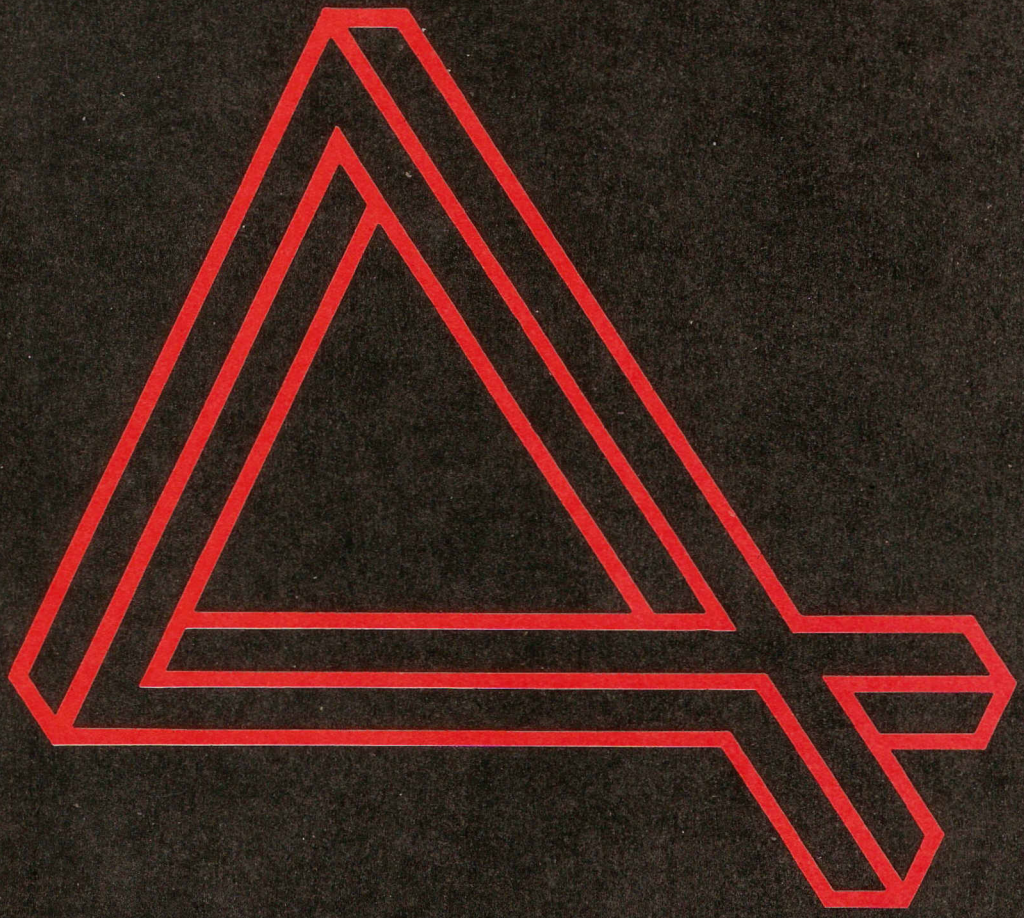
4th DIMENSION®

LANGUAGE REFERENCE

ACS



Date	Topic	Page
9 June 98	Precedence	108
	Menu/Procedures	69 **

Binder 2

Ver 2.1

4th DIMENSION LANGUAGE REFERENCE

4th DIMENSION by Laurent Ribardière

4th DIMENSION LANGUAGE REFERENCE

Written by Will Mayall
Technical assistance from Dave Terry
Copyedited by Silvio Orsino
Illustrated by Will Mayall
Layout by Will Mayall
Designed by Patrick Chédal C&C
Publication assistance from Lasselle-Ramsay, Inc.

SOFTWARE LICENSE AGREEMENT

ACI grants you a non-transferable, non-exclusive license to use this copy of the program and accompanying materials according to the following terms:

LICENSE:

You may:

a) use the program on only one computer at a time;

b) make one (1) copy of the program in machine readable form solely for backup purposes, provided that you reproduce all proprietary notices on the copy;

c) physically transfer the program from one computer to another, provided that the program is used on only one computer at a time; and

d) transfer the program onto a hard disk only for use as described above provided that you can immediately prove ownership of the original diskettes.

You may not:

a) use the program in a network unless you pay for a separate license for each terminal or workstation from which the program will be accessed;

b) modify, translate, reverse engineer, decompile, disassemble, create derivative works based on, or copy (except for the backup copy) the program or accompanying materials;

c) rent, transfer or grant any rights in the program in any form or accompanying materials to any person without the prior written consent of ACI which, if given, is subject to the conferee's consent to the terms and conditions of this license; or

d) remove any proprietary notices, labels or marks on the program and accompanying materials.

This license is not a sale. Title and copyrights to the program, accompanying materials and any copy made by you remain with ACI.

TERMINATION

Unauthorized copying of the program (alone or merged with other software) or the accompanying materials, or failure to comply with the above restrictions will result in automatic termination of this license and will make available to ACI other legal remedies. Upon termination you will destroy or return to ACI the program, accompanying materials and any copies.

LIMITED WARRANTY AND DISCLAIMER

THE PROGRAM AND ACCOMPANYING MATERIALS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

ACI does not warrant that the functions contained in the program will meet your requirements or that the operation will be uninterrupted or error free. The entire risk as to the use, quality, and performance of the program is with you. Should the program prove defective, you, and not ACI, assume the entire cost of any necessary repair.

However, ACI warrants the diskettes on which the program is furnished to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of delivery to you as evidenced by a copy of your receipt. The duration of any implied warranties on the diskettes is limited to the period stated above. ACI's entire liability and your exclusive remedy as to the diskettes (which is subject to you returning the diskettes to ACI or an authorized dealer with a copy of your receipt) will be the replacement of the diskettes or, if ACI or the dealer is unable to deliver a replacement diskette, the refund of the purchase price and termination of this Agreement.

SOME STATES DO NOT ALLOW LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY LASTS SO THE ABOVE LIMITATION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

LIMITATION OF LIABILITY

IN NO EVENT WILL ACI BE LIABLE FOR ANY DAMAGES, INCLUDING LOSS OF DATA, LOST PROFITS, COST OF COVER OR OTHER SPECIAL, INCIDENTAL, CONSEQUENTIAL OR INDIRECT DAMAGES ARISING FROM THE USE OF THE PROGRAM OR ACCOMPANYING MATERIALS, HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY. THIS LIMITATION WILL APPLY EVEN IF ACI OR AUTHORIZED DEALER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. YOU ACKNOWLEDGE THAT THE LICENSE FEE REFLECTS THIS ALLOCATION OF RISK. SOME STATES DO NOT ALLOW LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

GENERAL

This Agreement will be governed by the laws of France. In any dispute arising out of this Agreement, ACI and you each consent to the jurisdiction of the courts of France.

Use, duplication or disclosure by the U.S. Government is subject to restrictions stated in paragraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at 252.227-7013.

Licensor: **ACI, 5 Rue Beaujon, 75008 Paris, France**

This Agreement is the entire agreement between us and supersedes any other communications with respect to the program and accompanying materials.

If any provision of this Agreement is held to be unenforceable, the remainder of this agreement shall continue in full force and effect.

If you have any questions, please contact: **ACI Customer Service, (33) 1 42 27 37 25** or write us at the above address.

SIGN AND MAIL THE REGISTRATION CARD TODAY. Return of the registration card is required to receive any product updates and notices of new versions or enhancements.

CONTENTS

FIGURES AND TABLES

PREFACE

PREFACE

4th DIMENSION is a powerful relational database application and development tool for Apple's family of Macintosh computers.

You can use 4th DIMENSION to manage your own data or develop custom applications for different kinds of database management tasks.

For example you can

- Create a database structure of files and fields.
- Design layouts for entering, modifying, and displaying records.
- Search and sort records.
- Create reports and labels from information in the databases.
- Import and export data between 4th DIMENSION databases and other applications.

With 4th DIMENSION, you can enhance conventional data management tasks with these features:

- The powerful Layout editor that works like a full-featured drawing program to let you add object-oriented graphics and fonts to your layouts.
- The capacity to store graphics in database files.
- A password access system to protect sensitive data.
- Graphing functions that let you generate a variety of business graphs from your data.
- The capability to create custom applications from 4th DIMENSION with your own custom menus, dialog boxes, and buttons.
- A full-featured programming language that lets you incorporate procedures written in other languages.

4th DIMENSION's flexibility and power makes it ideal for a complete range of information management tasks. Novice users can quickly create databases and begin managing their data. Experienced users without programming experience can customize their databases with 4th DIMENSION's development tools. More experienced developers can use 4th DIMENSION's powerful programming language to add sophisticated features and capabilities to their databases, including file transfer and communications.

When you design a custom database with 4th DIMENSION, all the components of the Macintosh's user interface are at your disposal. You can add menus, dialog boxes, buttons, and windows to enhance your databases and make users more productive.

About the Manuals

The *4th DIMENSION Language Reference* is a guide to using the 4th DIMENSION language. Use this manual to learn how to use the language to customize a database. You should use it in conjunction with the other volumes in your documentation package.

The *4th DIMENSION Quick Start* and *4th DIMENSION Tutorials* lead you through example lessons where you create and use a 4th DIMENSION database. These examples provide hands-on experience and help you become familiar with the concepts and features of 4th DIMENSION.

The *4th DIMENSION Design Reference* serves as a reference guide to 4th DIMENSION's design environment and provides detailed descriptions of 4th DIMENSION operations that you can perform in this environment.

The *4th DIMENSION User Reference* provides a description of the environment where you will use the databases and layouts to enter and manipulate data.

The *4th DIMENSION Utilities Guide* provides a guide to the utilities available with 4th DIMENSION, such as 4D Tools, 4D Customizer, and 4D External Mover.

The *4th DIMENSION Glossary and Master Index* provides a glossary that defines terms and an index to all 4th DIMENSION documents.

About This Manual

This manual describes the 4th DIMENSION language. This manual assumes that you're familiar with terms such as file, field, and layout. Before you read this manual, you should

- use the *Quick Start* and *Tutorials* volumes to work through the database examples as needed.
- begin creating your own databases, referring to the *Design Reference* when you need to review a procedure or explanation.
- be comfortable with managing your database in the User environment. See the *4th DIMENSION User Reference* for more information on the User environment.

Part Descriptions

This manual is divided into three parts:

- Part I, "The Language," introduces you to the 4th DIMENSION language—why it exists, what it can do, and how to use it. It covers the fundamental components of the language and introduces the terminology used.
- Part II, "Language Definition," formally defines the components of the language. It contains reference information about how to name and refer to variables, files, and other objects in the language.
- Part III, "The Commands," documents the commands in the language. It gives the syntax for each command, a description of the command, and examples. The commands are organized by task. Part III contains discussions of topics, such as data entry, searching, and printing reports, that are directly relevant to most databases; and of other advanced topics, like transaction management and serial communication. You don't need to read about every command before you start using the language.

Aids to Understanding

This manual, and the other manuals in your documentation package, uses visual aids to help you understand the material.

Here are some examples of the visual aids in the manual:

Note: Text emphasized like this provides annotations and shortcuts that will help you become more productive with 4th DIMENSION.

Important: Notes like this alert you to important pieces of information.

Warning: Warnings like this alert you to situations where data might be lost.

Visual Conventions Used in This Manual

This manual uses a number of visual conventions to identify procedure code and commands.

- Code examples and commands are in a special font.
  For example: Piece of Code
- In code examples, commands appear in the special font, in bold.
  For example: **ADD RECORD**
- Commands that do not return a value are all uppercase.
  For example: **DEFAULT FILE**
- Commands that return a value (functions) have an initial capital letter.
  For example: **Records in file**
- Global procedures appear in the special font, in italic.
  For example: *My Proc*
- External procedures appear in the special font, in bold-italic.
  For example: ***My External***
- Parameters to commands appear in the normal font in italic.
  For example: *normal*

These conventions are used in the Procedure editor, in the Debugger, and in printed listings, as well as in this manual.

PART I

THE LANGUAGE

CHAPTER 1

INTRODUCTION

INTRODUCTION

This chapter introduces you to the 4th DIMENSION language. It discusses

- what the language is and what it can do for you
- how you'll use procedures
- developing your application

These topics are covered here in general terms—they're covered in more detail in later chapters.

What Is a Language?

The 4th DIMENSION language is not very different from the spoken language we use every day. It is a form of communication used to express ideas, to inform, and to instruct. Like a spoken language, 4th DIMENSION has its own vocabulary, grammar, and syntax, and you use it to tell 4th DIMENSION how to manage your database and data.

You do not need to know everything in the language. In order to speak you do not need to know the whole English language; in fact, you can have a small vocabulary and still be quite eloquent. The 4th DIMENSION language is much the same—there is only a small part of the language that you need to know in order to become productive, and you can learn the rest as the need arises.

Why Use a Language?

At first it may seem that there is little need for a programming language in 4th DIMENSION. The Design and User environments provide flexible tools that require no programming to perform a wide variety of data management tasks. All of the fundamental tasks, such as data entry, searching, sorting, and reporting, are handled with ease. In fact, many extra capabilities are available, such as data validation, data entry aids, graphing, and label generation.

Then why do we need a language? For several purposes:

- automating repetitive tasks—including data modification, generation of complex reports, and unattended completion of long series of operations
- controlling the user interface—including window management, menu management, layout control, and interface object control
- performing sophisticated data management—including transaction processing, complex data validation, multi-user management, and set operations
- controlling the computer—including serial port communications, document management, and custom error management
- creating applications—the creation of easy-to-use customized databases that use the Runtime environment

The language lets you take complete control over the design and operation of your database. Whereas the User environment gives you powerful "generic" tools, the language lets you customize your database to whatever degree you require.

Taking Control

The 4th DIMENSION language lets you take *complete* control of your data in a manner that is both powerful and elegant. The language is easy enough for a beginner to start with, and sophisticated enough for an application developer. It provides smooth transitions from built-in control over the database to a completely customized database.

The commands in the language provide the User environment editors that you are already familiar with. For example, when you use the SEARCH command, you are presented with the Search editor—using this command is almost as easy as choosing the Search menu item. But the SEARCH command is even more useful. If you want, you can tell the SEARCH command explicitly what to search for. For example, SEARCH ([People]Last Name = "Smith") will find all the people named Smith in your database.

The language is very powerful—one command often replaces hundreds or even thousands of lines of programming done in traditional computer languages. With power, surprisingly enough, comes simplicity. Commands have plain English names: To search, you use the command SEARCH; to add a new record, you use the command ADD RECORD.

The language is designed so that you can easily accomplish the most common tasks. Adding a record, sorting a file, searching for data, and similar operations are specified with simple and direct commands. But the language can also control the serial ports, read disk documents, control sophisticated transaction processing, and much more.

