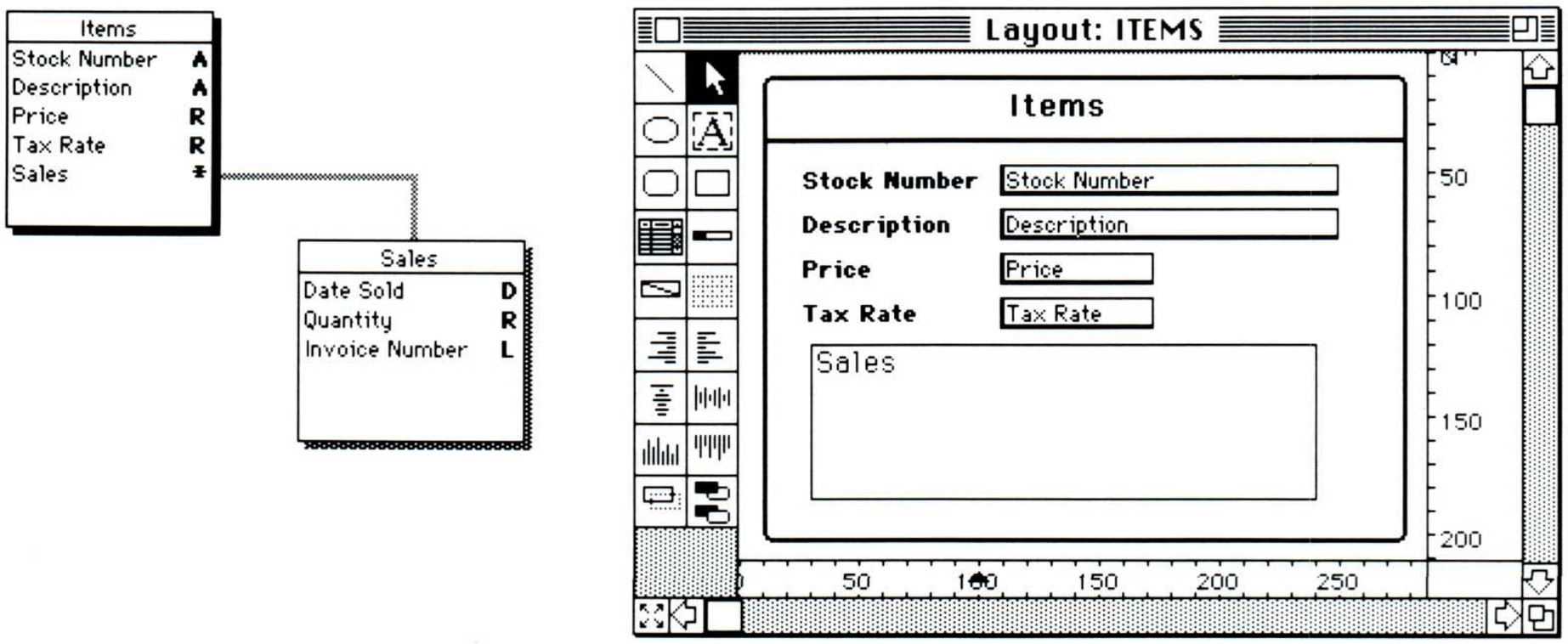


Figure 3-20
A file with its related record, subrecord layouts, and record/subfile output

Printing options in the subfile area

When creating a subfile area within a given layout, you may choose one of three framing options: Variable Frame, Fixed Frame (truncation), and Fixed Frame (multiple record). These options only apply to printing. They let you decide the way subrecords will be printed.

If you choose Variable Frame, 4th Dimension prints as many subrecords as are present, expanding the frame as necessary.

The Fixed Frame (truncation) choice always prints the same number of lines, whether you have enough subrecords to fill the space or not. If there is not enough space to accommodate all the subrecords, 4th Dimension does not print the extra records.

Fixed Frame (multiple record) will print as many records as necessary to print the subrecords for the current record. If the number of subrecords is less than the number of allotted lines, 4th Dimension still prints a whole page, but with the last line(s) blank. Figure 3-21 illustrates these three printing options.

Items

Stock Number BR006
Description BRICK-RED HAVERTON
Price 54.6
Tax Rate 0.065

Date Sold	Quantity	Invoice
07/12/86	3500	1548
08/10/86	3800	1602
09/05/86	4500	1741
10/16/86	3600	1845
11/21/86	4000	1874
12/12/86	4100	1920
01/05/87	3800	1985

Variable Frame

Items

Stock Number BR006
Description BRICK-RED HAVERTON
Price 54.6
Tax Rate 0.065

Date Sold	Quantity	Invoice
07/12/86	3500	1548
08/10/86	3800	1602
09/05/86	4500	1741

Fixed Frame (truncation)

Items

Stock Number BR006
Description BRICK-RED HAVERTON
Price 54.6
Tax Rate 0.065

Date Sold	Quantity	Invoice
10/16/86	3600	1845
11/21/86	4000	1874
12/12/86	4100	1920

Fixed Frame (multiple record)

Items

Stock Number BR006
Description BRICK-RED HAVERTON
Price 54.6
Tax Rate 0.065

Date Sold	Quantity	Invoice
01/05/87	3800	1985

Figure 3-21
Three options for printing subrecords

Subfile layout procedures and the execution cycle

The 4th Dimension execution cycle also applies to the execution of subfile layout procedures. This section discusses the input and output execution cycles for subfile layouts. Because you can nest subfiles to a depth of five, the terms *parent* and *offspring* refer not only to a file-level record and its subfile, but also to a higher-level (closer to the main record) subfile and the lower-level subfile connected to it.

❖ *Reminder:* The Multi-line layout lets you display subrecords within the record layout, and the Full Page layout lets you display a given subrecord when you double-click in the subfile area.

Execution cycle for input to a record with a subfile

A rule of thumb for input to a parent record having a subfile is that 4th Dimension executes each phase for the offspring before it executes the same phase for the parent. As with record-level input, user events can trigger a change of phase during input. These events include

- □ completing an entry to or a modification of a field by pressing Tab or Return or clicking another field
- □ clicking a button
- □ clicking an area
- □ selecting a menu
- □ forcing a redraw action by resizing a window, scrolling, or editing a Full Page (included subfile) layout

There are two input layouts that do not include subrecords: the simple input layout with no subrecords and the input layout with an included file. The order of phase execution for an input layout procedure with at least one subfile is as follows:

1. Before phase of the included subfile layout once for each subrecord
2. Before phase once for the parent layout
3. Display the parent record
4. During phase for each displayed subrecord
5. During phase for each user event, parent record, or subrecord
   - 5a. If the user acts on the subfile, a During phase occurs for the subfile layout procedure and then for the parent
   - 5b. If the user acts on the parent record, a During phase occurs for the parent layout procedure
6. After phase (only when the user validates the current record)
   - 6a. After phase for the subfile layout procedure once per subrecord
   - 6b. During phase for the parent layout procedure once

Execution cycle for output with subfiles

This section covers two output layouts with subfiles: those in which the Multi-line layout is designated as Fixed Frame (truncation) and those designated as Fixed Frame (multiple records).

The order of phase execution for an output layout in which the Multi-line layout is designated as Fixed Frame (truncation) is as follows:

1. Header phase for the parent (**Before selection** returns TRUE the first time only)
2. Before phase for each parent record before it prints
3. During phase for each parent record before it prints
4. Before phase for each displayed subrecord before it prints
5. During phase for each displayed subrecord before it prints
6. Break phase for level 0 only
7. Footer phase for the parent (**End selection** returns TRUE for the last footer only)
8. Return to step 1 if more records remain to be printed

A Fixed Frame (multiple records) subfile layout has a more complex cycle, because it must test to see if the fixed frame has room for another subrecord. If it does not have enough room, it should generate another page. The order of phase execution for an output layout in which the Multi-line layout is designated as Fixed Frame (multiple records) is as follows:

1. Header phase for the parent (**Before selection** returns TRUE the first time only)
2. Before phase for each parent record before it prints
   - 2a. Test: If room remains for the next subrecord, go to step 3
   - 2b. Break phase for level 0 only
   - 2c. Footer phase (**End selection** returns TRUE for the last footer only)
   - 2d. Header phase
   - 2e. Before phase
3. During phase for each parent record before it prints
4. Before phase returns TRUE for each displayed subrecord before it prints
5. During phase for each displayed subrecord before it prints
6. If subrecords remain to be printed:
   - 6a. If another subrecord will fit, do step 4
   - 6b. If another subrecord won't fit, do step 2a
7. If no subrecords remain, do step 2
8. Break phase for level 0 only
9. Footer phase for the parent (**End selection** returns TRUE for the last footer only)

Output from a record having at least one subfile

This section looks at the execution cycle for outputting a record having at least one subfile. This includes printing and screen display.

Printing a record having at least one subfile area

The order of execution when printing a record with at least one subfile is as follows:

1. **Before** file statements and **Before** file output layout statements execute once.
2. **Before** subfile statements and **Before** output layout statements used in the file output layout execute once for every subrecord contained in the record.
3. **During** file statements and **During** file output layout statements execute once.
4. **During** subfile statements and **During** output layout statements used in the file output layout execute once for every subrecord contained in the record.

Viewing a record selection having at least one subfile area

4th Dimension does not execute subfile procedures, but rather the file procedure (if any) for the parent file and the output layout procedure for the parent file. Only the subfile layout appears on the screen.

When to use a subfile

Subfiles and subfile levels are part of the concept of data hierarchy, because a subfile can generate a subfile for each record in a file. This subfile in its turn generates a subfile, and so on. As an example, assume that you need to manage a file of customers spread out over the country, which is divided into states, broken up into regions, and further subdivided into counties.

Given the concept of hierarchy, you would create a file named **Customers** with a **State** subfile, containing a **Region** subfile, entailing a subfile of lower level named **Counties**, in which you would enter the customers' addresses and phone numbers. The structure of your database would thus reflect that of the country. Layouts in your database would also reflect that hierarchy, because you'd have to work down the different levels to access a given record.

Nevertheless, several operations would become difficult to perform, especially a search by name or postal code. Sorting your customers by name will not be easy either, because subfiles are not related to one another. In this particular case, a hierarchical file structure is not an appropriate approach. Instead of using a hierarchy, create fields for customer last names, first names, addresses, etc., and three fields named **State**, **Region**, and **County**. This structure allows for easy search operations.

RAM costs of subfiles

Because subrecords are components of a record, and subrecords are loaded into memory whenever the parent record is, the number of subrecords you can relate to a given record depends on the size of each subrecord and how much RAM you have.

Creating an invoice system

Suppose you create a structure to deal with invoices and customers. The first analysis leads to the design shown in Figure 3-22.

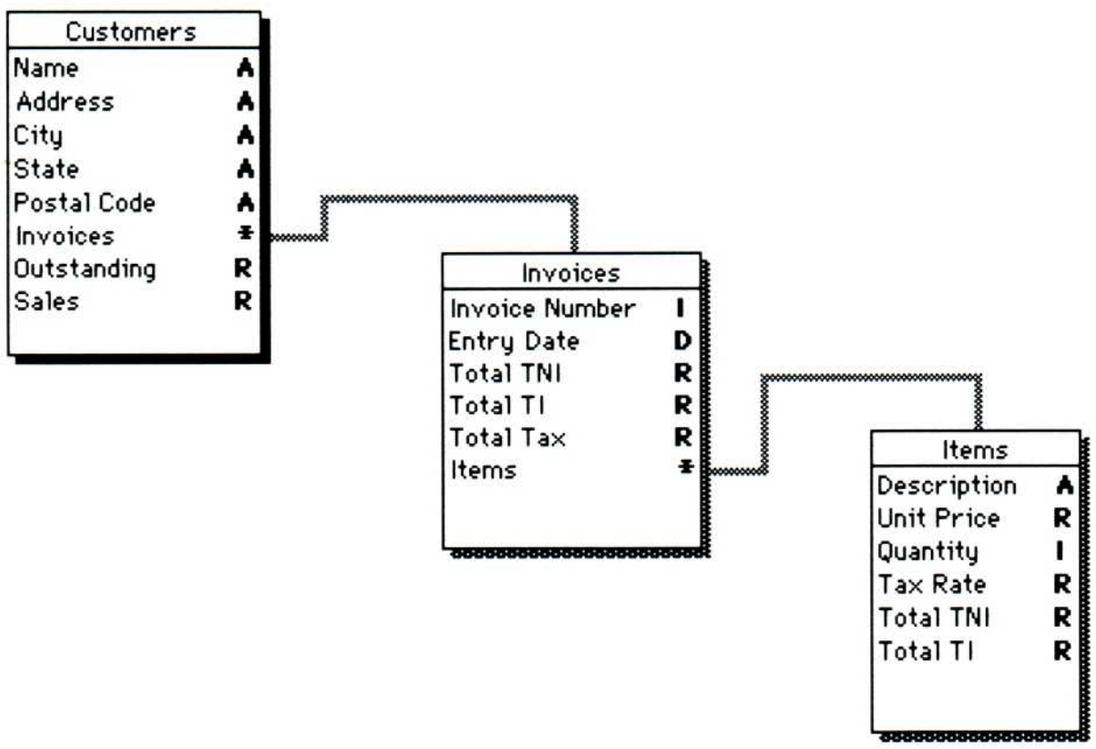


Figure 3-22
Invoice design with two levels of subfiles

This design lets you

- search customer records, sort the file selection by name, ZIP code, sales, and so on
- access all a customer invoices, because such information is stored in the customer's record
- calculate the total amount for every invoice using routines that 4th Dimension provides to add the items in all invoices (you will also use routines to calculate a customer's sales and outstanding amounts due)
- calculate annual sales by adding up all customers' sales (you will also be able to evaluate global outstanding by adding up all customers' outstanding)

On the other hand, the structure has several drawbacks:

- Whenever you wish to change a particular invoice, you have to select as the current record the corresponding customer record and then the specific invoice record as the current subrecord. In other words, you cannot search for a particular invoice without searching for the customer to whom it is related.
- Printing invoices related to different customers is not an easy operation, because a customer invoice consists of a customer subfile.
- Sorting all invoices by number or date is impossible.
- You may not have enough RAM space to load a customer's record if the customer has a lot of invoices with each having numerous items.

You soon realize that entering invoices into a **Customers** subfile is not appropriate for "file-type" operations, such as sorting, searching, or printing invoices. Creating two separate files in a relational database is a better solution. This structure is shown in Figure 3-23.

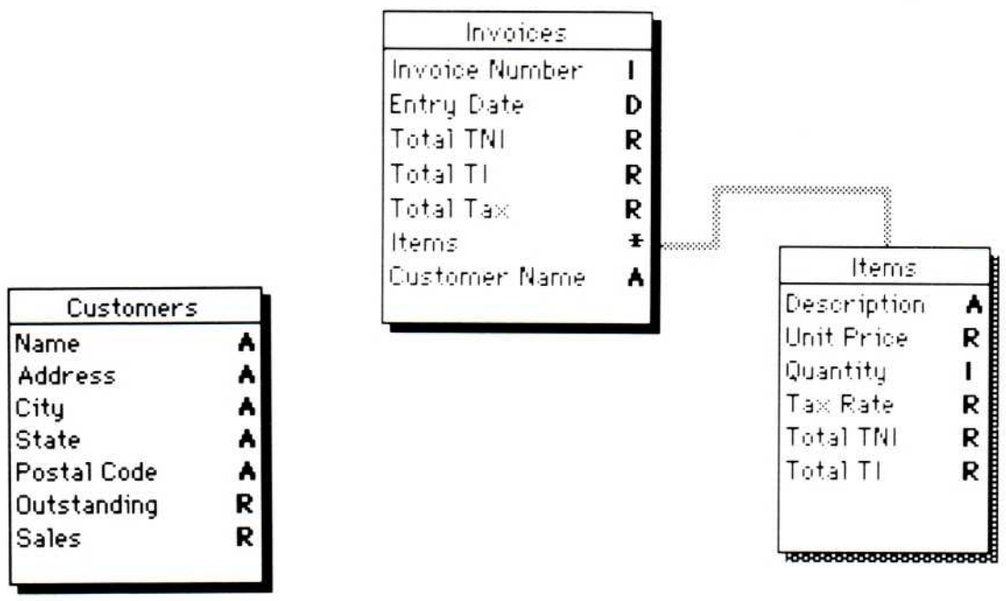


Figure 3-23
Invoice system with two files and one subfile

Such a structure provides for easy searching, sorting, and printing of the **Customers** file. It is designed for easy invoice management, because all information is now stored in a different file named **Invoices**. Add the **[Invoices]Customer Name** field to know the invoices belonging to a customer. A 4th Dimension relational database will automatically carry over invoice totals into customer **Sales** and **Outstanding** fields every time you add a new invoice. Refer to Chapter 5, "File Links," in this manual.

Chapter 4

Layouts

This chapter covers areas that relate to the programming of layouts. For details on how to work with layouts, layout tools, and various icons, see Chapter 2 of *4th Dimension User's Guide,* "Layout Design Basics." This chapter discusses the following topics:

- report layouts
- field formatting
- picture fields
- layout variables
- external areas

Report layouts

Look at the layout and the paper it will print on, shown in Figure 4-1. It raises the question of how 4th Dimension distributes printing areas given different printing scenarios. This section looks at three basic scenarios:

- Print a selection of unsorted records.
- Print a selection of sorted records.
- Print a selection of sorted records with subtotals and one break level.

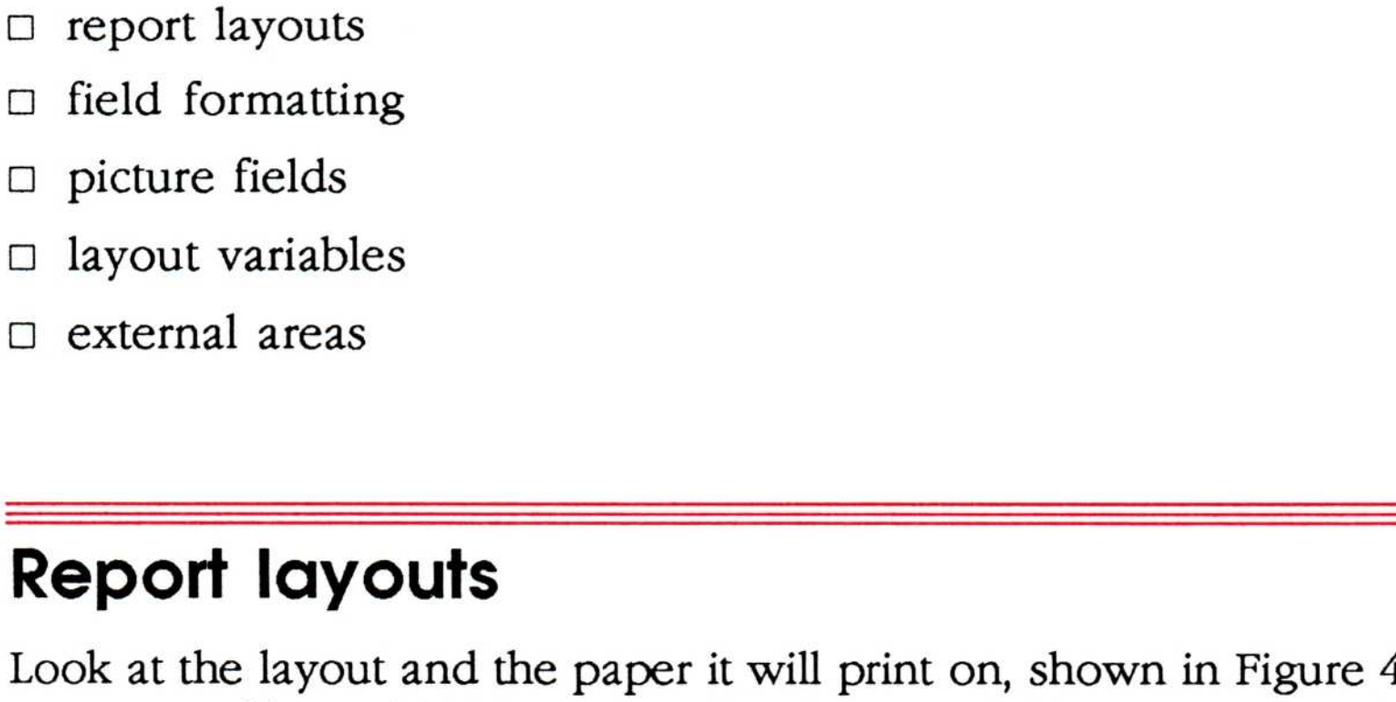


Figure 4-1
Output layout and a piece of paper

Printing a simple list

Printing a list of records can be a simple activity: header contents print at the beginning of every new page, detail prints as long as there are records to be printed, break contents print once under the last record of every page, and footer contents print at the end of every new page (see Figure 4-2).

❖ *Note:* The layout break prints once under the last record on a page, because it helps you frame a given group of records.

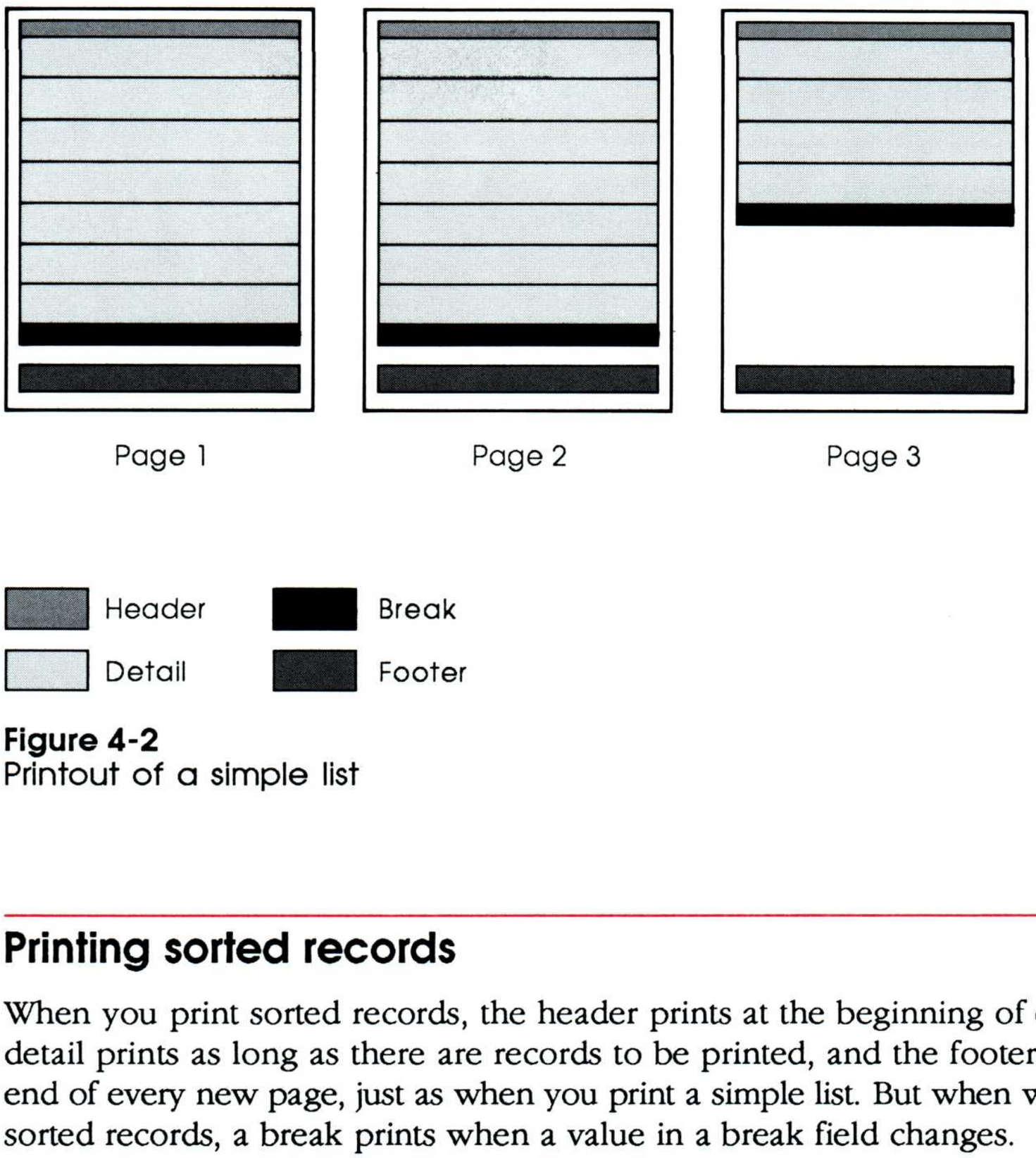


Figure 4-2
Printout of a simple list

Printing sorted records

When you print sorted records, the header prints at the beginning of every new page, detail prints as long as there are records to be printed, and the footer prints at the end of every new page, just as when you print a simple list. But when working with sorted records, a break prints when a value in a break field changes.

To print subtotals, statistics, or other information between records, you must create a break area and insert the appropriate variables within it. To create a break area, separate the D and B markers, and insert any areas you wanted printed in the resulting break area. (See the section "Layout Variables" later in this chapter.)

The 4th Dimension **Subtotal** function determines whether or not break processing will be active. The 4th Dimension interpreter scans the output layout procedure for the word **Subtotal.** If it finds **Subtotal,** it processes breaks. See Figure 4-3.

❖ *By the way:* 4th Dimension does not interpret or execute output layout procedures when looking for **Subtotal.** It simply searches for the word, written as an instruction.

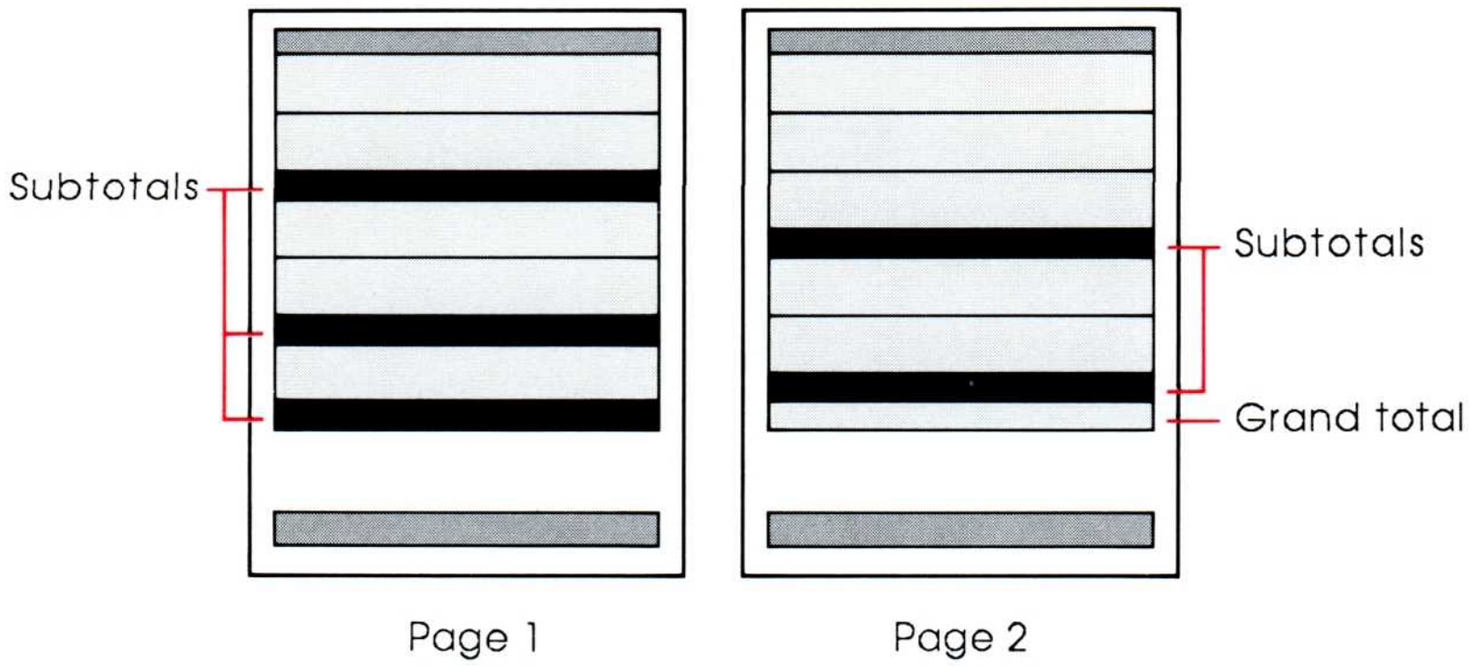


Figure 4-3
Breaks on sorted records with subtotals

Sort fields are fields that you have explicitly sorted. For example, if you included the statement

SORT SELECTION ([SALES] ; SalesRegion ; SalesPerson ; CustomerNumber)

in a procedure, SalesRegion, SalesPerson, and CustomerNumber become the sort fields. **Break fields,** on the other hand, include all the sort fields less the last sort field. Thus, the break fields for this statement are SalesRegion and SalesPerson, but not CustomerNumber.

Each break field has a different level. You can test this level with the **Level** function. The first sorted field, SalesRegion, generates a level 1 break when the value of SalesRegion changes. Likewise, the second sorted field, SalesPerson, generates a level 2 break when the value of SalesPerson changes.

Thus, if you have three records in which SalesRegion has the value of "North" followed by two records with the value "South," 4th Dimension generates a level 1 break and prints the break area after printing the three "North" records. It then prints the two "South" records, generates a level 2 break, and prints the appropriate break area.

4th Dimension also has a level 0 break. If the selection is unsorted, 4th Dimension generates a level 0 break after printing the last record of each page. If the selection is sorted, 4th Dimension generates a level 0 break after printing the last record in the selection. The following is an example of how an output procedure processes breaks. Figure 4-4 shows the form a printout would take under this procedure.

```
If (In break)
   Case of
      :(Level = 0)
         bEnd := "Total for the file: "
         bLine := "_" * 12 `Print 12 underscores
         bTotal := Subtotal(SalesAmount) `Subtotal activates break processing
      :(Level = 1)
         bEnd := "Total for region: "+vRegion
         bLine := "#" * 12
      :(Level = 2)
         bEnd := "Total for person: "+vPerson
         bLine := "=" * 12
   End case
End if
```

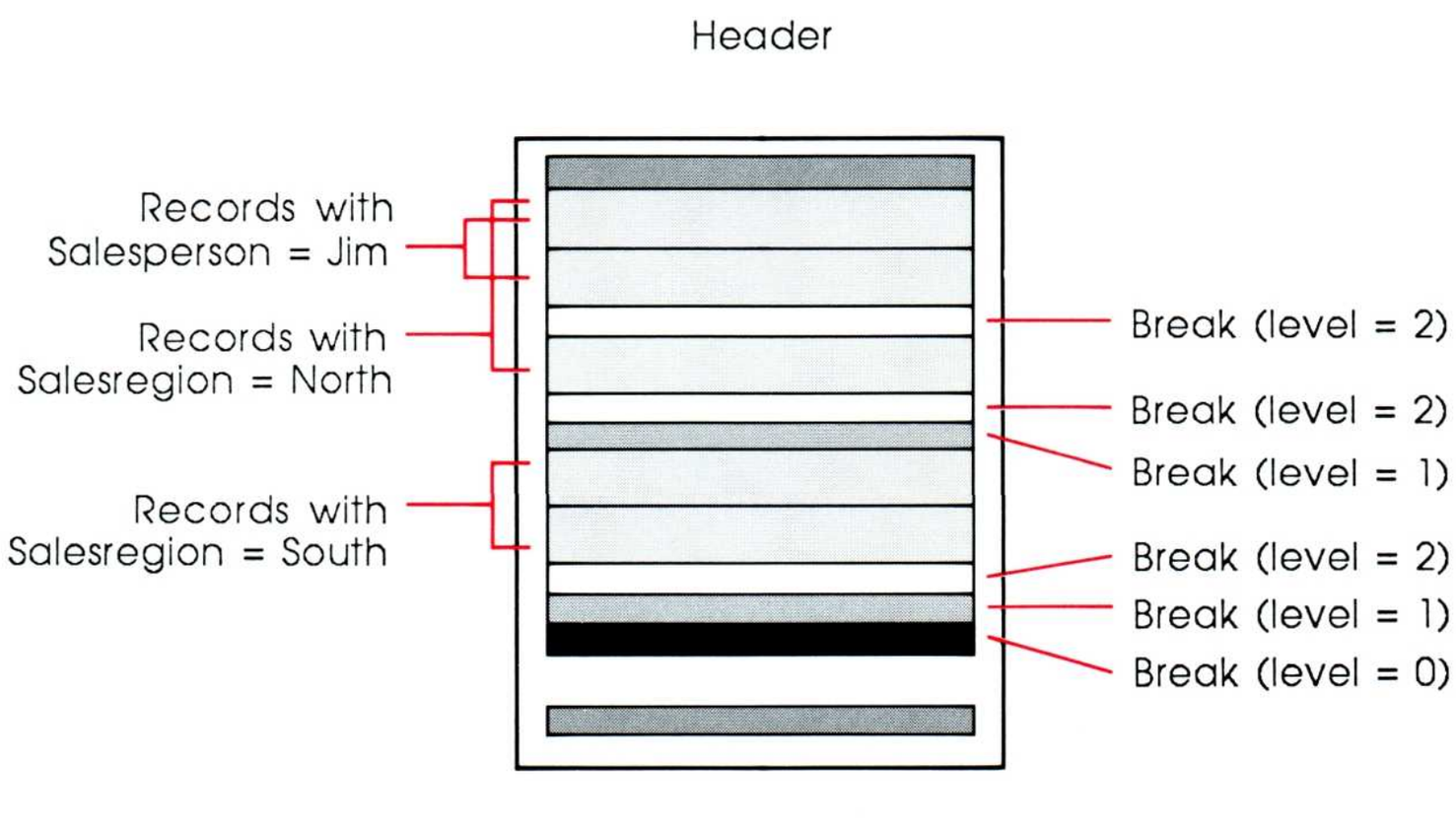


Figure 4-4
Report form with breaks for region and sales person

Printing sorted records with subtotals and a page break

The header prints at the beginning of every new page, detail prints as long as there are records to be printed, and footer prints at the end of every new page. The break prints when the break field changes. That is, when the value in a given field changes, 4th Dimension does a page break on each break. Figure 4-5 shows the result of such a scenario.

Break contents will print if you've assigned a subtotal variable to the values of a sorted selection. When you view records, this area appears once in the bottom part of a layout window, above the footer.