Even the most sophisticated tasks are specified with *relative* simplicity. To perform these tasks without using the 4th DIMENSION language would be unimaginable for many. Even with the language's powerful commands, some tasks can be complex and difficult. A tool by itself does not make a task possible; the task itself may be challenging and the tool can only ease the process. For example, a word processor makes writing a book faster and easier, but it will not write the book for you. Using the 4th DIMENSION language will make the process of managing your data easier and will allow you to approach sophisticated tasks with confidence.

Is It a "Traditional" Computer Language?

If you are familiar with traditional computer languages, this section may be of interest. If not, you may want to skip it.

The 4th DIMENSION language is *not* a traditional computer language. It is one of the most innovative and flexible languages available on a computer today. The language has been designed to work the way you do, not the other way around.

To use traditional languages, you must do extensive planning. In fact, planning is often one of the major steps in development. 4th DIMENSION allows you to start using the language at any time and in any part of your database. You may start by adding a script to a layout, then later add a procedure or two. As your database becomes more sophisticated, you might add a global procedure controlled by a menu. You can use as little or as much of the language as you want. It is not "all or nothing," as is the case with many other databases.

Traditional languages force you to define and pre-declare objects in formal syntactic terms. In 4th DIMENSION, you simply create the object and use it. 4th DIMENSION automatically manages the object for you. For example, to use a button, you draw it on a layout and name it. When the user clicks the button, the language automatically notifies your procedures.

Traditional languages have been rigid and inflexible, requiring commands to be entered in very formal and restrictive style. The 4th DIMENSION language breaks with tradition, and the benefits are yours.

Procedures—Gateway to the Language

It is through procedures that you use the 4th DIMENSION language. A procedure is nothing more than a series of instructions that causes 4th DIMENSION to perform a task. Each line of instruction in a procedure is called a *statement*. Each statement is composed of parts of the language.

Because you have gone through the *4th DIMENSION Tutorials* (you did go through the *Tutorials*, didn't you?), you have already written and used scripts and procedures.

There are four types of procedures you create when using 4th DIMENSION:

- scripts—short procedures used to control layout objects
- layout procedures—procedures that manage the display of a layout
- file procedures—similar to layout procedures, but applied to all layouts in a file
- global procedures—procedures that are available for use throughout your database

The next sections introduce each of these procedure types and give you a feel for how you can use them to automate your database. All of the procedure types are covered in more depth later in this manual.

Getting Started—Scripts

Any layout object that can perform an action—that is, an active object—can have a script associated with it. A script monitors and manages the active object, during both data entry and printing. The script is bound to the active object, staying with the active object when it is copied and taking control exactly when needed.

Scripts are the primary tools for managing the user interface. The user interface consists of the methods and conventions by which a computer communicates with the person operating it. The user interface is the doorway to your database. The goal is to make the user interface of your database as smooth, simple, and automated as possible. The user interface should make interaction with the computer a pleasant process, one that the user enjoys or does not even notice.

Figure 1-1 shows you some of the active objects that can have scripts.

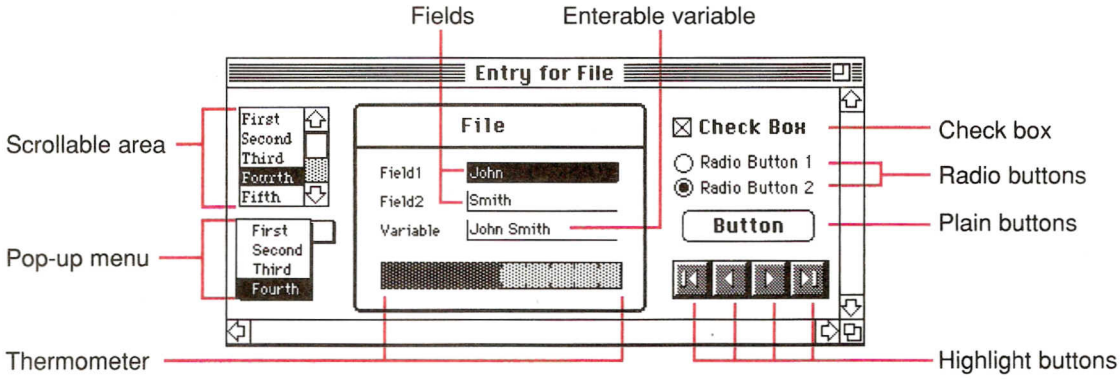


Figure 1-1
Some of the active objects that can have scripts

All active objects have built-in aids, like range checking and character filters for data entry areas, and automatic actions for buttons. Always use these aids before adding scripts. The built-in aids are similar to scripts in that they remain associated with the active object and are active only when the active object is being used. You will typically use a combination of built-in aids and scripts to control the user interface.

There are two basic types of active objects in a layout: active objects for entering and displaying data, such as fields and enterable variables; and active objects for control, such as buttons, scrollable areas, and thermometers.

A script associated with an active object used for data entry typically performs a data management task specific to the field or variable. The script might perform data validation, data formatting, do calculations, get related information from other files, and so on. Some of these tasks can, of course, also be performed with the built-in data entry aids for objects. Use scripts when the task is more complex than the built-in data entry aids can manage. See the *4th DIMENSION Design Reference* for more information on the built-in data entry aids.

Figure 1-2 shows a sample script that changes the field that it is associated with (Field1) to all uppercase characters.

Figure 1-2
An example script for a field

Scripts are also associated with active objects used for control, such as buttons. Active objects used for control are essential to navigating within your database: Buttons allow you to move from record to record, move to different layouts, and add and delete data. These active objects simplify the use of a database and reduce the time required to learn it. Buttons also have built-in aids and, as with data entry, you should use these built-in aids before adding scripts. Scripts give you the ability to add actions to your controls that are not built-in. For example, Figure 1-3 shows a script for a button that will display the Search editor when clicked.

Figure 1-3
An example script for a button

As you become more proficient with scripts, you will find that you can create libraries of objects with associated scripts. You can copy and paste these objects and their scripts between layouts, files, and databases. You can even keep them in the Scrapbook, ready to be used when you need them.

Controlling Layouts—Layout Procedures and File Procedures

In the same way that scripts are associated with the active objects in a layout, a layout procedure is associated with a layout, and a file procedure is associated with a file. Each layout can have one layout procedure; each file can have one file procedure.

A layout is a view of your data. Through the use of layouts, you can create attractive and easy-to-use data entry screens and printed reports. A layout procedure monitors and manages the use of a layout both for data entry and for printing. Figure 1-4 shows a sample layout procedure.

A file procedure also manages layouts, but only for data entry and for *every* layout in the file. In all other aspects, layout and file procedures are identical.

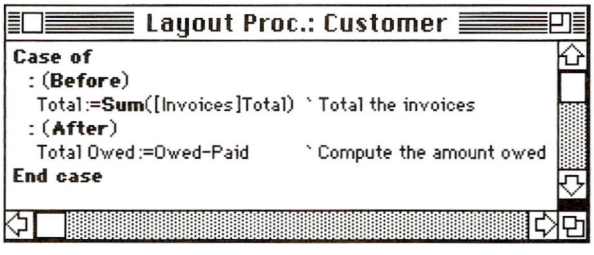


Figure 1-4
An example layout procedure

A layout procedure manages a layout at a higher level than do scripts. Scripts are activated only when the object is used; a layout procedure is activated when *anything* in the layout is used. Layout procedures are typically used to control the interaction between the different objects and the layout as a whole. Whenever a script is activated, the layout procedure is also activated.

Since layouts are used in so many different ways, it is useful to monitor what is happening while your layout is in use. You use the layout execution cycle for this purpose. It tells you what is currently happening with the layout. The execution cycle is broken into several phases, each phase occurring at different times in the layout. The execution cycle is described in Chapter 5.

Using Global Procedures—They're Everywhere

Unlike layout procedures and scripts, which are associated with a particular layout or active object, global procedures are available for use throughout your database. They are reusable, available for use by any other procedure. If you need to do a task over and over again, you don't to have to write identical procedures for each case.

You can *call* global procedures wherever you need them, from other global procedures or from scripts and layout procedures. When you call a global procedure, it acts just as if you had written the whole procedure at the place where you called it. Global procedures called from other procedures are often referred to as *subroutines*.

Figure 1-5 shows a global procedure that searches a file and then prints a report.

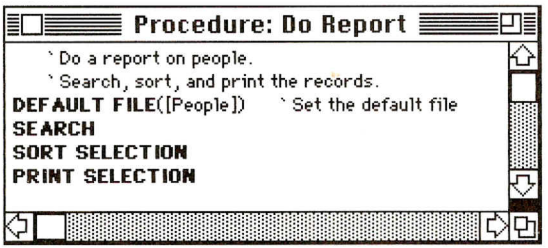


Figure 1-5
An example global procedure

There is one other way to use global procedures: associating them with menu items. When you associate a global procedure with a menu item, the procedure is executed when the menu item is chosen. You can think of the menu item as calling the global procedure. In an application, the procedures that are called from menus become the *master procedures*—the procedures that control the overall operation of the application.

Developing Your Database

Development is the process of customizing a database, using the language and other built-in tools.

The language lives in a world created by your database. By simply creating a database, you have already taken the first steps to using the language. All of the parts of your database—the files and fields, the layouts and their objects, and the menus—are intimately tied to the language. The 4th DIMENSION language "knows" about all of these parts of your database.

Perhaps your first use of the language is to add a script to a layout object, to control data entry. Later, you might add a layout procedure to control the display of your layout. As the database becomes more mature, you add a menu bar with global procedures to completely customize your database.

As with other aspects of 4th DIMENSION, development is a very flexible process. There is no formal path to take during development—you can develop in a manner comfortable to you. There are, of course, some general patterns in the process. You implement your design in the Design environment. You try out the design in the User environment and perhaps stay there to use your customized database. When your database is fully customized, you use it in the Runtime environment. If you find errors, you go back to the Design environment to fix them.

There are special support tools for development—they are woven into 4th DIMENSION, hidden until you need them. As your use of the language becomes more sophisticated, you will find that these tools ease the development process. For instance, the Procedure editors catch typing errors and format your work; the Interpreter (the engine that runs the language) catches errors in syntax and shows you where and what they are; and the Debugger lets you monitor the execution of your procedures to catch errors in design.

Putting It Together—Building Applications

You are by now familiar with the general uses of a database—data entry, searching, sorting, and reporting. You have performed these tasks in the User environment, using the built-in menus and editors. As you use a database, it becomes obvious that there are sequences of tasks that are performed over and over again. For example, in a database of personal contacts, you might search for your business associates, sort them by last name, and print a specific report, each time information about them is changed. These tasks may not seem difficult, but they certainly may seem time-consuming after you have done them 20 times. In addition, if you don't use the database for a couple of weeks, you may return to find that the steps used to generate the report are not so fresh in your mind. When you create procedures, the steps are chained together, so that choosing a single menu item performs all the tasks unassisted, and you don't have to worry about the specific steps any more.

An application takes database automation to its culmination. Applications have custom menus and perform tasks that are specific to the needs of the person using your database. An application is composed of all the pieces of your database: the structure, the layouts, the scripts and procedures, the menus, and the passwords.

An application can be as simple as a single menu that lets you enter people's names and print a report, or as complex as an invoicing, inventory, and control system. There are no limits to the uses of applications. Typically, an application grows from a database used in the User environment to a database controlled completely by custom menus.

Developing applications can be as simple or complex as you like. Chapter 7 describes the processes used to build an application.

COMPONENTS OF THE LANGUAGE

CHAPTER 2

COMPONENTS OF THE LANGUAGE

The 4th DIMENSION language is made up of various components that help you perform tasks and manage your data:

- variables—temporary storage places in memory for data
- operators—symbols that perform a calculation between two values
- expressions—combinations of other components that result in a value
- commands—built-in instructions to 4th DIMENSION to perform an action
- procedures—instructions that you write by using all the other parts of the language

This chapter discusses variables, operators, and expressions. Chapters 3, 4, 5, and 6 describe the different types of procedures, and Part III describes each command in the language.

Types of Data

There are many types of data that can be stored in a 4th DIMENSION database. In the language, the various types of data are referred to as *data types*.

There are seven data types:

- String. A series of characters, such as "Hello there". Alpha and Text fields are of the string data type.
- Numeric. Numbers, such as 2 or 1,000.67. This data type is also referred to as number. Integer, Long Integer, and Real fields are of the numeric data type.
- Date. Calendar dates, such as 1/20/89. Date fields are of the date data type.
- Time. Times, including hours, minutes, and seconds, such as 1:00:00 or 4:35:30 P.M. Time fields are of the time data type.
- Boolean. Logical values of TRUE or FALSE. Boolean fields are of the Boolean data type.
- Picture. Macintosh pictures. Picture fields are of the picture data type.
- Pointer. A special type of data used in advanced programming. There is no corresponding field type.

Notice that in the list of data types, the string and numeric data types are associated with more than one type of field. When the language is operating on a field of one of these types, it automatically converts the data to the data type the language supports. For example, if an integer field is used, its data is automatically treated as numeric. When data is put into a field, the language automatically converts the data to the correct type for the field. In other words, you need not worry about mixing similar field types when using the language; it will manage them for you.

However, it is important, when using the language, that you do not mix different *data* types. In the same way that it makes no sense to store "ABC" in a Date field, it makes no sense to put "ABC" in a variable used for dates. In most cases, 4th DIMENSION is very tolerant and will try to make sense of what you are doing. For example, if you add a number to a date, 4th DIMENSION will assume that you want to add that number of days to the date, but if you try to add a string to a date, 4th DIMENSION will tell you that the operation makes no sense.

There are cases where you need to store data as one type and use it as another type. The language contains a full complement of commands that let you convert from one data type to another. For example, you may need to create a part number that starts with a number and ends with characters such as "abc". In this case, you might write

```
Part Number := String (Number) + "abc"
```

where if Number is 17, then Part Number will get the string "17abc".

The data types are formally defined in Part II of this manual.

Operators

When you use the language, it's rare that you'll simply want a piece of data. It's more likely that you'll want to do something to or with that data. You do these calculations with *operators*. Operators, in general, take two pieces of data and perform an operation on them that results in a new piece of data. You are already familiar with many operators. For example, 1 + 2 uses the addition (or plus sign) operator to add two numbers together, and the result is 3. This table shows some familiar numeric operators.

Operator	Operation Performed	Example
+	Addition	1 + 2 results in 3.
–	Subtraction	3 – 2 results in 1.
*	Multiplication	2 * 3 results in 6.
/	Division	6 / 2 results in 3.

Numeric operators are just one type of operator available to you. Since 4th DIMENSION supports many different types of data, such as numbers, text, dates, and pictures, there are operators that perform operations on these different data types.

The same symbols are often used for different operations, depending on the kind of data that is operated on. For example, the plus sign (+) performs different operations with different data, as this table shows.

Data Type	Operation Performed	Example
Number	Addition	1 + 2 adds the numbers and results in 3.
String	Concatenation	"Hello " + "there" concatenates (joins together) the strings and results in "Hello there".
Date and Number	Date addition	!1/1/1989! + 20 adds 20 days to the date, January 1, 1989, and results in the date January 21, 1989.

The operators are formally defined in Chapter 10.

Expressions

Simply put, expressions just return a value. In fact, when using the language, you use expressions all the time and tend to think of them only in terms of the value they represent. Expressions are also sometimes referred to as *formulas*.

Expressions are made up of almost all the other parts of the language: commands, operators, variables, and fields. You use expressions to build statements (lines of code), which in turn are used to build procedures. The language uses expressions wherever it needs a piece of data.

Expressions rarely "stand alone." There are only two places in 4th DIMENSION where an expression can be used by itself: in the Search by Formula dialog box in the User environment; and in the Debugger, where the value of expressions can be checked.

An expression can simply be a *constant*, such as the number 4 or the string "Hello". As the name implies, a constant's value never changes.

It is when operators are introduced that expressions start to get interesting. In preceding sections you have already seen expressions that use operators. For example, 4 + 2 is an expression that uses the addition operator to add two numbers together and return the result, 6.

You refer to an expression by the data type it returns. There are seven expression types:

- string expression
- numeric expression (also referred to as number)
- date expression
- time expression
- Boolean expression
- picture expression
- pointer expression

Table 2-1 gives examples of each of the seven types of expressions. The data types are formally defined in Chapter 10.

Table 2-1
Example expressions

Expression	Type	Explanation
"Hello"	String	This is a string constant, the word *Hello*. Note the use of double quotation marks to indicate that this is a string constant.
"Hello " + "there"	String	Two strings, "Hello " and "there", are added together (concatenated) with the string concatenation operator (+). The string "Hello there" is returned.
"Mr. " + Name	String	Two strings are concatenated: the string "Mr. " and the current value of the field Name. If the field contains "Smith", the expression returns "Mr. Smith".
L331 **Uppercase** ("smith")	String	This expression uses "Uppercase," a command from the language, to convert the string "smith" to uppercase. It returns "SMITH".
4	Number	This is a number constant, 4.
4 * 2	Number	Two numbers, 4 and 2, are multiplied, using the multiplication operator (*). The result is the number 8.
My Button	Number	This is the name of a button. It returns the current value of the button: 1 if it was clicked, 0 if not.
!1/25/88!	Date	This is a date constant for the date 1/25/88 (January 25, 1988). Note the use of exclamation marks to indicate a date constant.
L335 **Current date** + 30	Date	This is a date expression that uses the command "Current date" to get today's date. It adds 30 days to today's date and returns the new date.
†8:05:30†	Time	This is a time constant that represents 8 hours, 5 minutes, and 30 seconds.
"† Option t †2:03:04† + †1:02:03†	Time	This expression adds two times together and returns the time 3:05:07.
L349 **True**	Boolean	This is a command that returns the Boolean value TRUE.
10 # 20	Boolean	This is a logical comparison between two numbers. The number sign (#) means "is not equal to." Since 10 "is not equal to" 20, the expression returns TRUE.
"ABC" = "XYZ"	Boolean	This is a logical comparison between two strings. They are not equal, so the expression returns FALSE.
My Picture + 50	Picture	This expression takes the picture in My Picture, moves it 50 pixels to the right, and returns the resulting picture.
» Option Shift Back Slash »[People]Name	Pointer	This expression returns a pointer to the field called [People]Name.
File (1) L320	Pointer	This is a command that returns a pointer to the first file.

Variables

Data in 4th DIMENSION is stored in two fundamental ways. Fields store data permanently on disk; variables store data temporarily in memory. When you set up your database, you tell 4th DIMENSION the names and types of fields that you want to use. Variables are much the same—you also give them names and different types.

There are seven variable types, corresponding to each of the data types:

- string
- numeric
- date
- time
- Boolean
- picture
- pointer

You can display variables on the screen, enter data into them, and print them in reports. In these ways they act just like fields, and the same built-in controls are available when you create them:

- data formats
- data validation
- character filters
- choice lists
- enterable or not enterable

Variables can also do some special things:

- control buttons
- control thermometers, rulers, and dials
- control scrollable areas and pop-up menus
- display results of calculations that don't need to be saved

Creating Variables

You create variables simply by using them; you do not need to formally define them as you do with fields. For example, if you want a variable that will hold the current date plus 30 days, you tell 4th DIMENSION

My Date := **Current date** + 30 L335

and My Date would be created and hold the date you need. The program line reads, "My Date *gets* the current date plus 30 days." You could now use My Date wherever you needed to in your database. For example, you might need to store the date in a field:

My Field := My Date

By the way, notice in the first line that the words "Current date" are in boldface. This is because "Current date" is a command from the language. Commands within procedures are shown in boldface by 4th DIMENSION, and this manual uses the same convention in its examples. In the rest of the manual, when commands are mentioned within text they are displayed like this: Current date. See the section, "Visual Conventions Used in This Manual," in the Preface of this manual, for more information on these and other visual conventions.

Assigning Data to Variables

Data can be put into and copied out of variables. Putting data into a variable is called *assigning* the data to the variable and is done with the *assignment operator* (:=). (The assignment operator is also used to assign data to fields.)

The assignment operator is the primary way both to create a variable and to put data into it. You put the name of the variable that you want to create on the left side of the assignment operator. So, for example,

My Number := 3

creates the variable My Number and puts the number 3 into it. If My Number already exists, then the number 3 is simply put into it.

Of course, variables would not be very useful if you could not get data out of them. Once again, you use the assignment operator. If you needed to put the value of My Number in a field called Size, you would place My Number on the right side of the assignment operator:

```
Size := My Number
```

In this case, Size would be equal to 3. This example is rather simple, but it illustrates the fundamental way that data is transferred from one place to another by using the language.

Important: Be careful not to confuse the assignment operator (:=) with the comparison operator *equal* (=). Assignment and comparison are very different operations. See Chapter 10 for more information on the comparison operators.

Global and Local Variables

Most variables you create are *global variables*—variables available throughout your database. Anywhere global variables are needed, they can be used. There is one other type of variable: the *local variable*. A local variable, as its name implies, is local to a procedure—accessible only within the procedure in which it is created, and not accessible outside the procedure. Being local only to the procedure is formally referred to as being local in *scope*. Conversely, global variables are global in scope.

Why would you want to restrict a variable to work only within one procedure? There are two reasons:

- to avoid conflicts with the names of other variables
- for temporary use of data

When you are working in a database with many procedures and variables, you often find that you need to use a variable only within the procedure you are working on. You create and use such a variable as a local variable, without having to worry about whether you have already used the same variable name somewhere else.

The name of a local variable always starts with a dollar sign ($). This naming rule ensures that local variables are always identified with names different from the names of global variables. For example, $My Var is the name of a local variable, and My Var is the name of a different variable, a global variable.

Frequently, in a database, small pieces of information are needed from the user. The Request command can be used to obtain this information; it displays a dialog box with a message prompting the user for a response. When the user enters the response, the command returns the information the user entered. You usually do not need to keep this information in your procedures for very long. This is a perfect place to use a local variable.

Here is an example:

```
$Response := Request ("Please enter your ID:")
SEARCH ([People]ID = $Response)
```

This procedure simply asks the user to enter an ID; it puts the response into a local variable, $Response, and then searches for the ID that the user entered. When this procedure finishes, the $Response local variable is erased. This is fine, since the variable is needed only once, and only in this procedure.

Local variables are also used to pass data to and from procedures (parameter passing). See the section "Passing Parameters to Subroutines," in Chapter 6, for more information on using procedure parameters.

Layout Object Variables

In the Layout editor, the name given to each active object—buttons, check boxes, scrollable areas, thermometers, and so on—automatically creates a variable with the same name. For example, if you create a button named My Button, then a variable named My Button is also created. Note that this variable name is not the label for the button.

The variables allow you to control and monitor the objects. For example, when a button is clicked, its variable is set to 1; at all other times, it is 0. The variable associated with a thermometer or dial lets you read the current setting and change the setting. For example, if you drag a thermometer to a new setting, the value of the variable changes to reflect the new setting. Similarly, if your procedure changes the value of the variable, the thermometer is redrawn to show the new value.

Variables associated with layout objects are discussed in more detail in Chapter 4, "Layouts and Scripts."

System Variables

4th DIMENSION maintains a number of variables called *system variables*. These variables let you monitor many operations. The system variables are all global variables, accessible from anywhere in your database.

The most important system variable is the OK system variable. As its name implies, it essentially tells you if everything is OK. Did the record get saved? Did the importing operation complete? Did the user click the OK button? The OK system variable is set to 1 when a task was completed successfully, and to 0 when it was not.

All of the 4th DIMENSION system variables are discussed in detail in Appendix C.

USING PROCEDURES

CHAPTER 3

USING PROCEDURES

To make the commands, operators, and other parts of the language work, you put them in procedures. This chapter describes features common to all types of procedures. There are several kinds of procedures: layout procedures, file procedures, and global procedures. Scripts are also procedures of a special type.

A procedure is composed of *statements,* each statement consisting of one line in the procedure. A statement performs an action. For example, the following line is a statement that will add a new record to the [People] file:

ADD RECORD ([People])

A statement may be simple or complex. Although a statement is always one line, that one line can be as long as needed (up to 32,000 characters, which is probably enough for most tasks).

A procedure can be written as a flowchart or as a text listing. In either case, the execution inside procedures is fundamentally the same: line-by-line. It begins at the first line and works its way down to the last line. Making a procedure work is called *executing* or *running* the procedure.

Types of Procedures

There are five types of procedures in 4th DIMENSION:

- Scripts. A script is a short procedure associated with an active layout object. Scripts generally "manage" the object while the layout is displayed or printed.
- Layout procedures. A layout procedure belongs to a layout. It executes each time the layout is used, that is, when the layout is displayed or printed. You can use a layout procedure for data and object management, but it is generally simpler and more efficient to use a script for these purposes.
- File procedures. A file procedure belongs to a file. It executes when *any* layout belonging to the file is used for data entry. A file procedure is used to carry out data entry management that is common to *all* layouts belonging to a file. In practice, file procedures are rarely used.
- Global procedures. A global procedure is not directly associated with any specific part of the database. It is available for use throughout the database. A global procedure may also act as a function, returning a value when it executes.
- External procedures. External procedures are procedures that are created outside of 4th DIMENSION.

All procedures, except external procedures, are created by using 4th DIMENSION. Scripts, file procedures, and layout procedures are covered in more detail in Chapters 4 and 5. Global procedures are covered in Chapter 6.

An Example Procedure and Terminology

This section examines a procedure in detail in order to establish some of the terminology, concepts, and common aspects of procedures. Everything presented in this section is covered in greater detail in other parts of this manual.

All procedures are fundamentally the same—they start at the first line and work their way through each statement (line of instruction) until they reach the last line. Here is an example global procedure:

```
DEFAULT FILE ([People])      L134 • L131   ` Set the default file
SEARCH                       L194          ` Display the Search editor
If (Records in selection = 0)  L117 • L185 ` If no one was found...
  ADD RECORD                 L141          ` Let the user add a new record
End if                       L117          ` The end!
```

First, let's establish some terminology and features of the language. Each line in the example is a statement or *line of code*. Anything that you write by using the language is loosely referred to as *code*. Code is *executed*, or *run*—this simply means that 4th DIMENSION performs the task that the code specifies.

We will examine the first line in detail and then move on more quickly.

```
DEFAULT FILE ([People])      L134          ` Set the default file
```

The first element in the line, DEFAULT FILE, is a *command*. A command is part of the 4th DIMENSION language—it performs a task. In this case, DEFAULT FILE selects which file will be used (similar to choosing a file in the User environment). Notice that the command is in bold in the example; this is the way that commands are displayed by 4th DIMENSION, and the convention is used in all examples. When a command is referenced in text, it appears like this: DEFAULT FILE. Also notice that the command's name is in all uppercase letters. This is the naming convention used for 4th DIMENSION commands that do not return a value.

```
DEFAULT FILE ([People])      L131          ` Set the default file
```

The parentheses specify an *argument* to the DEFAULT FILE command. An argument (or *parameter*) is data that a command needs in order to complete its task. In this case, [People] is the name of a file. Files are always specified inside square brackets ([...]). You would say, "The People file is an argument to the DEFAULT FILE command."

DEFAULT FILE ([People]) Set the default file

Finally, there is a *comment* at the end of the line. A comment tells you (and anyone else who might read your code) what is happening in the code. It is indicated by the reverse apostrophe (`). Anything on a line following the comment mark will be ignored when the code runs. A comment can be put on a line by itself, but you should try to put comments to the right of the code, as in the example. Liberally sprinkle comments throughout your code; it makes it easier for you and others to read and understand.

The next line contains the SEARCH command:

SEARCH ` Display the Search editor

This command displays the Search editor. After the search is performed, there is a test to see if any records were found.

If (**Records in selection** = 0) ` If no one was found...

The If statement is a *control-of-flow* statement—a statement that controls the step-by-step execution of your procedure. The If statement performs a test, and if the test is TRUE, execution continues with the subsequent line(s). Notice that TRUE is written in all capital letters, because it refers to a *logical* or *Boolean* value. You will see the two Boolean values, TRUE and FALSE, presented this way throughout this manual.

Records in selection is a *function*—a command that returns a value. Here, Records in selection returns the number of records in the current selection. (You should already know what the current selection is; it is the group of records you are working on at any one time.) Notice that only the first letter of the function's name is capitalized. This is the naming convention used for 4th DIMENSION functions.

If the number of records is equal to 0 (in other words, if no one's record was found), then the following line is executed:

ADD RECORD ` Let the user add a new record

The ADD RECORD command displays a layout so that the user can add a new record. Notice that this line is indented. 4th DIMENSION formats your code automatically for you; this line is indented to show you that it is dependent on the control-of-flow statement (If), above.

End if ` The end!

The End if statement concludes the If statement's section of control. Whenever there is a control-of-flow statement, you need to have a corresponding statement telling the language where the control stops.

Be sure you feel comfortable with the concepts in this section. If they are all new, you may want to review them until they are clear to you.

Procedure Control

Regardless of the simplicity or complexity of a procedure, you will always use one or more of three types of programming structures. Programming structures control whether and in what order statements are executed within a procedure. The three types of structures are the sequence, the branch, and the loop.

There are statements in the language that control each of these structures. These statements are introduced in this section and are formally defined in Part II of this manual.

Sequence Structure

The sequence structure is a simple linear structure. A sequence is simply a series of statements that 4th DIMENSION executes one after the other, from first to last. For example,

DEFAULT FILE ([Employees]) L134
OUTPUT LAYOUT ("Listing") L138
ALL RECORDS L184
DISPLAY SELECTION L143

A one-line routine, frequently used for scripts, is the simplest case of a sequence structure. For example,

Last Name := **Uppercase** (Last Name) L331

Branching Structures

A branching structure allows procedures to test a condition and take alternative paths, depending on the result. The condition is a Boolean expression, an expression that evaluates to TRUE or FALSE. One branching structure is the If…Else…End if structure, which directs program flow along one of two paths. The other branching structure is the Case of…Else…End case structure, which directs program flow to one of many paths.