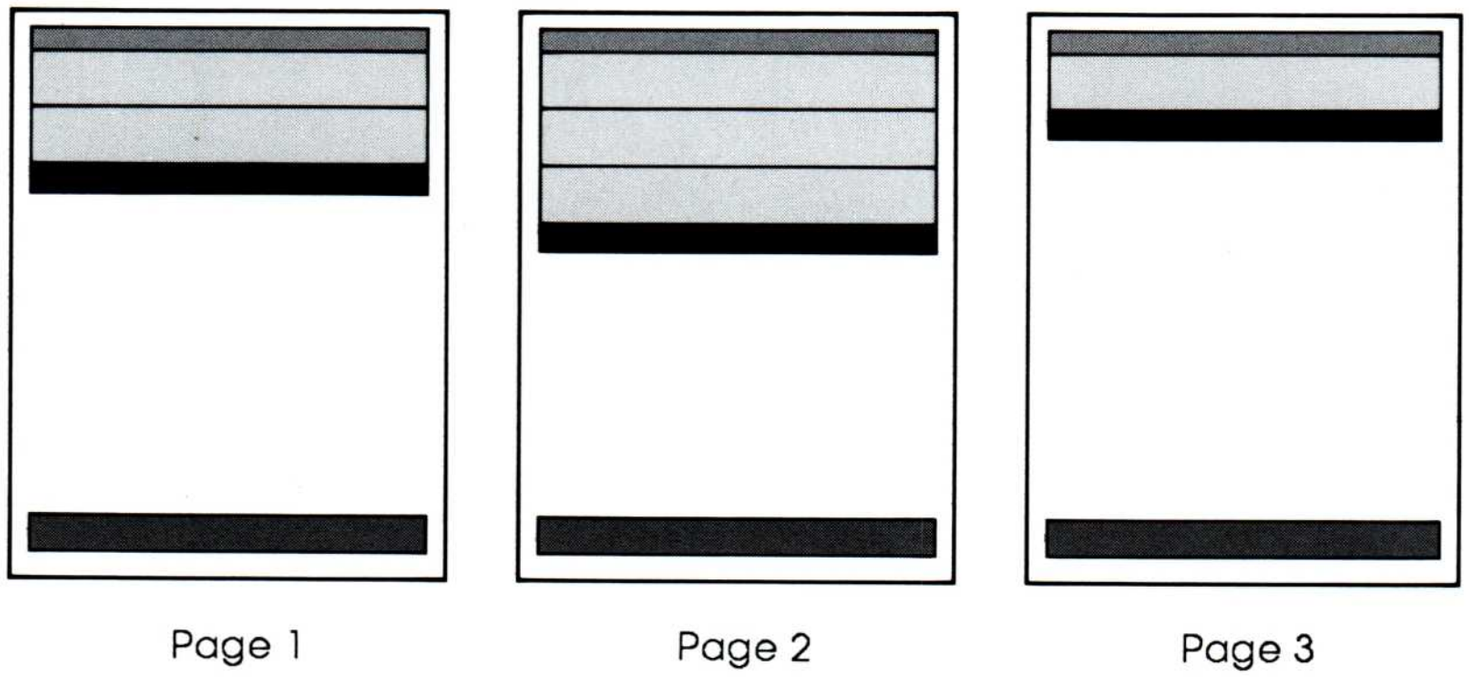


Figure 4-5
Report with page breaks for each break

❖ *Note:* The break also prints once under the last record of the file selection to display the grand total.

Formatting fields within a layout

Data and data display are two different concepts. Take a Real field in a given record as an example. A numeric value stored in that field can have as many as 19 significant digits. This value can be displayed as two decimal, three decimal, and so forth. Whatever the format, the field contents remain unchanged.

Alphanumeric and Text fields

Displayed values stored in these fields can be left aligned, right aligned, or centered. Choose the appropriate alignment using the Style menu when designing a layout.

❖ *Note:* Alphanumeric and Text fields display faster when you insert an exclamation point (!) in format specifications. This usage only displays one line. Inserting an exclamation point in a numeric field left justifies the display.

Real, Integer, and Long Integer fields

Displayed values stored in these fields can be left aligned, right aligned, or centered. Choose the appropriate alignment using the Style menu. To format a numeric field, double-click the field or choose Format from the Layout menu. The Format of field dialog box, shown in Figure 4-6, will appear on the screen.

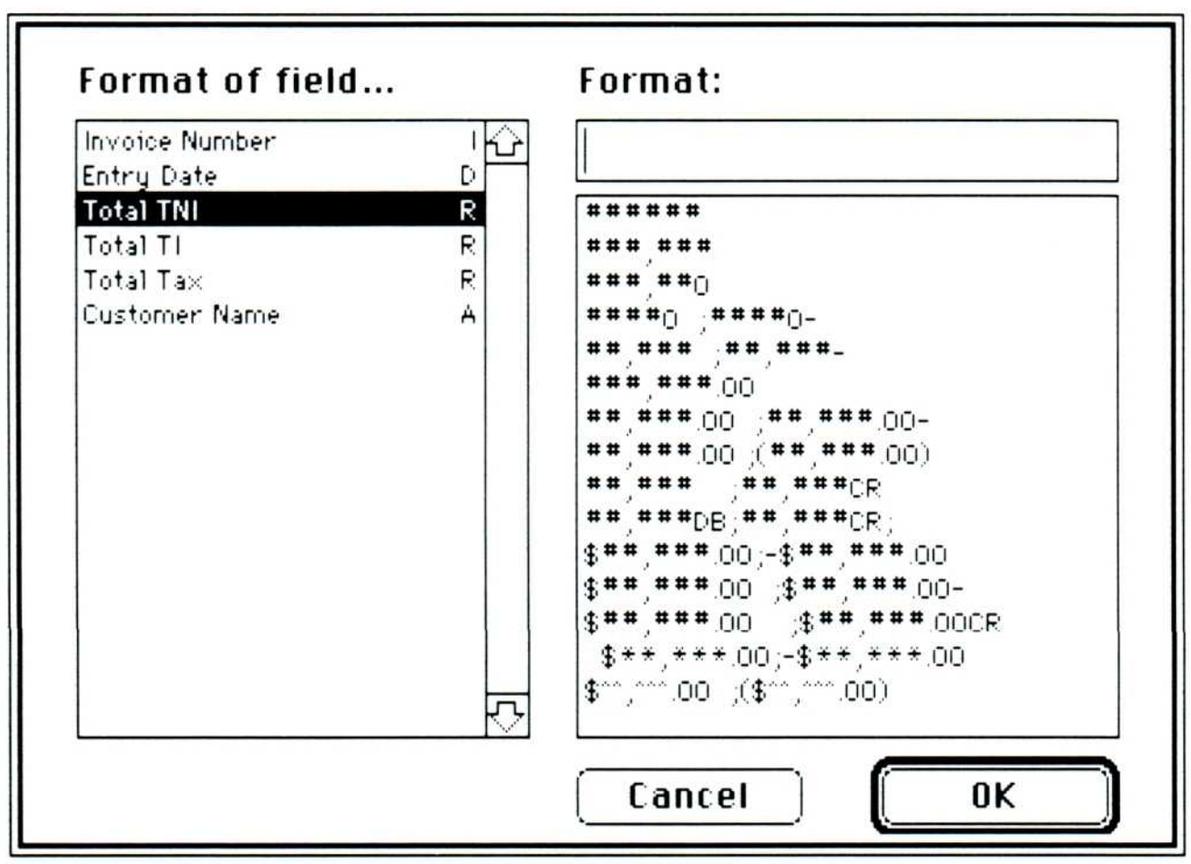


Figure 4-6
Format of field dialog box

You can format a numeric field using alphanumeric expressions. You can specify three formats, each separated by a semicolon. The first format applies when the field value is positive, the second when the field value is negative, and the third when the value is null (zero).

Separate the expression into three parts, separating each part with a semicolon. For example:

```
Positive;Negative;Zero
```

4th Dimension prints the first string (**Positive**) when you enter a positive value, the second (**Negative**) is enabled when the field value is negative, and the third format (**Zero**) when the value is null (zero).

You may specify any formatting character except Tab, Return, or control characters. Four specific symbols determine the maximum number of digits you wish to display on the screen. These are

- number sign (#)
- asterisk (*)
- caret (^)
- zero (0)

If you have no zero format, zero uses the positive format. If you have no negative format, negative uses the positive format.

Number sign (#)

This symbol is replaced by a digit when the field value is displayed. When there are fewer digits than number signs, leading number signs are disabled as well as any other character surrounded by number signs.

Asterisk (*)

This symbol is replaced by a digit when the field value is displayed. When there are fewer digits than asterisks, leading asterisks are forced and displayed; any other character surrounded by asterisks also appears as an asterisk.

Caret (^)

This symbol is replaced by a digit when the field value is displayed. When there are fewer digits than carets, leading spaces are forced. They appear as non-breaking spaces, and any other character surrounded by carets also appears as a non-breaking space.

❖ *Note:* The caret character does not generate a space character, but a non-breaking space (ASCII code 202). This non-breaking space is not a word separator, and has the same character width as a digit.

Zero (0)

This symbol is replaced by a digit when the field value is displayed. When there are fewer digits than zeros, leading zeros are forced and displayed. Any other character surrounded by zeros also appears as a zero.

What happens at display time

Whatever option you choose, the following rules remain in effect:

- Characters placed to the left or to the right of the last symbol are displayed.
- Characters inserted between two symbols that are replaced by digits will be displayed.
- Whenever a formatting error prevents the entire value from being displayed, less-than characters (<) appear instead.
- When a field value is negative, the sign is accounted for as a digit and is displayed on your screen. It does not show if you've set a specific format for negative numbers.

The Format of field dialog box lets you choose among default settings for numeric values. Should these settings be inappriopriate, enter a format in the text area.

Numeric formatting examples

Table 4-1
How 4th Dimension displays numeric fields for various formats (for display purposes only) and in its three different configurations (positive, negative, and zero)

Format	1234	-1234	0
###	<<<	<<<	
####	1234	<<<<	
#####	1234	-1234	
#####.##	1234	-1234	
####0.00;-####0.00	1234.00	-1234.00	0.00
####0	1234	-1234	0
###0+;###0~; 0~	1234+	1234-	0~
###0~;###0-	1234~	1234-	0~
###0;###0-	1234	1234-	0
+###0;-###0;0	+1234	-1234	0
####0CR	1234CR	-1234CR	0CR
###0~~;###0CR	1234~~	1234CR	0~~
###0DB;###0CR; 0~~	1234DB	1234CR	0~~
###0CR;###0DB; 0~~	1234CR	1234DB	0~~
###0~;(###0)	1234~	(1234)	0~
##,##0	1,234	-1,234	0
##,##0.00	1,234.00	-1,234.00	0.00

Table 4-1 (continued)
How 4th Dimension displays numeric fields for various formats (for display purposes only) and in its three different configurations (positive, negative, and zero)

Format	1234	-1234	0
^^^^^ (see *Note*)	~1234	–1234	~~~~~
^^^^0	~1234	–1234	~~~~0
^^,^^0	~1,234	–1,234	~~~~~0
^^,^^0.00	~1,234.00	–1,234.00	~~~~~0.00
*****	*1234	–1234	*****
****0	*1234	–1234	****0
,0	*1,234	–1,234	*****0
,0.00	*1,234.00	–1,234.00	*****0.00
$**,**0.00;–$**,**0.00	$*1,234.00	–$*1,234.00	$*****0.00
$^^^^0	$~1234	$–1234	$~~~~0
~$^^^0;–$^^^0	~$1234	–$1234	~$~~~0
~$^^^0~;($^^^0)	~$1234~	($1234)	~$~~~0~
~$^,^^0.00~;($^,^^0.00)	~$1,234.00	($1,234.00)	~$~~~~0.00

Note: When you use the ^ character in your format, it generates the non-breaking space character (ASCII 202) rather than the true space character (ASCII 20). Reals are accurate to 19 places total, on either side of the decimal point. The ~ character represents a non-breaking space (used here only to show that 4th Dimension shows a space). The <<< characters represent numbers too large for a given format to display.

Formatting a Date field

You can specify alignment and appropriate display format for dates, as shown in Table 4-2. You cannot generate additional date formats.

Table 4-2
Date formats

Format	Display
Short	1/1/90
Abbreviated	Mon, Jan 1, 1990
Long	Monday, January 1, 1990

Working with Picture fields

You may display Picture fields in three different ways:

- Truncated
- Scaled to fit
- On background

Truncated pictures

4th Dimension centers a Truncated picture in the Picture field and trims the picture if it's bigger than the Picture field. Figure 4-7 shows Truncated pictures.

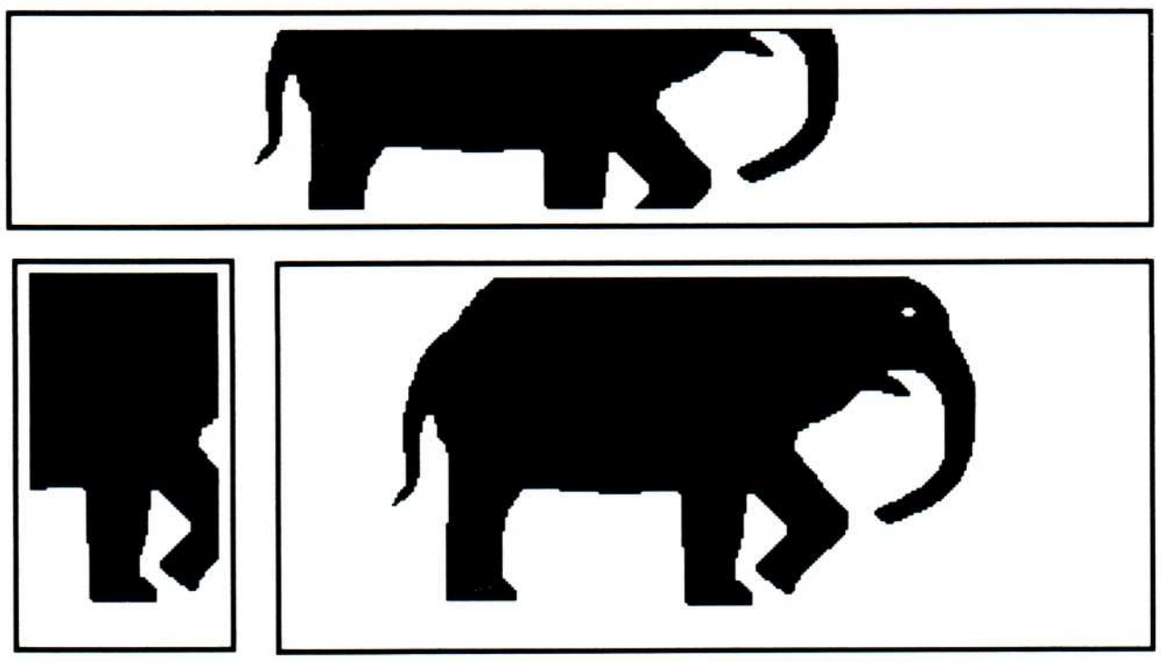


Figure 4-7
Truncated pictures

Scaled to fit pictures

4th Dimension automatically enlarges or reduces a Scaled to fit picture to fit in the Picture field. Figure 4-8 shows Scaled to fit pictures.

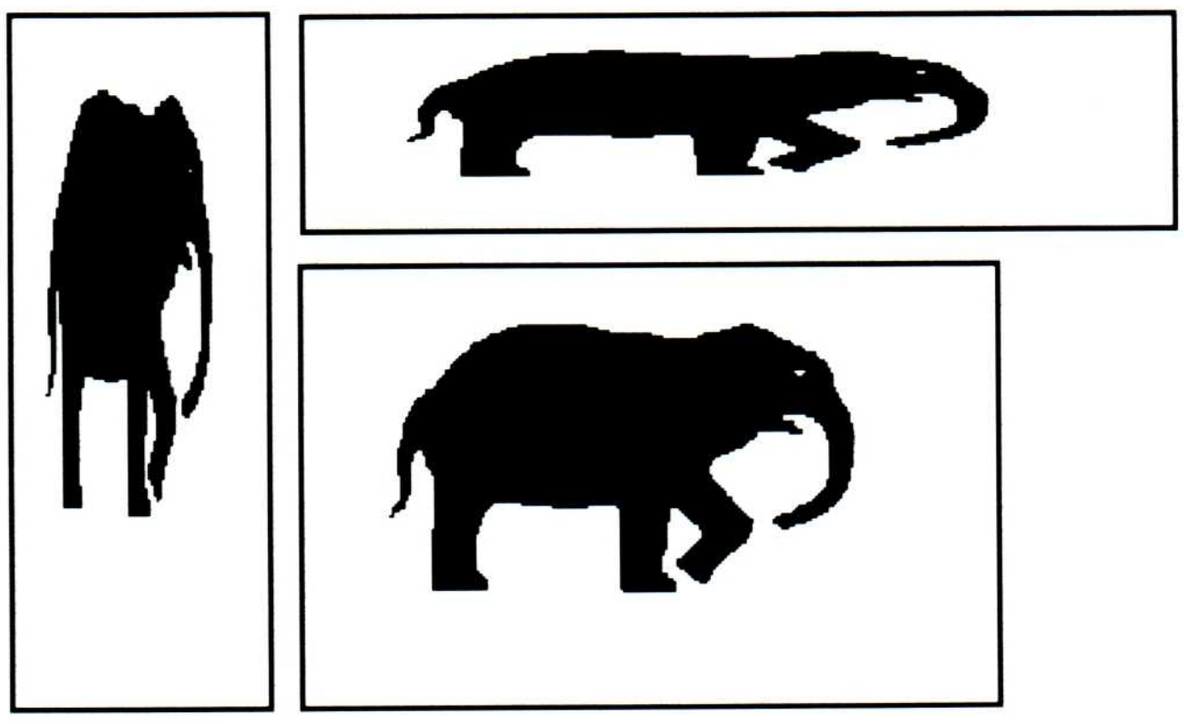


Figure 4-8
Scaled to fit pictures

On background pictures

A picture stored in an area designated as On background can be moved by the user by dragging the picture. You first need to design the background in the layout. The user can control the display of the picture on the background for each record. Figure 4-9 shows an example of an On background picture.

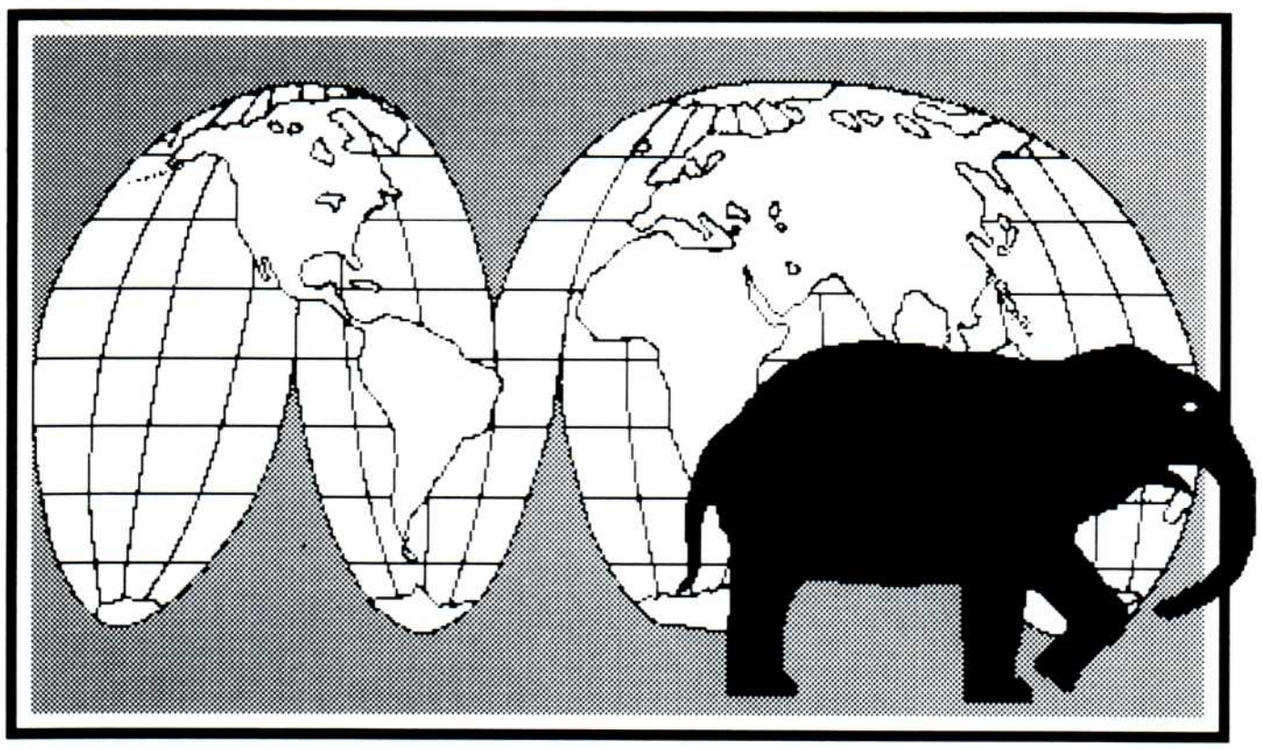


Figure 4-9
On background picture

Double-clicking anywhere in the Picture field brings up the Choice of mode dialog box, shown in Figure 4-10. The user can select the display of the picture on the background desired by clicking one of the eight mode choices.

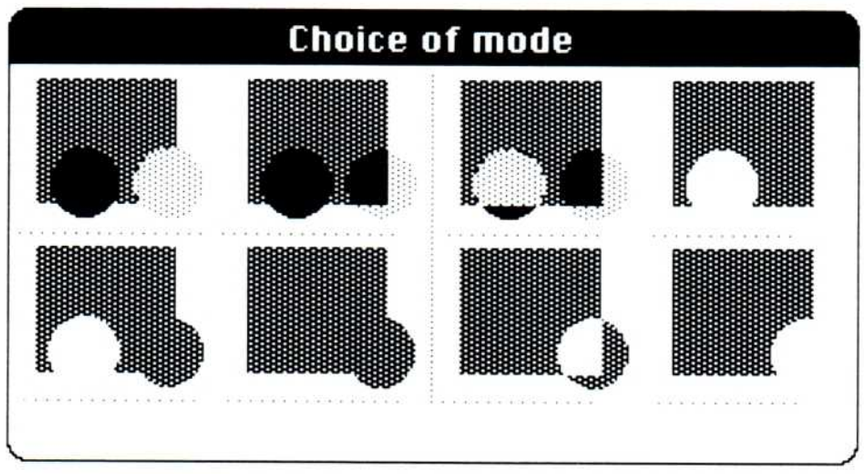


Figure 4-10
Choice of mode dialog box

Choosing any of these options determines the contrast between the picture and the background. Pixels can be black or white. The combination of two pixels determines the shade of the resulting point according to the background and picture patterns. 4th Dimension offers four basic operations—**Copy**, **Or**, **Xor**, and **Bic** (Bit Copy)—and a **Not** operation for each operator for a total of eight modes. Table 4-3 summarizes the eight modes, showing the action taken on the destination pixel, based on whether the source pixel is black or white.

❖ *Note:* Picture fields functions may be combined with arithmetic operations on pictures (see Chapter 8, "Operations on Pictures").

Table 4-3
Pixel transfer modes

Source transfer mode	If black pixel	If white pixel	Figure reference
srcCopy	Force black	Force white	Figure 4-11
srcOr	Force black	Leave alone	Figure 4-12
srcXor	Invert	Leave alone	Figure 4-13
srcBic	Force white	Leave alone	Figure 4-14
notSrcCopy	Force white	Force black	Figure 4-15
notSrcOr	Leave alone	Force black	Figure 4-16
notSrcXor	Leave alone	Invert	Figure 4-17
notSrcBic	Leave alone	Force white	Figure 4-18