1. The If...Else...End if Structure

The If...Else...End if structure lets your procedure choose between two alternative actions, depending on whether a test (a Boolean expression) is TRUE or FALSE. When the Boolean expression is TRUE, the statements immediately following the test are executed. If the Boolean expression is FALSE, the statements following the Else statement are executed. The Else statement is optional; if you omit Else, execution continues with the first statement (if any) following the End if. For example,

```
CONFIRM ("Press OK to print.")      ` Sets OK to 1 or 0
If (OK = 1)                         ` Test the Boolean expression
  PRINT SELECTION                   ` Executes if TRUE
Else
  ALERT ("Printing canceled.")      ` Executes if FALSE
End if
```

See the section "Controlling Procedure Flow," in Chapter 10, for more information on the If...Else...End if structure.

2. The Case of...Else...End case Structure

Like the If...Else...End if structure, the Case of...Else...End case structure also lets your procedure choose between alternative actions. Unlike the If...Else...End if structure, the Case of...Else...End case structure can test an unlimited number of Boolean expressions and take action depending on which one is TRUE.

Each Boolean expression is prefaced by a colon (:). This combination (the colon and the Boolean expression) is called a *case*. For example, the following line is a case:

```
: (Number = 1)
```

Only the statements following the *first* TRUE case will be executed. If none of the cases is TRUE, none of the statements will be executed. You can include an Else statement after the last case; then, if all of the cases are FALSE, the statements following the Else will be executed.

This example tests a numeric variable and displays an alert box with a word in it:

```
Case of                                          L118
  : (Number = 1)                                 ` Test if the number is 1
    ALERT ("One.")                               ` If it is 1 display an alert
  : (Number = 2)                                 ` Test if the number is 2
    ALERT ("Two.")                               ` If it is 2 display an alert
  : (Number = 3)                                 ` Test if the number is 3
    ALERT ("Three.")                             ` If it is 3 display an alert
  Else                                           ` If it is not 1, 2, or 3 do a special alert
    ALERT ("It was not one, two, or three.")
End case                                         L118
```

For comparison, here is the If…Else…End if version of the same procedure:

```
If (Number = 1)                                  L117  ` Test if the number is 1
  ALERT ("One.")                                 L239  ` If it is 1 display an alert
Else
  If (Number = 2)                                      ` Test if the number is 2
    ALERT ("Two.")                                     ` If it is 2 display an alert
  Else
    If (Number = 3)                                    ` Test if the number is 3
      ALERT ("Three.")                                 ` If it is 3 display an alert
    Else                                               ` If it is not 1, 2, or 3 do a special alert
      ALERT("It was not one, two, or three.")
    End if
  End if
End if
```

Remember that with a Case of…Else…End case structure, only the *first* TRUE case is executed. Even if two or more cases are TRUE, only the statements following the first TRUE case will be executed.

See the section "Controlling Procedure Flow," in Chapter 10, for more information on the Case of…Else…End case structure.

Loop Structures

It is very common when writing procedures to find that you need a sequence of statements to repeat a number of times. To deal with this need, the language provides three loop structures: For, While, and Repeat. There are two ways that the loops are controlled: Either they loop until a condition is met, or they loop a specified number of times. Each loop structure can be used in either way, but While loops and Repeat loops are more appropriate for repeating until a condition is met, and For loops are more appropriate for looping a specified number of times.

1. The While Loop

A While loop executes the statements inside the loop as long as the Boolean expression is TRUE. It tests the Boolean expression at the beginning of the loop and does not enter the loop at all if the expression is FALSE.

It is common to *initialize* the value tested in the Boolean expression immediately before entering the While loop. Initializing the value means setting it to something appropriate, usually so that the Boolean expression will be TRUE and the loop will be entered.

A common task for a While loop is to add records to a database:

```
CONFIRM ("Add a new record?")     ` Does the user want to add a record?
While (OK = 1)                    ` Loop as long as the user wants to
  ADD RECORD                      ` Add a new record
End while                         ` The loop always ends with End while
```

In this example, the OK system variable is set by the CONFIRM command before the loop is entered. If the user clicks the OK button in the confirmation dialog box, the OK system variable is set to 1 and the loop is entered. Otherwise, the OK system variable is set to 0 and the loop is skipped entirely. Once the loop is entered, the ADD RECORD command keeps the loop going, because it sets the OK system variable to 1 when the user saves the record. When the user cancels (does not save) the last record, the OK system variable is set to 0 and the loop stops.

The Boolean expression must be set by something inside the loop or else the loop will continue forever. The following loop continues forever, because Never Stop is always TRUE.

```
Never Stop := True
While (Never Stop)
End while
```

> **Note:** If you find yourself in a situation like the one just described, where a procedure is executing uncontrolled, you can use the trace facilities to stop procedure execution and track down the problem. See Chapter 8 for more information on the trace facilities.

See the section "Controlling Procedure Flow," in Chapter 10, for more information on the While...End while loop.

2. The Repeat Loop L120

A Repeat loop is similar to a While loop except that it tests the Boolean expression after the loop rather than before. Thus, it always executes the loop once, whereas if the Boolean expression is initially FALSE, a While loop does not execute the loop at all.

The other difference with a Repeat loop is that the loop continues *until* the Boolean expression is TRUE. Compare the following example with the example for the While loop. Notice that the Boolean expression does not need to be initialized—there is no CONFIRM command to initialize the OK variable. Also notice that the test is the opposite: OK # 1 (OK is not equal to 1).

```
Repeat                L120
  ADD RECORD          L191
Until (OK # 1)
```

In the example, the loop is always executed at least once, and continues as long as the user keeps accepting the new records (and setting OK to 1).

See "Controlling Procedure Flow," in Chapter 10, for more information on the Repeat...Until loop.

3. The For Loop L121

The For loop is a loop controlled by a *counter*. The counter is a numeric variable that the For loop initializes to a value, and that is then incremented each time the loop is executed. When the counter passes a specified value, the loop stops. The first value used by the For statement is the counter variable, and the second and third values are the starting and terminating values, respectively. Unless specified, the increment is 1. The following loop starts at 1 and loops 100 times:

```
For ($i; 1; 100)      L121
End for
```

It is interesting to see how the While loop and Repeat loop would perform the same action. Here is the equivalent While loop:

```
$i := 1                          ` Initialize the counter
While ($i <= 100)                ` Loop 100 times
   $i := $i + 1                  ` Need to increment the counter
End while
```

And here is the equivalent Repeat loop:

```
$i := 1                          ` Initialize the counter
Repeat
   $i := $i + 1                  ` Need to increment the counter
Until ($i > 100)                 ` Loop 100 times
```

A typical use of the For loop is to move through each record in a selection of records. Here is the code that does this task:

```
For ($i; 1; Records in selection)
   ` Do something with the record here
   NEXT RECORD                   ` Move to the next record
End for                          ` Always end a For loop with End for
```

The loop initializes the counter, $i, to 1, and loops until $i is greater than the number of records in the selection. Notice that if there is only one record, the loop still executes once, and that if there are no records, the loop is not executed at all.

The For loop is faster than the While and Repeat loops because 4th DIMENSION tests the condition internally for each cycle of the loop and increments the counter. Therefore, use the For loop whenever possible.

See "Controlling Procedure Flow," in Chapter 10, for more information on the For…End for loop.

Trivia: In the example loops in this section, you may have noticed that the counters are represented by a local variable called $i. A local variable is appropriate in this case, since the variable is used only inside the loop and does not need to be available globally. The use of the letter *i* is historic. It originated with an ancient programming language called Fortran. Fortran used the letter *i* to indicate an integer numeric variable. It was very common to use this variable to control loops. The convention has remained in use, and you will see it used in this manual.

LAYOUTS AND SCRIPTS

CHAPTER 4

LAYOUTS AND SCRIPTS

Layouts are perhaps the most empowering aspect of 4th DIMENSION. With a simple unified system of tools, you can create layouts that show your data in any fashion you desire. Layouts intended for data entry have all of the superb Macintosh interface at their disposal. Layouts used for printing can create attractive reports.

4th DIMENSION provides built-in tools that allow you to manage layouts without using the language. These tools include data validation, data entry filters, data formatting, range checking, choice lists, default values, and buttons with associated actions. The language extends the built-in tools to allow you to control and monitor your layouts to an even greater degree. You don't need to give up any of the tools you are already familiar with—in fact, you should take advantage of those tools as much as you can. The language just extends the power those tools have already given you.

Three types of procedures are used to manage layouts:

- scripts
- layout procedures
- file procedures

Scripts are the most common of the procedure types—a layout may have a script for every active object. Scripts are used for data entry and reporting.

File and layout procedures work in basically the same way, with a few differences. Both types of procedures are activated when the layout is used. File procedures are used for data entry only; layout procedures are used for both data entry and reporting. A file procedure is associated with a particular file and applies to the use of any layout in the file; a layout procedure is associated with a particular layout.

File procedures are less commonly used than layout procedures. The rest of this manual refers only to layout procedures; any discussion of layout procedures used for data entry also applies to file procedures.

Controlling Layouts

The control of layouts is one of the most challenging aspects of customizing a database. When a global procedure is running, it is in control—the procedure does only what it was told to, and there are few outside events that can affect it. When you use layouts, suddenly there is a new source of events that your procedures must handle—the user.

Users like to do things in their own way; they want to click buttons, move from field to field, change to a different layout page, choose a menu item, and in general try to confuse your procedures.

In addition, there are special events that come from the use of the layout itself: When a layout is first displayed, it may need initializing; when a record is accepted, there is often a need for special processing; when a report is printing, different parts of the report need custom preparation.

How are all of these events managed? If your procedures had to monitor and respond to each of the many possible events, your procedures would never get anything done—they would be tied up just testing events. Fortunately, 4th DIMENSION manages events for you and informs your procedures only when it is appropriate.

The management of events is done through two primary means: scripts and the execution cycle. Scripts are object-oriented event managers—they respond to the events that happen to layout objects. The execution cycle is the "grand" event manager. It monitors the major events (and the minor ones) that happen to your layouts.

The rest of this chapter addresses the use of scripts. Chapter 5 covers the execution cycle and its effect on both layout procedures and scripts. You should read both chapters to fully understand how to use the 4th DIMENSION language to manage layouts.

Using Scripts

A script is a procedure that is associated with an active object in a layout. Its role is very specific to that object. A script should be used only to manage its associated object.

Scripts are executed when:

- performing data entry
- listing records on screen
- printing reports with layouts
- importing and exporting with layouts

Each active object can have one script, although a script is not mandatory.

During data entry, an active object can be used for entry (such as a field or variable), or as part of the interface (such as a button, pop-up menu, or thermometer). The script will be executed each time the object is activated. For example, the script attached to a button will be executed when the button is clicked; the script attached to a field will be executed when data is entered or modified.

When records are listed on screen and reports are printed with a layout, an object's script is executed as the object is displayed or printed. In these cases, the script usually affects the format or appearance of the object.

Scripts tend to be short—often only one line. For many databases, scripts may be the only type of procedure that you need to write. Once a script is attached to an active object, the object retains the actions specified by the script when it is cut, copied, or duplicated. Thus, you can build a library of active objects in your Scrapbook, and paste them into different layouts.

Scripts are executed according to the execution cycle, but in most cases you need not be aware of the execution cycle in order to use scripts. For more information on the execution cycle, see Chapter 5.

Scripts and Data Entry

For active objects such as fields and variables, scripts are used for operations like these:

- validating data as it is entered into the database
- assigning values to variables
- manipulating strings, such as concatenating fields or converting from lowercase to uppercase
- performing arithmetic and date calculations, such as computing totals, averages, and counts
- managing information in related files

When the data has been changed in a field or variable during data entry, the script for that field or variable is executed. The script is executed before the file and layout procedures.

Scripts and Interface Objects

Layouts can contain a wide variety of objects that interface with the user. Many of these objects can be completely controlled by built-in tools, such as automatic actions for buttons. Some of the objects are so flexible and can be used in so many ways that the language is required to monitor and control them. This section describes the interaction between the objects and the language.

Figure 4-1 shows the panel in the Object Definition dialog box where active objects are defined. (Refer to the *4th DIMENSION Design Reference* for more information regarding the creation and use of active objects.)

The name of the object, as shown in Figure 4-1, is also the name of a variable that 4th DIMENSION creates automatically. The variable is often used to monitor the status of the object.

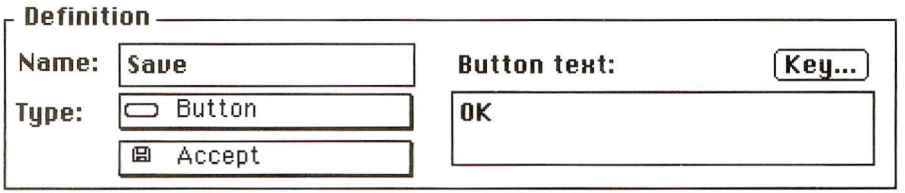


Figure 4-1
The object definition area of the Object Definition dialog box

There are 15 different types of objects. Figure 4-2 shows the Object Type pop-up menu that defines the type of active object.

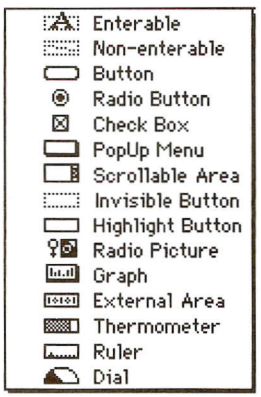


Figure 4-2
The Object Type pop-up menu

In all cases, scripts make taking care of the objects an easy task. You can use the combination of script and object to let the user communicate with your procedures. The sections that follow discuss how to use scripts with each type of object.

Buttons

The button is an extremely common interface object in layouts. Using scripts with buttons is very simple—when a button is clicked, its script is executed.

The name of the button is also the name of a variable—the variable is automatically created and associated with the button. When the button is clicked, the variable is set to 1. At all other times, the variable's value is 0. If the button has an automatic action, the action is performed after the script has executed.

Figure 4-3 shows the automatic actions that are available for buttons.

Take advantage of these actions; they can be used in conjunction with your scripts to make a flexible and useful user interface.

Figure 4-3
The Action pop-up menu for buttons

This list tells you the six types of buttons and the result of clicking on each of them. Remember that in all cases the script for the button is executed after the button's variable is set.

- Button. The button's variable is set to 1.
- Radio button. Radio buttons are usually in a group, where the first letter of each button's name is the same. When a radio button is clicked, the button's variable is set to 1 and the button is highlighted. All of the other buttons in the group are set to 0, and they are not highlighted.
- Check box. The check box's variable is set to 1 if the box is checked, and to 0 if the box is unchecked.
- Invisible button. The button's variable is set to 1. (The button does not get highlighted.)
- Highlight button. The button's variable is set to 1. (The button is highlighted when clicked.)
- Radio picture. Radio pictures are highlight buttons that work like radio buttons. All of the button names have the same first letter. When a radio picture is clicked, the picture button's variable is set to 1 and the button stays highlighted. All of the other buttons in the group are set to 0, and they are not highlighted.

For information on managing buttons with advanced techniques, see "Using Pointers to Buttons" and "Setting a Button Using a Pointer," in Chapter 9.

Scrollable Areas and Pop-up Menus

These two active objects operate the same way from the language's point of view. Both contain a list of items (an array of elements) and both allow the user to pick one of those items. Pop-up menus and scrollable areas can display four data types:

- string
- numeric
- date
- time

In addition, scrollable areas have the special capability of displaying pictures.

Because of their flexible nature, these objects require more interaction with the language than do other objects. There are three things you need to do in order to use these objects:

- Fill the scrollable area or pop-up menu with items.
- Respond when the user selects an item.
- Change the list if needed.

The list of items is an *array*. The array is specified as the name of the object when you create the object. (See Figure 4-1.) The array is created with one of the array commands described in the section "Managing Arrays," in Chapter 18.

For an example of using pop-up menus and scrollable areas, see "Displaying Arrays—An Example" and "Using Grouped Scrollable Areas," in Chapter 9.

Filling the Scrollable Area or Pop-up Menu

The array can be filled in any manner you like. It is often convenient to fill the array from a list created in the Design environment, or from a selection of records. The commands to create and fill arrays are described in the section "Managing Arrays," in Chapter 18. For example, to fill an array from a list you could use this line:

L360 **LIST TO ARRAY** ("Pop List"; My Pop)

Filling the array can be done at any time, but is typically done at one of three times: in a startup procedure (described in Chapter 6); in a global procedure before the layout is displayed; or in the Before phase of the execution cycle (described in Chapter 5). The best time to fill the array depends on how you are using the area: If the elements never change, do it in the startup procedure; if the elements do not change for each record, do it in the global procedure; if the elements change for each record, do it in the Before phase of the layout procedure or script.

Responding When the User Selects an Item

When the user chooses a pop-up menu item or selects an item in a scrollable area, the following events occur:

1. The name of the array is set to the item number (array element) selected.
2. A During phase is executed.
3. The script is run.

The name of the array is set to a positive number representing the item that the user selected. For example, consider the pop-up menu and scrollable area shown in Figure 4-4.

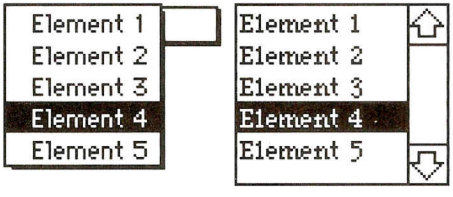


Figure 4-4
A pop-up menu and a scrollable area

If the array that each is displaying is called My Array and the user has selected the fourth element (as in the figure), the variable, My Array, is set to 4.

If the user clicks in an empty area of a scrollable area, or chooses no menu item, the array name is set to 0. You may need to test for this special case in order to perform the correct task or to change a pop-up menu to display an appropriate value.

If you want an item to always be selected, you can save the number of the last selected item and set it back to that item. For example, the following code sets the item back to its previous value when the user selects nothing:

```
Case of
  : (Before)
    Item := 1                          ` Initialize the item
    My Array := 1
  : (My Array = 0)                     ` If they select nothing...
    My Array := Item                   ` set the selected item back to previous
  Else
    Item := My Array                   ` Otherwise, save the selected item
End case
```

Use the object's script to see if an item has been selected. If the script is executing, it means the object was selected (in the During phase).

Changing the Items in the Scrollable Area or Pop-up Menu

When you change the array in any way, 4th DIMENSION automatically recognizes it and updates the object to reflect the change. Changes you might make include changing an element in the array, deleting and adding elements, or even deleting the whole array.

You can use the name of the array to "select" an item. To do this, just set the name of the array to the item that you want selected. The item will be highlighted in a scrollable area, or shown as the displayed menu item in a pop-up menu. You *must* use the assignment operator to set the name of the array; otherwise, the change will not be recognized.

Figure 4-5 shows a pop-up menu and a scrollable area with the second element selected.

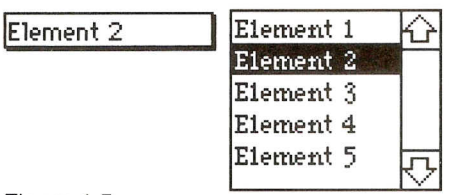


Figure 4-5
Setting the choice in a pop-up menu and scrollable area

If the pop-up menu and scrollable area shown in Figure 4-5 represent My Array, then the following line sets the pop-up menu and scrollable area as shown. Here is the line:

```
My Array := 2
```

Thermometers, Rulers, and Dials

Thermometers, rulers, and dials show a numeric value as a percentage of an area. The variable associated with the object changes to reflect the display, and, conversely, the display changes to reflect the value of the variable. In other words, if the user drags on the object, the value of the variable changes, and if the value of the variable is changed, the object is redrawn to reflect the new value.

The script for the object typically initializes the object in the Before phase. The During phase is executed each time the user changes the object. (See Chapter 5 for a discussion of the Before phase and the During phase.)

Graph Areas

Graph areas are completely controlled by the GRAPH command. For more information, see the section, "Graphing," in Chapter 13.

External Areas

External areas are areas controlled by procedures created in languages other than 4th DIMENSION's. There are no limits to what an external area can do; when it is selected, it takes complete control and manages all events until it returns control to 4th DIMENSION. When it returns control to 4th DIMENSION, a single During phase is generated. (See Chapter 5 for a discussion of the During phase.)

For more information on the creation and use of external areas and external procedures, contact ACIUS or ACI.

Scripts and Reporting

Scripts associated with fields and variables will be executed when the layout containing them is printed. The script will execute only when the object's layout area is printed, and then only if the "Only if modified" check box is unchecked.

Design Manual 203

For example, given the layout in Figure 4-6, notice the variable named vTotal in the B0Break area. B0

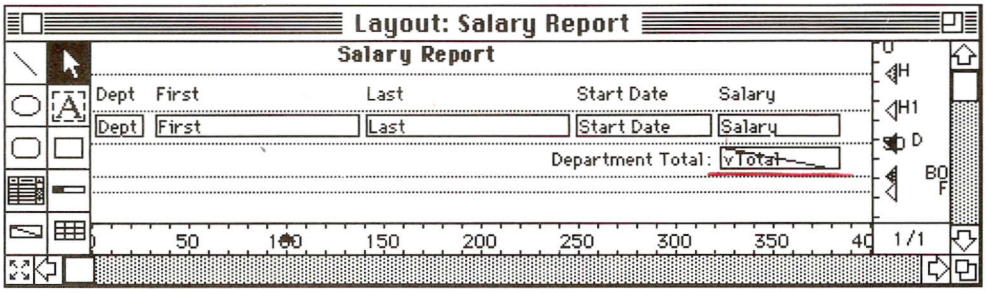


Figure 4-6
A report layout containing a variable with a script

The script associated with this variable will be printed at a level 0 break. In this layout, the script for vTotal is

vTotal := **Subtotal** (Salary) L160

file must be sorted to calculate "Subtotal" L160

This script assigns the subtotal for the salary field to the variable.

CHAPTER 5

THE LAYOUT EXECUTION CYCLE

THE LAYOUT EXECUTION CYCLE

Layout procedures and scripts are executed according to the layout execution cycle. The execution cycle determines when your procedures are executed. It is an aid to your procedures: It tells them what is happening to the layout. The execution cycle starts when *something happens* to a layout. Here are the execution cycle phases for a layout being used for data entry:

- Before phase—The layout is about to be displayed.
- During phase—Something in the layout just changed.
- After phase—The user accepted the record.

As you can see, each part of the execution cycle is referred to as a *phase*. The procedures of your layouts are executed at each phase. The phases called *Before, During,* and *After* are the most important ones, and you will probably use them the most.

There are also execution cycle phases for printing:

- In Header phase—The page header is about to be printed.
- Before phase—A record is about to be printed.
- During phase—A record is being printed.
- In Break phase—A break area is about to be printed.
- In Footer phase—The page footer is about to be printed.

The execution cycle affects all procedures associated with layouts. Although the execution cycle is integral to the use of layouts, in many cases it can be ignored—it is there only when you need it. By using the phases of the execution cycle, you can control when various types of data and object management take place.

When you are building simple databases that use scripts for object management, you may not need to be aware of the execution cycle and its phases. Typically, a script does not test for the phase of the execution cycle, because it is simply executed whenever the object with which it is associated is activated. In many cases, this rule ensures that scripts execute at the appropriate time. However, you can explicitly test for the phase of the execution cycle in a script, if you need to restrict data processing to a particular phase.

Monitoring the Execution Cycle Phases

Each phase of the execution cycle has a function in the language. Each function returns TRUE when its phase is executing, and the functions of the other execution cycle phases return FALSE at that time.

This list shows the name of each function, with a short description of the phase it is associated with. As with all built-in functions and commands, the functions are more completely described in the section "Monitoring the Layout Execution Cycle," in Chapter 13. Page 178

- Before—A Before phase occurs for each record before the layout is displayed or printed.
- During—A During phase occurs during data entry each time data has been changed or an active object is used. The During phase occurs during printing each time the Detail area of a layout is printed.
- After—An After phase occurs only for data entry, after a record has been accepted.
- Before and During—This special phase occurs when records are listed on-screen.
- In header—An In Header phase occurs when a layout Header area is about to be printed or displayed on-screen. One header is printed for each page and multiple headers may be printed for each break level. You can test for the first header, with the Before selection function. You can test what break level is printing, with the Level function.
- In break—An In Break phase occurs during break processing, when a layout break area is about to be printed. You can test what break level is printing, with the Level function.
- In footer—An In Footer phase occurs when a layout footer area is about to be printed. One footer is printed for each page. You can test for the last footer, with the End selection function.

Procedures typically use a Case of structure to test for the phases of the execution cycle. For example, here is a typical structure for a layout procedure used for data entry:

```
Case of
  :(Before)
    ` Do initialization here
  :(During)
    ` Monitor the data entry process here
  :(After)
    ` Do any "clean-up" required when a record is accepted
End case
```

General Rules for the Execution Cycle

Here are some general rules concerning the execution cycle:

- For each phase of the execution cycle, the procedures that exist are executed in the following order:

  1. The script(s) are executed.
  2. The file procedure is executed (only for data entry).
  3. The layout procedure is executed.

- In the Before and After phases, all scripts are executed, except scripts that are designated as "Only if modified."
- The scripts for objects used for data entry are executed in the data entry order.
- The scripts for objects not used for data entry are executed in an undefined order. Your scripts should not rely on the order in which they are executed.
- During data entry, only the script for the activated object is executed. For example, when a field is modified, only that field's script will be executed.
- When records are being printed or displayed on-screen, only the scripts of the objects being printed or displayed are executed, and then only if the script is not designated as "Only if modified." For example, a variable in a header will have its script executed only when the header is displayed (the In Header phase).

The Execution Cycles

The following sections describe the execution cycle for each of the different uses of a layout.

For Data Entry

When a user is entering data into a layout, there are three phases of the execution cycle:

- Before—Before the layout is displayed.
- During—During data entry each time data has been changed or an active object is used.
- After—After the record has been accepted.

The Before phase occurs before the layout is displayed. It is used, for example, to test whether the record is a new record or an existing record. (A layout may be used both for adding new records or modifying existing records.) The Before phase is also used to initialize variables, display default values in fields, or assign a sequence number to the record.

The During phase occurs when the user does something to the layout. Some of the user actions that can cause a During phase are

- changing data in a field or variable and leaving the field or variable
- clicking a button
- manipulating a thermometer, ruler, or dial
- choosing an item from a menu
- choosing an item from a pop-up menu
- selecting an item from a scrollable area
- pressing an assigned key combination
- clicking an external area

The During phase is typically used for tasks such as data validation, formatting entered data, managing data in related files, monitoring and responding to the selection of controls, and other tasks specific to layout objects.

The After phase takes place after the user has accepted a record. It will not happen if the user cancels the record.

For Files in Included Layouts

When 4th DIMENSION is displaying a list of records in an included layout, there are three phases of the execution cycle:

- Before—When a new record is added to the included file.
- During—For each record that is displayed. A During phase occurs when the records are first displayed, and during modification of the records. A parent record During phase is executed after the included layout's During phase.
- After—For each record that is changed and accepted.

For each phase of the execution cycle, the scripts, the file procedure, and the layout procedure are executed.

For Subfiles in Included Layouts

When 4th DIMENSION is displaying a list of subrecords in an included layout, there are three phases of the execution cycle:

- Before—For each subrecord, before the Before phase of the parent layout, and when a new subrecord is added.
- During—For each subrecord that is displayed. A During phase occurs when each subrecord is first displayed, and during modification of a subrecord. A parent record During phase is executed after the included layout's During phase.
- After—For each subrecord before the parent record's After phase.

There is no After phase when a full-page layout for a subrecord is accepted.

For each phase of the execution cycle, the scripts, the file procedure, and the layout procedure are executed.

For User Environment List of Records

When 4th DIMENSION displays the list of records on screen in the output layout, there are four phases of the execution cycle:

- In Header—Before the Header area is displayed. This phase can be used to generate a title or summary information for the records.
- Before and During—Before and During are TRUE simultaneously. This phase occurs once for each record that is displayed.
- During—Occurs only when a field in a record is modified while in the "Enter in List" mode.
- After—Occurs only when a record has been modified and accepted while in the "Enter in List" mode.

For each phase of the execution cycle, the layout procedure is executed. For each layout area, such as the Header and Detail areas, only the scripts for layout objects in that area are executed, and then only if the "Only if modified" check box is unchecked.

For MODIFY SELECTION and DISPLAY SELECTION

When 4th DIMENSION displays the list of records in the output layout for MODIFY SELECTION or DISPLAY SELECTION, there are four phases of the execution cycle for the output layout. If a record is double-clicked, the input layout procedure for that record is executed according to the rules for data entry. The phases of the execution cycle are:

- Before—Occurs once, before any records are displayed.
- In Header—Occurs once, before the header area is displayed. This phase can be used to generate a title or summary information for the records.
- Before and During—Before and During are TRUE simultaneously. This phase occurs once for each record that is displayed.
- During—Before is FALSE and During is TRUE when a menu is selected, a button is clicked, or a record is double-clicked.

For each phase of the execution cycle, the layout procedure is executed. For each layout area, such as the header and detail areas, only the scripts for layout objects in that area are executed, and then only if the "Only if modified" check box is unchecked.