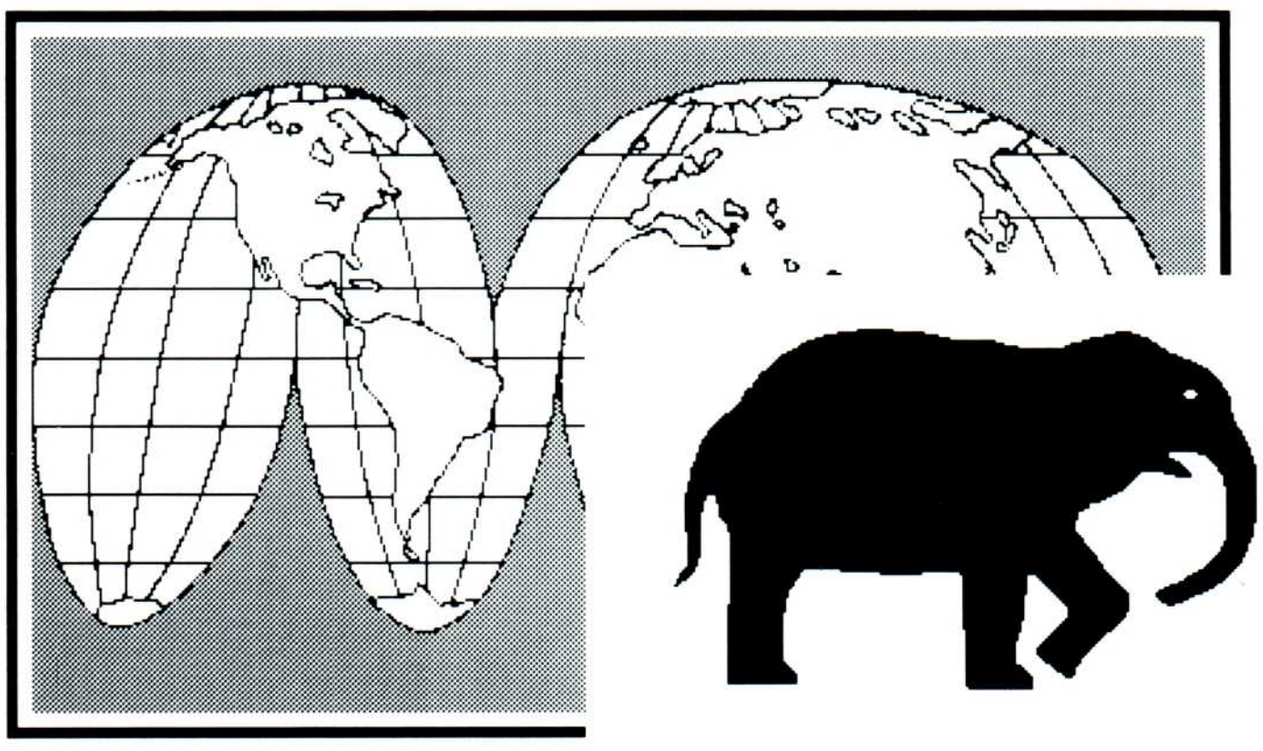


Figure 4-11
srcCopy example

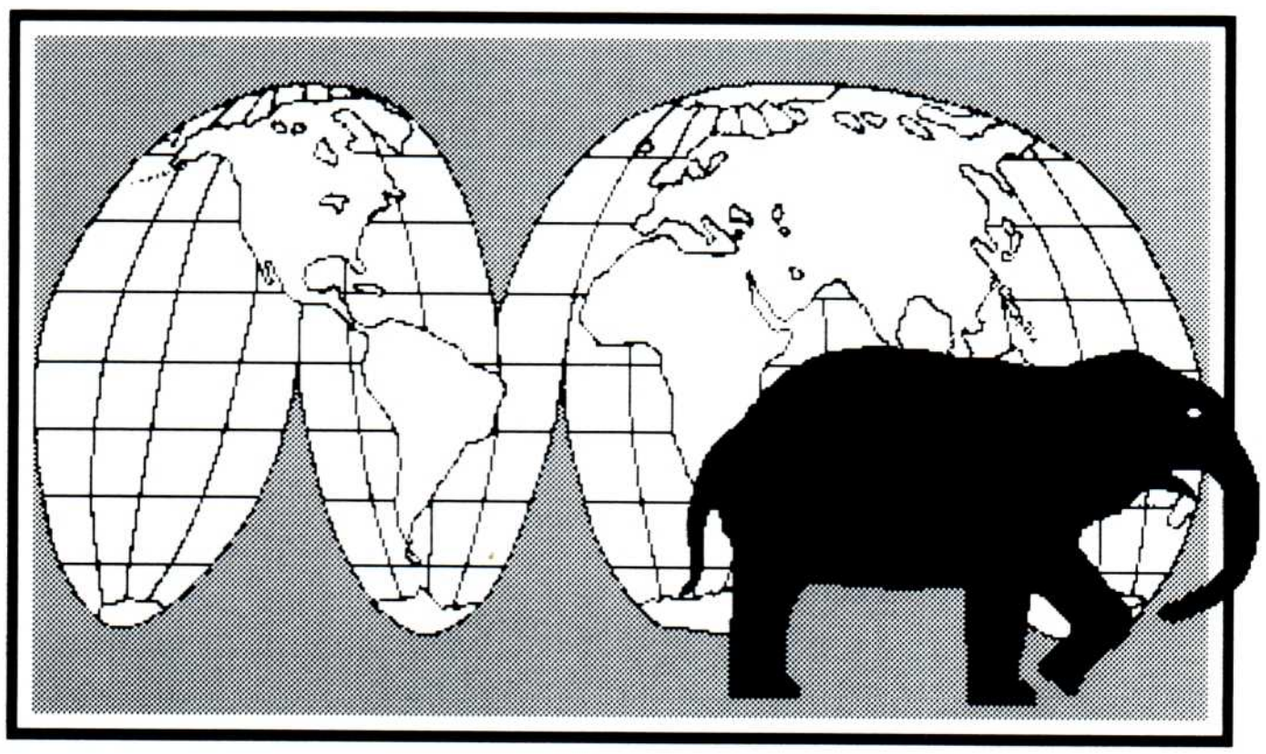


Figure 4-12
srcOr example

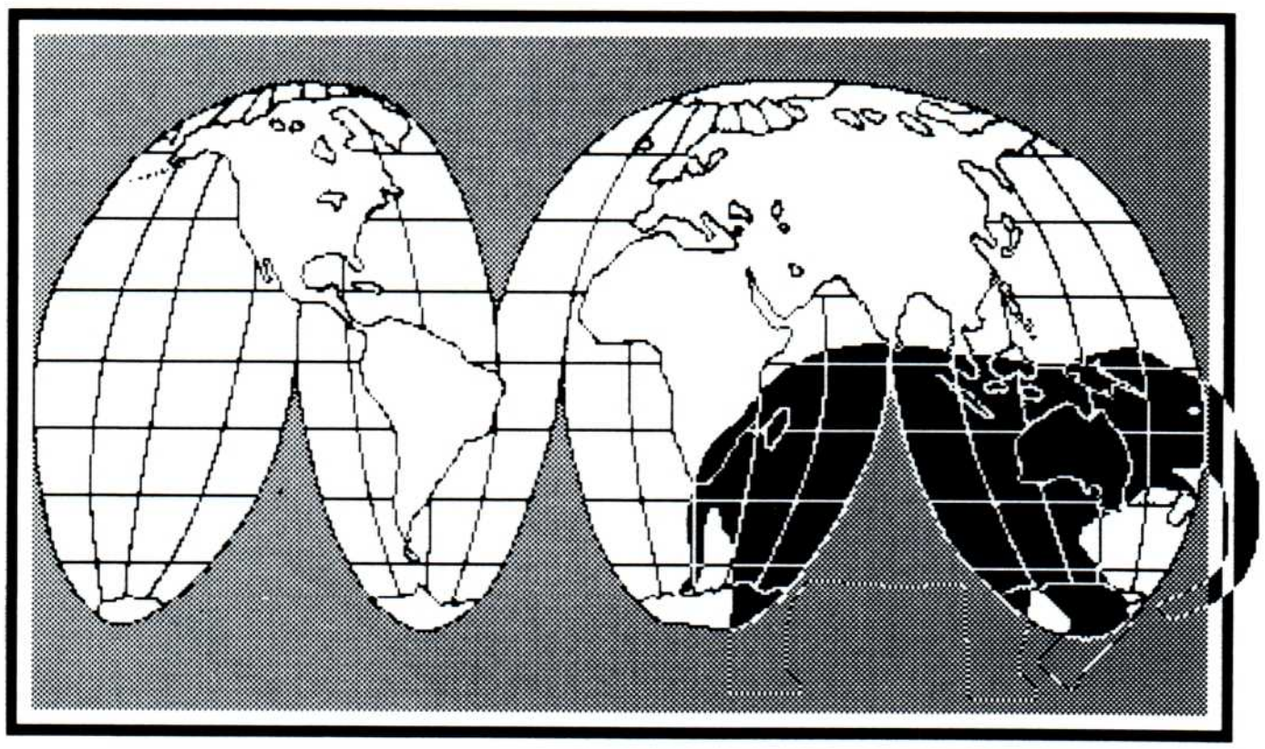


Figure 4-13
srcXor example

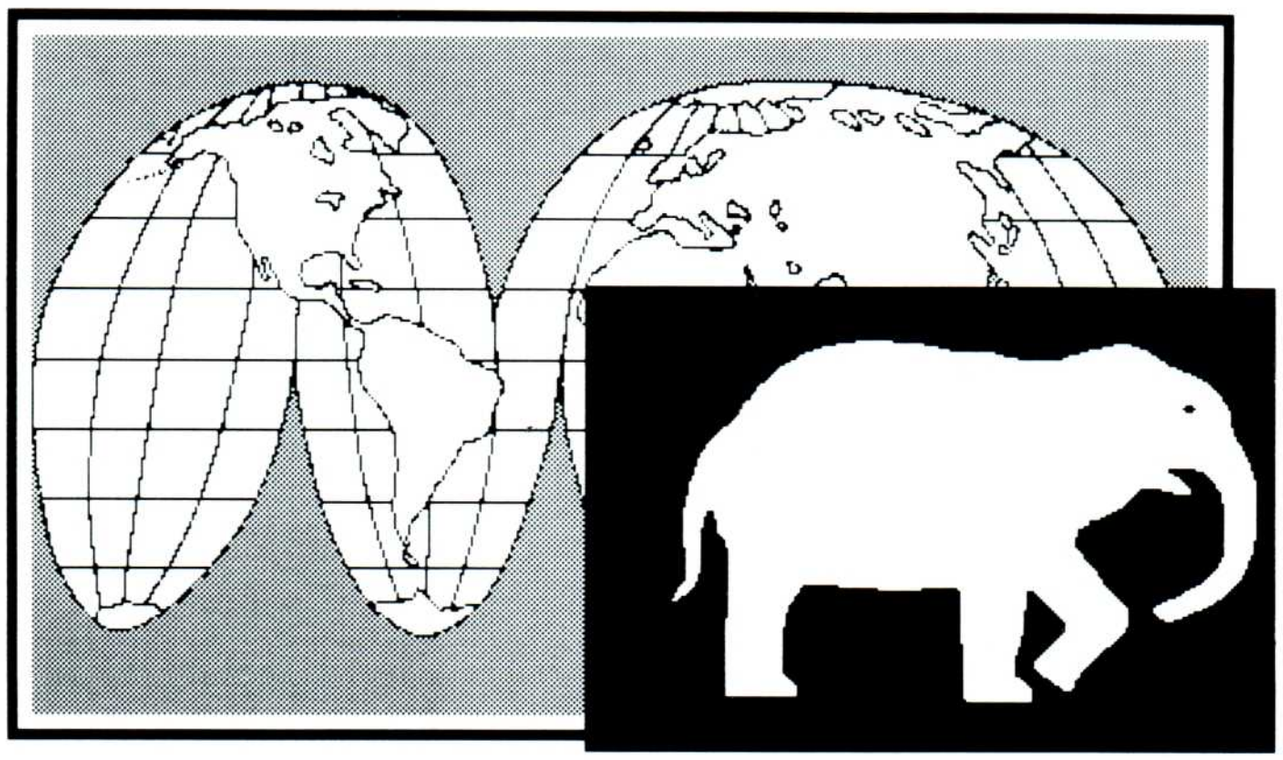


Figure 4-14
srcBic example

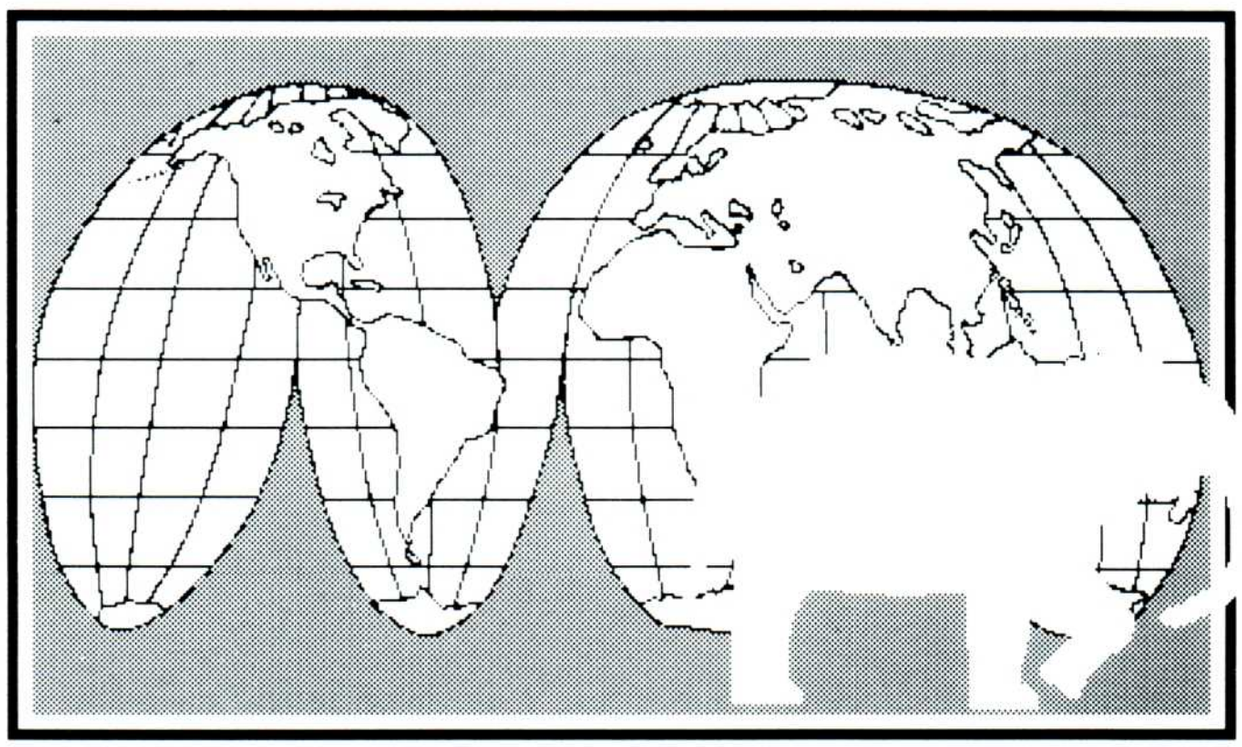


Figure 4-15
notSrcCopy example

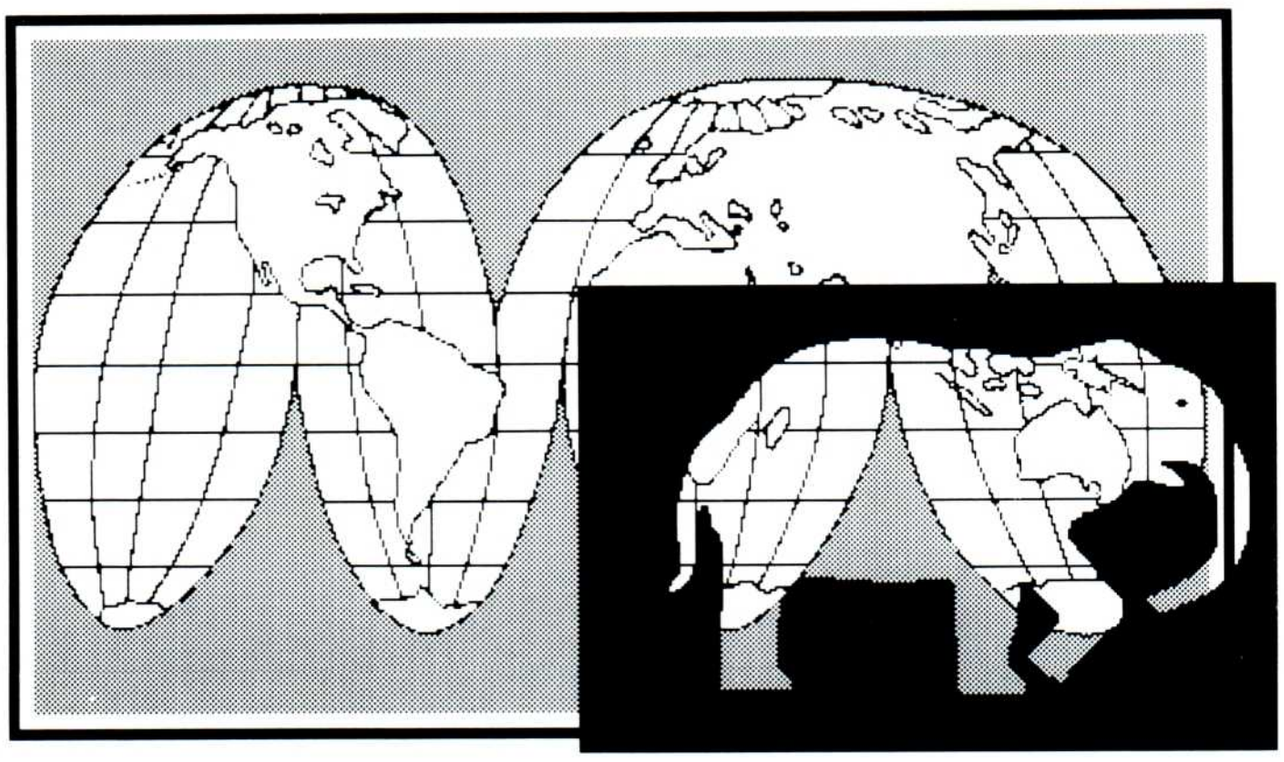


Figure 4-16
notSrcOr example

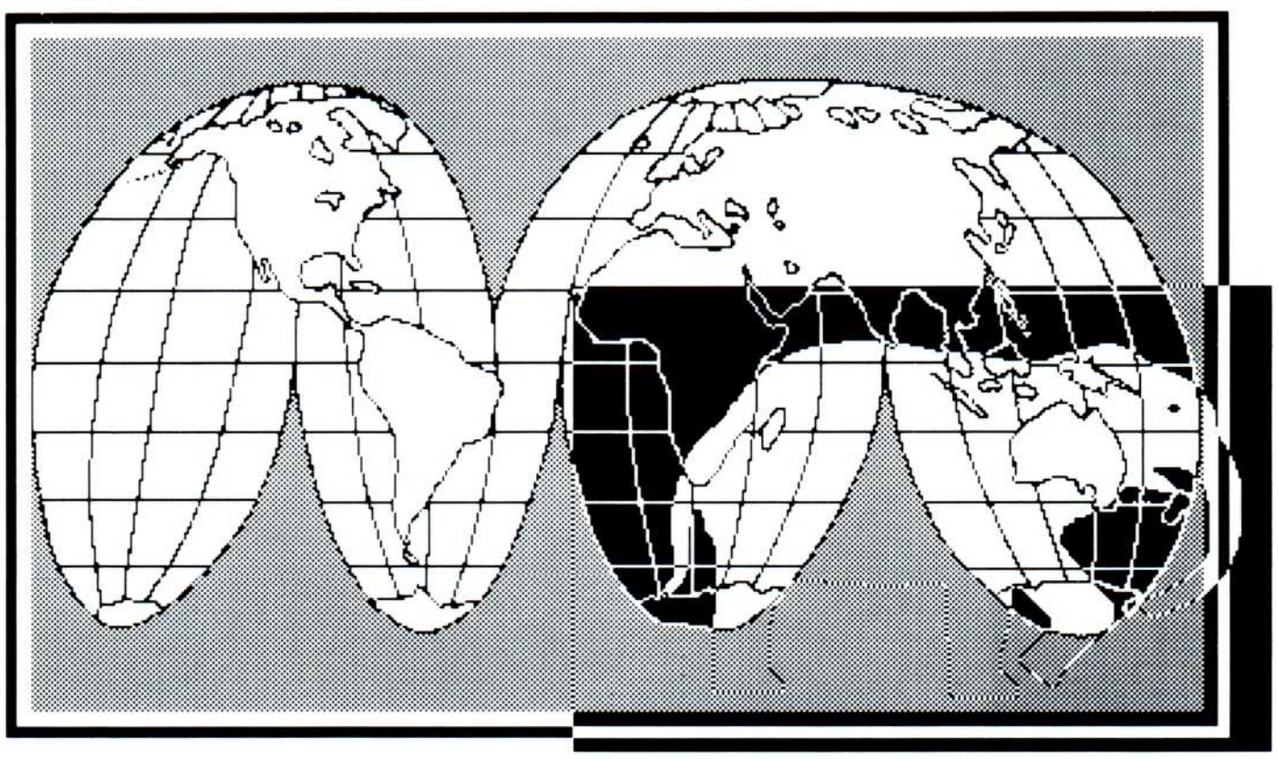


Figure 4-17
notSrcXor example

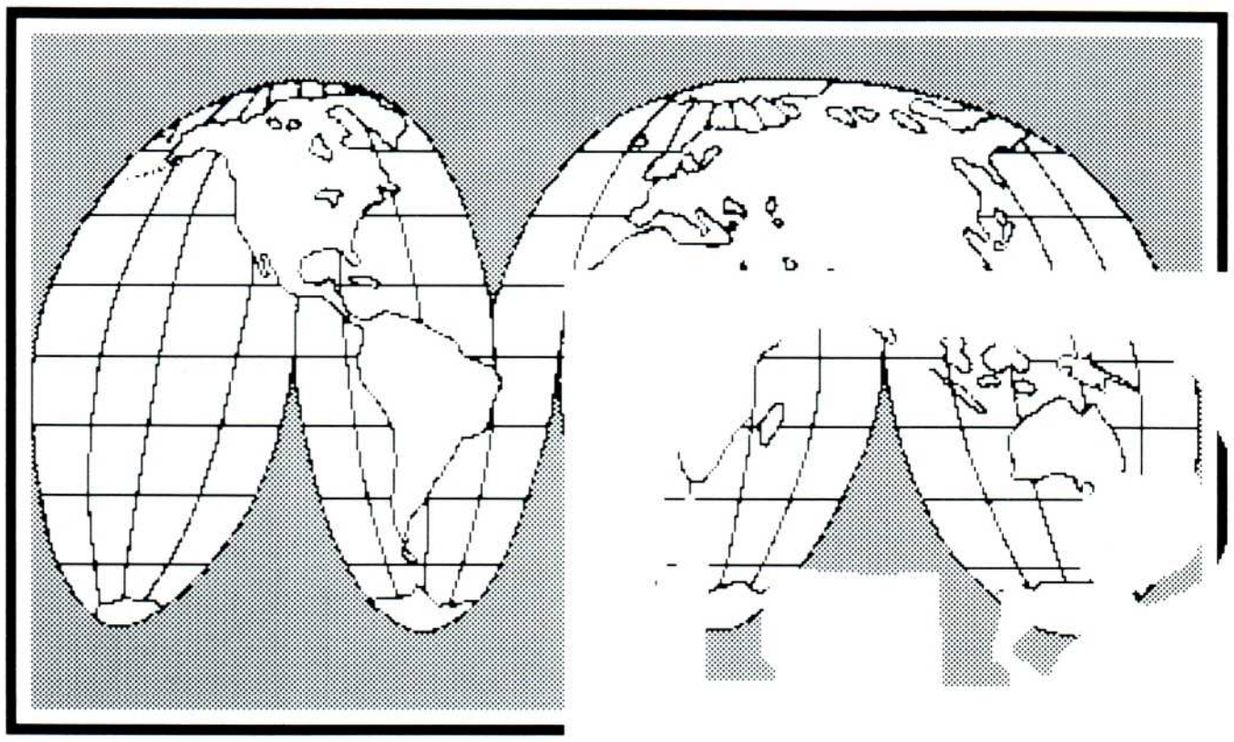


Figure 4-18
notSrcBic example

❖ *Note:* The combination of pixels is calculated only for the picture area; the remaining background is unchanged.

Layout variables

In addition to fields, text, and graphic objects. 4th Dimension layouts can contain variables. You can set up layouts for data entry using variables. You can also design dialog boxes to suit Macintosh user interface standards. You can also create complex page setups, which include data extracted from other files and substructures. Figure 4-19 shows a layout containing text, fields, and all of the types of layout variables.

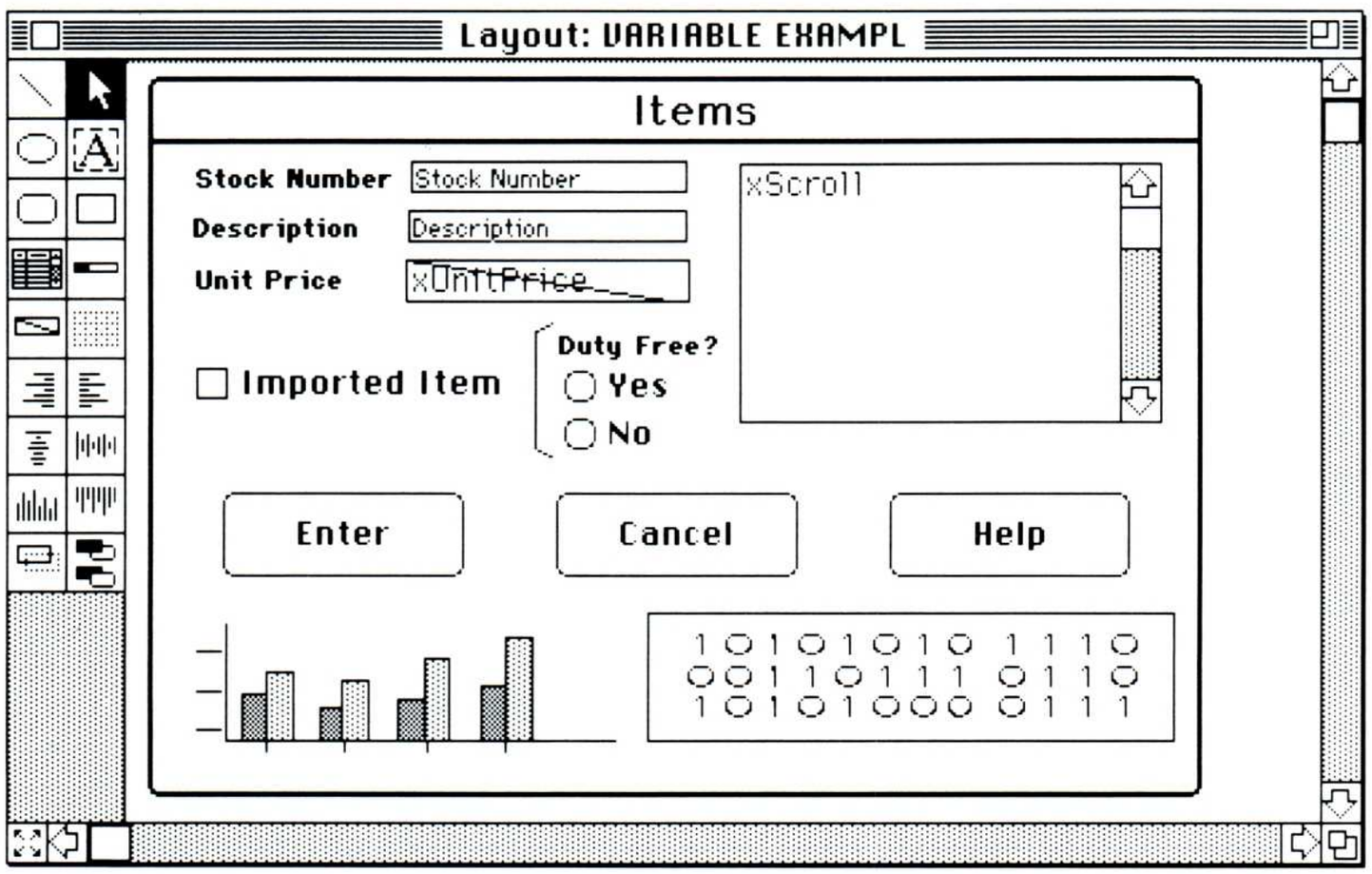


Figure 4-19
Standard layout variables

When you create a variable within a layout, the Format of variable dialog box appears (see Figure 4-20). Use the dialog box to specify the type, name, and format for a variable and for labeling button variables.

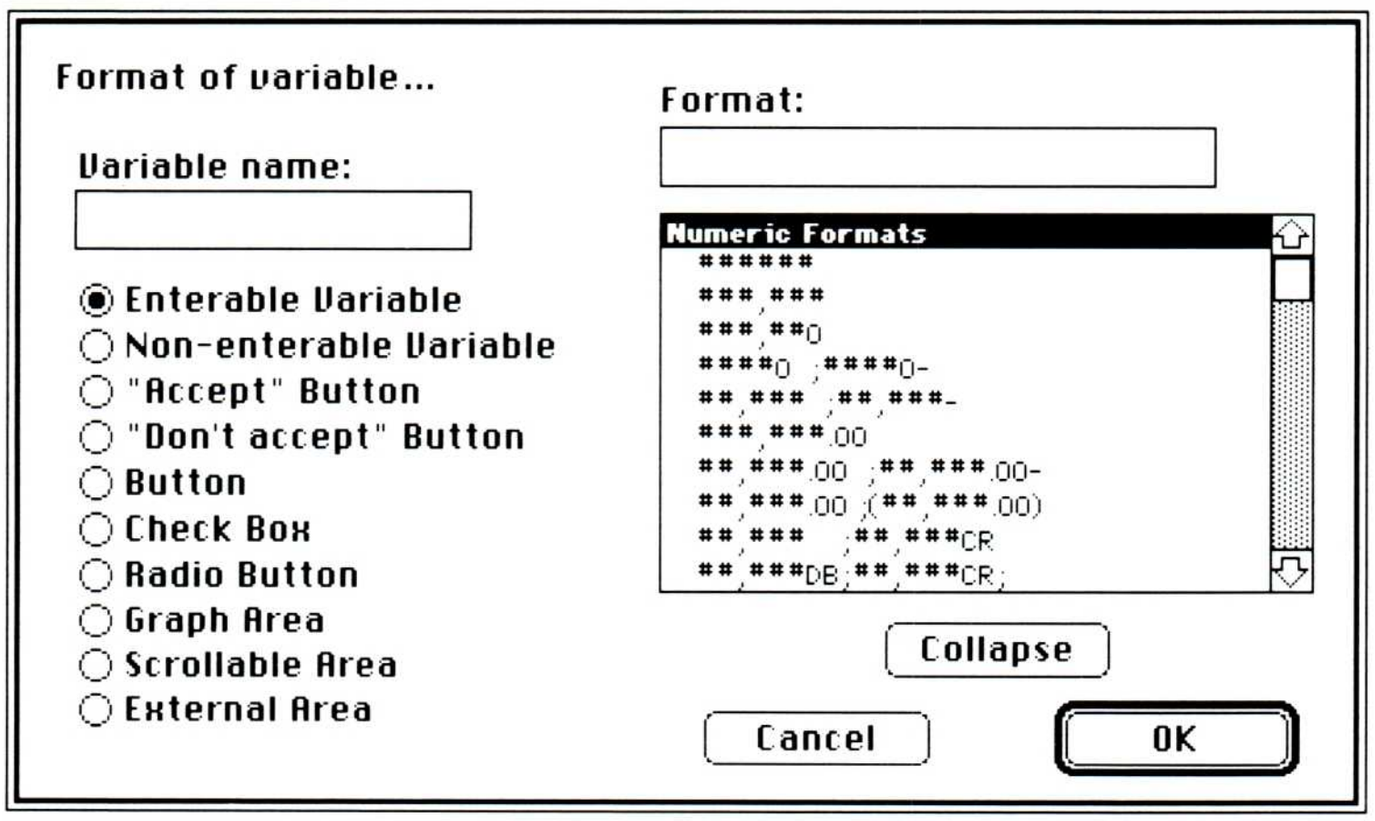


Figure 4-20
Format of variable dialog box

The dialog box shown in Figure 4-20 lets you specify the format for Numeric, Date, and Picture type variables. You format variables in the same way, with the same symbols, that you format a field. See the earlier section, "Formatting Fields Within a Layout."

Enterable and Non-enterable variables

Enterable and Non-enterable variables enable you to view, in a given layout, data stored in a file other than the current file. 4th Dimension supports five types of variables:

- Alphanumeric
- Numeric
- Date
- Boolean
- Picture

Variable typing takes place when a statement assigns a value to a variable. The variable takes on the type of the value assigned. You can use variables in layouts for a number of purposes:

- By placing variables in a layout, you can view data contained in a linked record you loaded during data entry.
- You can create a mailing label, in which the entire layout consists of a single variable, by concatenating all necessary values into one extended string (including carriage returns).
- You can concatenate data from a customer file and display it in a layout. For example:

  ```
  Title+" "+Last Name +" "+ First Name+" "+Age
  ```

- You can assign the total of numeric values stored in a substructure and display the result.

4th Dimension ignores the Enterable or Non-enterable attributes in an input layout, because you can't enter data into variables in an input layout. You can only enter data into variables in a dialog layout. When you use a layout as a dialog, you may enter data into Enterable variables, and use Non-enterable variables to display data. This becomes particularly handy for displaying instructions and values. You can reuse the same dialogs and change text on the fly through procedures. Just change the Non-enterable variable contents.

❖ *Note:* When you use a layout as a dialog, the layout displays current field values. These fields are not enterable.

Accept, Don't Accept, and Button buttons

A button variable generates a Macintosh-type button within a layout. A Macintosh button is an area you click to validate a choice or confirm an action. The button appears as a rounded-corner box containing a label. You choose the name of the button variable and its label in the 4th Dimension Format of variable dialog box.

❖ *Button:* This book uses the term "button" to indicate any of the layout variables the user can click to choose a state or action. Thus, "button" includes not only Accept, Don't Accept, and Button buttons, but radio buttons and check boxes.

Accept and Don't Accept buttons

An Accept button validates a data entry. Thus, it is equivalent to the action of the Enter button in a 4th Dimension record input layout and to pressing the Enter key. A Don't Accept button cancels data entry. Its action is similar to Cancel in a 4th Dimension record input layout and to pressing the Command-. (period) key combination.

Important

Including **any** kind of button in an input layout automatically cancels 4th Dimension's default button panel. Failing to supply both an Accept button and a Don't Accept button in a layout will leave you in an undesirable state. For example, if you include only an Accept button or a Don't Accept button, you can only Accept by pressing the Enter key or Cancel by pressing Command-. (period).

Understanding how to test to see if a user has validated an entry is important. How your tests work depends on what kind of buttons and how many of each kind you've installed in your layout. Unless you've made assignment statements to the contrary, all buttons are set to 0 when you summon a layout. Therefore, you can use a sequence of **If...End if** statements or a **Case of...End case** construction to test button values and take appropriate action. You can write tests either in a global procedure or within a layout procedure.

For example, if you create a layout with one Accept button and one Don't Accept button, you can find out which button was clicked by testing the OK system variable or the Accept button variable. Both will equal 1 if the user clicked the Accept button and 0 if the user clicked the Don't Accept button.

On the other hand, if your layout includes more than one Accept button or more than one Don't Accept button, you need to test each button variable, rather than the OK system variable.

Example

You create a procedure that lets the user choose options in a dialog box , like search, print, view, change, or delete the current selection. The variables in this dialog box include one Don't Accept button whose variable is bCancel and whose label is Cancel and five Accept buttons:

- bSrch labeled Search
- bList labeled List Selection
- bPrnt labeled Print Selection
- bMdfy labeled Modify Selection
- bDel labeled Delete Selection

Here is the procedure:

```
DEFAULT FILE ([filename])
bCancel := 0
While (bCancel = 0)
      `While the user does not cancel, display the dialog box layout named
       strexpr belonging to the [filename] file.
   DIALOG (strexpr)
   If (bCancel = 0)
     Case of 'User has not cancelled
       : (bSrch = 1)
          `DoSearch
       : (bList = 1)
          `DoDisplay
       : (bPrnt = 1)
          `DoPrint
       : (bMdfy = 1)
          `DoModify
       : (bDel = 1)
          `DoDelete
     End case
   End if
End while
```

Button buttons

Button buttons react to clicks just like other buttons, except that they don't complete data entry when clicked. Thus, the user can perform actions without having to quit the current entry (whether in a record or a dialog box). Examples of Button buttons include summoning a help screen, advancing to the next page of a long input layout, and performing calculations.

Unless you make an assignment to the contrary, 4th Dimension sets Button variables to 0 before displaying the layout. When clicked, a Button variable returns 1. Always test a Button button within its layout procedures and not from a global procedure.

To test a button, test the value of the Button variable. It contains 1 if the button was clicked and 0 if not.

Check boxes

Use check boxes when you want the user to be able to make one or more choices from among related alternatives. A check box is a rectangular area in which you can place a mark (actually an X, not a check). The box's label sits to the right of the check box. When such a box is checked, its value is 1; if not, its value is 0. You set a check box by assigning it a 1. To clear a check box, assign it a 0. Specify the variable name and check-box title within the Format of variable dialog box. Assign and test check-box values only in a layout procedure.

Radio buttons

Use radio buttons when you want to limit the user to one choice from among several related alternatives. For example, choose one communications protocol. When the user clicks a radio button, 4th Dimension sets its value to 1 and the values of all other radio buttons in the group to 0. You should initialize a default radio button by assigning it a 1. Assign and test radio buttons only in a layout procedure.

To create a set of radio buttons, begin the variable name of each button with the same letter, for example, **B1, B2, B3,** and so on. When testing radio buttons, remember that only one radio button within the group can be set to 1. The best testing strategy is to test the value of each radio-button variable in the group inside a **Case of...End case** structure. Only one button can be 1 at a given time.

4th Dimension automatically sets all radio buttons to 0 before the layout is displayed. Initialize a default radio button for the layout by assigning it a 1 during the Before phase. Specify variable names and labels in the Format a variable dialog box.

Graph areas

You draw a graph in a Graph area by executing a **GRAPH** command in a layout procedure. You can choose among eight different graph types:

- ☐ column
- ☐ proportional column
- ☐ stacked column
- ☐ line
- ☐ area
- ☐ scatter
- ☐ pie
- ☐ picture

The **GRAPH** command requires an Alpha value to label the X-axis and one or more Numeric values to graph on the Y-axis. You can have stored the numeric values in a subfile, an array, or a variable table. Further, data in both subfiles and arrays may be displayed on the same graph. Specify the name of the Graph area in the Format of variable dialog box.

Scrollable areas

4th Dimension provides a Scrollable area to display any array of values.

Specify the name of the Scrollable area in the Format of variable dialog box. This name is the same as the name of the array containing the data. As an example, give the area and the array the name List. Values that will appear in this area when the layout is displayed will belong to that variable table and will be named List1, List2, through List*n*. (The maximum value of *n* is 32,767.) Specify the number of items to be contained in the table in the List0 variable. You can tell which value was selected by testing the List variable. This variable contains the number of the item selected.

Different elements of the same array can contain different types of data.

❖ *The 0 variable:* The X0 variable contains the number of items shown in the Scrollable area X. If you have 28 items to display, set X0 to 28. Additional variables—X29, X30, etc.—may exist in memory. If X0 is greater than the number of items in the list, 4th Dimension provides selectable blank lines.

If you change the value contained in an element of the array, you must explicitly update the screen with the **REDRAW** command, which takes the Scrollable area name as argument.

Example

Figure 4-21 shows a layout named "Generator." The List variable at the top left of the layout is the Scrollable area.

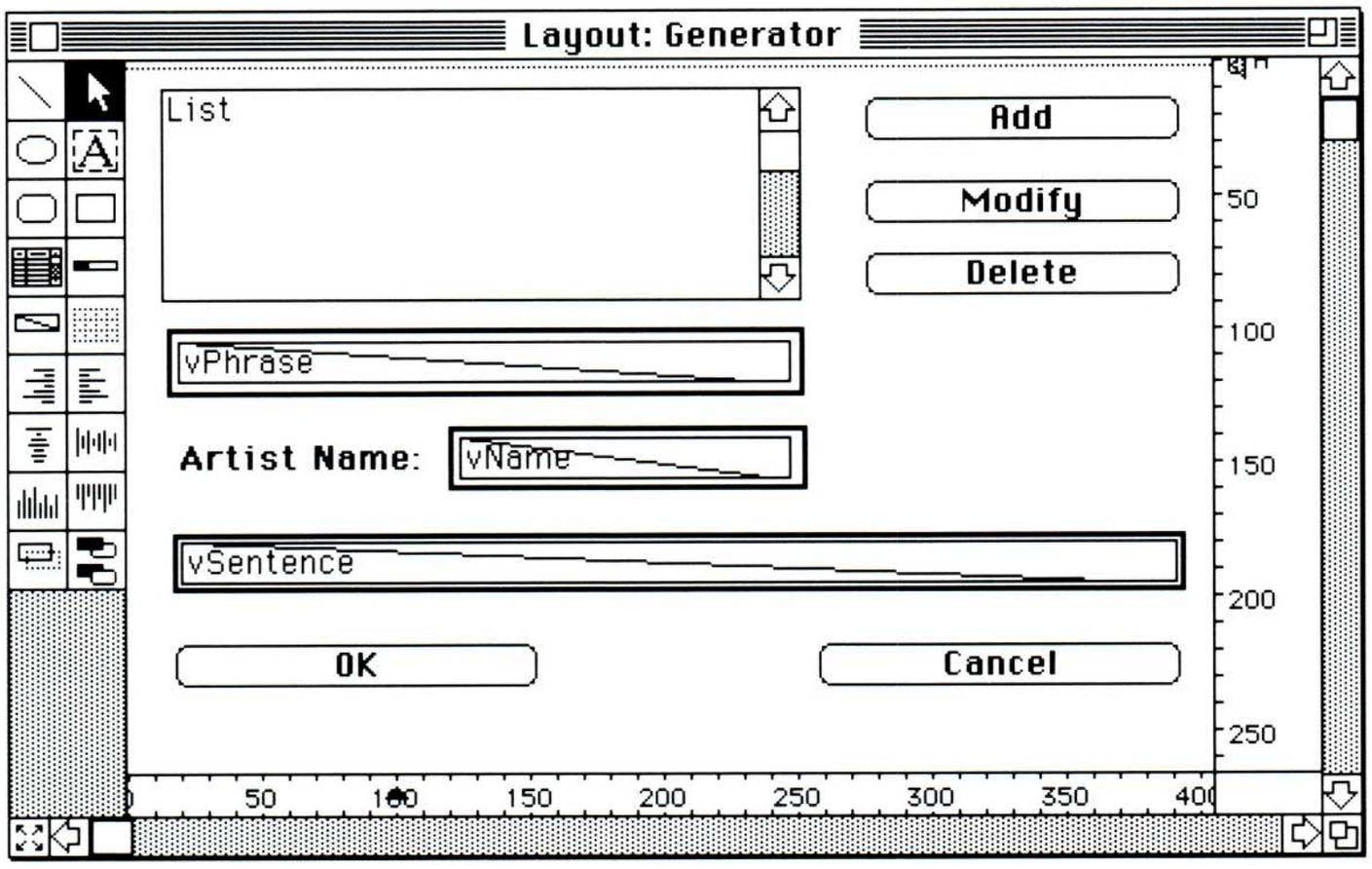


Figure 4-21
Generator layout with Scrollable area List

List is a Scrollable area. Add (bAdd), Modify (bMod), and Delete (bDel) are Button buttons. vName and vPhrase are Enterable variables, while vSentence is a Non-enterable variable. OK is an Accept button. Cancel is a Don't Accept button.

Create a menu bar with the menu item Writer to invoke the global procedure WriteSentence :

```
`Procedure: WriteSentence
`Calls the dialog "Generator" which generates a sentence given a variable vName
`entered by the user and a choice from the list List in the scrollable area.
DIALOG([Writings];"Generator")
While(bOK=1) `User clicked the Accept button bOK
   CREATE RECORD([Writings])
   [Writings]Phrase:=vSentence `Assign sentence into field in record
   SAVE RECORD([Writings])
   DIALOG([Writings];"Generator")
End while
```

The WriteSentence procedure activates a dialog box using the Generator layout. Here is the layout procedure for Generator:

```
   `Layout procedure: "Generator"
   `Manages the dialog that creates a descriptive sentence about an artist.
If (Before)
      `Before the layout appears on the screen.
    If (Undefined (List0) ) `initialize array containing descriptive attributes for artists
         `Undefined is a 4th Dimension standard function that returns TRUE when the variable
         `passed to the procedure is of type undefined.
         `Here, check whether List0 is undefined to create the array. If it is
         `defined,you won't go any further.
       List1 := "Impressionism"
       List2 := "painting cityscapes"
       List3 := "fine sculptures"
       List4 := "studying in Paris"
       List5 := "many reasons"
          `Assign the number of List items to List0.
       List0 := 5
    End if
      `Disable the Delete and Modify buttons, before the layout is displayed.
    DISABLE BUTTON (bDel)
    DISABLE BUTTON (bMod)
      `Initialize the dialog Enterable variables.
    vName:= "Picasso"
    vSentence:=""
    vPhrase:=""
    List:=0
End if ` End the Before phase
If (During)
      `Here, the layout statements are invoked when you modify a variable or when you click a
      `Button button or any value in the scrollable area.
    Case of
    : (bAdd = 1)
         `The user clicked the Add button. Add new phrase to the end of the list.
If (vPhrase # "")
    List0 := List0 + 1
    List{List0}:=vPhrase
    List:=List0
End if
```

```
: (bMod = 1)
        `The user clicked the Modify button.
    If (vPhrase # "")
            `If the vPhrase Enterable variable is not empty, copy it to the selected item.
        List{List}:=vPhrase
        REDRAW (List)
    Else `user has selected an item to modify.
        vPhrase:=List{List}
    End  If
: (bDel = 1)
        `The user clicked the Delete button.
    If (List0 # 0)
            `Copy the selected item number to i.
        i := List
            `While i is less than the number of items in the table.
        While (i < List0)
              `Copy the value of the i + 1 item to preceding item.
           List {i} := List{i+1}
           i := i + 1
        End  while
           `Specify in List0 that the table contains one item less.
        List0 := List0 - 1
           `Specify in List that there is no is selected item.
        List := 0
    End  If
End  case
If ((List # 0)&(vName#""))
        `If an item is selected in the scrollable area, calculate the following sentence:
    Sentence := "The artist " + vName + " is known for " + List{List} + "."
        `Enable the Delete and Modify buttons.
    ENABLE  BUTTON (bDel)
    ENABLE  BUTTON (bMod)
Else
        `Else, place the empty string in the sentence:
    Sentence := ""
        DISABLE  BUTTON (bDel)
        DISABLE  BUTTON (bMod)
    End if `Test choice from List and name entered
End if `End During phase.
```

The dialog box will look like the one shown in Figure 4-22.