For Export Through Layouts

When you are exporting records through a layout, there is one phase of the execution cycle for each record:

- Before—Before each record is exported.

The scripts, the file procedure, and the layout procedure are executed for the execution cycle.

You can use this execution cycle to perform processing on the data before it is exported; for example, concatenating fields or padding data for fixed-length fields.

For Import Through Layouts

When you are importing records through a layout, there is one phase of the execution cycle for each record:

- After—After the record is imported and before the record is saved.

The scripts, the file procedure, and the layout procedure are executed for the execution cycle.

You can use this execution cycle to perform processing on the data before it is saved; for example, stripping spaces from fixed-length data.

For Layout Reports

When you are printing a report, there are five phases of the execution cycle:

- In Header—A layout Header area or break Header area is about to be printed. There is one header for each page printed. There may be many break headers. You can test which break header is printing with the Level function. You can test for the first header with the Before selection function.
- Before—Occurs once for each record. The layout Detail area is about to be printed.
- During—Following each Before phase.
- In Break—During break processing when a layout Break area is about to be printed. You can test what break level is printing with the Level function.
- In Footer—A layout Footer area is about to be printed. There is one footer for each page printed. You can test for the last footer with the End selection function.

Layout procedures are executed for each execution cycle. When a layout area is printed, only the scripts for layout objects in the area are executed, and then only if the "Only if modified" check box is unchecked. For example, if a variable was placed in the Header area of a report, its script would be executed only when each header was printed.

Whenever possible, it's recommended that you use scripts to control processing of your reports. By simply placing the objects in the various print areas of a layout, you ensure that their scripts will be executed at the appropriate times.

CHAPTER 6

GLOBAL PROCEDURES

GLOBAL PROCEDURES

Global procedures are aptly named. Whereas a layout procedure or script is intimately associated with a specific layout or object, a global procedure is available anywhere—it is not specifically associated with anything. A global procedure can have one of two very different roles:

- master procedure—acting as a traffic cop for your customized database
- subroutine—acting as a servant to other procedures

A *master procedure* is a global procedure called from a custom menu. It acts as a traffic cop, directing the flow of your application. The master procedure takes control, branching where needed, presenting layouts, generating reports, and otherwise managing your database.

The other type of global procedure can be thought of as a servant—being asked to perform tasks by other procedures. This type of procedure is called a *subroutine*.

The sections that follow describe each of these types of global procedures. This chapter also covers startup procedures.

Master Procedures—Procedures Called From Menus

A master procedure is called from a custom menu item. You assign the procedure to the menu item by using the Menu editor. (See the *4th DIMENSION Design Reference* for more information on the Menu editor.) When the menu item is chosen, the procedure executes. This process is one of the major aspects of customizing a database. By creating custom menus with master procedures that perform specific actions, you personalize your database.

Custom menu items can cause one or more activities to take place. For example, a menu item for entering records might call a procedure that does two tasks: displaying the appropriate input layout, and calling the ADD RECORD command until the user cancels the data entry activity.

Automating sequences of activities is a very powerful capability of the programming language. Using custom menus, you can automate sequences of tasks that would be carried out manually in the User environment. With custom menus, you provide more guidance to the users of the database.

Chapter 7 gives examples of using master procedures called from menus.

Subroutines—Procedures Called From Procedures

When you create a global procedure, it becomes part of the language for the database in which you create it. You can then call the global procedure just like you can call 4th DIMENSION's built-in commands. A global procedure used in this way is called a *subroutine*.

There are four reasons to use subroutines:

- to reduce the amount of repetitious coding you must do
- to clarify your procedures
- to ease changes to your procedures
- to modularize your code

For example, let's say you have a database of customers. As you customize the database, you find that there are some tasks that you perform repeatedly. One of those tasks might be to find a customer and modify their record. The code to do this task might look like this:

```
DEFAULT FILE ([Customers])        L134    ` Set the default file
SEARCH BY LAYOUT ("Find Cust")    L193    ` Search for a customer
INPUT LAYOUT ("Input Cust")       L137    ` Select the input layout
MODIFY RECORD                     L141    ` Modify the customer's record
```

If you do not use subroutines, you will have to write the code each time you want to modify a customer's record. If there are ten places in your custom database where you need to do this task, you will have to write the code ten times. If you use subroutines, you will only have to write it once. This is the first advantage of subroutines: to reduce the amount of coding you must do.

If the code above was a procedure called *Modify Cust,* you would execute it simply by using the name of the procedure in another procedure. For example, to modify a customer's record and then print the record, you would write this procedure:

```
Modify Cust
PRINT SELECTION ([Customers])     L163
```

This capability can dramatically simplify your procedures. In the example, you do not need to know *how* the *Modify Cust* procedure works, just *what* it does. This is the second reason for using subroutines: to clarify your procedures.

If you find that you need to change the method you use to find customers in the example database, you will need to change only one procedure, not ten. This is the next reason to use subroutines: to ease changes to your procedures.

Using subroutines, you *modularize* your code. This simply means breaking up your code into *modules* (subroutines), each of which performs a logical task. Consider the following code from a checking account database:

```
Find Cleared        ` Find the cleared checks
Reconcile           ` Reconcile the account
Check Report        ` Print a checkbook report
```

Even for someone who doesn't know the database, it is quite clear what this code does. It is not necessary to examine each subroutine. Each subroutine might be many lines long and perform some complex operations, but here it is only important that it performs its task.

It is recommended that you break up your code into logical tasks, or modules, whenever possible. If you find that a procedure is more than about 20 lines long, you should consider breaking it into modules. This is the last reason for using subroutines: to modularize your code.

Passing Parameters to Subroutines

You'll often find that you need to pass data to your subroutines. This is easily done with parameters. Parameters (or arguments) are pieces of data that a subroutine needs in order to perform its task. The terms *parameter* and *argument* are used interchangeably throughout this manual.

Parameters are also passed to built-in 4th DIMENSION commands. In this example, the string "Hello" is an argument to the ALERT command:

```
ALERT ("Hello")
```

Parameters are passed to subroutines the same way. For example, if a procedure called *My Proc* accepted three parameters, a call to the subroutine might look like this:

```
My Proc (This; That; The Other)
```

The parameters are separated by semicolons.

In the subroutine, the value of each parameter is automatically copied into sequentially numbered local variables: $1, $2, $3, and so on. The numbering of the local variables represents the order of the parameters. The local variables *are not* the actual parameters; they simply *contain the values* of the parameters. Since they are local variables, they are available only within the subroutine, and are cleared at the end of the subroutine.

Changing the value of a local variable *does not* change the value of the parameter. For example, the following subroutine, called *Cat*, concatenates two strings and displays the result in an alert box.

```
$1 := $1 + $2          ` Concatenate the two strings
ALERT ($1)             ` Display the new string
```

Notice that $1 is changed by the first statement. The following lines pass two parameters to *Cat*:

```
My Var := "You may "
Cat (My Var; "ask yourself")
```

These lines display a dialog box with the words "You may ask yourself" in it. The local variable, $1, receives the string "You may ". It changes $1 to the string "You may ask yourself", but does not change the variable My Var.

Note: You can refer to parameters within a subroutine by using curly braces. For example, ${i} refers to the same parameter as $1 if i contains 1.

Subroutines as Functions

Data can be returned from subroutines. A subroutine that returns a value is called a *function*. Commands that return a value are also called functions.

For example, the following line is a statement that uses the built-in function, Length, to return the length of a string. The statement puts the value returned by Length in a variable called My Length. Here is the statement:

```
My Length := Length ("How did I get here?")
```

Any subroutine can return a value. The value to be returned is put into the local variable $0. For example, the following function, called *Up4*, returns a string with the first four characters of the string passed to it in uppercase:

```
$0 := Uppercase (Substring ($1; 1; 4)) + Substring ($1; 5)
```

If you execute the next line, the string "ONCE in a lifetime" is put into the variable Byrne. Here is the line that uses the *Up4* function:

```
Byrne := Up4 ("once in a lifetime")
```

Startup Procedures

You can specify that a global procedure be executed when you open a database. If you name a global procedure *Startup*, it will be executed when the database is opened.

Using the Password Access editor, you can assign different startup procedures to each user. The user startup procedure specified for a user is executed after the procedure named *Startup*.

DATABASE APPLICATIONS

CHAPTER 7

DATABASE APPLICATIONS

An application is a database designed to fulfill a specific need. An application has a user interface designed specifically to ease the use of the application. The tasks that an application performs are limited to those appropriate for the application. 4th DIMENSION makes the creation of applications smoother and more accessible than does traditional programming.

4th DIMENSION can be used to create applications such as

- an invoice system
- an inventory control system
- an accounting system
- a payroll system
- a personnel system
- a customer tracking system

It is possible that a single application could even contain all of these systems.

Applications like these are traditional uses of databases. In addition, the tools in 4th DIMENSION allow you to create innovative applications, such as

- a document tracking system
- a graphic image management system
- a catalog publishing application
- a serial device control and monitoring system
- an electronic mail system (E-mail)
- a multi-user scheduling system

An application typically starts as a database used in the User environment. The database "evolves" into an application as it is customized. What differentiates an application is that the processes required to manage the database are hidden from the user. Database processing is automated, and users use menus to perform specific tasks.

When you use a database in the User environment, you must know the steps to be taken in order to achieve a result. In an application, those steps are automated—described by using the 4th DIMENSION language. By creating an application, you expend a little bit more effort up front so that you can save a lot of effort in the long run.

The sections that follow give examples showing how the language can automate the use of a database.

A Custom Menu Example

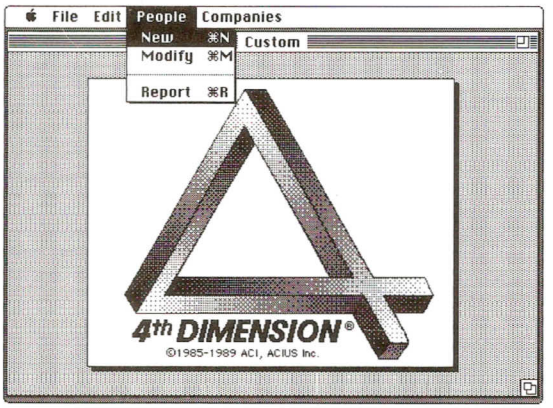
Custom menus are the primary interface in an application. Creating custom menus is very simple—you connect procedures to each menu item by using the Menu editor. Custom menus make a database easier to learn and use.

This section gives a very simple example of a custom menu and what happens when the user chooses a menu item. Although the example is simple, it should be apparent that the database is easier to use and learn. Instead of the "generic" tools and menu items in the User environment, the user sees only things that are appropriate to his or her needs.

In the example, the left column is what the user sees. The right column explains what is going on behind the scenes, and the design work that made it happen.

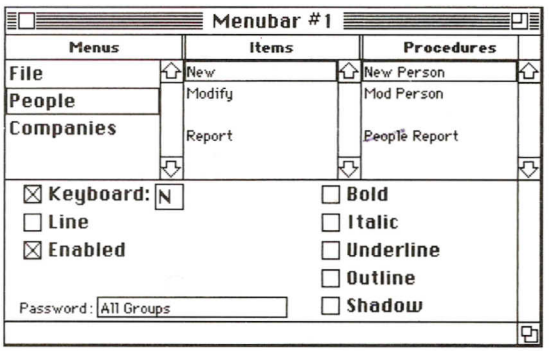
The User's Perspective

The user chooses a custom menu item called New to add a new person to the database.

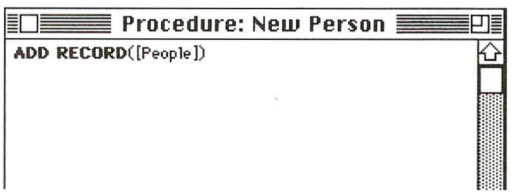


Behind the Scenes

The menu bar was created in the Design environment, using the Menu editor.



The menu item, New, has a global procedure named *New Person* associated with it. This procedure was created in the Design environment, using the Procedure editor.

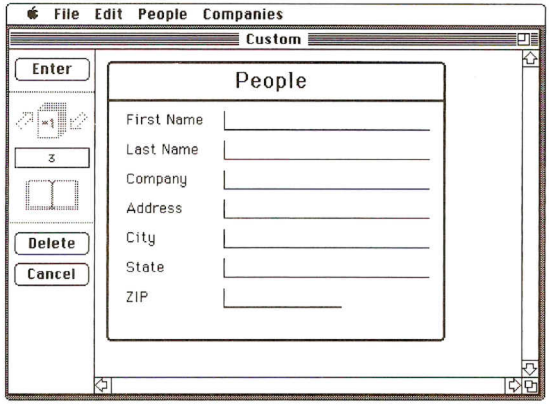


The *New Person* procedure executes:

ADD RECORD ([People])

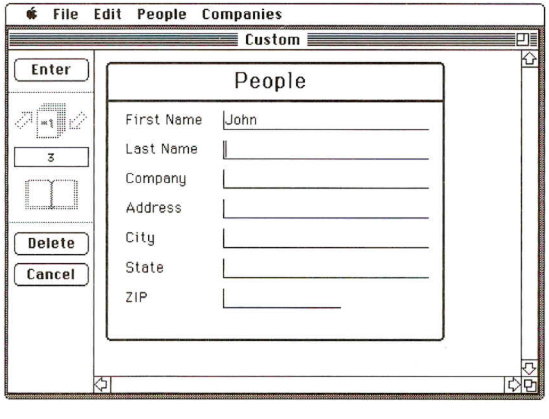
Saved to disk "Language Ch7f"

The input layout for the People file is displayed.



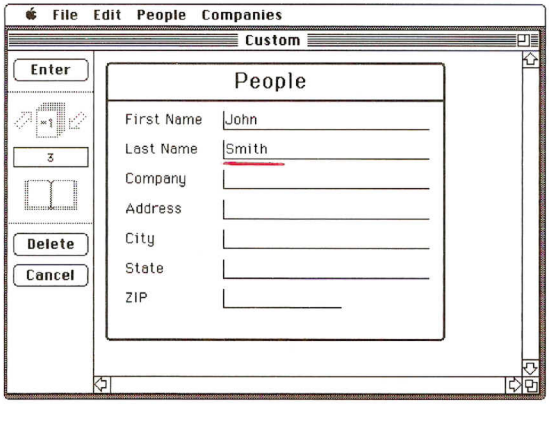
The ADD RECORD command acts just like the New Record menu item in the User environment. It displays the input layout to the user, so that he or she can add a new record.

The user enters the person's first name and tabs to the next field.

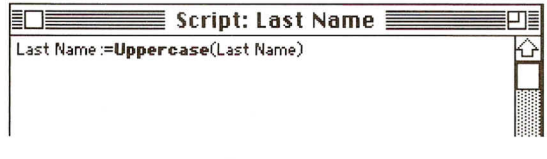


There is no script for the First Name field, so nothing executes.