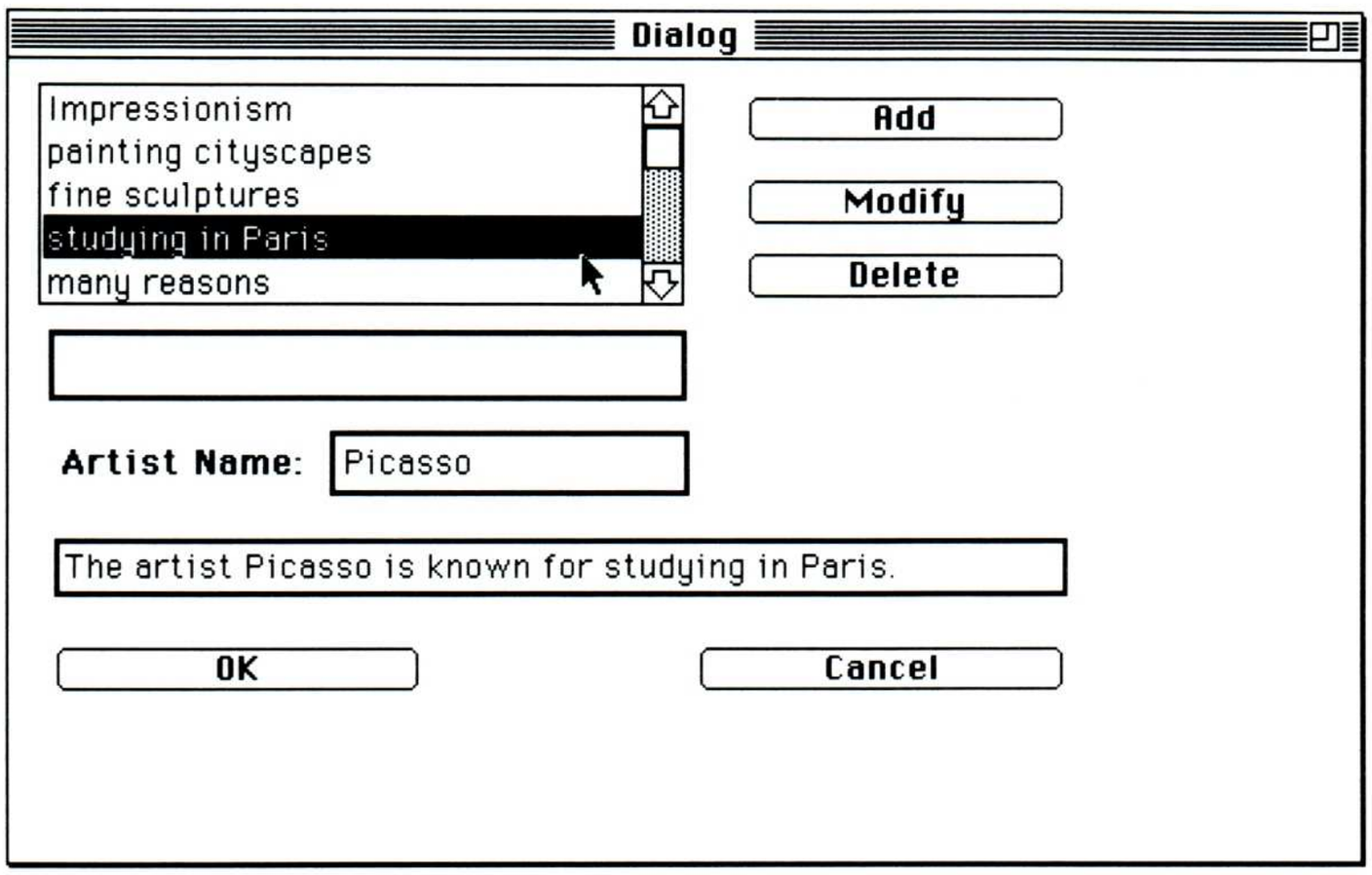


Figure 4-22
Working Generator dialog box displaying values

External areas

External areas call external routines written to augment 4th Dimension's built-in routines. See Appendix D in *4th Dimension Command Reference* for a complete discussion of external areas.

Chapter 5

File Links

Multiple-file databases have a number of advantages over single-file databases. These advantages include

- more efficient use of development time
- more efficient use of disk space
- more efficient data entry
- fewer errors during data entry

This chapter demonstrates these advantages through an example that begins with a single-file database and develops it into a multiple-file database. Most particularly, the chapter shows you how to use indexed searches and file links to work with multiple files and follows this order:

- a single-file database
- the **SEARCH BY INDEX** instruction
- various link instructions, beginning with **LOAD LINKED RECORD**
- mandatory links
- handling duplicate linked records
- using the at sign (@) with links

Single-file approach

The example deals with a typical business database situation: managing data about professional associates. It begins with a single-file example, named Contacts. Figure 5-1 shows its structure.

Contacts	
Last Name	A
First Name	A
Title	A
Position	A
Company Name	A
Addr 1	A
Addr 2	A
City	A
State	A
Postal Code	L
Telephone	A

Figure 5-1
Single-file database structure: Contacts file

In the [Contacts]Last Name, [Contacts]First Name, [Contacts]Title fields, you store the last name, first name, and title of a business associate. In the [Contacts]Position field, you store the position of the person in the company. In the remaining fields, you store the address and phone number of the company. Strictly speaking, this structure looks correct. However, it's quite inadequate in practice, because

- every time the user enters a new name, the user must enter the address of the company, even if the file already contains names of others belonging to the same company

  This is a waste of time and creates a strong potential for error when retyping data.

- the redundant company information takes up unnecessary space on the disk

  Suppose the file contains 5000 names, that there are approximately 10 persons for every company, and that the alphanumeric fields are 75% entered. Table 5-1 shows the size of a record on disk.

Table 5-1
Record size for Contacts file

Field	Type	Size in bytes
Last Name	Alpha 25 75% entered, 19 bytes + 1 for length	20
First Name	Alpha 20 75% entered, 15 bytes + 1 for length	16
Title	Alpha12 75% entered, 9 bytes + 1 for length	10
Position	Alpha 20	16
Company Name	Alpha 25	20
Company Adr1	Alpha 25	20
Company Adr2	Alpha 25	20
City	Alpha 25	20
State	Alpha 2	2
Postal Code	Long Integer	4
Telephone	Alpha 12	10

The table shows that an average Contacts record takes up 158 bytes on the disk.

If there are approximately 10 persons for every company and 500 different companies in the file, the file will contain 5000 records for a disk storage total of about 790,000 bytes. 4500 records contain redundant information. From the table, you can see that company information takes up 96 bytes. The 4500 redundant records contain approximately 432,000 bytes of redundant data. Thus, more than half of the file contains duplicated information. Anyone with a hard disk might not have space problems, but it still takes time to type and access 432,000 bytes of redundant information.

Two-file solution

Because 4th Dimension can work with more than one file in a database, you can create dramatic economies in entry time and disk storage by removing the redundant information from the **Contacts** file and putting it in a second file, **Companies**. Figure 5-2 shows the new database structure. Notice that the two files have one field of information in common: **[Contacts]Company Name** and **[Companies]Name**. Having a duplicate item of information is crucial when building relationships between files.

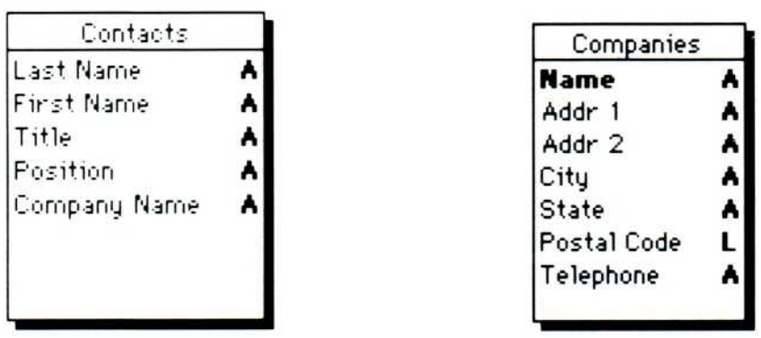


Figure 5-2
Two-file database structure: Contacts and Companies files

Here's how the new database works. The user stores the 500 companies in a file named **Companies** and the 5000 persons in the **Contacts** file. You must, however, keep the **[Contacts]Company Name** field in the **Contacts** file to know which company the person belongs to. The **Contacts** file will take up approximately 410,000 bytes, and the **Companies** file will take up approximately 48,000 bytes. Thus, the database will take up about 458,000 bytes. This structure frees 332,000 bytes on disk. Further, the user won't have to type redundant information. The two-file database saves disk space and time by placing data in more than one file.

However, the structure needs some improvement. When the user enters a record in the **Contacts** file, he needs to know right away whether the company specified already exists in the **Companies** file. This means the user must make a quick search of the **[Companies]Name** field. To do so, you must index the search field. For 500 records, the index file for the **[Companies]Name** field will take up about 33,000 bytes. Even so, this database takes far less space than the first one.

Once you have created the new structure, you build layouts and procedures to manipulate the two files. You first create a layout for the **Contacts** file, which you name **Entry**. Then you use it to enter business associates. You also create a layout named **List** for listing data about associates. Figure 5-3 shows the **Entry** layout of the **Contacts** file.

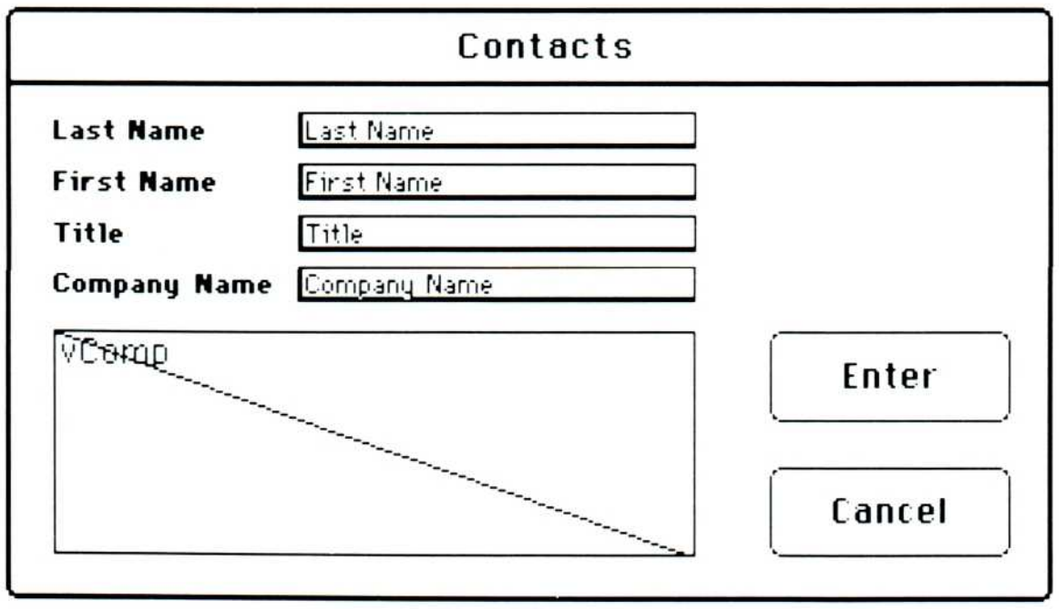


Figure 5-3
Entry layout for Contacts file

The **vComp** variable will display the address and phone number of the company where the person works. The address and phone number will be taken from the **Companies** file and placed in the variable by the layout procedures in the **Entry** layout. Figure 5-4 shows the **List** layout of the **Contacts** file.

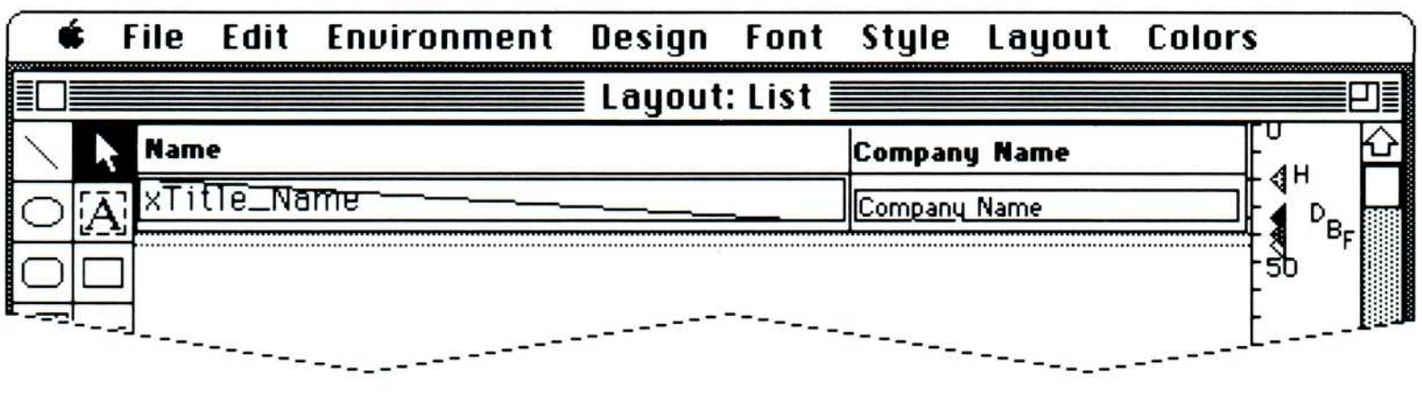


Figure 5-4
List layout for Contacts file

To concatenate and show the title, last name, and first name of the person, you assign the three corresponding fields to the **xTitle_Name** variable using the layout procedure for the **List** layout:

```
xTitle_Name := Title + " "+LastName + " " + First Name
```

You also create a layout for the **Companies** file, which will be used for entries and another layout for listing companies. Figure 5-5 shows the **Entry** layout for the **Companies** file.

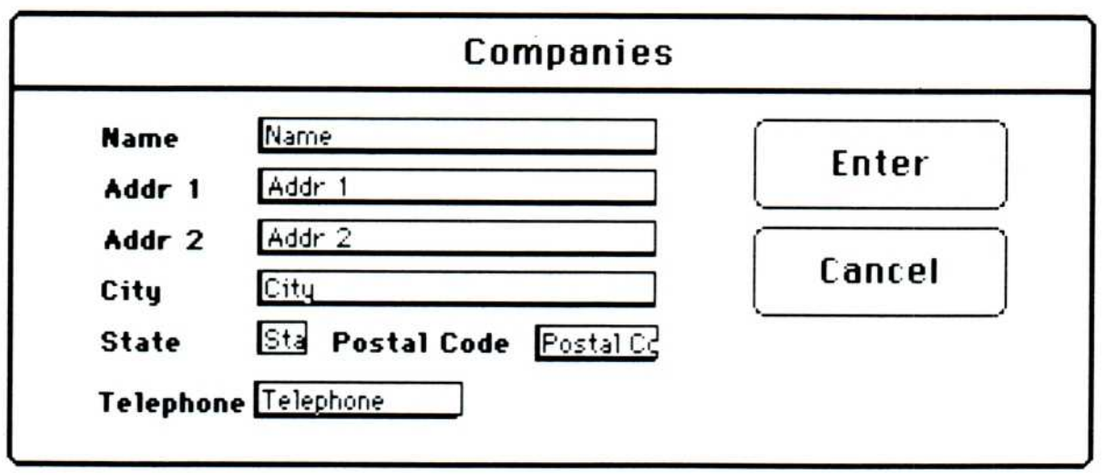


Figure 5-5
Entry layout for Companies file

Figure 5-6 shows the List layout for the Companies file.

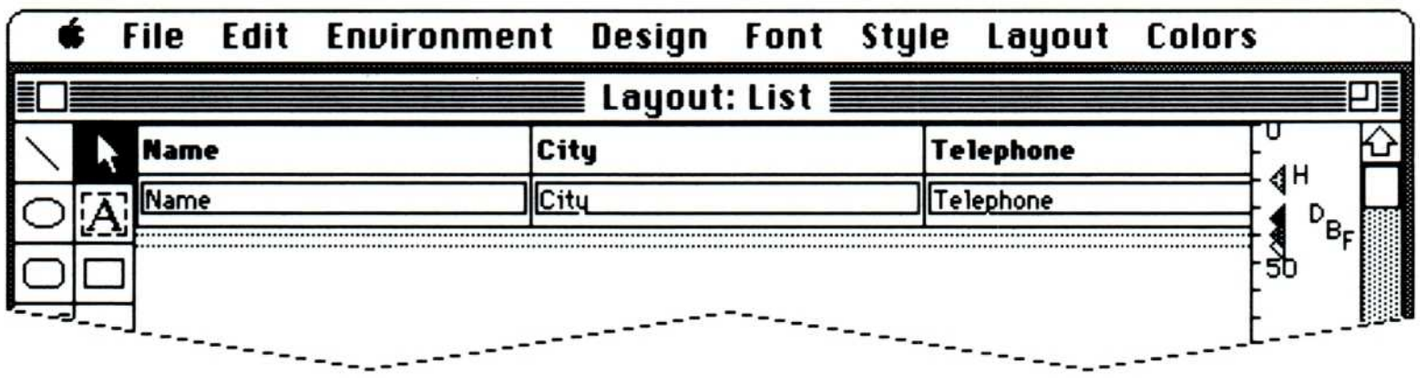


Figure 5-6
List layout for Companies file

The layout procedure for the Contacts file Entry layout will manage the Companies file when entering records into the Contacts file:

```
   `Entry layout procedure for the Contacts file
If (During)
      `During data entry.
   Last Name:= Uppercase (Name) `Automatically convert last name into uppercase letters.
      `Automatically convert the first letter of the first name to uppercase and the
      `remaining letters to lowercase.
   First Name := Uppercase (Substring (First Name ; 1; 1)) + Lowercase (Substring (First
      Name ; 2; Length (First Name)))
   If (Modified (Company Name))
         `Modify is a 4th Dimension standard function. Modified returns TRUE if the user modifies
         `the fieldname during data entry.
         `If the user modified the Company Name field, proceed to a search by index in the
      `[Companies] file to find the company whose name is equal to the value the user typed.
SEARCH BY INDEX ([Companies]Name = Company Name)
If (Records in selection ([Companies]) # 0)
      `Records in selection is a standard procedure returning the number of records belonging
      `to the file selection given as parameter.
      `If at least one record exists in the Companies file, assign 1 to Comp Exists variable.
   Comp Exists := 1
Else
      `If no record is found in the [Companies] file, assign 0 in the Comp Exists variable.
   Comp Exists := 0
      `Specify if you want to create the company record. CONFIRM is a built in
      `4th Dimension command: it creates a dialog box displaying the strexpr as a
      `prompt message. A CONFIRM box has two buttons: OK and Cancel. If you click
      `OK, the OK variable takes on the value 1, else it takes on the value 0.
   CONFIRM ("The record " + Company name + " doesn't exist, do you want to create it?")
   If (OK = 1)
         `The user is given the opportunity to add a record in the Companies file in the
         `[Contacts] field entry. Considering the user interface, take an additional
         `precaution. If the Entry layout of the Companies file is used to add the
         `company record, you might change the CompaniesName field by mistake. To
      `prevent this, create another layout, named Entry2 in which a variable will
      `be placed to display the company name instead of the Name field and name that
      `variable vName.
vName := Company name
   `Select the Entry2 layout as the input layout of the Companies file.
INPUT LAYOUT ([Companies] ; "Entry2")
   `Add the Companies record entry by using the ADD RECORD standard procedure.
ADD RECORD ([Companies])
```

```
        If (OK = 1)
            `If you validate the Companies record entry, assign 1 to the Comp Exists
            `variable.
            Comp Exists := 1
        End  if
    End  if
End  if
If ( Comp Exists = 1)
        `If the Comp Exists variable contains 1, it either means that the company exists
        `or that it didn't exist but that the user added it during data entry: in that case,
        `display the address and the phone number of the company in the relations layout
        `with the help of the vComp variable.
    CR := Char (13)
    vComp := [Companies]Adr1 + CR + [Companies]Adr2 + CR + String ([Companies]Postal
      Code ; "00000") + " "
        vComp := vComp + [Companies]City + CR + [Companies]Phone Number
    Else
        vComp := ""
    End  if
End  if
```

Figure 5-7 shows the Companies file layout named Entry2.

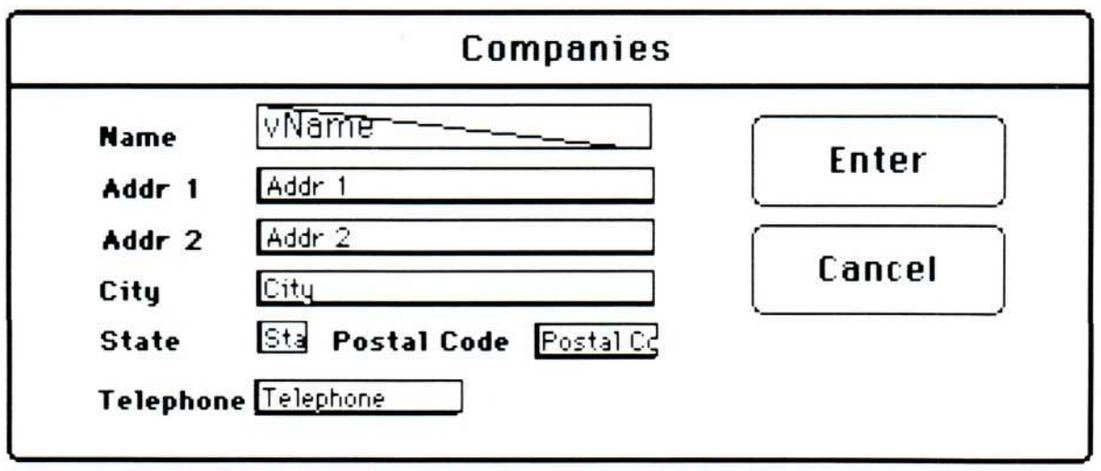


Figure 5-7
Entry2 file layout

To make sure the Companies record will be saved with the name, include the following statement in the layout procedure:

```
Name := vName
```

Thus programmed, the application lets the user

- search records in a file, while entering data into another file
- add records to a file, while entering data into another file

In the above example, you've seen that you can copy some Companies record values to variables and display them in the Contacts file layout. With the help of procedures, you can copy the company name entered in the Contacts file to the [Companies]Name field.

The database lets you

- □ display file data in another file layout
- □ exchange data between files

Suppose that the user has already entered quite a number of records and then realizes he entered the wrong phone number for company X. With the first structure mentioned above, the user had to search all the business contacts belonging to that company and change the phone number in all the records found in the **Contacts** file. With the second structure, the user has to change the value only in one record of the **Companies** file. Thus, the database saves time when processing data.

Suppose you're dealing with the three-file database structure shown in Figure 5-8.

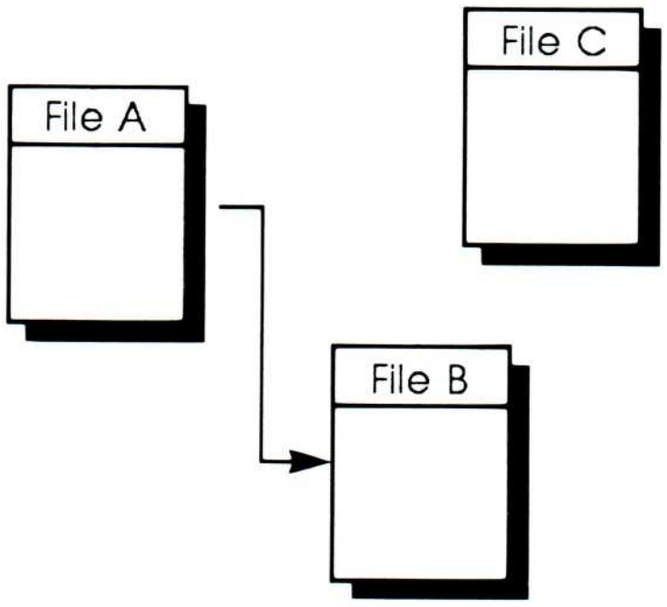


Figure 5-8
Three-file database structure

This database consists of three files: **[A]**, **[B]**, and **[C]**. If you search one of the files—for example, file **[C]**—4th Dimension changes the file selection and the current record. It's important to note that the **[A]** and **[B]** files are not modified nor are their current selections, if any. This means that

- □ a table representing the current selection is kept in memory for every file of the database
- □ there is one current record, at the most, loaded in memory for every file of the database

See Figure 5-9.

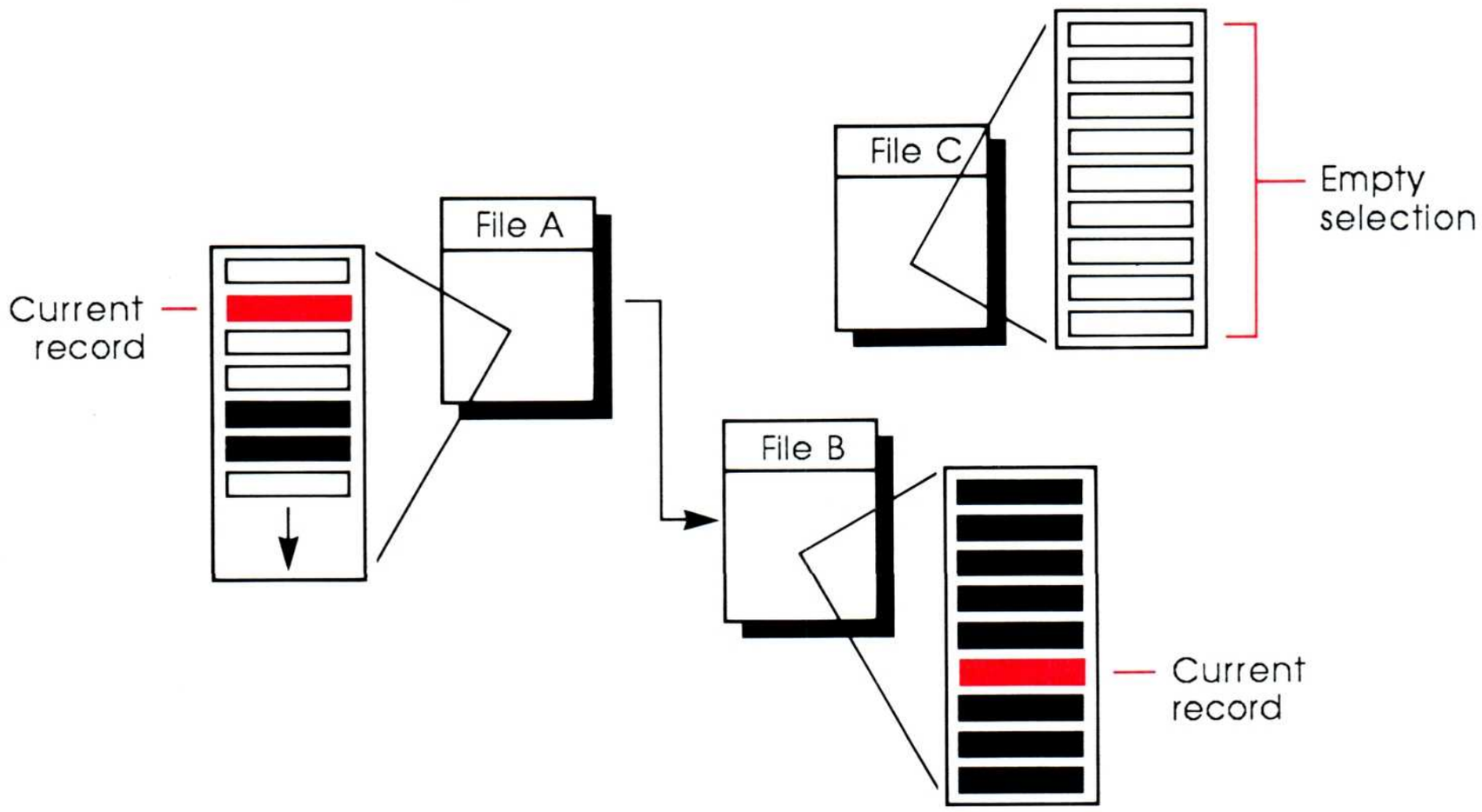


Figure 5-9
Current record and current selection for three files

Linking files

The above example shows that every time you work on a **Contacts** record (whether you're entering or modifying data) you have to do a search by index in the **Companies** file to find the company record of a business contact. What happens when 4th Dimension searches by index?

Figure 5-10 illustrates the use of the index table in the search process. This process takes four steps:

1. 4th Dimension goes through the index table until it encounters the searched value.
2. When 4th Dimension finds the value, it also finds the pointer to the record in the data file.
3. Using the record pointer, 4th Dimension reads the record directly.
4. 4th Dimension loads the record into memory.

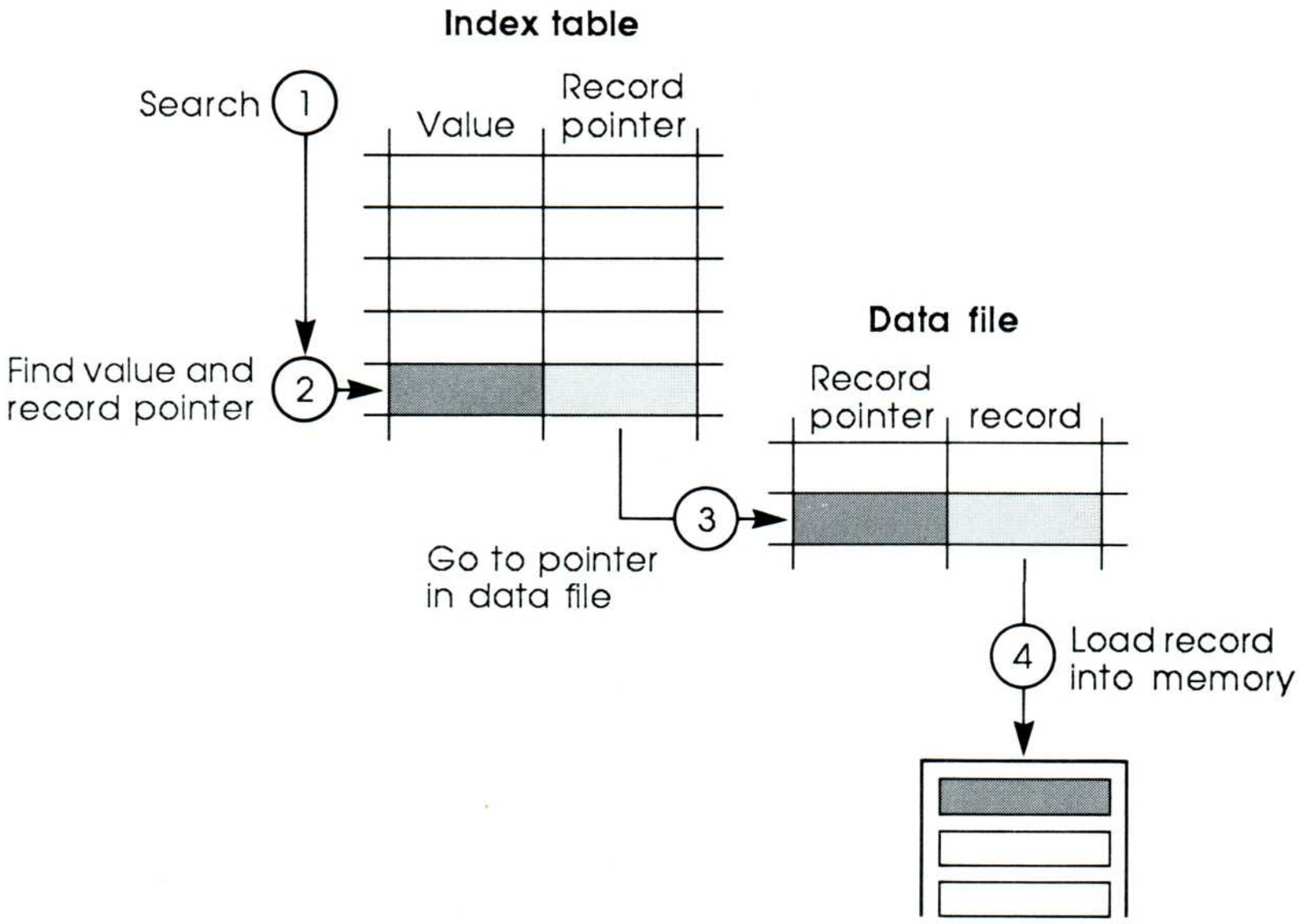


Figure 5-10
Searching and the index table

If you could store the pointer from step 2, subsequent searches would not be required. This is exactly what links do.

To create a link, make sure you're in the Structure window and drag the pointer from one field in the linking file to the appropriate field in the linked file. See Figure 5-11. (Refer to Chapter 1, "Database Structures," in *4th Dimension User's Guide* for details.)

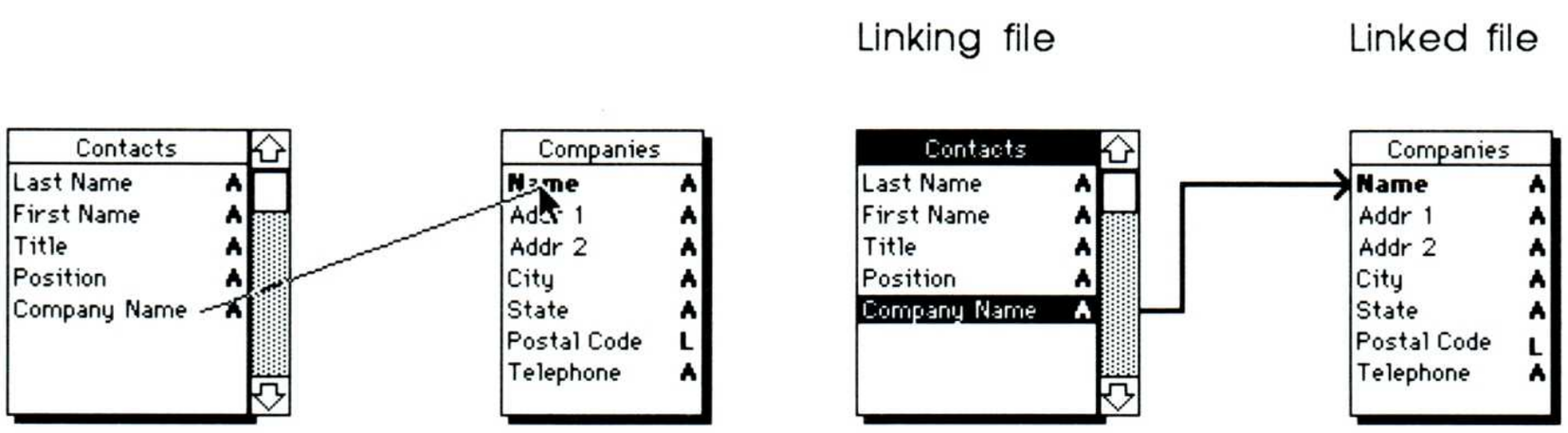


Figure 5-11
Drawing a link in Structure window

Always drag the pointer from the field in which you will enter an identifier, like a stock number (the linking field), to the field containing similar information (the linked field). These fields are the basis of the link. Through them, the linking record can draw data from the linked file and save data to the linked file.

4th Dimension automatically indexes linked fields.

How links work

When 4th Dimension establishes a link for the first time, it does an indexed search through the linked file to find the matching record. That record is loaded into memory and its pointer saved with the linking record.

Loading a linked record for the first time

Loading a linked record for the first time takes 4th Dimension five steps (shown in Figure 5-12):

1. 4th Dimension goes over the index table until it encounters the searched value.
2. When 4th Dimension finds the value, it also finds the pointer to the record in the data file.
3. Using the record pointer, 4th Dimension reads the record directly from the file.
4. 4th Dimension loads the record into memory.
5. 4th Dimension copies the pointer of the linked record into a special area of the linking record.

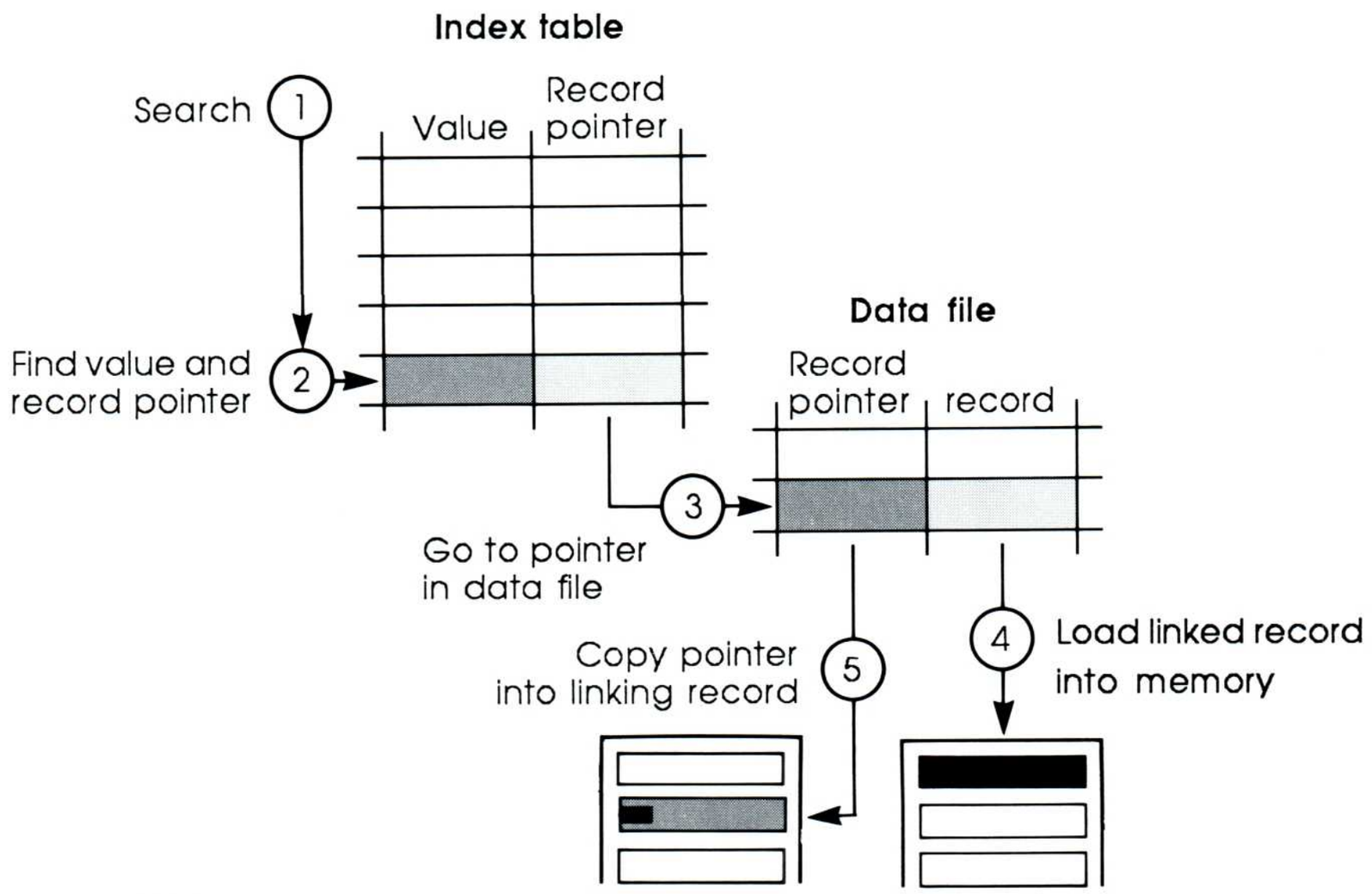


Figure 5-12
Loading a linked record for the first time

The next time you load a linked record

Once 4th Dimension has established a link for a record, getting the linked record takes only two steps (shown in Figure 5-13):

1. Because the pointer of the linked record is saved in the linking record, 4th Dimension can read it directly from the data file.
2. 4th Dimension loads the record into memory.

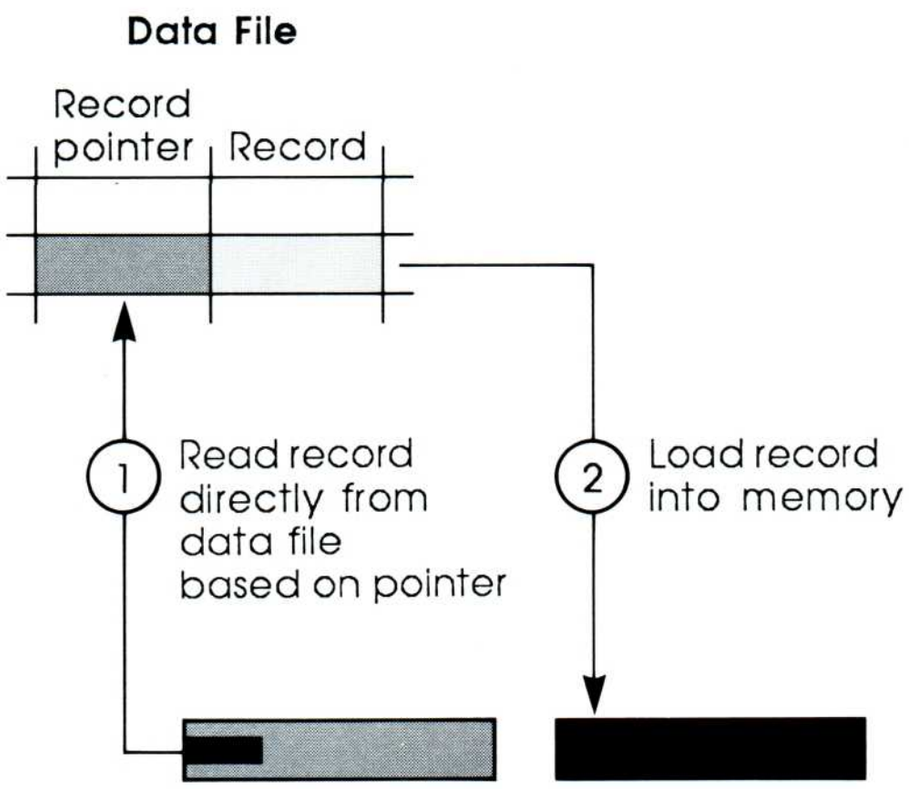


Figure 5-13
Loading subsequent linked records

Important considerations

Keep three things in mind when dealing with links:

1. When working with links, 4th Dimension takes care of record modification and deletion. All the records of all the files of a database are saved one after the other in a single file: the data file. If, when you modify a linked record, you change its size, the record might not necessarily be saved in the same place as before. Moreover, you can delete a linked record. Thus, the pointer to a linked record saved in a linking record can become incorrect while you're operating the database.

   4th Dimension automatically detects the problem and finds the new pointer to the linked record which changed after you modified the record. If the record has been deleted, 4th Dimension places a special value in the linking record to indicate that it no longer links to the record which has been deleted.

2. Using a link changes the current selection and the current record of the linked file. Upon loading a linked record for the first time, 4th Dimension does a search by index to find the record. Consequently, the file selection is changed and the linked record becomes the current record. The subsequent loading of that linked record will change the current selection of the linked file, and the linked record will become the current record.
3. A link handles records differently than a search by index when the linked file has two or more equal field values. In a file, you may have more than one record with the same value in a single field. Doing a search by index on that field will return a selection containing all those records and the first record of the selection. That is, the first record 4th Dimension encounters becomes the current record.

   If there are duplicate values in a particular field of a linked file, the **LOAD LINKED RECORD** command returns the first record of the selection. Unlike a search by index, which returns the selection of records containing the value and the first record of the selection as the current record, loading on link returns as a selection only the first record encountered with the correct value. The linked record becomes the current record. See "Dealing With Duplicate Values in Linked Fields" later in this chapter.

LOAD LINKED RECORD command

You establish a link between two fields with the **LOAD LINKED RECORD** command.

Let's go back to the example above and see what happens to the layout procedure for the Contacts file Entry layout when using a link:

```
   `Layout procedure for the Contacts file "Entry" layout:
If (During)
   Last Name:= Uppercase (Last Name)
   First Name := Uppercase (Substring (First Name ; 1; 1)) + Lowercase (Substring (First
      Name; 2; Length (First Name)))
   If (Modified (Company Name))
      LOAD LINKED RECORD (Company Name) `<--- the link is activated
      If (Records in selection ([Companies]) =1)
         CR := Char (13)
         vComp := [Companies]Adr1 + CR + [Companies]Adr2 + CR + String ([Companies]Postal
            Code ; "00000") + " "
         vComp := vComp + [Companies]City + CR + [Companies]telephone
      Else
         vComp := ""
      End if
   End if
End if
```

Notes

1. This procedure has the same effect as a search by index and the procedures as they were originally written. There are, however, two differences. The next time the user accesses the record, 4th Dimension won't have to do a search by index. The linked record will automatically be loaded into memory. However, if the Companies record doesn't exist, the procedures won't allow the user to create a record during data entry. Later on, you'll see how to deal with this problem.
2. Every record of the Contacts file points, at the most, to one record of the Companies file. This means that a Contacts record points either to a single record in the Companies file or to no record at all.
3. Many business colleagues can belong to a single company. This means that a record in the Companies file can be linked to more than one Contacts file record.

What this means is that numerous records in the linking field can point to one record in the linked field. Thus, a link always proceeds from the many to the one. See Figure 5-14.

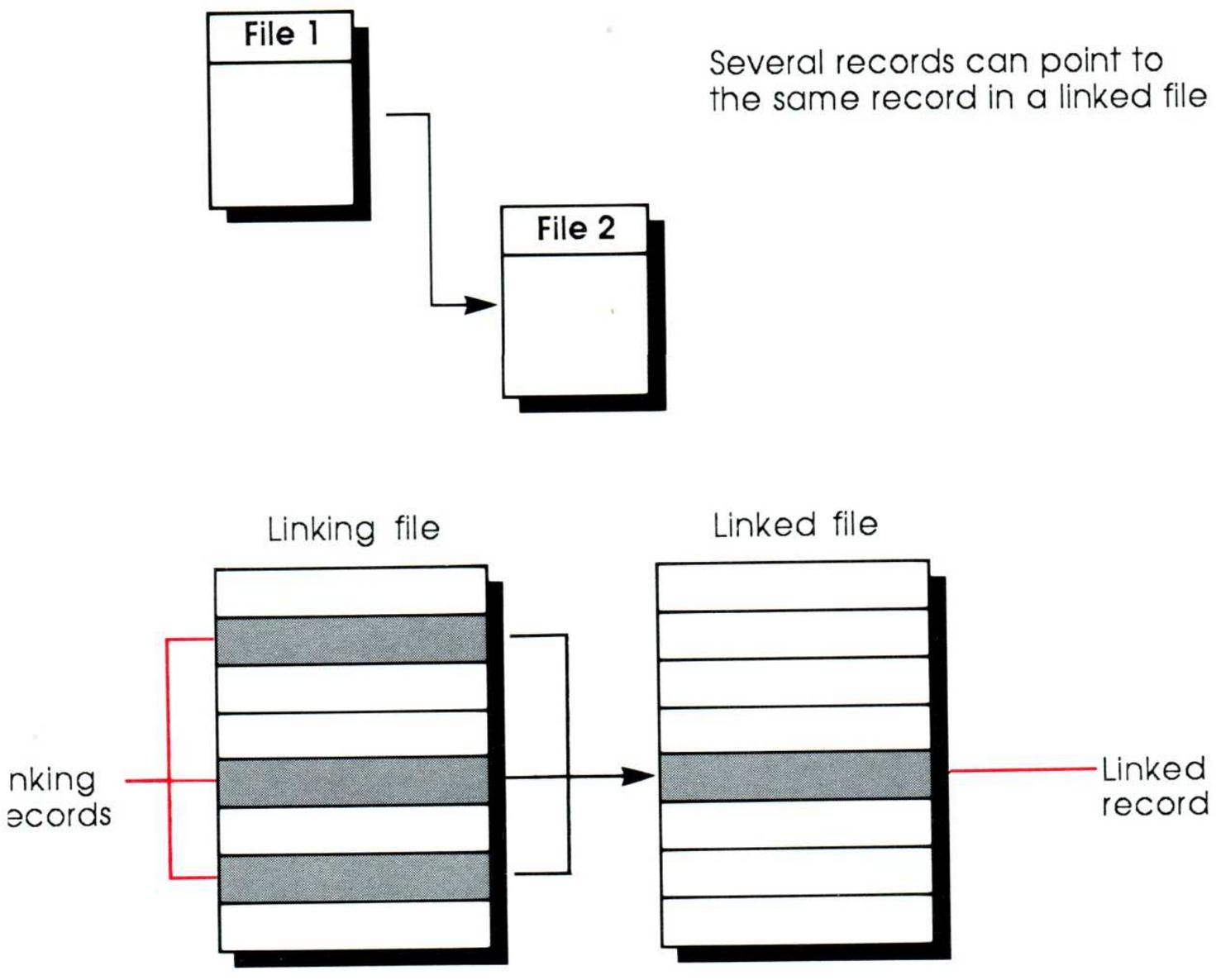


Figure 5-14
Linking from the many to the one

Mandatory attribute

The question may arise, how do you create a linked record when the record doesn't exist? This section provides two solutions. If a person belongs to a company, Solution 1 does the job, using the Mandatory attribute. Solution 2 works if the person doesn't belong to a company.

Solution 1

In the Contacts and Companies files example, you establish that an entry for an associate cannot be validated, if the company for which the contact works hasn't been specified. You give the [Contacts]Company name field the Mandatory attribute. This means that a Contacts record can be validated only if a Companies record is linked to it. 4th Dimension will automatically manage the creation of a new Companies record if the record doesn't exist. When you have given the linking field the Mandatory attribute, the **LOAD LINKED RECORD** command automatically displays the dialog box shown in Figure 5-15, if the Companies record doesn't exist. 4th Dimension use the current input layout to create the new record.

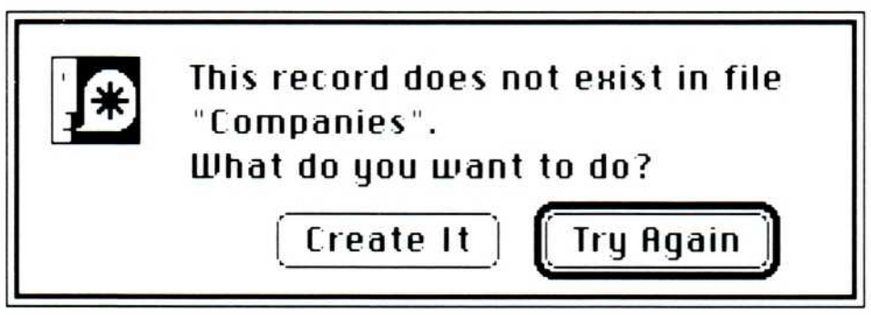


Figure 5-15
Create a record dialog box

If the user clicks the Create It button, 4th Dimension will create a Companies record linked to the Contacts. If the user clicks Try Again, the company record name can be entered again in the Contacts record, if it was incorrectly spelled the first time. The user can go to the next field of the Contacts record only after creating the company record or after specifying the name of an existing company.

In summary, if the field where the link originates is mandatory, 4th Dimension will automatically manage record creation during data entry, if the linked record doesn't exist.

Solution 2

A user may decide that a business colleague does not necessarily belong to a company. In that case, the [Contacts]Company Name field won't be Mandatory. You'll have to explicitly manage the creation of the linked record in the Companies file, if the record doesn't exist. Use the procedure mentioned earlier that works with the **SEARCH BY INDEX** command and replace it with **LOAD LINKED RECORD.** In most cases, it's necessary for an entered record to have a linked record. In that way, you take advantage of the automatic creation of the linked record if the record doesn't exist.

Dealing with duplicate values in linked fields

Let's go back to the Contacts and Companies files example. It's possible that two different companies bear the same name. In that case, **LOAD LINKED RECORD** will return the first company encountered with the entered name. It may be that this is not the desired record.

A second syntax for the LOAD LINKED RECORD command

There is a second syntax for **LOAD LINKED RECORD:**

LOAD LINKED RECORD (*fieldname1*;[*filename*]*fieldname2*)

fieldname1 is the linking field of the file where the link originates.
[*filename*]*fieldname2* is a field belonging to the linked file, which is usually different from the field to which *fieldname1* points.

With this syntax, the linked record is loaded in the same way as with the regular syntax with one exception. If more than one record is found in the linked file, 4th Dimension displays a list of those records, so that the user can choose the desired record. The window lists the records showing the matching values contained in the field which *fieldname1* points to and to the values contained in the [*filename*]*fieldname2*. In that way, the user can differentiate among records in which duplicate values exist in linked fields.

In the Contacts and Companies example, you replace in the layout procedures:

LOAD LINKED RECORD (Company Name)

with

LOAD LINKED RECORD (Company Name;[Companies]City)

For instance, if the user types SMITHS LTD in the Contacts record and there is more that one SMITHS LTD record in the Companies file, 4th Dimension displays a window similar to the one shown in Figure 5-16.

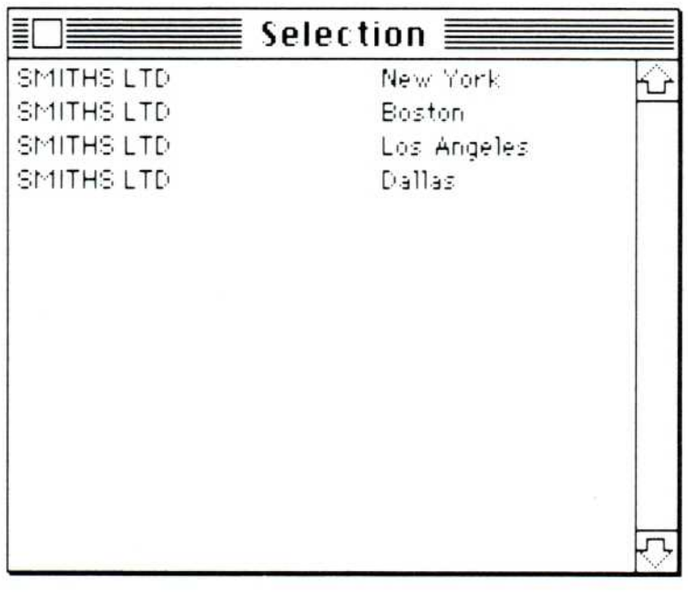


Figure 5-16
Scrollable window of duplicate values

Here, the user chooses the Companies record which will be linked to the Contacts record entered by clicking the desired name in the window.

❖ *Note:* If in the Companies file, there are two SMITHS LTD's in the same city, the case looks quite hopeless, because the user won't be able to differentiate the two companies. However, you can use a second argument for **LOAD LINKED RECORD,** different from [Companies]City for which you know no duplicate will exist. For example: [Companies]Phone Number.

Wildcards and the LOAD LINKED RECORD command

You can gain additional benefits from the second syntax

LOAD LINKED RECORD (*fieldname1*;[*filename*]*fieldname2*)

by placing the at sign (@) wildcard character at the end of a search string.

In the User environment, you can do a search on an alphanumeric field by typing some characters (the beginning of a name, for example) followed by an at sign. Such a search returns all file records containing a value beginning with the letters you typed. This possibility has been extended to **LOAD LINKED RECORD** using the second syntax. In the Contacts and Companies example, if you type SMI@ in the [Contacts]Company Name field, 4th Dimension displays the Selection window listing all Companies records with names beginning with "SMI" (see Figure 5-17).

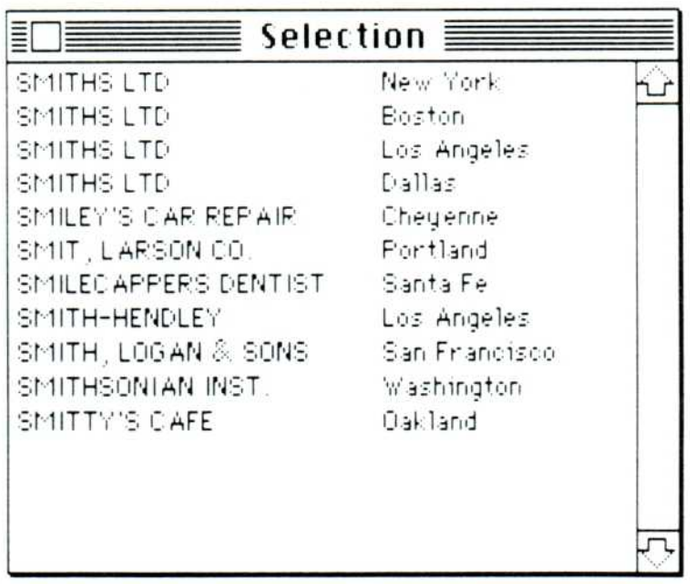


Figure 5-17
Selection window after wildcard search

Using the at sign is convenient, because the user doesn't have to memorize all the company names or enter an entire value.

SAVE LINKED RECORD command

This section introduces a new example database: an invoice system. The rest of the chapter concentrates on using various link commands to implement this system and add inventory and sales files to it. This section introduces the **SAVE LINKED RECORD** command and the **Old** function.

The database: an analysis

Figure 5-18 shows the database's structure. Notice that it includes a subfile for invoice detail and that it links to a customer address file.

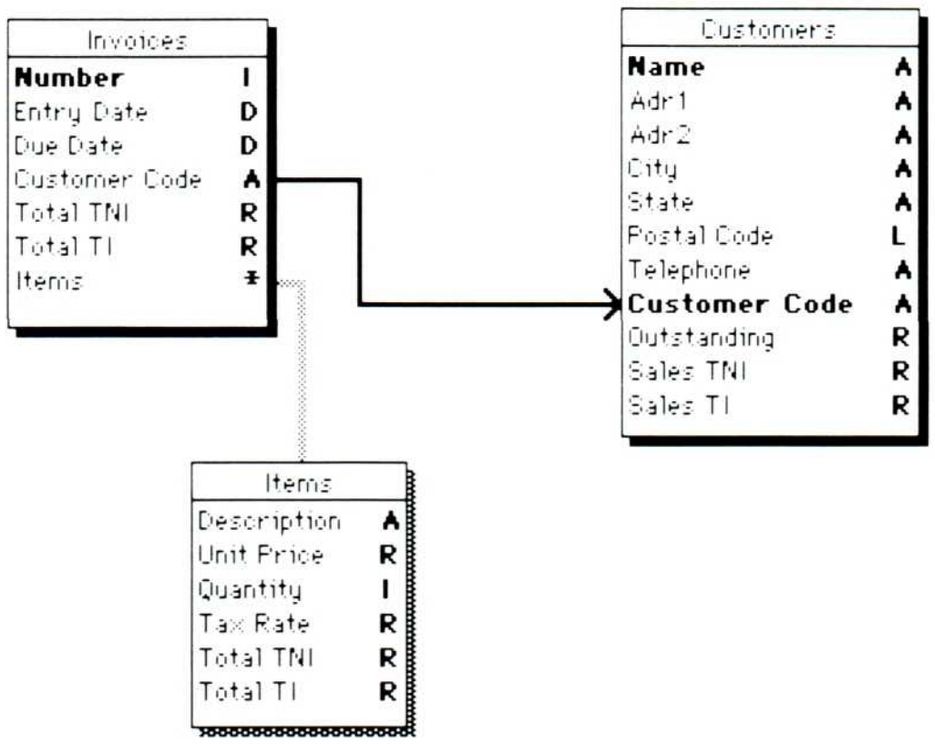


Figure 5-18
Structure window view of Invoices database

Figure 5-19 shows a printout of the structure for the linked **Customers** file. In the **Customers** file, the **Customer Code** field has Unique and Indexed attributes, because you'll search that field to find a customer when issuing an invoice. This field cannot be modified. The outstanding **Total TI** and **Total TNI** fields cannot be entered. These fields will be updated automatically every time an invoice is issued.

- *Field-name abbreviations:* The field name **Total TI** stands for *Total tax included* and **Total TNI** stands for *Total tax not included.*

Structure: Customers		
Name	Alpha 20	Indexed; Enterable; Modifiable
Adr1	Alpha 25	Enterable; Modifiable
Adr2	Alpha 25	Enterable; Modifiable
City	Alpha 20	Enterable; Modifiable
State	Alpha 20	Enterable; Modifiable
Postal Code	Long Integer	Enterable; Modifiable
Telephone	Alpha 20	Enterable; Modifiable
Customer Code	Alpha 8	Indexed; Unique; Enterable
Outstanding	Real	Enterable; Modifiable
Sales TNI	Real	Modifiable
Sales TI	Real	Modifiable

Figure 5-19
Structure for Customers file

Figure 5-20 shows the structure for the linking Invoices file. In the Invoices file, the Total TI and Total TNI fields have the Non-enterable attribute, because the input layout Invoices file layout procedure will assign them the sum of the values contained in the Total TI and Total TNI fields of the subfile [Invoices]Items subrecords. The [Invoices]Due Date field is Non-enterable, because it will be calculated from the [Invoices]Entry Date field.

Structure: Invoices		
Number	Integer	Indexed; Enterable; Modifiable
Entry Date	Date	Enterable; Modifiable
Due Date	Date	Enterable; Modifiable
Customer Code	Alpha 8	Enterable; Modifiable
Total TNI	Real	Modifiable
Total TI	Real	Modifiable
Items	Subfile	

Figure 5-20
Structure for Invoices file

Figure 5-21 shows the structure for the [Invoices]Items subfile. In the [Invoices]Items subfile, the Total TI and Total TNI fields have the Non-enterable attribute, because they'll be calculated by subfile procedures from the Unit Price, Quantity, and Tax Rate fields. In the [Invoices]Items subfile procedures or in the layout procedures of the input layout Invoices file, you write the following statements to calculate the Total TNI and the Total TI for the line:

```
Total TNI := Quantity * Unit price

Total TI := Total TNI + ((Total TNI * Tax Rate) / 100)
```

Structure: Items		
Description	Alpha 20	Enterable; Modifiable
Unit Price	Real	Enterable; Modifiable
Quantity	Integer	Enterable; Modifiable
Tax Rate	Real	Enterable; Modifiable
Total TNI	Real	Modifiable
Total TI	Real	Modifiable

Figure 5-21
Structure for [Invoices]Items subfile

Managing the link between Invoices and Customers files

You want to update the outstanding balance, the total sales tax not included (TNI), and the total sales tax included (TI) of a customer automatically every time the user makes out a bill in the customer's name. The **LOAD LINKED RECORD** routine lets you link the searched customer to an invoice. Once the customer record is modified, the **SAVE LINKED RECORD** routine saves the linked customer record you've loaded.

SAVE LINKED RECORD saves a loaded linked record. You always give the field in which the link originates (the linking field) to the **SAVE LINKED RECORD** command. With links, you can load a record linked to another record and change a record, by calculating the values contained in the record it is linked to.

The input layout procedure for the Invoices file is as follows:

```
LOAD LINKED RECORD (Customer Code)
   `When applied to a subfield, the SUM function returns
   `the sum of the values contained in the subrecords of the current record only.
Total TNI := Sum (Items'Total TNI)
Total TI := Sum (Items'Total TI)
If (After)
      `The user validates the bill entry:
      `The customer accounts will be incremented by the bill amount:
   [Customers]Outstanding := [Customers]Outstanding + Total TI
   [Customers]Sales TNI := [Customers]Sales TNI + Total TNI
   [Customers]Sales TI := [Customers]Sales TI + Total TI
      `SAVE LINKED RECORD updates the customer record on the disk.
   SAVE LINKED RECORD (Customer code)
End if
```

❖ *Note:* The Invoices file Customer Code field has the Mandatory attribute. 4th Dimension will manage the creation of the Customers record during an invoice entry if the record doesn't exist.

Improving procedures

The above procedure only manages the addition of items in an invoice; it does not handle invoice modifications. This can create problems. As an example, let's say SMITHS LTD phones in an order. The user makes out an invoice for $15,000.00 TNI ($17,790.00 TI). When the user validates the record, **SAVE LINKED RECORD** adds these numbers to the SMITHS LTD account balance. Then, a few minutes later, SMITHS LTD phones back with additional purchases on the same invoice worth $18,000.00 TNI ($21,348.00 TI). When the user validates the modified invoice, SMITHS LTD's balance shows a balance of $33,000.00 TNI and $39,138.00 TI. This substantial over-ring happens because the current procedure doesn't subtract old invoice totals before adding modified totals.

To solve this problem, you can use the 4th Dimension **Old** function. When the user modifies a record, **Old** returns the value the record contained before the change. Thus, you can use it maintain correct amounts for things like account balances. In essence, **Old** has no effect when a new record is added, because it returns a null string, a zero, or the null date depending on field type. Rewrite the procedure as follows:

```
.
.
.
If (After)
    [Customers]Outstanding := [Customers]Outstanding - Old (Total TI) + Total TI
    [Customers]Sales TNI := [Customers]Sales TNI - Old (Total TNI) + Total TNI
    [Customers]Sales TI := [Customers]Sales TI - Old (Total TI) + Total TI
    SAVE LINKED RECORD (Customer code)
End if
```

Here are the numbers calculated by the revised procedure. The invoice shows

[Invoices]Total TNI equals $15,000.00

[Invoices]Total TI equals $17,790.00

After the new invoice is entered, the SMITHS LTD customer accounts shows

[Customer]Outstanding equals $0.00 – $0.00 + $17,790.00

[Customers]Sales TNI equals $0.00 – $0.00 + $15,000.00

[Customers]Sales TI equals $0.00 – $0.00 + $17,790.00

After modifying the invoice,

[Invoices]Total TNI equals $18,000.00

[Invoices]Total TI equals $21,348.00

and in the SMITHS LTD customer accounts

[Customer]Outstanding equals $17,790.00 – $17,790.00 + $21,348.00, which is $21,348.00

[Customers]Sales TNI equals $15,000.00 – $15,000.00 + $18,000.00, which is $18,000.00

[Customers]Sales TI equals $17,790.00 – $17,790.00 + $21,348.00, which is $21,348.00

With the **Old** function, the database remains accurate. Whether adding or modifying an invoice, the carried-over amounts are always correct in the customer accounts.

Working with old links

This section introduces two link commands that work on old links: **LOAD OLD LINKED RECORD** and **SAVE OLD LINKED RECORD. LOAD OLD LINKED RECORD** loads the previously linked record. **SAVE OLD LINKED RECORD** saves the previously linked record. You always pass these two instructions to the record where the link originates. They become important for maintaining records so that no discrepancy appears between the sum of all customer balances and the sum of all the invoices made out to these customers.

Let's look at another problem, by modifying the previous scenario. A data entry user makes out the earlier invoice using the SMITHS LTD customer code; then, as previously described, modifies the record from $15,000 to $18,000 (TNI). The user realizes that the invoice was meant for BROWN LTD instead of SMITHS LTD. The user's solution is to change the customer number to that of BROWN LTD. Under the revised procedure, here's what happens with this series of actions.

After adding the invoice, the SMITHS LTD account will be incremented:

[Customer]Outstanding equals $0.00 – $0.00 + $17,790.00

[Customers]Sales TNI equals $0.00 – $0.00 + $15,000.00

Customers]Sales TI equals $0.00 – $0.00 + $17,790.00

After modifying the invoice, BROWN LTD accounts will first be decremented before being incremented:

[Customer]Outstanding equals $0.00 – $17,790.00 + $21,348.00, which is $3,558.00

[Customers]Sales TNI equals $0.00 – $15,000.00 + $18,000.00, which is $3,000.00

Customers]Sales TI equals $0.00 – $17,790.00 + $21,348.00, which is $3,558.00

As you can see, the results are incorrect and the procedures are inadequate. You have to decrement the old customer's accounts and increment the new customer's accounts. To do this, you will use two 4th Dimension commands: **LOAD OLD LINKED RECORD** and **SAVE OLD LINKED RECORD. LOAD OLD LINKED RECORD** loads the record previously linked to another record. **SAVE OLD LINKED RECORD** saves a linked record previously loaded in memory by **LOAD OLD LINKED RECORD.**

Here is the final version of the procedure:

```
.
.
.
If (After)
    [Customers]Outstanding := [Customers]Outstanding + Total TI
    [Customers]Sales TNI := [Customers]Sales TNI + Total TNI
    [Customers]Sales TI := [Customers]Sales TI + Total TI
    SAVE LINKED RECORD (Customer Code)
    LOAD OLD LINKED RECORD (Customer Code)
    [Customers]Outstanding := [Customers]Outstanding - Old (Total TI)
    [Customers]Sales TNI := [Customers]Sales TNI - Old (Total TNI)
    [Customers]Sales TI := [Customers]Sales TI - Old (Total TI)
    SAVE OLD LINKED RECORD (Customer Code)
End if
```

Considering the SMITHS LTD and BROWN LTD example, after adding the invoice, the SMITHS LTD accounts will be incremented:

[Customers]Outstanding equals $0.00 + $17,790.00

[Customers]Sales TNI equals $0.00 + $15,000.00

[Customers]Sales TI equals $0.00 + $17,790.00

The added invoice doesn't yet have a loaded linked customer record. Here **LOAD OLD LINKED RECORD** and **SAVE OLD LINKED RECORD** have no effect.

After modifying the invoice, the BROWN LTD accounts will be incremented:

[Customers]Outstanding equals $0.00 + $21,348.00

[Customers]Sales TNI equals $0.00 + $18,000.00

[Customers]Sales TI equals $0.00 + $21,348.00

This is carried out by **SAVE LINKED RECORD.** The SMITHS LTD accounts will then be decremented:

[Customers]Outstanding equals $17,790.00 – $17,790.00, which is $0.00

[Customers]Sales TNI equals $15,000.00 – $15,000.00, which is $0.00

[Customers]Sales TI equals $17,790.00 – $17,790.00, which is $0.00

CREATE LINKED RECORD command

The **CREATE LINKED RECORD** routine is more sophisticated than the **LOAD LINKED RECORD** routine. If the file contains a record corresponding to the search requirements, the record is selected and loaded into memory just as with **LOAD LINKED RECORD.** If that record doesn't exist, 4th Dimension creates it. You must then assign values to the record's fields through calculations, and save the record with **SAVE LINKED RECORD.**

To demonstrate **CREATE LINKED RECORD,** we'll add a linked Products file to improve the efficiency of the invoice system. Such a file can supply the invoice with a product's unit cost and tax rate, while maintaining figures on the quantity of the item sold. Figure 5-22 shows the database with the linked Products file.

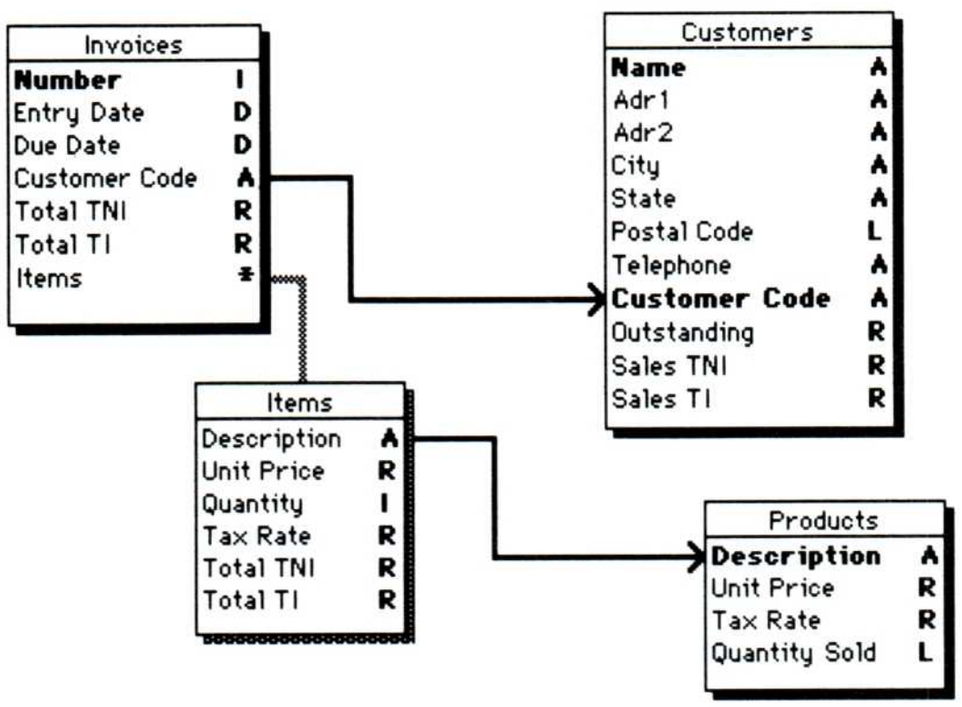


Figure 5-22
Structure showing addition of linked Products file

Figure 5-23 shows the field structure of the Products file.

Structure: Products		
Description	Alpha 20	Indexed; Modifiable
Unit Price	Real	Modifiable
Tax Rate	Real	Modifiable
Quantity Sold	Long Integer	Modifiable

Figure 5-23
Field structure of Products file

All fields of the **Products** file are Non-enterable. Every time the user makes out an invoice, you want to create the **Products** record automatically if the record doesn't exist. You also want to increment the **[Products]Quantity Sold** field. To do this, you'll use the **CREATE LINKED RECORD** command. In the above example, you create the input layout of the **Invoices** file, shown in Figure 5-24.