The user enters the person's last name and tabs to the next field.



There is a script for the Last Name field. This script was created in the Design environment, using the Procedure editor.

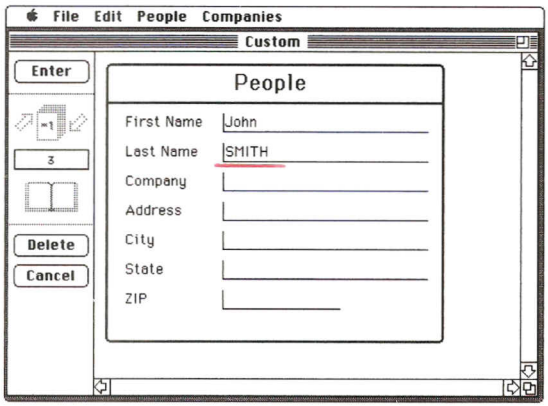


The script executes:

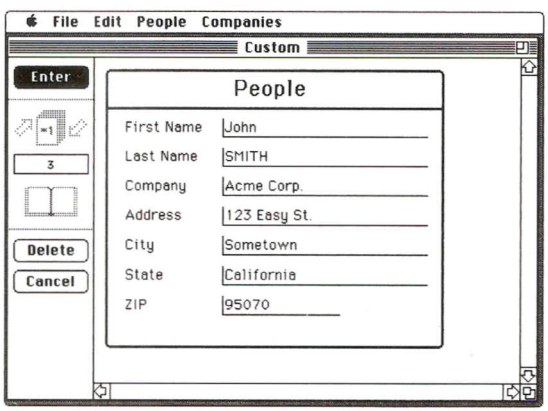
Last Name := **Uppercase** (Last Name)

This line converts the Last Name field to uppercase characters.

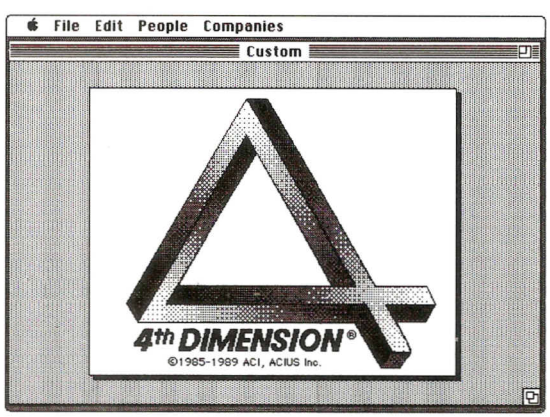
The user sees that the last name has been converted to uppercase.



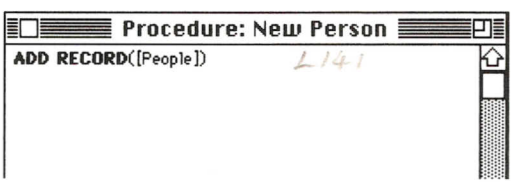
The user finishes entering the record and clicks the Enter button.



The user is returned to the menu bar.



The ADD RECORD command saves the new record and returns to the *New Person* procedure.



Since there are no more statements to execute, the *New Person* procedure stops executing, and control returns to the menu bar.

Saved to

Comparing an Application With the User Environment

Let's compare the way a task is performed in the User environment and the way the same task is performed by using the language. The task is a common one: to find a group of records, sort them, and print a report.

The first comparison uses the built-in editors of 4th DIMENSION in both the User environment and the language. In this case, the language partially automates the process. The column on the left shows the actions that the user takes in the User environment. The column on the right shows the same tasks being performed in an application.

Notice that although both methods perform the same task, the steps on the right are automated by using the language.

Using a Database in the User Environment

The user chooses Search Editor from the Select menu.

Using an Application With the Built-in Editors

The user chooses a custom menu item that starts a procedure.

Even at this point, using an application is easier for the users. They did not need to know that searching is the first step, and the menu item, Report, is very specific to their needs.

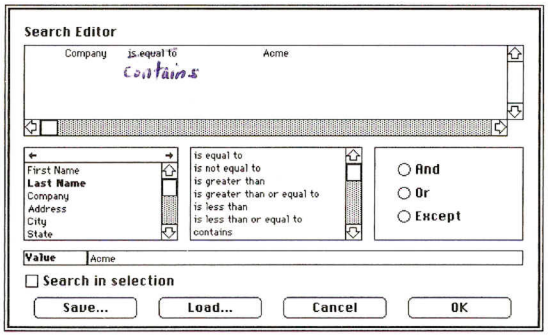
A procedure called ~~*My Report*~~ People is attached to the menu item. It looks like this:

SEARCH ([People]) L194
SORT SELECTION ([People]) L205
OUTPUT LAYOUT ([People]; "Report") L138
PRINT SELECTION ([People]) L163

The first line is executed:

SEARCH ([People]) L194

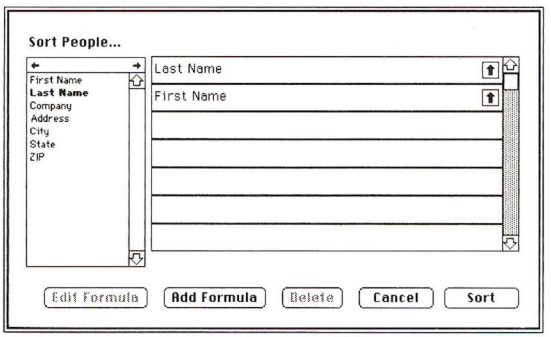
The Search editor is displayed.



The user enters the search criteria and clicks OK. The search is performed.

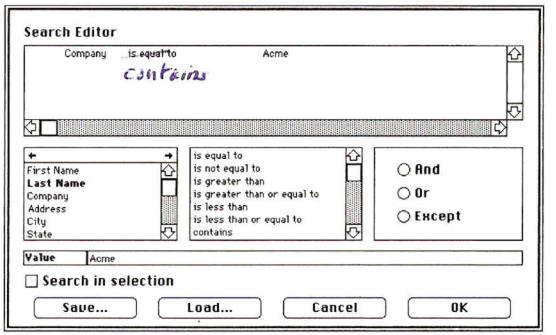
The user chooses Sort Selection from the Select menu.

The Sort dialog box is displayed.



The user enters the sort criteria and clicks Sort. The sort is performed.

The Search editor is displayed.



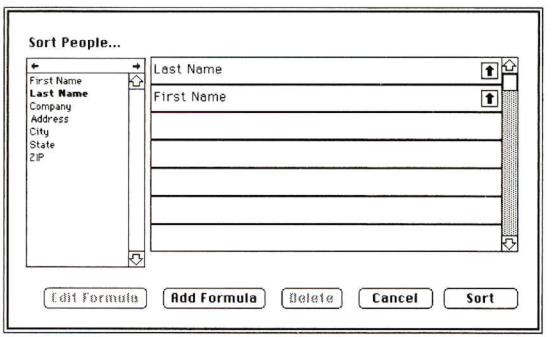
The user enters the search criteria and clicks OK. The search is performed.

The second line of the *My Report* procedure is executed:

SORT SELECTION ([People]) L205

Notice that the user did not need to know that sorting was the next step.

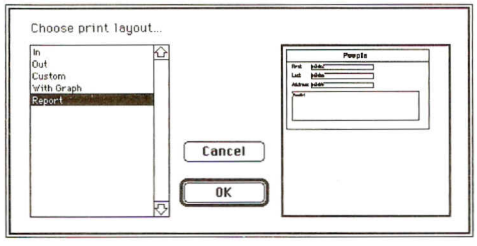
The Sort dialog box is displayed.



The user enters the sort criteria and clicks Sort. The sort is performed.

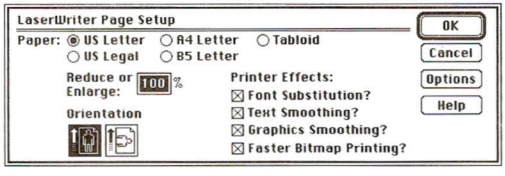
The user chooses Print from the File menu.

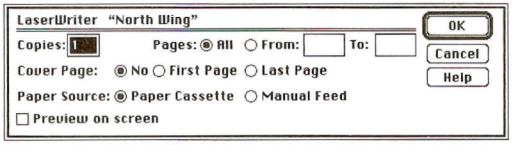
The "Choose print layout" dialog box is displayed.



Users need to know which layout to choose. They choose the layout and press Return.

The printer dialog boxes are displayed.





The user chooses the settings, and the report is printed.

The third line of the *My Report* procedure is executed:

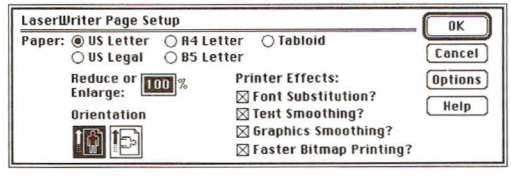
OUTPUT LAYOUT ([People]; "Report")

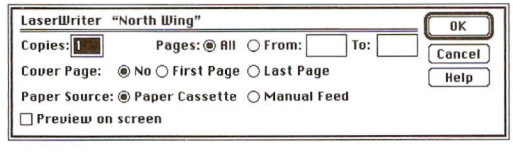
Once again, the user did not need to know what to do next. The path was already defined.

The final line of the *My Report* procedure is executed:

PRINT SELECTION ([People])

The printer dialog boxes are displayed.





The user chooses the settings, and the report is printed.

Further Automating the Application

The same commands that were used in the preceding comparison can be used to further automate the database.

In the next comparison, the application from the previous example that uses the built-in editors is in the left column. The right column shows the language completely automating the process. Notice that the user only needs to know to select the Report menu item in order to generate the report. Also notice how the same commands are used to more completely describe the actions that need to be taken.

Using an Application With the Built-in Editors

The user chooses a custom menu item that starts a procedure.

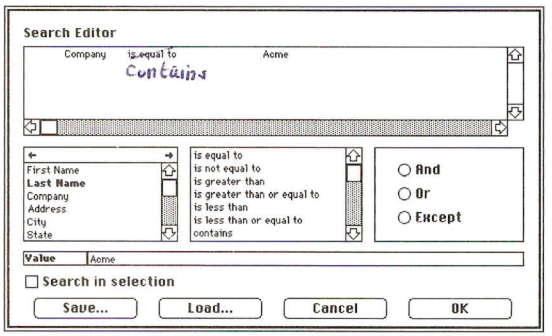
A procedure called ~~My~~ People *Report* is attached to the menu item. It looks like this:

SEARCH ([People]) L194
SORT SELECTION ([People]) L205
OUTPUT LAYOUT ([People]; "Report") L138
PRINT SELECTION ([People]) L163

The first line is executed:

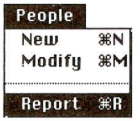
SEARCH ([People]) L194

The Search editor is displayed.



Using an Application With Complete Automation

The user chooses a custom menu item that starts a procedure.



A procedure called ~~My~~ People *Report2* is attached to the menu item. It looks like this:

SEARCH ([People]; [People]Company = "Acme")
SORT SELECTION ([People]; "Acme@"
[People]Last Name; > ;
[People]First Name; >)
OUTPUT LAYOUT ([People]; "Report") ! Contains - (Begins with) a wild card
PRINT SELECTION ([People]; *)

The first line is executed:

SEARCH ([People]; [People]Company = "Acme") @ Contains

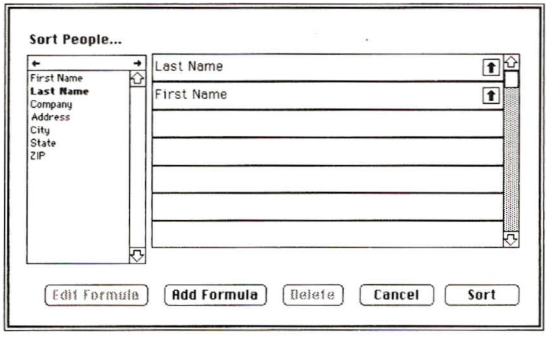
The Search editor is *not* displayed. Instead, the search is specified *and* performed by the SEARCH command. The user does not need to do anything.

The user enters the search criteria, clicks OK, and the search is performed.

The second line of the *My Report* procedure is executed:

SORT SELECTION ([People])

The Sort dialog box is displayed.

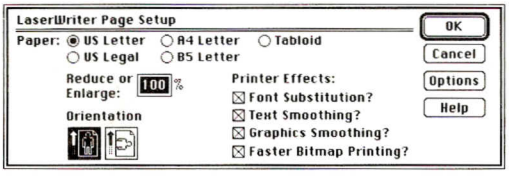


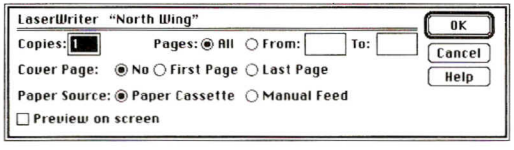
The user enters the sort criteria, clicks Sort, and the sort is performed.

The final lines of the *My Report* procedure are executed:

OUTPUT LAYOUT ([People]; "Report")
PRINT SELECTION ([People])

The printer dialog boxes are displayed.





The user chooses the settings, and the report is printed.

The second line of the *My Report2* procedure is executed:

SORT SELECTION ([People];
[People]Last Name; > ;
[People]First Name; >)

The Sort dialog box is not displayed, and the sort is immediately performed. Once again, no user actions are required.

The final lines of the *My Report2* procedure are executed:

OUTPUT LAYOUT ([People]; "Report")
PRINT SELECTION ([People]; *)

The printer dialog boxes are *not* displayed. The PRINT SELECTION command accepts an optional asterisk (*) parameter that instructs the command to use the print settings that were in effect when the report layout was created. The report is printed.

User Environment Menus and Equivalent Commands

As you've seen in the examples, there are commands in the language that perform the same actions as the User environment menu items. These commands provide an easy means of customizing a database.

Each of the menu items either performs an action or presents an editor or dialog box. Using the commands, the editors and actions can be "strung together" into a custom sequence, allowing repetitive tasks to be automated. The commands can also be used to actually specify the action, such as searching, without presenting the editor and without any user intervention.

Table 7-1 lists User environment menu items and the corresponding commands, and gives an example of each command in use.