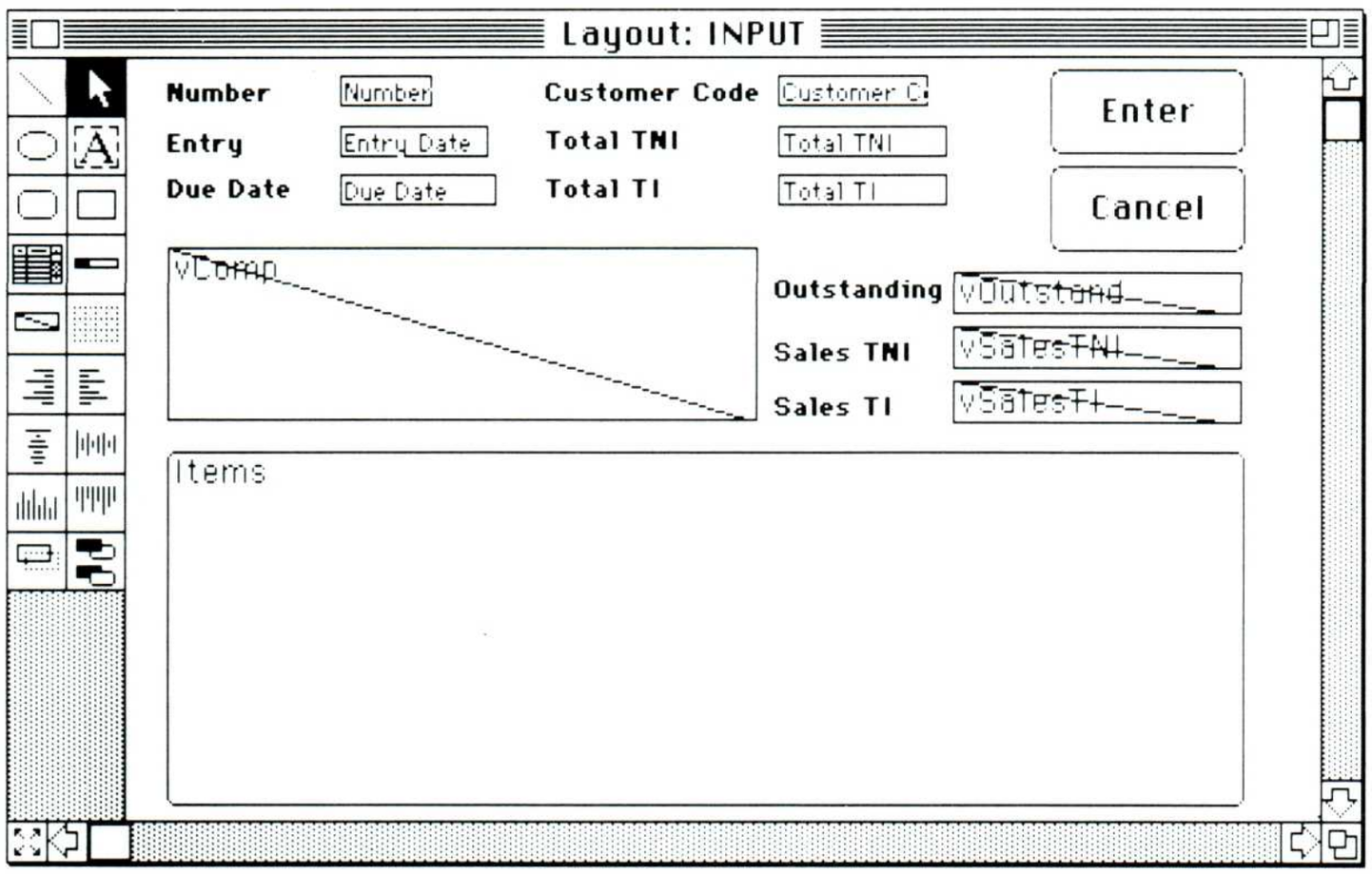


Figure 5-24
Input layout for Invoices file

In the subfile area, you specify the **[Invoices]Items** subfile with the layout shown in Figure 5-25.

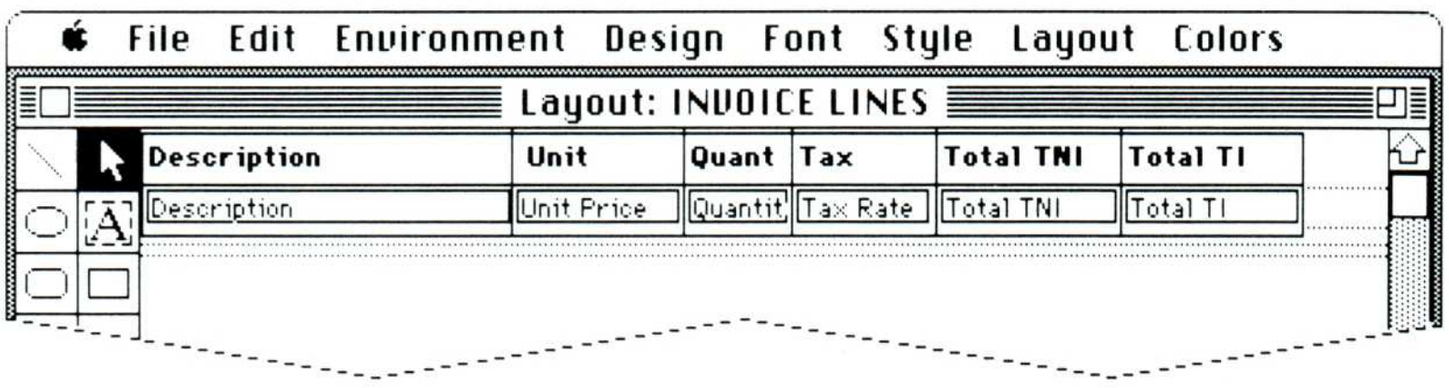


Figure 5-25
Subfile layout for (Invoices)Items subfile

You're going to modify the invoice input layout procedures, so that the Products file will be changed automatically every time an invoice is created or modified. The subfile layout procedures remain unchanged:

```
   `Input layout procedure (final version) of the Invoices file.
If (Before)
    If (Entry Date = !00/00/0000!)
          `If entry date equals null date, you're dealing with a created invoice.
          `Enter the current date in the field by using the Current date standard routine.
       Entry Date := Current date
    End if
End if
   `Load the linked customer record
LOAD LINKED RECORD (Customer Code)
   `Display the customer accounts by using the vOutstand, vSalesTNI and vSalesTI variables
   `specified in the invoice layout.
vOutstand := [Customers]Outstanding
vSalesTNI := [Customers]Sales TNI
vSalesTI := [Customers]Sales TI
   `Calculate vComp to display the customer's address and phone number in the invoice layout.
vCR := Char (13)
vComp := [Customers]Name + vCR + [Customers]Adr1 + vCR + [Customers]Adr2 + vCR
vComp := vComp + String ([Customers]Postal Code) + " " + [Customers]City
   `Calculate the total amount, tax not included, and the total amount, tax included, of the invoice.
Total TNI := Sum (Items'Total TNI)
Total TI := Sum (Items'Total TI)
   `Calculate the due date at 30 days following the invoice creation date.
Due date := Entry date + 30
If (After)
      `If you validate the invoice entry, increment the customer accounts.
    [Customers]Outstanding := [Customers]Outstanding + Total TI
    [Customers]Sales TNI := [Customers]Sales TNI + Total TNI
    [Customers]Sales TI := [Customers] Sales + Total TI
    SAVE LINKED RECORD (Customer Code)
      `If the invoice is modified, decrement the old customer accounts.
    LOAD LINKED RECORD (Customer Code)
    [Customers]Outstanding := [Customers]Outstanding - Old (Total TI)
    [Customers]Sales TNI := [Customers]Sales TNI - Old (Total TNI)
    [Customers]Sales TI := [Customers] Sales - Old (Total TI)
    SAVE OLD LINKED RECORD (Customer Code)
```

```
   `You must now carry over the sales in the Products file.
   `To do this, select all items in the invoice with ALL SUBRECORDS.

ALL SUBRECORDS (Items)
While (Not (Last subrecord (Items)))
      `While you don't try to go below the last item of the invoice. Assign the
      `Items'Designation field to itself to force the link to the Products file. This is
      `necessary because you're in the after step of the record, and that at that point,
      `4th Dimension, no longer considers the entry fields as modified. The link is not
      `activated by calling LOAD LINKED RECORD or CREATE LINKED RECORD because
      `these routines are optimized: they only activate the link on the first call, if the field
      `has been modified. Assigning the field to itself forces the field to be modified. This
      `method is necessary only if you call a link activation routine in an after step.
   Items'Description := Items'Description
      `Call CREATE LINKED RECORD:
      `if the Products record exists, 4th Dimension loads that linked record.
      `If it doesn't, 4th Dimension creates the linked record.
CREATE LINKED RECORD (Items'Description)
      `Assign to the Products record fields, the values contained in the invoice item:
[Products]Description := Items'Description
[Products]Unit Price := Items'Unit Price
[Products]Tax Rate := Items'Tax Rate
      `Increment the sales with the invoiced quantity.
[Products]Quantity Sold := [Products]Quantity Sold + Items'Quantity
      `Save the Products record
SAVE LINKED RECORD (Items'Description)
      `As for the customer accounts, if you modified the invoice, look for the old product
      `record linked to the item:
LOAD OLD LINKED RECORD (Items'Description)
      `Decrement these sales from the previously invoiced quantity.
[Products]Quantity sold := [Products]Quantity Sold - Old (Items'Quantity)
      `Update the record on the disk.
SAVE OLD LINKED RECORD (Items'Description)
      `The carry over of the items being made, go to the next item:
NEXT SUBRECORD (Items)
   End while
End if
```

In the User environment, you enter the invoice shown in Figure 5-26.

Entry for Invoices

Number 1 **Customer Code** MAR001
Entry 10/17/86 **Total TNI** 38,400.00
Due Date 11/16/86 **Total TI** 40,896.00

Enter

Cancel

MARTINSON CONSTRUCTION
350 Fifth Ave.
San Ramon, CA 99935

Outstanding 40,896.00
Sales TNI 38,400.00
Sales TI 40,896.00

Description	Unit	Quant	Tax	Total TNI	Total TI
Norton Fitting Mach	15,000.00	2	0.065	30,000.00	31,950.00
Reversing Gasket	1.50	2000	0.065	3,000.00	3,195.00
Straight Seal Gasket	3.60	1500	0.065	5,400.00	5,751.00

Figure 5-26
Completed invoice form

You then have to select the **Products** file to display the result of the procedures (shown in Figure 5-27).

Products: 3 of 3

Description	Unit Price	Tax Rate	Quantity Sold
Norton Fitting Mach	15,000.00	0.065	2
Reversing Gasket	1.50	0.065	2000
Straight Seal Gasket	3.60	0.065	1500

Figure 5-27
Products file output displaying results

You've just seen that a link can originate in a subfield. In such a case, every subfield of the record points to at most one record and not that one record points to many records. See Figure 5-28.

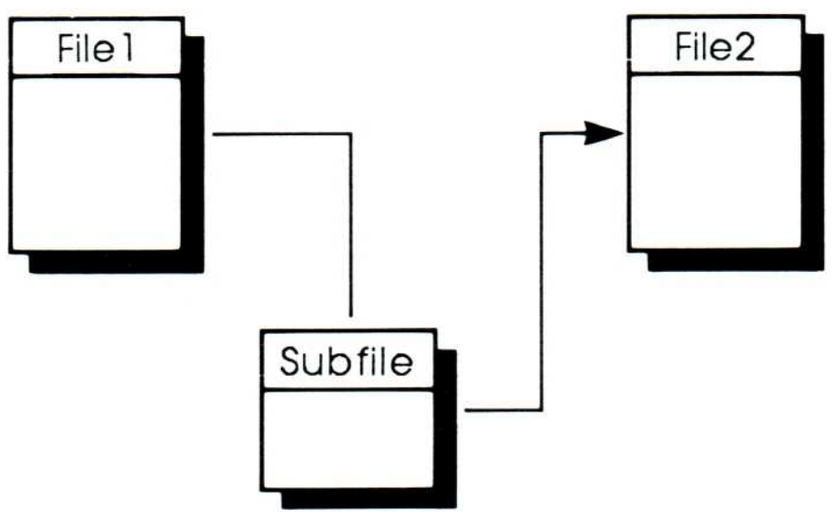


Figure 5-28
How a subfile links to a record in a file

To reach linked records, you must call **LOAD LINKED RECORD** for each subrecord. See Figure 5-29.

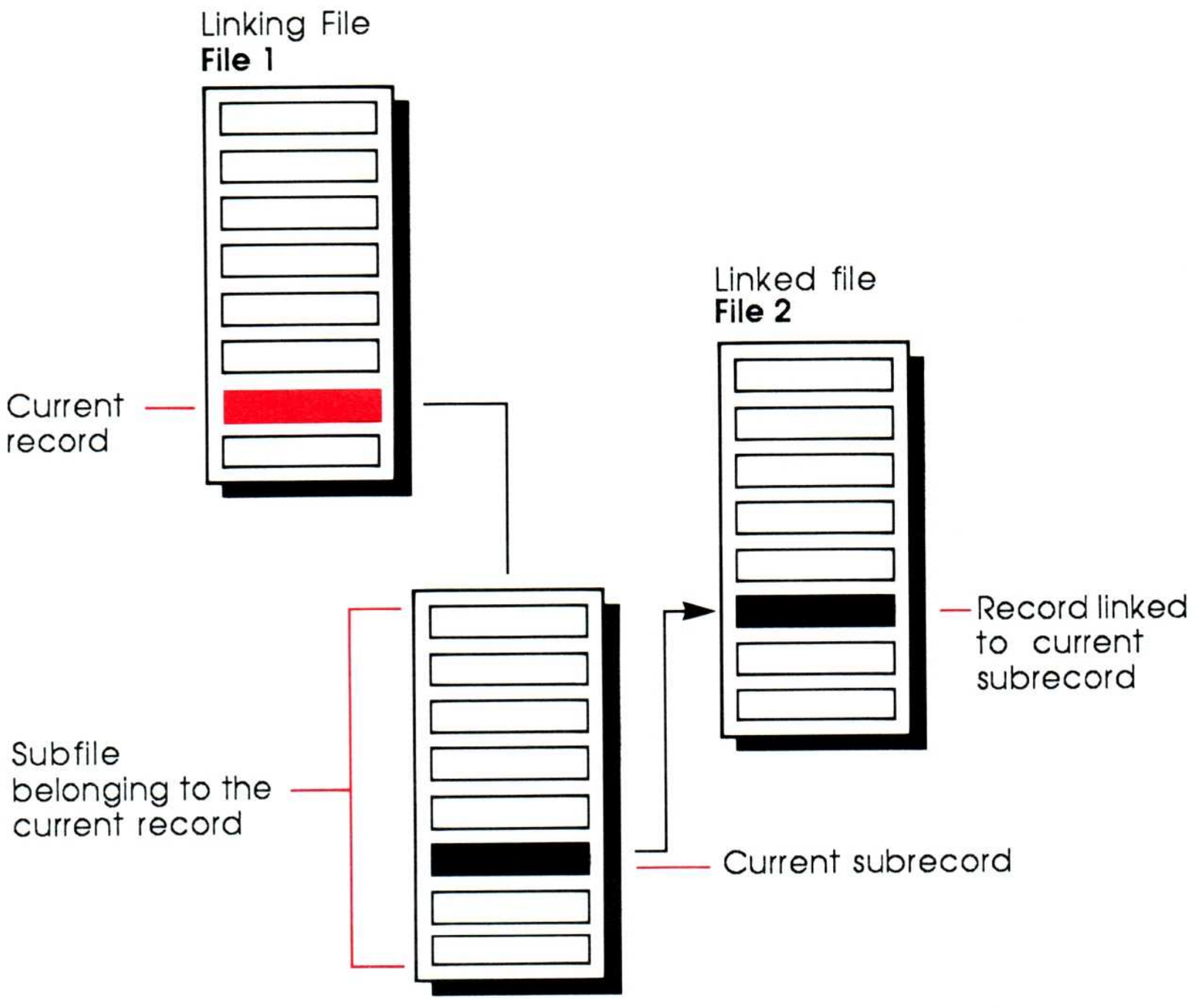


Figure 5-29
Current subrecord pointing to one record in file linked to subfile

In the above example, the [Invoices]Customer Code field is Modifiable. To manage a customer change when modifying an invoice, you must include procedures to decrement the old customer's accounts. Here the **LOAD OLD LINKED RECORD** and **SAVE OLD LINKED RECORD** routines were used.

If the [Invoices]Customer Code field is not modifiable, the customer won't change when you modify the invoice, so you won't have to write statements to decrement the old customer's accounts. However, you'll have to decrement the old totals of the customer accounts. This case refers to the procedures written in the section above, "Working With Old Links."

Linking to a subfile

The database will undergo a last change: you'll add a subfile called Sales to the Products file. Figure 5-30 shows the revised structure.

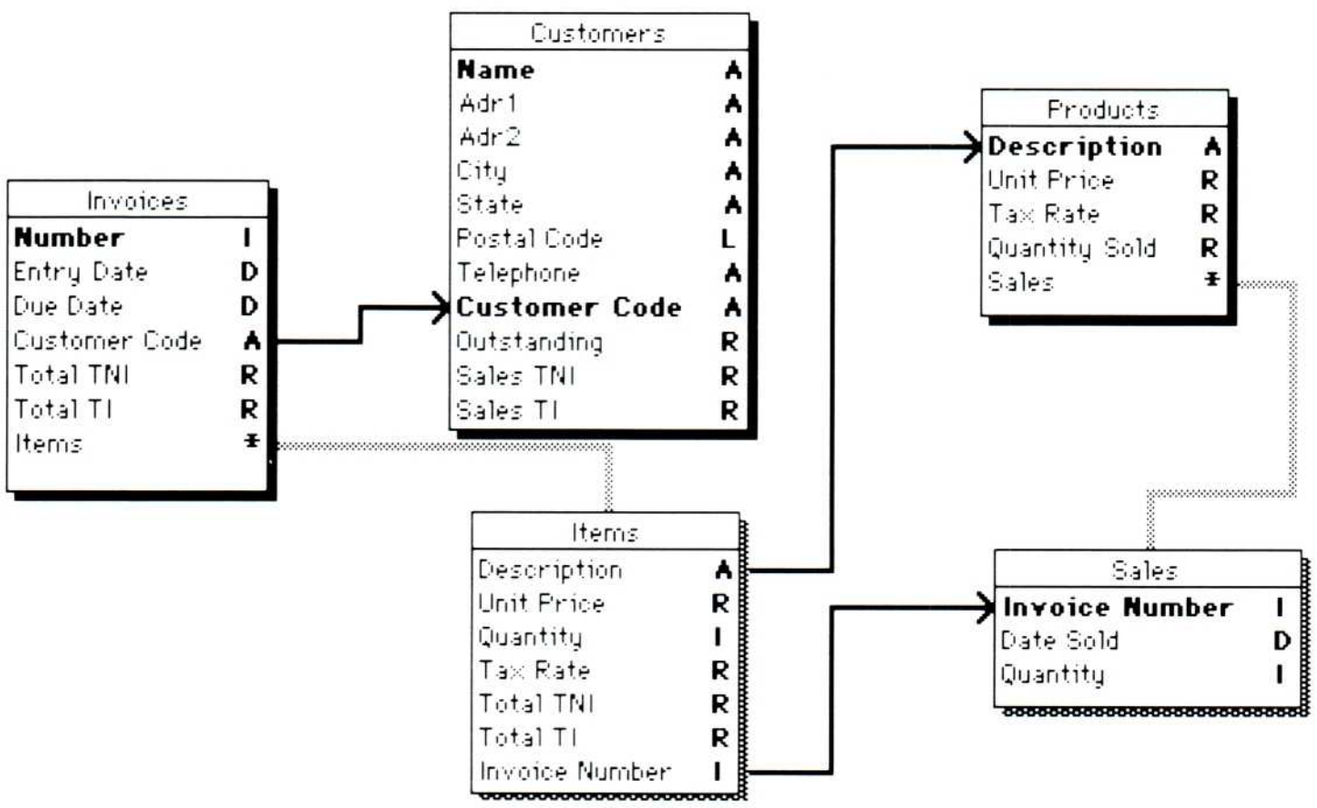


Figure 5-30
Structure with addition of Sales subfile

You want to save the chronology of the sales for every item in that subfile. You create a field called Invoice Number in the [Invoices]Items subfile and link [Invoices]Items'Invoice Number to [Products]Sales'Invoice Number. You'll use **CREATE LINKED RECORD.**

In the layout procedure for the [Invoices]Items subfile, you add a procedure to carry over the invoice number:

```
Total TNI := Quantity * Unit price
Total TI := Total TNI + ((Total TNI * Tax Rate) / 100)
Invoice Number := [Invoices] Number
```

You modify the Invoices layout procedures to create the linked Sales record in the [Products] file:

```
If (Before)
    If (Entry Date = !00/00/0000!)
        Entry Date := Current date
    End if
    vComp := 33
    vOutstand := 0
    vSalesTNI := 0
    vSalesTI := 0
End if
LOAD LINKED RECORD (Customer Code)
vOutstand := [Customers]Outstanding
vSalesTNI := [Customers]Sales TNI
vSalesTI := [Customers]Sales TI
vCR := Char (13)
vComp := [Customers]Name + vCR + [Customers]Adr1 + vCR + [Customers]Adr2
vComp := vComp + vCR + String ([Customers]Postal Code) + " " + [Customers]City
Total TNI := Sum (Items'Total TNI)
Total TI := Sum (Items'Total TI)
Due Date := Entry Date + 30
If (After)
    [Customers]Outstanding := [Customers]Outstanding + Total TI
    [Customers]Sales TNI := [Customers]Sales tNI + Total TNI
    [Customers]Sales TI := [Customers] Sales + Total TI
    SAVE LINKED RECORD (Customer Code)
    LOAD LINKED RECORD (Customer Code)
    [Customers]Outstanding := [Customers]Outstanding - Old (Total TI)
    [Customers]Sales TNI := [Customers]Sales TNI - Old (Total TNI)
    [Customers]Sales TI := [Customers] Sales - Old (Total TI)
    SAVE OLD LINKED RECORD (Customer Code)
    ALL SUBRECORDS (Items)
    While (Not (Last subrecord (Items)))
        Items'Description := Items'Description
CREATE LINKED RECORD (Items'Description)
[Products]Description := Items'Description
[Products]Unit Price := Items'Unit Price
[Products]Tax Rate := Items'Tax Rate
[Products]Quantity Sold := [Products]Quantity Sold + Items'Quantity
   `You're in the after step, so you must assign the field to itself to make it modified.
Items'Invoice Number := Items'Invoice Number
```

```
   `Create the sales linked record in the [Products]Sales subfile.
CREATE LINKED RECORD (Items'Invoice Number)
   `Assign the sales fields as follows:
[Products]Sales'Invoice Number := Items'Invoice Number
[Products]Sales'Date Sold := Entry Date
   `Increment the quantity sold:
[Products]Sales'Quantity := [Products]Sales'Quantity + Items'Quantity
   `IMPORTANT:
   `CREATE LINKED RECORD creates or loads a Sales subfile subrecord. This operation
   `takes place in memory. You must then save the changes made to the subrecord.
   `The subrecord will be saved when you'll save the record containing it. SAVE
   `LINKED RECORD of the products record will save the record along with its
   `subrecords. Therefore, you don't have to call SAVE LINKED RECORD for the subrecord.
SAVE LINKED RECORD (Items'Description)
LOAD OLD LINKED RECORD (Items'Description)
      [Products]Quantity Sold := [Products]Quantity Sold - Old (Items'Quantity)
      SAVE OLD LINKED RECORD (Items'Description)
      NEXT SUBRECORD (Items)
   End while
End if
```

A link can point to a subfield. When a linked subrecord is created or loaded, the record containing that subrecord must be saved to save the subrecord. Similarly, to create or load a linked subrecord, you must select the record which contains the subfile and load it as the current record.

After entering two invoices, you obtain the list of items shown in Figure 5-31.

Products: 5 of 5

Description	Unit Price	Tax Rate	Quantity Sold
Norton Fitting Mach	15,000.00	0.065	2
Reversing Gasket	1.50	0.065	2000
Straight Seal Gasket	3.60	0.065	1500
Norton 5000 FM	20,000.00	0.065	1
Norton 10000 FM	35,000.00	0.065	1

Figure 5-31
Five new records in Products file

Figure 5-32 displays a particular product so the user can see its sales history.

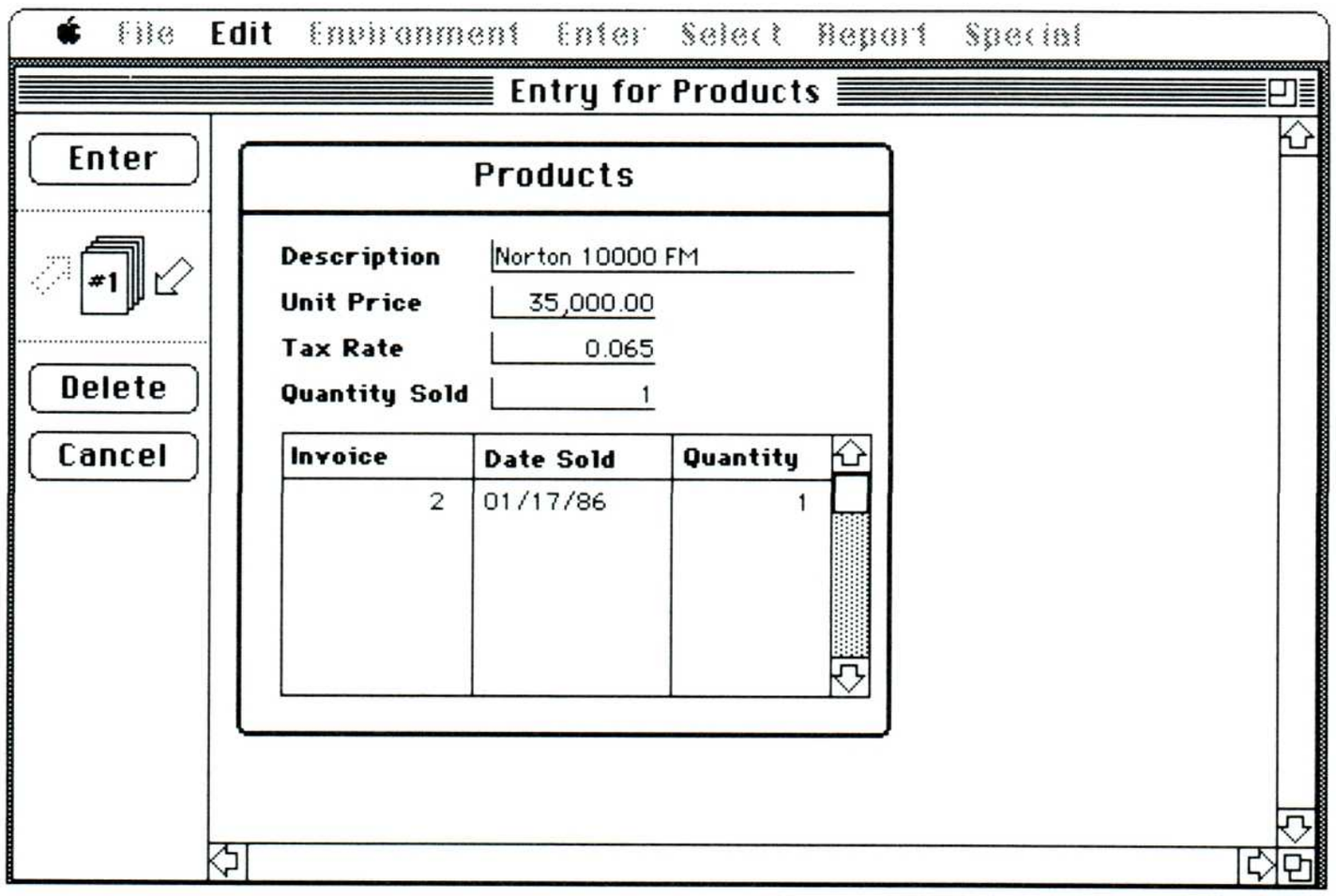


Figure 5-32
Displaying a product's sales history

❖ *Note:* The example studied here for creating linked subrecords doesn't manage bill modifications or deletions.

Chapter 6

Sets

4th Dimension sets offer the developer a powerful, swift means for manipulating file selections. Besides the ability to create sets, relate them to the curent selection, and store, load, and clear sets, 4th Dimension offers three standard set operators:

- Intersection
- Union
- Difference

Sets defined

The idea of sets is closely bound to the idea of the current selection. The current selection is a list or table in memory that can point to all records in the file or any subset of them (including a null selection). Whatever the case, the list exists in memory. A selection doesn't actually contain the records, but only a list of the records. Only the current record of the file is in memory. When you work on a file, you always work with the records in the current selection.

A selection and a set are two different kinds of objects. A selection is the list of the file records you're working on. A set is an object you create in memory, consisting of one bit for every record in the file to which it belongs. Set operations are, in effect, binary operations on the bit array portion of the set and are thus very fast. For example, the **UNION** command performs an OR operation on the bit arrays of the two specified sets. Table 6-1 compares the current selection with sets.

Table 6-1
Current selection and sets concepts compared

Comparison	Current selection	Sets
Number per file	1	0 to many
Sortable	Yes	No
RAM per record	4 bytes	1 bit (1/8 of a byte)
Combinable	No	Yes
Contains current record	Yes	Yes, as of the last time the set was used

4th Dimension has commands with which you can create a set from a file selection or create a new file selection from a set belonging to that file. Set operations make it easy to store and combine results from several searches. You can either create an empty set or create a set from a selection or from existing sets. As a result, a given set may contain more, fewer, or the same number of records as the current selection. The two are not necessarily related. You always name the sets you create. For example:
Mail Dupes.

❖ *Note:* The size of a set, in bytes, is always equal to the total number of records contained in the file to which it belongs divided by 8. If you create a set belonging to file containing 10,000 records, the set will take up 1250 bytes, which is about 1.2K in RAM. Sets are very economical in terms of RAM and disk space.

Important

When you create a set, it belongs to the file under which you created it. The operations you perform on sets can only be performed on sets belonging to the same file.

Operations on sets

The seven primary set commands in 4th Dimension are (in alphabetical order)

- **ADD TO SET:** Put a record in the set.
- **CREATE EMPTY SET:** Create an empty set.
- **CREATE SET:** Put all records in the current selection into a new set.
- **DIFFERENCE:** Put unique records from two sets into a third set.
- **INTERSECTION:** Put common records of two sets into a third set.
- **UNION:** Put elements belonging to set 1, set 2, or both into a third set.
- **USE SET:** Make the current selection reflect the contents of a set.

This section describes each of these seven commands with graphic depictions of their workings.

4th Dimension also has set commands that return the number of records in a set (**Records in set**), clear a set from memory (**CLEAR SET**), save a set to disk (**SAVE SET**), and load a set into memory that you previously saved to disk (**LOAD SET**).

CREATE EMPTY SET command

CREATE EMPTY SET (*«filename;»strexpr*)

Figure 6-1 illustrates the **CREATE EMPTY SET** command. This command creates an empty set belonging to *filename* and gives the empty set the name *strexpr*.

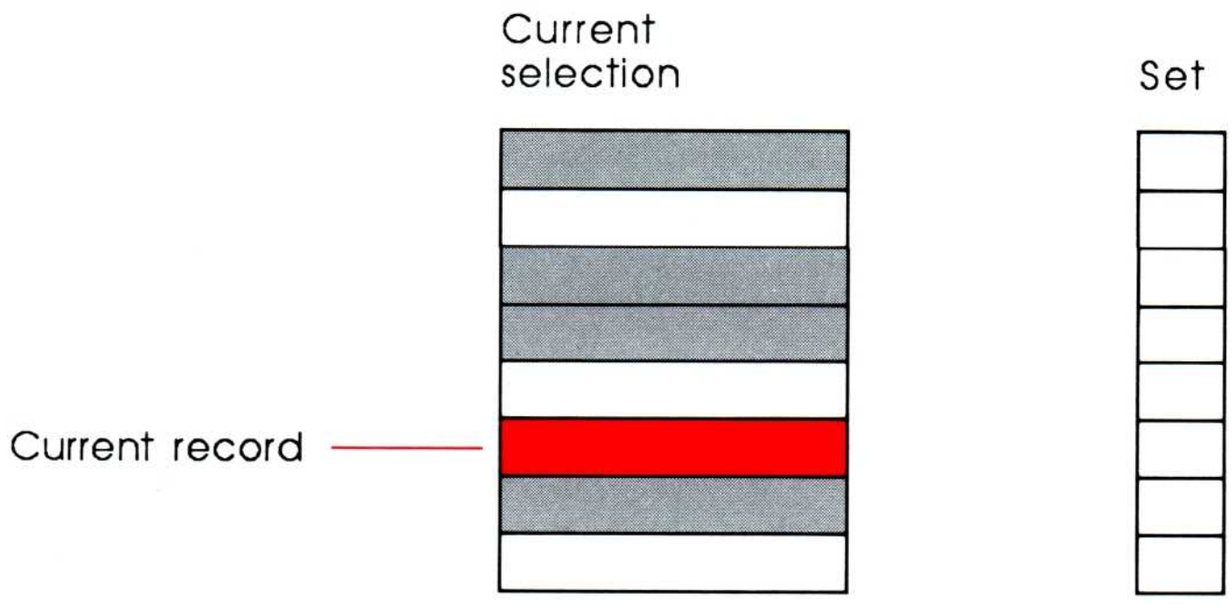


Figure 6-1
CREATE EMPTY SET command

CREATE SET command

CREATE SET (*«filename;»strexpr*)

Figure 6-2 illustrates the **CREATE SET** command. This command creates a set named *strexpr*. *strexpr* will belong to *filename*. The records belonging to the set are the ones contained in the current selection of *filename*. Compare **CREATE SET** and **USE SET.**

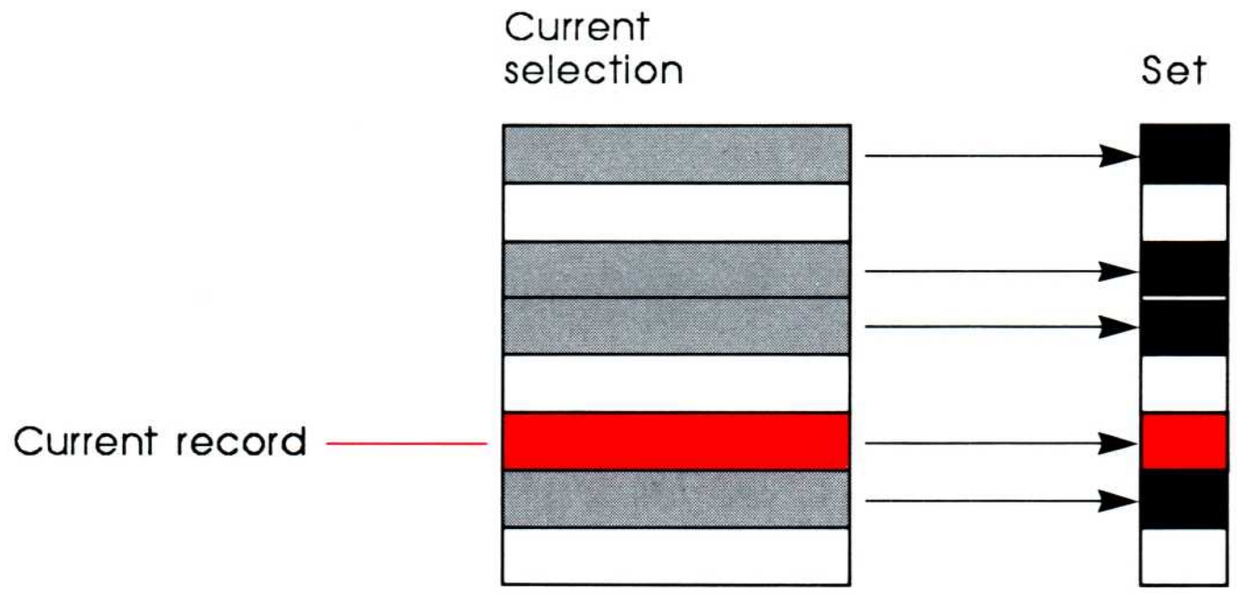


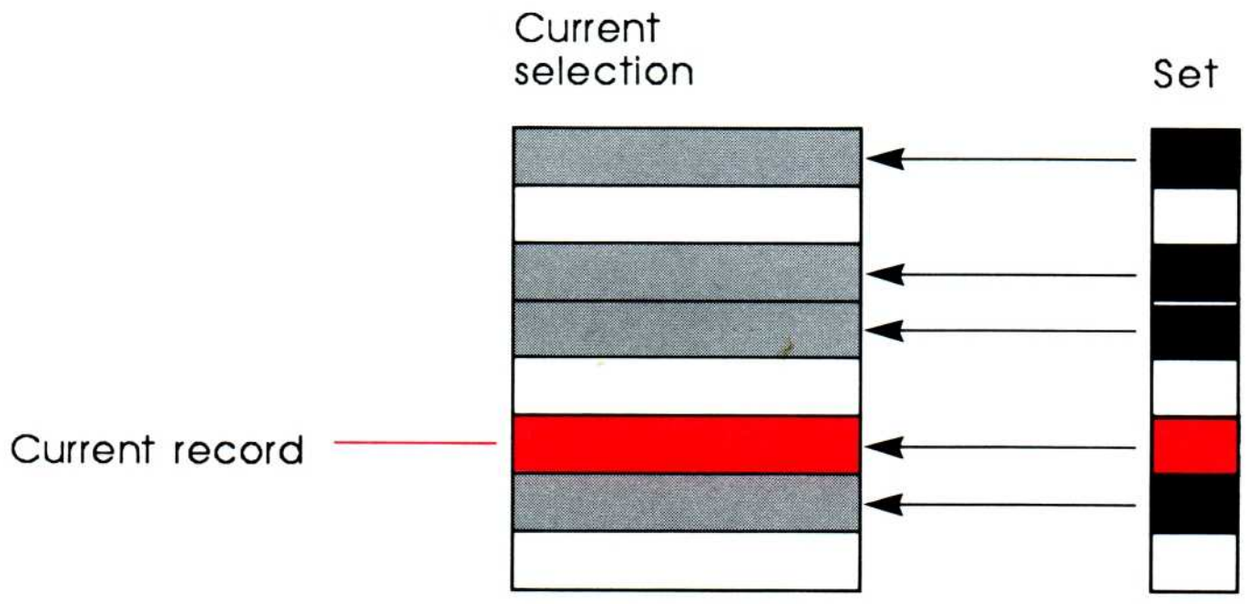
Figure 6-2
CREATE SET command

USE SET command

USE SET (*strexpr*)

Figure 6-3 illustrates the **USE SET** command. This command creates a new current selection for *filename* from the set named *strexpr*. *strexpr* must belong to *filename*. The records of the new selection are the ones belonging to the set. Compare **USE SET** and **CREATE SET.**

❖ *The current record and sets:* The first record in a set does not necessarily become the current record under **USE SET.**



ADD TO SET command

ADD TO SET (*«filename;»strexpr*)

Figure 6-4 illustrates the **ADD TO SET** command. This command adds the current record to a set named *strexpr*. *strexpr* must belong to *filename*. If the current record doesn't exist, **ADD TO SET** has no effect.

ADD TO SET has two common uses. First, inside a global procedure called by the **APPLY TO SELECTION** command, you can use **ADD TO SET** to create sets based on complex criteria. Second, to keep track of records that have changed since some specific event. You might want to track modified records.

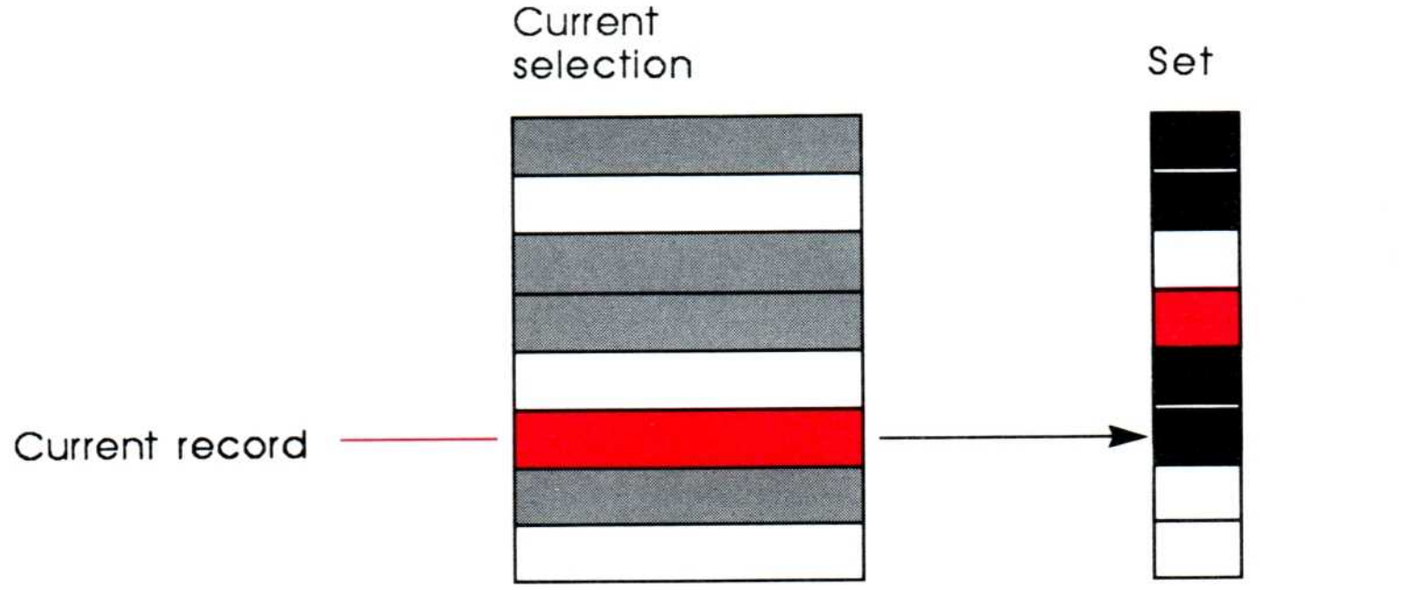
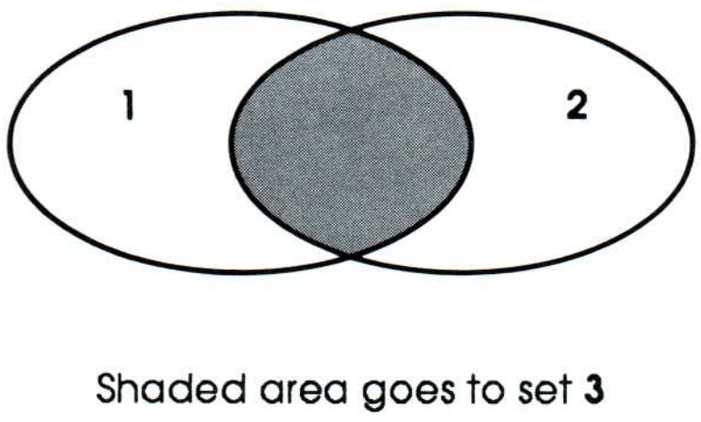


Figure 6-4
ADD TO SET command

INTERSECTION command

INTERSECTION (*strexpr1*;*strexpr2*;*strexpr3*)

Figure 6-5 illustrates the **INTERSECTION** command. This command creates a set named *strexpr3*. Its records are the ones belonging to both the *strexpr1* and *strexpr2* sets. *strexpr3* can be a third set name or it can be the same as *strexpr1* or *strexpr2*.



Intersection
Is record in set number *n*?

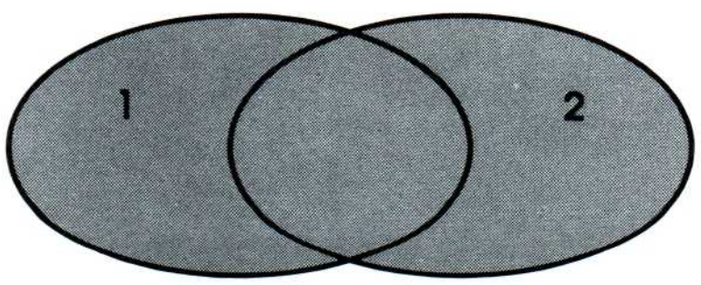
1	2	3
N	N	N
N	Y	N
Y	N	N
Y	Y	Y

Figure 6-5
INTERSECTION command

UNION command

UNION (*strexpr1*;*strexpr2*;*strexpr3*)

Figure 6-6 illustrates the **UNION** command. This command creates a set named *strexpr3*. Its records are the ones belonging to either the *strexpr1* or *strexpr2* set or to both sets. *strexpr3* can be a third set name or it can be the same as *strexpr1* or *strexpr2*.



Union
Is record in set number *n*?

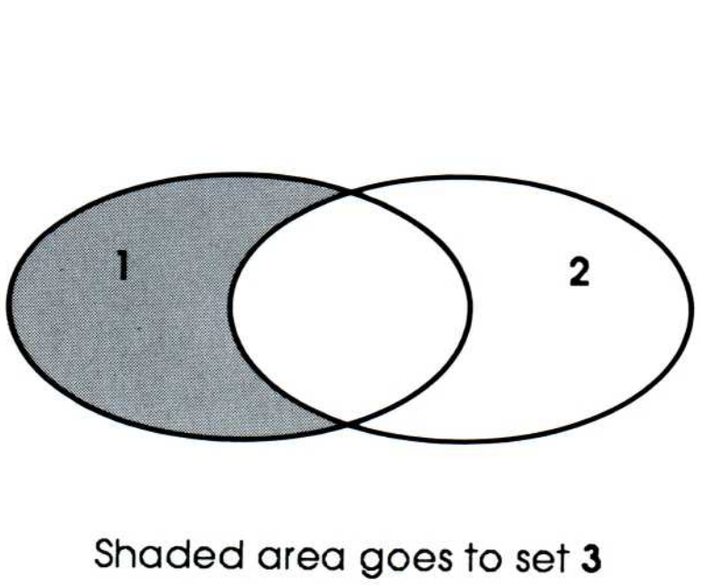
1	2	3
N	N	N
N	Y	Y
Y	N	Y
Y	Y	Y

Figure 6-6
UNION command

DIFFERENCE command

DIFFERENCE (*strexpr1*;*strexpr2*;*strexpr3*)

Figure 6-7 illustrates the **DIFFERENCE** command. This command creates a set named *strexpr3*. Its records represent the difference between the *strexpr1* and *strexpr2* sets. *strexpr3* can be a third set name or it can be the same as *strexpr1* or *strexpr2*. A record will appear in the *strexpr3* set if it is in the *strexpr1* set and not in the *strexpr2* set.



Difference
Is record in set number *n*?

1	2	3
N	N	N
N	Y	N
Y	N	Y
Y	Y	N

Figure 6-7
DIFFERENCE command

Using sets: deleting duplicate records

Clearing duplicate records out of mailing lists is a perfect job for sets. In this example, the procedure tests to see if two records have the same last name, first name, and title. If they do, the procedure assigns one of the records to a set named Doubles and continues the search. Once the search is over, the **USE SET** command puts everything in Doubles into the current selection. The procedure then deletes the current selection. The file structure is shown in Figure 6-8.

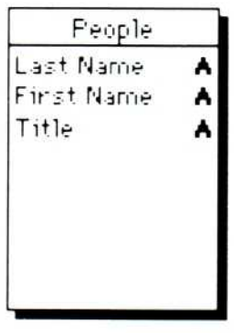


Figure 6-8
File structure

Here is the procedure named Destroy doubles:

```
    `You're going to work on the [People] file. Select it as if it were the default file.
    `Thus you won't have to enter the filename as parameter for routines requiring that
    `parameter. This will simplify the writing of the algorithm.
DEFAULT FILE ([People])
    `Select all records in the [People] file with the ALL RECORDS command.
ALL RECORDS
    `Initialize the three variables, vLast Name, vFirst Name and vTitle, in which you'll
    `store the name, first name and status of the record preceding the one studied in order
    `to compare values.
vLast Name :=""
vFirst Name :=""
vTitle :=""
    `Sort the [People] file by last name, first name and title in ascending order.
SORT ([People]Last Name ; > ; [People]First Name ; > ; [People]Title ; >)
    `Assign an empty set named "Doubles" to the [People] file.
EMPTY SET ("Doubles")
    `Start at the first sorted selection.
While (Not (End selection))
        `While you don't try to go beyond the last record of the sorted selection.
    If ([People]Last Name = vLast Name)
            `Is that last name equal to the preceding last name?
            `If so, check whether the first name is equal to the preceding one.
            `Note: For the first selected record there obviously cannot be a preceding record.
            `This is why you must initialize the variables.
        If ([People]First Name = vFirst Name)
                `Both names being identical, try to determine whether the first name is equal to
                `the preceding first name. If so, check whether the title is equal to the preceding one.
            If ([People]Title = vTitle)
                    `Is the title equal to the preceding one?
                    `If so, that record is redundant. Add the record (which is the current
                    `record) to the"Doubles" set.
                ADD TO SET ("Doubles")
            Else
                    `The title is different. Place it in the vTitle variable.
                vTitle := [People]Title
        End if
Else
        `Only the last name is different. Place the title and first name in the
        `corresponding variables.
    vTitle := [People]Title
    vfirst name := [People]First Name
End if
```

```
    Else
            `Everything is different.
        vTitle := [People]Title
        vFirst Name := [People]First Name
        vLast Name := [People]Last Name
    End if
            `Move to the next record.
    NEXT RECORD
End while
        `When exiting the "Doubles" set loop, the set contains all double records.
        `Create a new file selection from that set by calling USE SET.
USE SET ("Doubles")
        `Delete the selection of records.
DELETE SELECTION
        `Delete in memory the set and the unwanted variables.
CLEAR VARIABLE ("v")
CLEAR SET ("Doubles")
```

Figure 6-9 shows the file before the procedure was executed.

People: 7 of 7

Title	First Name	Last Name
Hon.	Jean	Davis
Mr.	Henry	Drummond
Mr.	Henry	Drummond
The Rev.	William	Mackenzie
Ms.	Anne	Martin
Mrs.	Anne	Martin
Mrs.	Isabel	Martin

Figure 6-9
Output of file before removing duplicates

Figure 6-10 shows the file after the procedure was executed.

People: 6 of 6

Title	First Name	Last Name
Hon.	Jean	Davis
Mr.	Henry	Drummond
The Rev.	William	Mackenzie
Ms.	Anne	Martin
Mrs.	Anne	Martin
Mrs.	Isabel	Martin

Figure 6-10
Output of file after removing duplicates

UserSet system set

4th Dimension has a system set named UserSet. UserSet automatically stores the most recent set of selections by the user. Thus, you can display a group of records, ask the user to select from among them, and turn the results of that selection into a set that you name or into a selection. The brief procedure below illustrates how you can do this:

```
   `Userpick
   `Display all records and allow user to select any number of them.
   `Then display this selection by using UserSet to change current selection.
DEFAULT FILE([Models])
OUTPUT LAYOUT("Display")
ALL RECORDS
ALERT("Press Command and Click to select a document required")
DISPLAY SELECTION
USE SET("UserSet")
ALERT("You chose the following documents")
DISPLAY SELECTION
OUTPUT LAYOUT("Screen1")
```

Chapter 7

Menus

Pull-down menus have become one of the hallmarks of the Macintosh user interface. Thanks to menus, users can choose activities by name, rather than relying on often-cryptic typed command codes. Through custom menus, you can create applications that look to the user as if you built them "from scratch." 4th Dimension contains a complete menu construction kit with which you can create menus and Command key combinations to select items without resorting to a menu, you can password-protect menu items, and you can enable and disable items. See Chapter 6 of *4th Dimension User's Guide* for details on implementing these features. This chapter concentrates on issues involving programming and menus.

Menu components

The bar at the top of the screen is called the **menu bar.** Each name on the bar represents a **menu.** When you pull down the menu, you see the menu's **items.** Figure 7-1 shows these components.

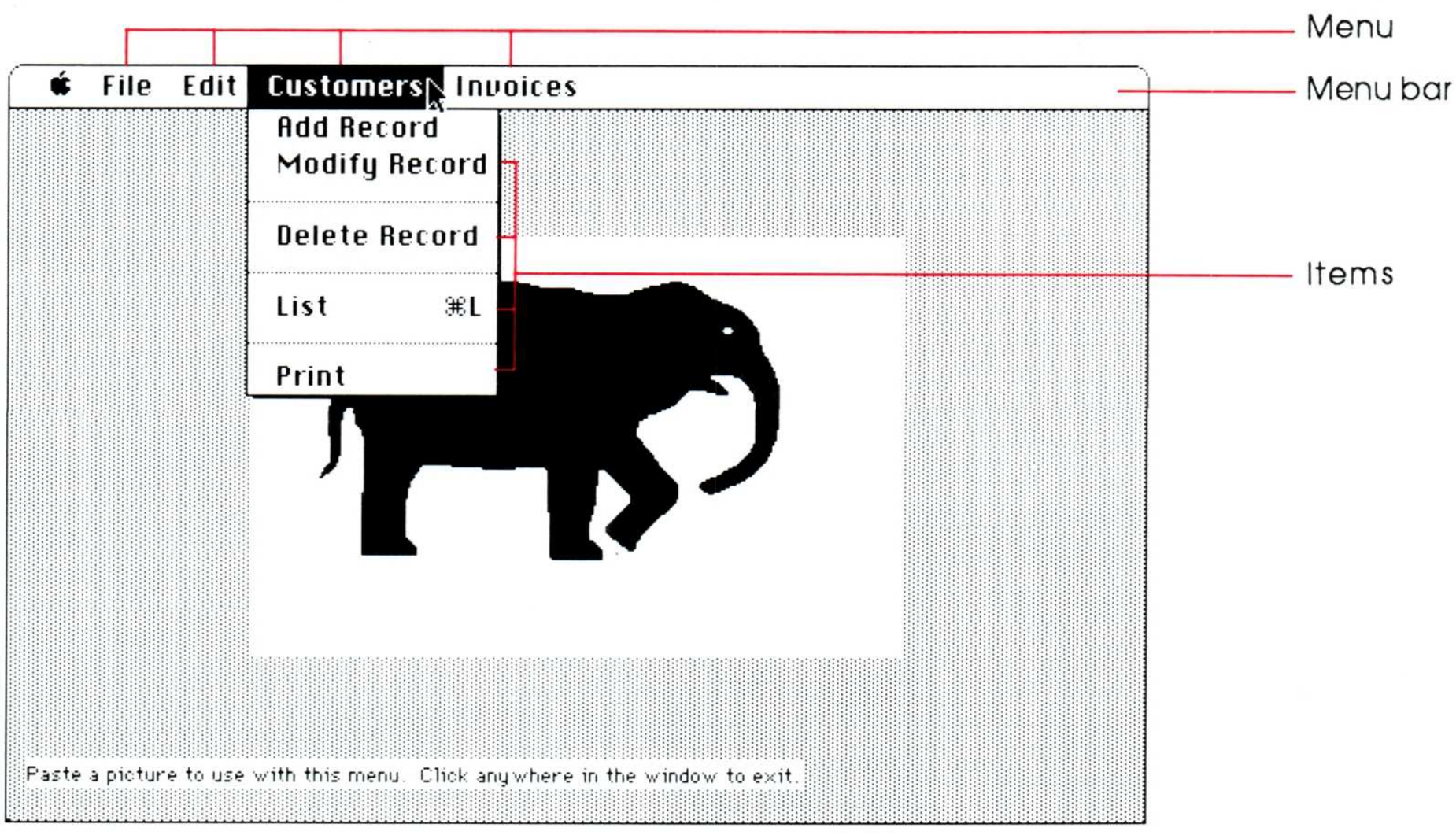


Figure 7-1
Menu components

You create menus in the Design environment's Menu window. Unlike other 4th Dimension entities, menus are identified by number, rather than name. The first menu is named Menubar #1. It is also the default menu. If you wish to open an application with a menu other than Menubar #1, you must force it with the **MENU BAR** command in a startup procedure.

Each item can have one and only one developer-written procedure attached to it. You associate a procedure with an item by naming the procedure in the Procedures column of the Menu window. The user executes the procedure by selecting the item to which the procedure belongs. If you don't assign a procedure to an item, selecting that item causes 4th Dimension to quit the menu system. In the case of run-time installations, this means returning to the Finder.

A menu bar comes pre-equipped with three menus—the Apple, Edit, and File menus. The Apple menu contains "About 4th Dimension" and any desk accessories currently installed in the System file. The Edit menu contains the editing commands. File has only one item—Quit. Notice that Quit has no procedure associated with it. That's how it causes a quit. You can rename the File menu, add items to it, or keep it as is. It is recommended that you always keep the File menu with Quit as the last item. The Apple and Edit menus are permanent.

❖ *Quitting time:* Any time you execute a menu item that has no procedure attached to it, you leave the Custom environment.

If 4th Dimension encounters an **ABORT** command in a global procedure, it stops execution and returns to the menu bar. This is like clicking the Abort button in the debugger.

Like menu bars, menus themselves are numbered. The Apple and Edit menu are not counted. Instead, File is menu 1. Thereafter (reading from left to right), menus are numbered sequentially (2, 3, 4, and so on). Menu numbering becomes important when working with the **Menu selected** function. Similarly, items within each menu are numbered from 1 for the topmost item down. Item numbering comes into play when working with the **CHECK ITEM** command.

Menu window features

When you select a particular item within the Items column in the Menu window, you apply any of five features to the selected item (see Figure 7-2):

- **Keyboard:** Checking the Keyboard box and typing a character assigns the character as a command character. The user can call an item by typing the appropriate Command key combination.
- **Line:** Checking the Line box draws a dividing line between two menu items. You should disable any lines you create.
- **Enabled:** The default state for any item is enabled. You can toggle the enabled state off and on by clicking this box.
- **Font styles:** Check the style(s) you want for the menu item. The default is Plain. Use styles only in a Style menu.
- **Passwords:** You can block unauthorized users from gaining access to particular items by assigning a password to the item. For details on passwords, see Chapter 6 of *4th Dimension User's Guide*.

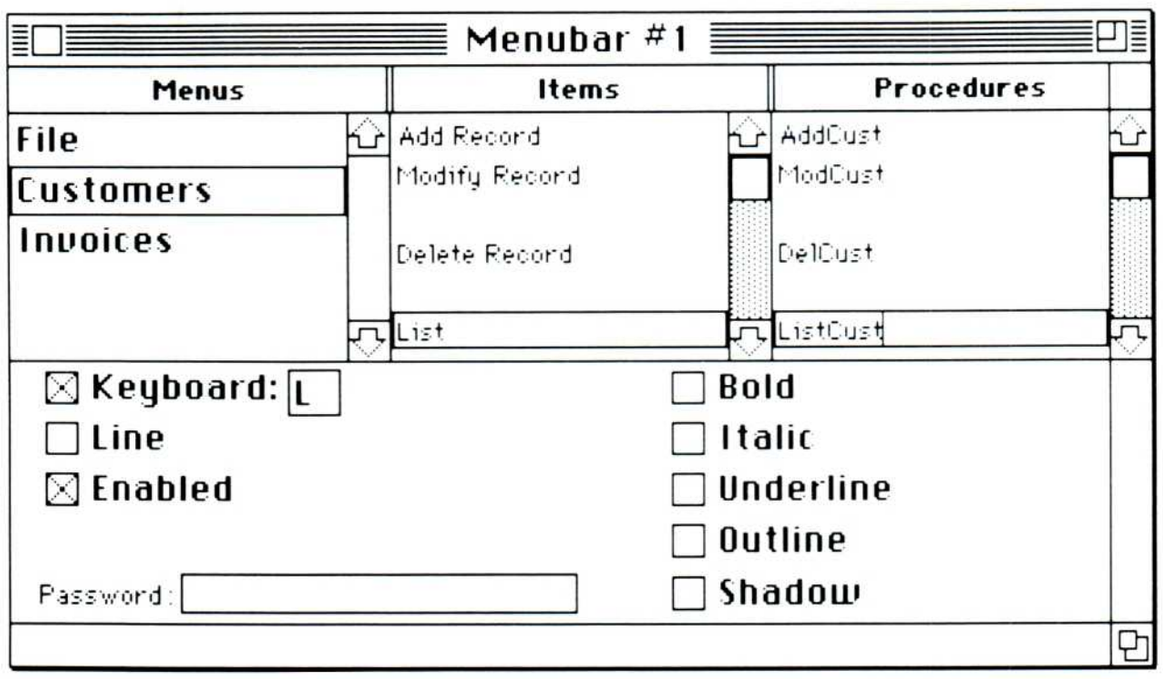


Figure 7-2
Menu window

Of these five features, only Enabled is programmable. A line is normally used to group items on a menu. In 4th Dimension, a line is an enabled item unless you disable it. This means that if someone selects an enabled line and you have associated a procedure with it, 4th Dimension will execute the procedure. If you have not associated a procedure with an enabled line, selecting the line will cause the 4th Dimension to quit the Custom environment. As a general rule, disable all lines. When you disable an item, the selection bar skips over it as you drag down the menu. Enabled and disabled lines look the same.

Do not use the following command keys:

- Z: Undo
- X: Cut
- C: Copy
- V: Paste
- Q: Quit (except for the Quit menu item)

Programmable menu features

4th Dimension has a number of programmable menu features:

- **ENABLE ITEM** and **DISABLE ITEM** programmatically toggle the Enable state on and off.
- **CHECK ITEM** places a check mark next to the designated item.
- **MENU BAR** takes a menu bar number as an argument and activates the menu bar identified by this number.
- **Menu selected** returns either the menu or the item selected, depending on how you manipulate it.

You can enable and disable items as appropriate from within procedures with **ENABLE ITEM** and **DISABLE ITEM.** As a general rule, if you find yourself disabling a particular item a lot, set it as disabled in the menu window. The same principle works for enabling items.

CHECK ITEM comes in handy as a way of reminding the user what item is in effect. You can either write constants into **CHECK ITEM** or feed **CHECK ITEM** the necessary arguments with the **Menu selected** function. A check is generated with an ASCII code 18. To erase a check mark, use a null string or a space.

When you are entering or modifying a record or working with a dialog, you can select menu items, but nothing will happen unless you have written a procedure to execute a command. To do this, you can read the menu and menu item positions with **Menu selected** and evaluate the numbers returned through a case statement. Each case should have a procedure name to execute if the number is found to be the case.

Chapter 8

Operations on Pictures

Introduction

In the preceding sections of this manual, you've seen that you can define Picture fields and use Picture variables. These fields support Macintosh pictures.

Picture type fields and variables can be displayed and printed in three different ways:

- The picture can be truncated: 4th Dimension centers the graphic in the Picture area and trims any part of the graphic that is bigger than the size of the Picture area.
- The picture is scaled to fit: It's automatically enlarged or reduced to fit in the Picture area.
- The picture can be placed on background: The picture can be moved with the pointer over the layout background. The user can control the contrast between the picture and the layout background.

With picture arithmetic operations, you can build more complex pictures: calculations are performed on Picture expressions and the results are placed in Picture fields or variables. In the same way you assign Alphanumeric, Numeric, or Date expressions, you assign a graphic to a field or variable with the assignment operator (:=) .

Consider the two-file database structure shown in Figure 8-1.

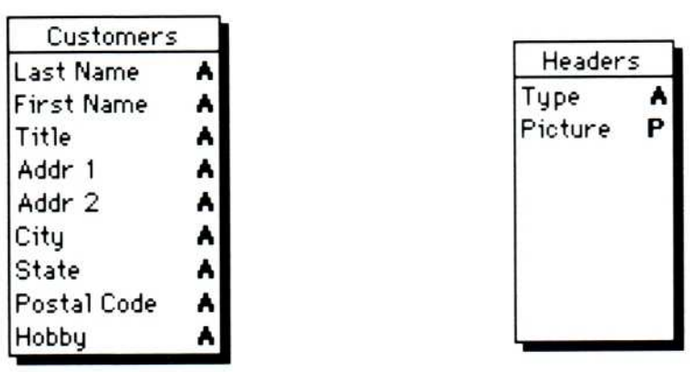


Figure 8-1
Two-file database structure

You want to send a customized letter to every customer and place a picture in the header reflecting the customer's hobby, which you stored in the **[Customers]Hobby** field. You create the **Letter 1** layout shown in Figure 8-2.

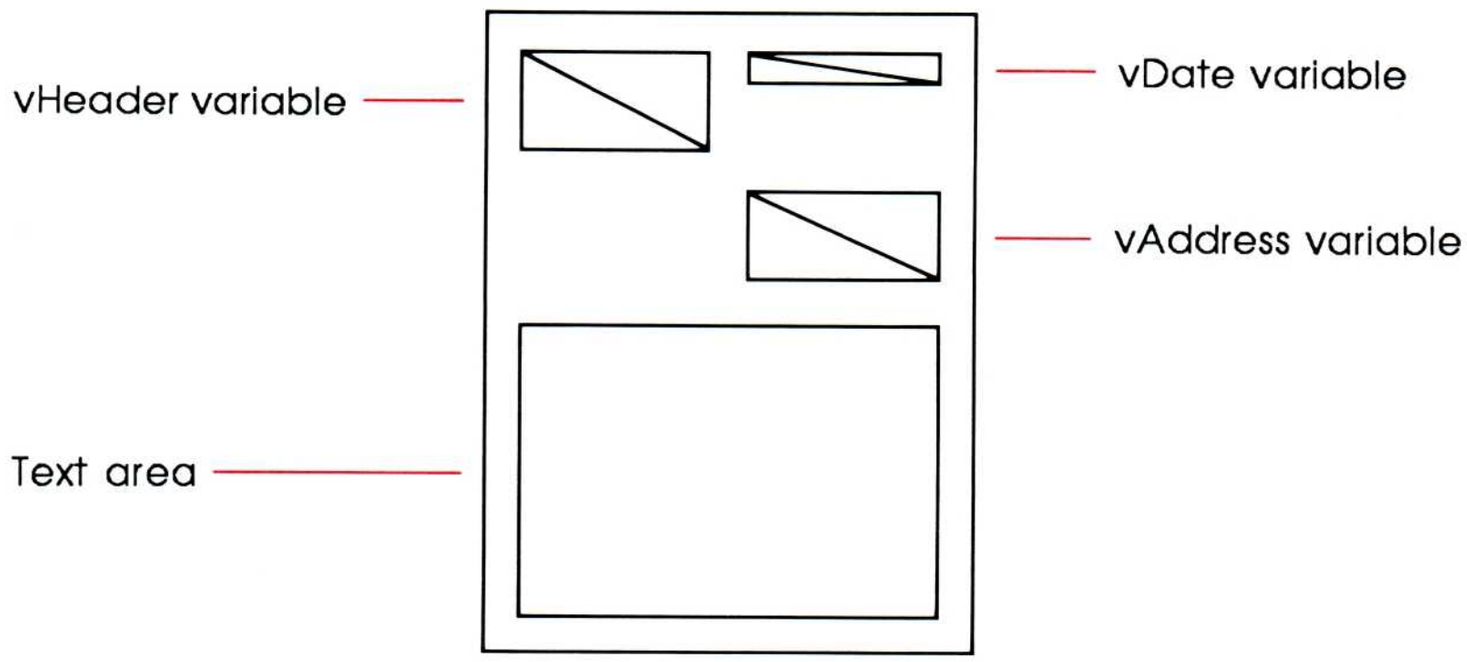


Figure 8-2
Overview of layout for Letter 1

You assign the header picture to the vHeader variable, the customer's address to the vAddress variable, and the date to the vDate variable. In the layout text, you can insert variables enclosed between angle brackets (<>). They'll be replaced by their values upon printing. The layout procedure will calculate the variables:

```
If (Before)
      `If the customer doesn't have any hobby, or if his hobby is unknown in the [Headers]
      `file, copy to the IDefault variable the standard picture stored in the [Headers] file
      `for that purpose.
   If (Undefined (IDefault))
      SEARCH SELECTION ([Headers]Type= "Standard")
      IDefault := [Headers]Picture
   End if
   vDate := String (Current date)
   CR := Char (13)
   vAddress := Title + " " + First Name + " " + Last Name + CR + Adr1 + CR +
      Adr2 + CR
   vAddress := vAddress + City + " " + State + " " + String (Postal Code ; "000000")
      `Search for the header corresponding to the customer's hobby.
   SEARCH SELECTION ([Headers]Type = Hobby)
   If (Records in selection ([Headers]) # 0)
      vHeader := [Headers]Picture
   Else
         `If the header doesn't exist, use the default header.
      vHeader := IDefault
   End if
         `And so on...
End if
```

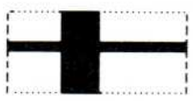
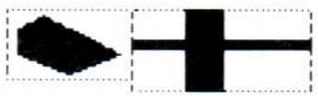
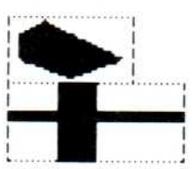
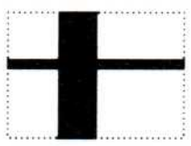
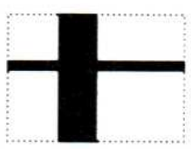
Operations on Picture expressions

4th Dimension offers nine arithmetic and one logical operation on pictures. These are

- horizontal concatenation (+)
- vertical concatenation (/)
- exclusive superimposition (&)
- inclusive superimposition (|)
- horizontal move (+)
- vertical move (/)
- point symmetry (*)
- horizontal scaling (*+)
- vertical scaling (*/)
- negation (Not)

Table 8-1 demonstrates how the concatenation and superimposition operators work. A description of each command follows.

Table 8-1
Concatenation and superimposition operations

Operation	First picture	Second picture	Resulting picture
+ Horizontal concatenation			
/ Vertical concatenation			
& Exclusive superimposition			
| Inclusive superimposition			

Horizontal concatenation (+)

NewPic := *picturexpr1+picturexpr2*

Horizontal concatenation places the *picturexpr2* expression to the right of the *picturexpr1* expression. Both expressions are top aligned. *picturexpr1* + *picturexpr2* and *picturexpr2* + *picturexpr1* are two different statements. There is no commutation in horizontal concatenation.

Vertical concatenation (/)

NewPic := *picturexpr1/picturexpr2*

The *picturexpr2* expression is placed under the *picturexpr1* expression. Both expressions are left aligned *picturexpr1/picturexpr2* and *picturexpr2/picturexpr1* are two different statements. There is no commutation in vertical concatenation.

Exclusive superimposition (&)

NewPic := *picturexpr1&picturexpr2*

The *picturexpr2* expression and the *picturexpr1* expression are superimposed and centered. One point of the resulting picture expression is black, if only one corresponding point of the *picturexpr1* and *picturexpr2* expressions is black, and not both.

Inclusive superimposition (|)

NewPic := *picturexpr1|picturexpr2*

The *picturexpr2* expression and the *picturexpr1* expression are superimposed and centered. One point of the resulting picture expression is black, if one of the corresponding points of *picturexpr1* and *picturexpr2* expressions is black or if both points are black.

The following sections discuss and illustrate the rest of the operations in detail.

- *Background:* In the following discussion, picture movement is relative to the background only if it applies to placing pictures on a background. Pictures remain centered within individual display areas.

Horizontal move (+)

NewPic := *picturexpr+numexpr*

The *picturexpr* expression is moved horizontally by a number of pixels equal to *numexpr*. If *numexpr* expresses a positive value, the picture is moved from left to right. If *numexpr* expresses a negative value, the picture is moved from right to left. See Figure 8-3.

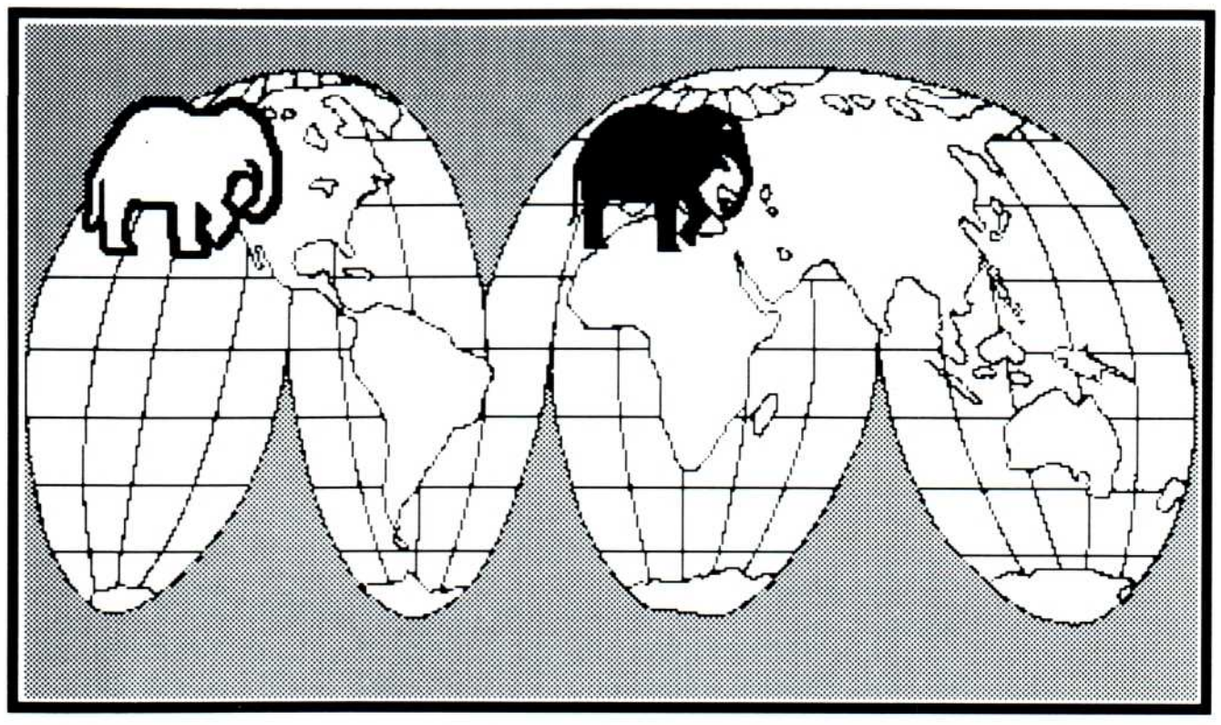


Figure 8-3
Horizontal move

Vertical move (/)

NewPic := *picturexpr*/*numexpr*

The *picturexpr* expession is moved vertically by a number of pixels equal to *numexpr*. If *numexpr* expresses a positive value, the picture is moved from top to bottom. If *numexpr* expresses a negative value, the picture is moved from bottom to top. See Figure 8-4.

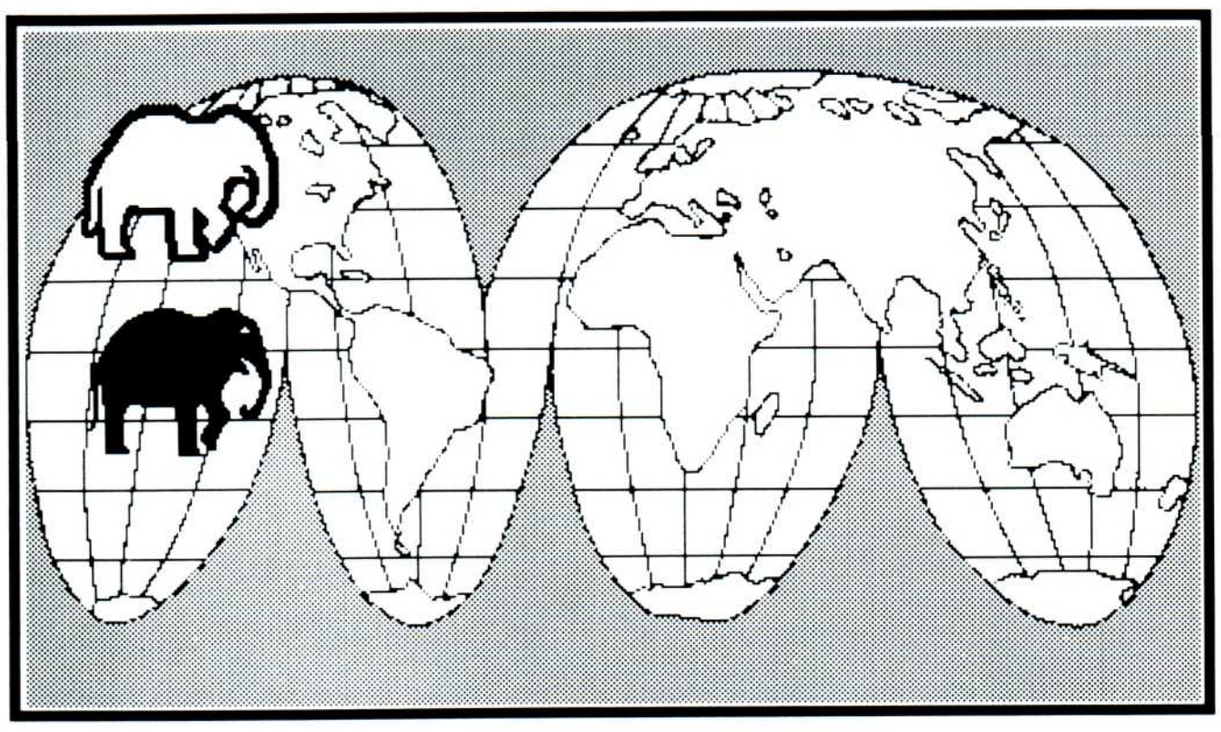


Figure 8-4
Vertical move

Point symmetry(*)

NewPic := *picturexpr* numexpr*

The *picturexpr* expression is resized according to the *numexpr* coefficient. When *numexpr* is positive, either of two things can happen, depending on the value of *numexpr*. If *numexpr* is less than 1, the picture is reduced. If *numexpr* is greater than 1, the picture is enlarged. When *numexpr* is negative, either of two things can happen, depending on the value of *numexpr*. If *numexpr* lies between 0 and –1, the picture is reduced and flipped both horizontally and vertically. If *numexpr* is less than –1, the picture is enlarged, but the picture is drawn by point symmetry through to the upper-left corner of the picture. See Figure 8-5.

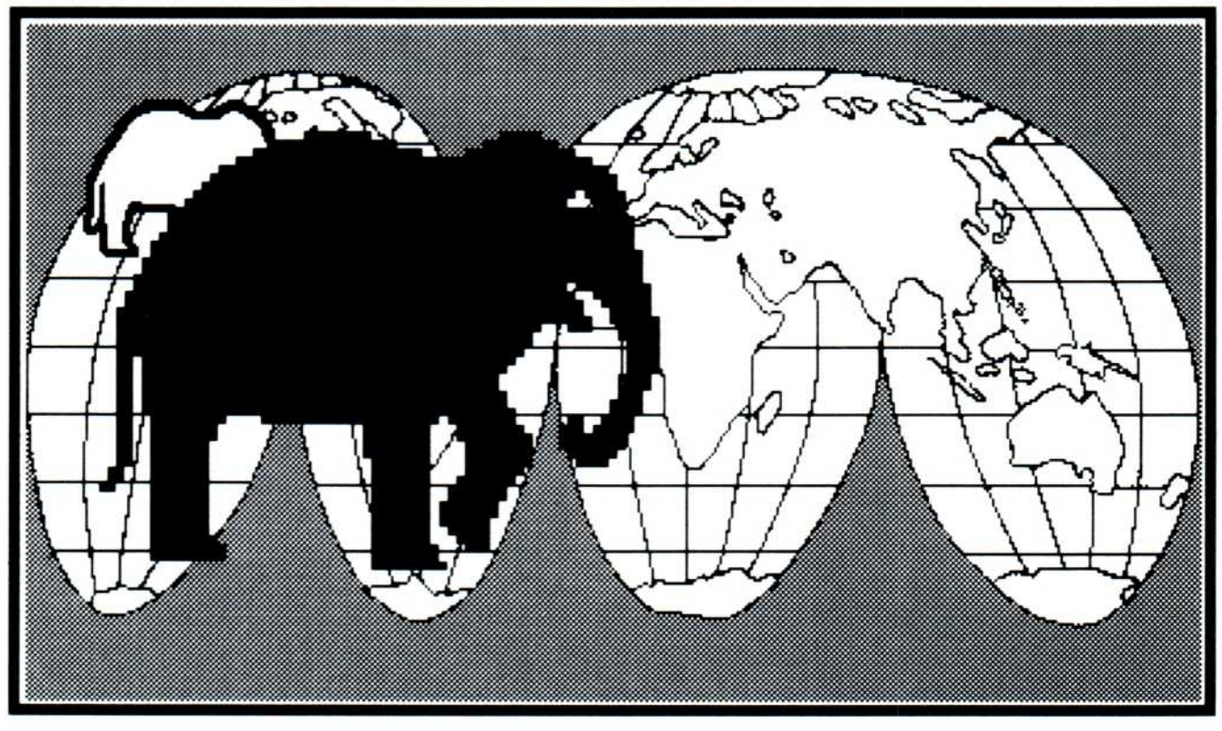


Figure 8-5
Point symmetry

Horizontal scaling (*+)

NewPic := *picturexpr**+*numexpr*

The *picturexpr* expression is scaled horizontally according to the *numexpr* coefficient. When *numexpr* is positive, either of two things can happen, depending on the value of *numexpr*. If *numexpr* lies between 0 and 1, the picture is reduced. If *numexpr* is greater than 1, the picture is enlarged. When *numexpr* is negative, either of two things can happen, depending on the value of *numexpr*. If *numexpr* lies between 0 and –1, the picture is reduced. If *numexpr* is less than –1, the picture is enlarged. In either case, the picture is flipped horizontally. *picturexpr**+-1 returns the horizontal "flip" of *picturexpr*. See Figure 8-6.

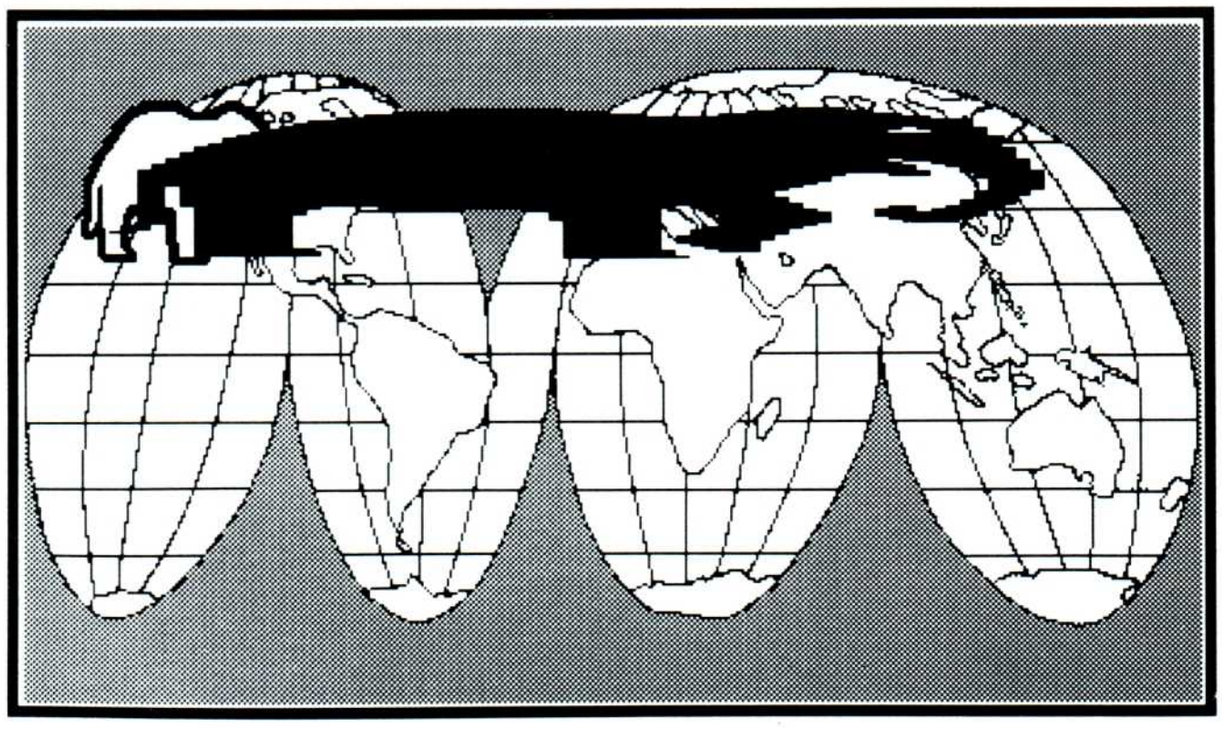


Figure 8-6
Horizontal scaling

Vertical scaling (*/)

NewPic := *picturexpr* / numexpr*

The *picturexpr* expression is resized vertically according to the *numexpr* coefficient. When *numexpr* is positive, either of two things can happen, depending on the value of *numexpr*. If *numexpr* lies between 0 and 1, the picture is reduced. If *numexpr* is greater than 1, the picture is enlarged. When *numexpr* is negative, either of two things can happen, depending on the value of *numexpr*. If *numexpr* lies between 0 and –1, the picture is reduced. If *numexpr* is less than –1, the picture is enlarged. In either case, the picture is flipped vertically. *picturexpr* / -1* returns the vertical "flip" of *picturexpr*. See Figure 8-7.

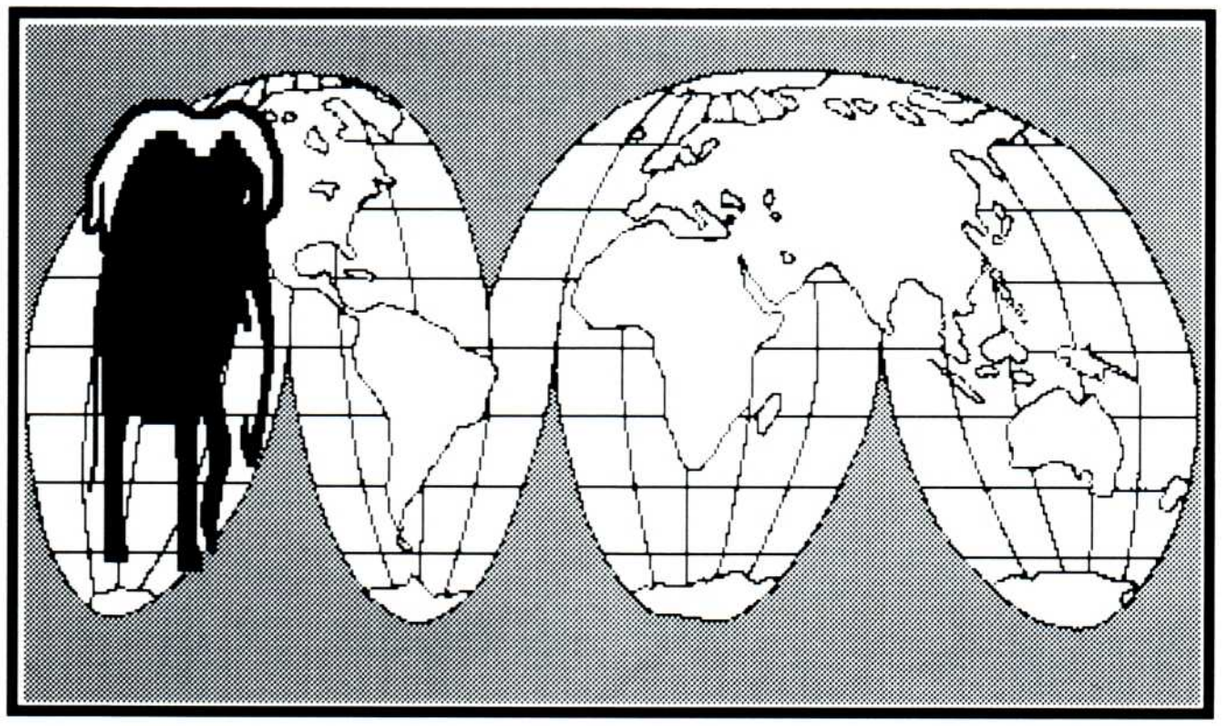


Figure 8-7
Vertical scaling

Negation (Not)

NewPic := **Not** (*picturexpr*)

Not, applied to a Picture expression, returns a Picture expression where all points are inverted. See Figure 8-8.

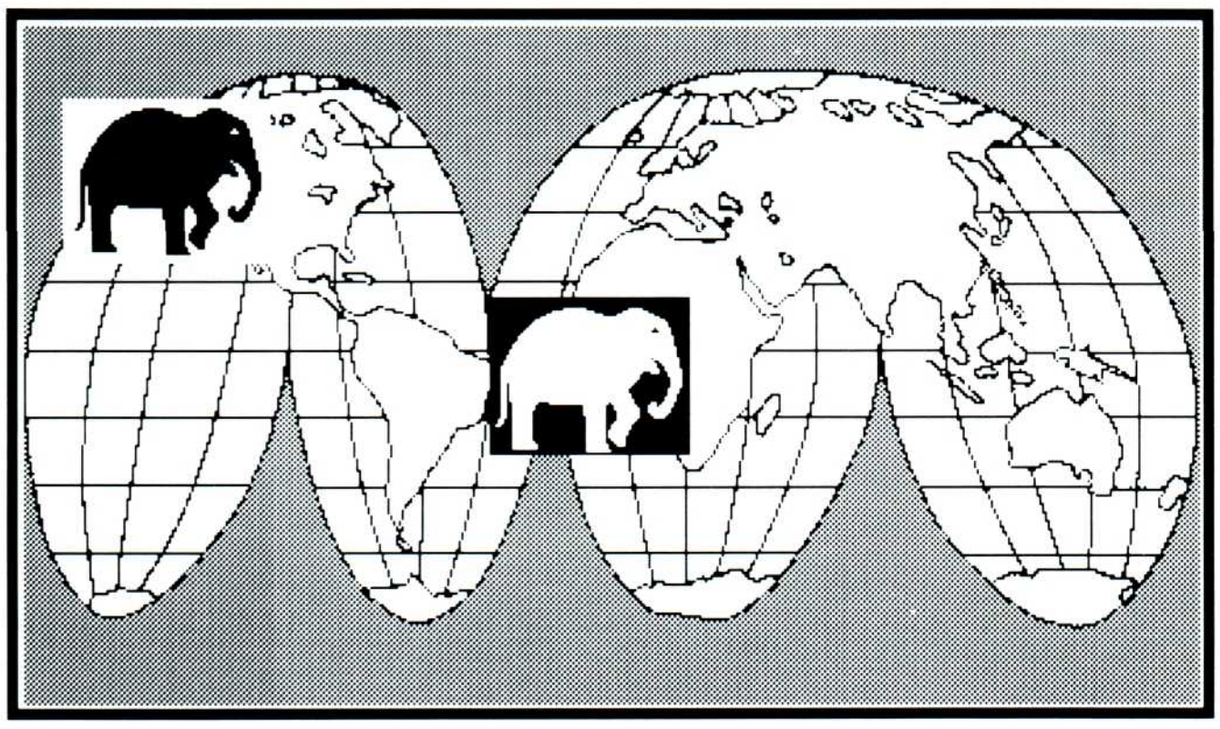


Figure 8-8
Negation

Picture operations examples

You can create interesting and useful graphics using 4th Dimension's picture operators. The moving process flow bar shown in Figure 8-9 is created with the procedure that follows.

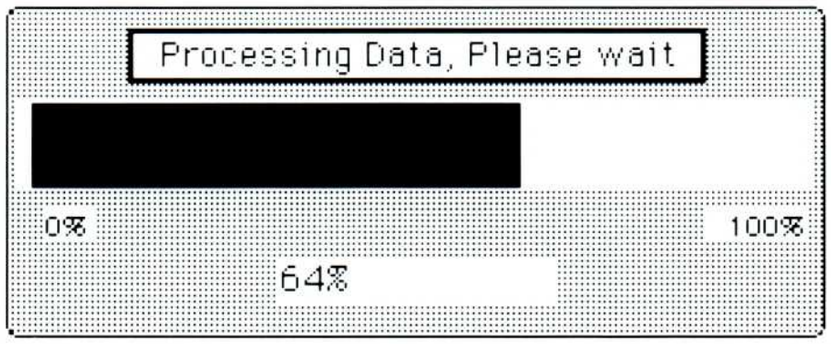


Figure 8-9
Processing indicator

This routine

1. Reads a black box and an (almost) empty box into picture variables.
2. Scales the black and empty boxes, adds them together, and assigns them to a File field.
3. Copies the % done to a variable.
4. Does a DISPLAY RECORD to cause the variables to be updated on the screen.

```
    `Global Procedure: Run Thermo
    `Get the boxes for displaying
SEARCH BY INDEX([PictureFile]PictureName="Black Box")
Box:=[PictureFile]Picture   `Get a filled box
SEARCH BY INDEX([PictureFile]PictureName="Blank")
Blank:=[PictureFile]Picture   `Get a filled box
Empty:=Blank&Blank   `Create even more empty box, remove 2 pixels
ALL RECORDS([PictureFile])   `restore selection
CREATE RECORD([PictureFile])
INPUT LAYOUT([PictureFile];"d.Thermometer")   INPUT layout is used to DISPLAY records
    `Main loop, Display progress
i:=0   ` "i" represents the % complete
Scale:=10
[PictureFile]Picture:=Empty   `start at 0%
While (i<=100)
    VarLabel:=String(i)+"%  "
    [PictureFile]Picture:=(Box*+(i/Scale))+(Empty*+((100-i)/Scale))
    If (i>=95)
        `Ok, now near 100%.  If we don't do this, we get blanks for 95% and above.
        [PictureFile]Picture:=Box
    End if
    DISPLAY RECORD([PictureFile])
    i:=i+4
End while
    `Don't save record!
UNLOAD RECORD([PictureFile])
INPUT LAYOUT([PictureFile];"PictureInput")   `Restore old input layout as the default.
CLEAR VARIABLE("Box")
CLEAR VARIABLE("Blank")
CLEAR VARIABLE("Empty")
CLEAR VARIABLE("VarLabel")
CLEAR VARIABLE("Scale")
```

Figure 8-10 shows a bar chart drawn with the procedure that follows.

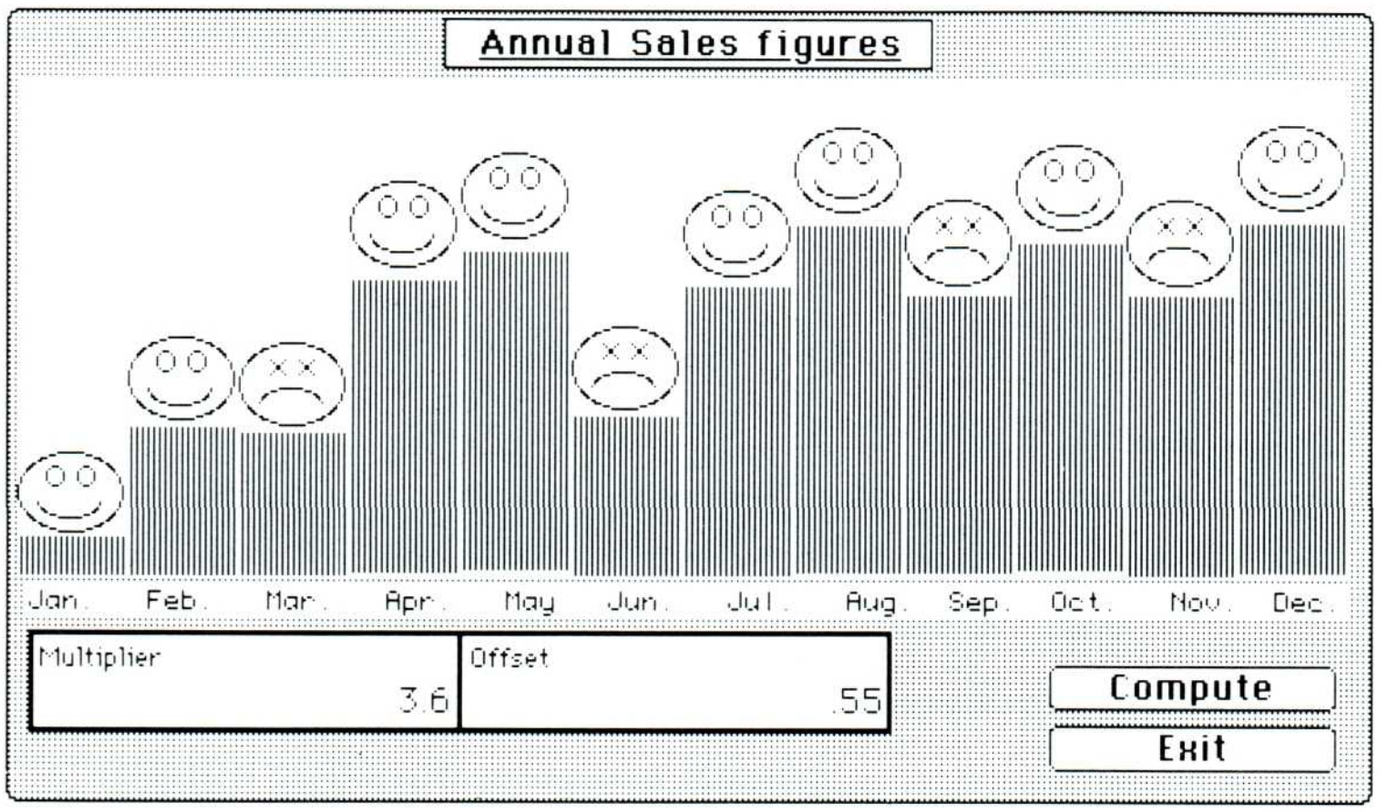


Figure 8-10
Bar graph

This routine:

1. Demonstrates the power of 4th Dimension picture arithmetic by creating a graph from a series of existing pictures.
2. Creates random sales figures for each month (from 0 to MaxSales).
3. Plots a scaled box and either a happy or a sad face for each month—happy if sales are up, sad if sales are down.
4. Sums plots into a single picture variable for display.
5. Supports the user by allowing changes to the scaling factors and recomputing the graph based on these factors.

```
      `Layout procedure: d.graph
      `File: [PictureFile]    Even though this dialog box is within the [PictureFile]
      `file, there is no requirement that it be.

      `You will likely have to adjust the size of the picture variable in the dialog
      `layout if you want your patterned areas to come out consistent.
      `Remember that the entire variable is scaled to fit in the variable
      `layout region at the time it is displayed.
If (Before)
      `Define constants.
   ScaleMul:=3.6
   ScaleOff:=0.55
   MaxSales:=120   `Just pick a figure for the maximum sales value.  Normally, you'd have to compute
      `this figure from the file data.
      `Note: The pictures read are all the same size, 38 pixels wide.
```

```
SEARCH BY INDEX([PictureFile]PictureName="Box")  `Box with vertical stripes
pBox:=[PictureFile]Picture
pBlank:=[PictureFile]Picture  `Assign blank to the same size/shape box
pBlank:=pBlank&pBlank  `XOR to remove all black lines, leaving white that is just the right size.
SEARCH BY INDEX([PictureFile]PictureName="Happy")  `Happy face
pHappy:=[PictureFile]Picture
SEARCH BY INDEX([PictureFile]PictureName="Sad")  `Sad face
pSad:=[PictureFile]Picture
ALL RECORDS([PictureFile])
    `Create sales figures (Mostly random)
Sales0:=0  `Just to have something to compare against.
Sales1:=0  `Start with a very small sales month.  Done for range checking of the result.
month:=2  `First month assigned, now do months 2–11.

While (month<=11)
    Sales{month}:=Int((Random/32676)*MaxSales)  `range from 10 to 120
    month:=month+1
End while
Sales12:=MaxSales  `End on a high note.
End if  `(Before)
If ((During)&(bCompute=1)|(Before))  `The before is here just to save the user
    `from having to press the [Compute] button.
    `Compute Vertical Bar and picture for each month.
    `Valid values for "SolidScale" seem to be limited to: 0.0666 to 1.333
    month:=1
    While (month<=12)
        SolidScale:=(ScaleMul*(Sales{month}/MaxSales))
        pBarSolid:=pBox*/(SolidScale+ScaleOff)
        pBarEmpty:=pBlank*/((ScaleMul-SolidScale)+ScaleOff)
            `Put it all together with the appropriate face.
        If (Sales{month}>=Sales{month-1})
            pMonth{month}:=pBarEmpty/pHappy/pBarSolid
        Else
            pMonth{month}:=pBarEmpty/pSad/pBarSolid
        End if
        month:=month+1
    End while
        `String them all together for a complete graph for 1 month
    month:=1
    CLEAR VARIABLE("var")
    While (month<=12)
        var:=var+pMonth{month}
        month:=month+1
    End while
End if
```

Chapter 9

ASCII Maps

This chapter looks at ASCII maps—what they are and how to use them—in the context of file import and export. (*ASCII* stands for *American Standard Code for Information Interchange.*)

File import and export

In the User or Custom environment, you may exchange, import, or export data between your 4th Dimension database and other documents stored on disk or in another computer. A 4th Dimension **ASCII map** is a table that translates from one character representation code to another. The table can be active during an application's storage operation or while exchanging data through the serial port, a network, or a direct connection between computers.

Several 4th Dimension commands that take advantage of the ASCII map facility work in the User and Custom environments. They can read from or write documents to disk and communicate with a Macintosh or other computer through the serial port. To work with documents on disk, choose **EXPORT DIF, EXPORT SYLK,** and **EXPORT TEXT,** to export data and **IMPORT DIF, IMPORT SYLK,** and **IMPORT TEXT** to import data.

When in the User environment, choose the Export Data command from the File menu to export data stored in records belonging to the current file selection. Choose the Import Data command from the File menu to add records to the current file. You can export or import data in the SYLK, DIF, or TEXT formats.

Using 4th Dimension's communications commands—**SET CHANNEL, RECEIVE PACKET, SEND PACKET,** and **RECEIVE BUFFER**—you can read from and write to documents stored on disk and communicate with another computer through the serial port.

Uses for an ASCII map

There are any number of reasons for exchanging data. Here are a few of the things you may want to do:

- Import text documents stored on another computer.
- Replace carriage return characters (ASCII decimal code 13) with line feed characters (ASCII decimal code 30).
- Do rudimentary substitution encryption for file security.
- Send 8-bit ASCII codes directly to an ImageWriter printer (7 bit) which only uses ASCII characters 1 through 127.

Exchanging data between several 4th Dimension databases, or between one 4th Dimension database and a document created with another Macintosh application, is a simple procedure. Conversely, if you want to exchange data between a 4th Dimension database and a document created with an application program designed for another type of computer, you may have to use the ASCII map.

In a computer, each character is assigned a unique ASCII code number from 0 to 255. When two computers communicate, they exchange codes corresponding to characters. There is, however, a problem: while all computers use the same code for the first 128 characters, they use different codes for all the remaining characters. For example, a Macintosh computer assigns the 131 code to an accented capital E (É), whereas another type of computer may assign the 144 code to that character. The host computer sends ASCII code 144 standing for É, and your Macintosh looks it up in its code table and "translates" the 144 code as an ê.

Working with ASCII maps

An ASCII map can solve this communication problem.. When creating an ASCII map, you create a correspondence between the ASCII table of your Macintosh and that of the host computer. For instance, you will specify that ASCII code number 131 on your Macintosh is equivalent to number 144 on the host computer. Then you select the appropriate map as the current ASCII map; that is, the default map that 4th Dimension will use during a communication session.

Important

Once invoked, an ASCII map remains in effect until you leave the database in which you created it. Therefore, If you choose to use an ASCII map, be sure to reset the map with the command **USE ASCII MAP (*)**. This way, after you have finished an ASCII map setup, you can return to your Macintosh's standard ASCII interpretation.

The way maps work is rather simple. When 4th Dimension sends a given character to a disk or to a serial port, it actually sends the corresponding map character. When 4th Dimension receives a given character from a disk or serial port, it displays the corresponding map character on screen.

You may create any number of maps. You will thus have the ability to communicate with many types of computers. You won't need to exchange SYLK, DIF, or ASCII documents through character-converting utilities, but rather you can work with documents directly in the 4th Dimension environment. This means you don't have to modify your application; you just create a new map where necessary.

You create maps in the User environment; choose the Edit Map command from the Special menu. (For details on creating and using ASCII maps, see Chapter 3 of *4th Dimension User's Guide.*) You can also use maps from within the User Environment. When in the Custom environment, select any map as the current map with the USE ASCII MAP command. Figure 9-1 shows the Edit map dialog box.

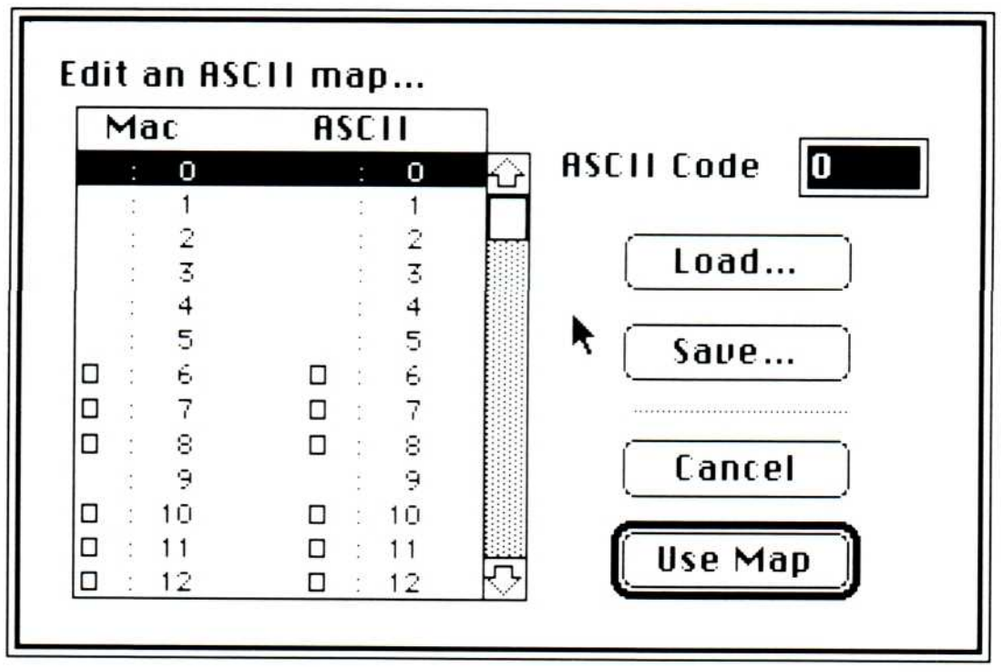


Figure 9-1
Edit map dialog box

Index

D

E

F

G

R

S

T

U

V

W, X, Y, Z