Table 7-1
User environment menus with their equivalent commands

Menu Item	Page	Command	Example
Apply Formula	186	APPLY TO SELECTION	**APPLY TO SELECTION** ([People]; *Format Name*)
Choose File/Layout	134	DEFAULT FILE	**DEFAULT FILE** ([People])
	137	INPUT LAYOUT	**INPUT LAYOUT** ([People]; "In")
	138	OUTPUT LAYOUT	**OUTPUT LAYOUT** ([People]; "Out")
Edit ASCII Map	314	USE ASCII MAP	**USE ASCII MAP** ("Map Name")
Export	212	EXPORT TEXT	**EXPORT TEXT** ([People]; "")
Graph	176	GRAPH FILE	**GRAPH FILE** ([People])
Import	213	IMPORT TEXT	**IMPORT TEXT** ([People]; "")
Labels	170	PRINT LABEL	**PRINT LABEL** ([People]; "")
Modify Record	141	MODIFY RECORD	**MODIFY RECORD** ([People])
New Record	141	ADD RECORD	**ADD RECORD** ([People])
Print	163	PRINT SELECTION	**PRINT SELECTION** ([People])
Quick Report	162	REPORT	**REPORT** ([People]; "")
Search by Formula	200	SEARCH BY FORMULA	**SEARCH BY FORMULA** ([People]Age = 20)
Search by Layout	193	SEARCH BY LAYOUT	**SEARCH BY LAYOUT** ([People]; "Find")
Search Editor	194	SEARCH	**SEARCH** ([People])
Show All	184	ALL RECORDS	**ALL RECORDS** ([People])
	143	MODIFY SELECTION	**MODIFY SELECTION** ([People])
Sort File	206	SORT FILE	**SORT FILE** ([People])
Sort Selection	205	SORT SELECTION	**SORT SELECTION** ([People])

DEBUGGING

CHAPTER 8

DEBUGGING

When developing your procedures, it is important that you find and fix errors in your procedures.

Different types of errors are possible when you are using the language. There are three types of errors you can make:

- A typing error—This type of error is caught by the Procedure editor and marked with bullets (•). See the *4th DIMENSION Design Reference* for more information on the Procedure editors.
- A syntax error—This type of error is caught when you execute the procedure. The Syntax Error window is displayed when a syntax error occurs.
- An error in design or logic—This is generally the most difficult type of error to find. You use the Debugger to track down the error.

This chapter describes the tools that you use to track down syntax errors and errors in design or logic: the Syntax Error window and the Debugger.

The Syntax Error Window

The Syntax Error window is displayed when procedure execution is halted. Procedure execution can be halted for either of two reasons:

Option key

- 4th DIMENSION halts execution because there is a syntax error that prevents further procedure execution.
- You generate a *user interrupt* by holding down the Option key down and clicking the mouse (Option-clicking) while a procedure is executing.

The Syntax Error window is shown in Figure 8-1.

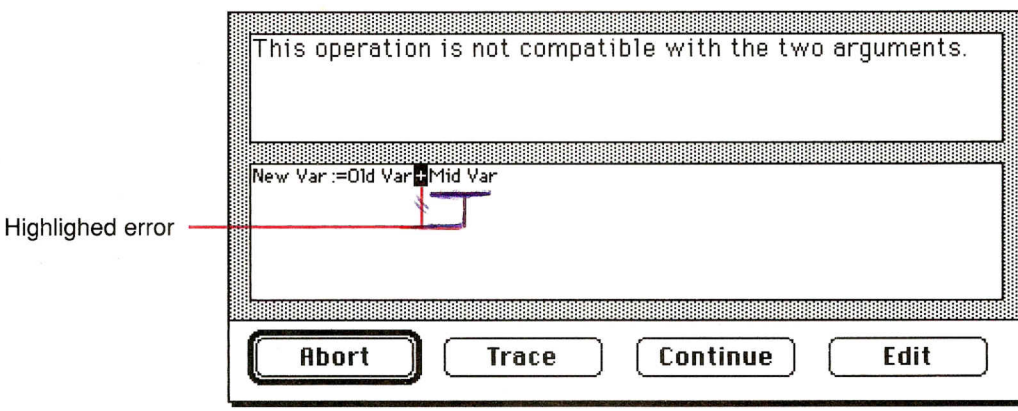


Figure 8-1
The Syntax Error window

The upper text area of the Syntax Error window displays a message describing the error. The lower text area shows the line that was executing when the error occurred, and highlights the area where the error occurred.

The buttons at the bottom of the window give you four options:

- Abort—All procedure execution is halted, and you return to where you were before you started procedure execution. If you are in a layout execution phase, the phase is stopped and you return to the layout. If the procedure was executed because of a menu choice, you return to the menu.
- Trace—You enter Trace mode, and the Trace window is displayed. See the next section for information on tracing.
- Continue—Execution continues. The line with the error may be partially executed, depending on where the error was. Continue with caution—the error may prevent your procedure from continuing properly.
- Edit—All procedure execution is halted. 4th DIMENSION changes to the Design environment, and the procedure where the error occurred is opened in the Procedure editor, allowing you to correct the error.

Note: The Syntax Error window will not be displayed if an error-handling procedure has been installed with the ON ERR CALL command. For information on this command, see "Controlling the Execution of Procedures," in Chapter 18.

The Debugger

When an error has occurred, or when you need to monitor the execution of your procedures, you can use the Debugger. The term *debug* comes from the term *bug*. A bug in a procedure is a mistake that you want to eliminate. A debugger helps you find the bug by allowing you to slowly step through your procedures and examine procedure information. This process of stepping through procedures is called *tracing*.

You can display the Debug window and trace procedures in one of two ways:

- clicking Trace in the Syntax Error window
- using the TRACE command described in Chapter 18 page 365

Note: If there is a password system, only the Designer (that is, the user with the top-level password) may trace procedures.

The Debug window is shown in Figure 8-2. It may be moved and resized.

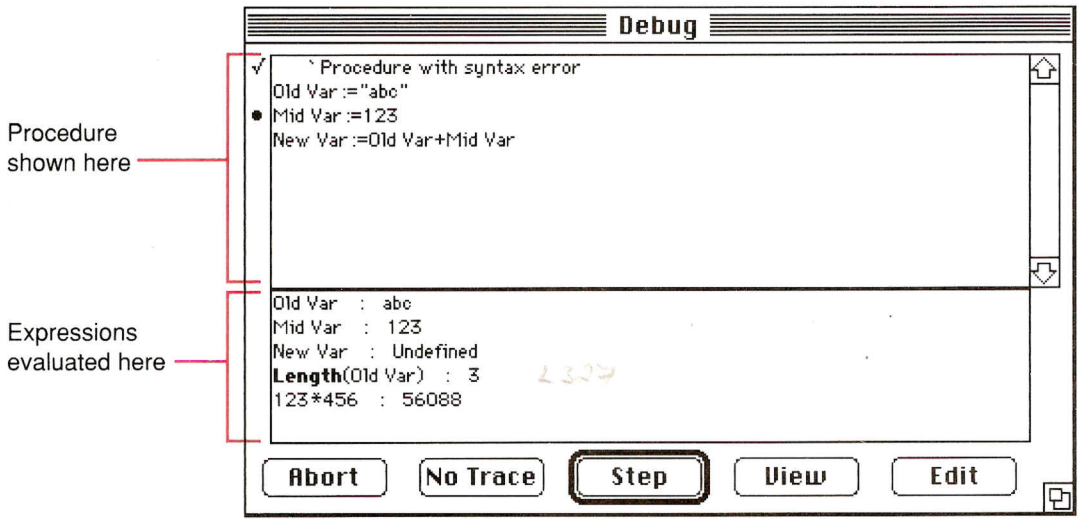


Figure 8-2
The Debug window

The buttons at the bottom of the Debug window give you five options:

- Abort—All procedure execution is halted, and you return to where you were before you started procedure execution. If you are in a layout execution phase, the phase is stopped and you return to the layout. If the procedure was executed because of a menu choice, you return to the menu.
- No Trace—Tracing is halted, and normal procedure execution resumes.
- Step—The current procedure line is executed, and the Debugger steps to the next line.
- View—The Debug window is hidden so that you may view what is behind it.
- Edit—All procedure execution is halted. 4th DIMENSION changes to the Design environment, and the procedure last displayed in the Debug window is opened in the Procedure editor.

The upper text area of the Debug window shows the currently executing procedure. If the procedure is longer than will fit in the text area, you may scroll to view other parts of the procedure. The lower text area is described in the next section.

Evaluating Expressions

The lower text area of the Debug window is used to evaluate expressions. Any type of expression can be evaluated, including fields, variables, pointers, calculations, built-in functions, your functions, and anything else that returns a value. In the Debug window shown in Figure 8-2 you can see several of these items: two variables; an undefined variable; the result of a built-in function; and a calculation.

There are several things you can do in the expression evaluation area:

- You can resize the expression evaluation area by holding down the Option key and dragging the line that separates the procedure from the expression evaluation area.
- You enter an expression by clicking in the area. A text insertion point appears.
- After you enter an expression, you evaluate the expression by clicking in the procedure listing area.
- If you hold down the Option key and click at the insertion point, a pop-up menu of your files and fields appears (see Figure 8-3). If you have a complex structure, the menu may take a couple of seconds to display. When you choose one of the fields, its name is placed in the expression evaluation area.
- If you hold down the Command key and click at the insertion point, a pop-up menu of the built-in commands appears (see Figure 8-4). This menu is large, and may take a couple of seconds to display. When you choose one of the commands, its name is placed in the expression evaluation area.

Figure 8-3 shows an example of a menu of files and fields displayed by holding down the Option key and clicking.

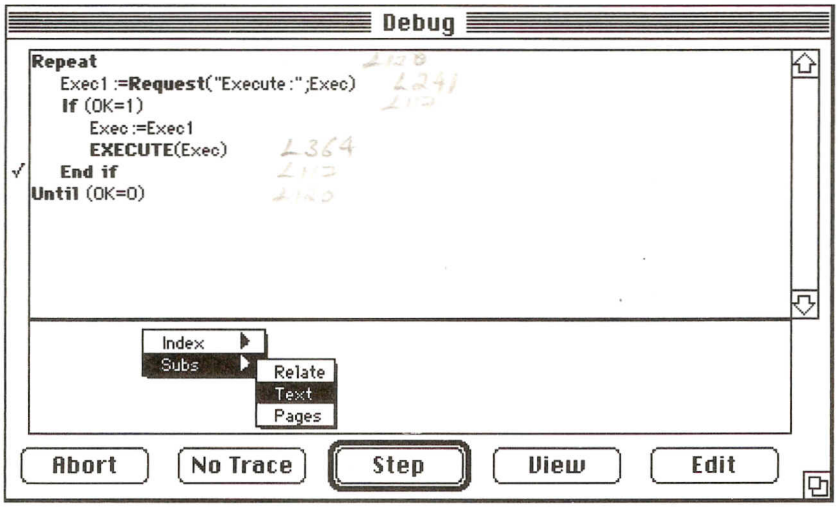


Figure 8-3
Menu of files and fields in the Debug window

Figure 8-4 shows the menu of built-in commands displayed by holding down the Command key and clicking.

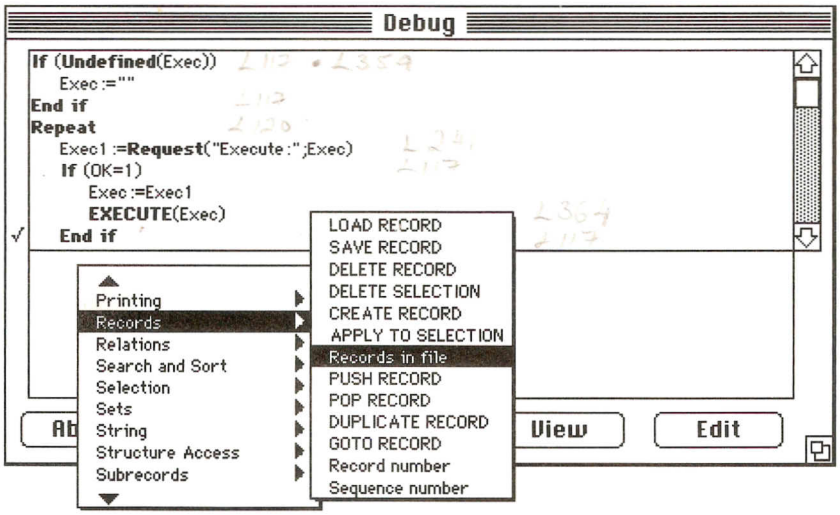


Figure 8-4
Menu of built-in commands in the Debugger

Stepping and Breakpoints

In the Debug window, a check mark in the left margin next to the procedure marks the next line that will be executed. In the Debug window in Figure 8-5, the first line has not been executed; the check mark next to it indicates that it will be executed next.

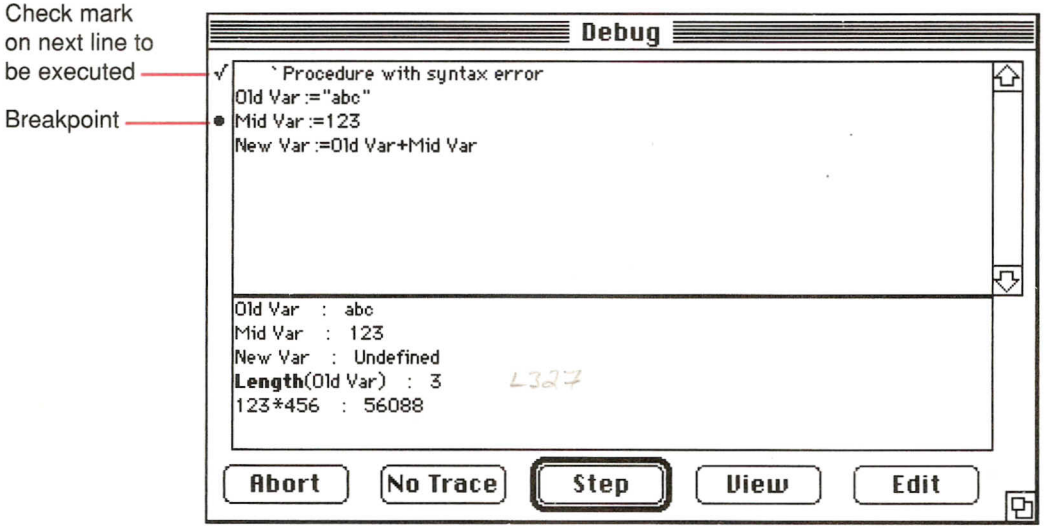


Figure 8-5
Check mark on first line in the Debug window

When you click the Step button (or press Return or Enter), the line is executed and the check mark moves to the next line. Any expressions in the expression evaluation area are updated at this time.

If the line being executed calls another procedure, the Debugger will trace the called procedure. You can optionally execute the called procedure without tracing it, by holding down the Shift key when you click the Step button.

You can set a *breakpoint* in the same margin as the check mark. A breakpoint marks a point at which to halt procedure execution. Here are the rules for breakpoints:

- A breakpoint is indicated by a bullet (•). Figure 8-5 shows a breakpoint next to the third procedure line.
- You set a breakpoint by moving the pointer to the left of the line on which you want to halt. The pointer turns into a bullet. When you click, the breakpoint is set.
- You clear a breakpoint by clicking on it again.
- If you have set a breakpoint, you can click No Trace, and the procedure will execute normally. It will halt procedure execution and display the Debugger when the breakpoint is reached.
- You can set multiple breakpoints.
- All breakpoints are cleared either when procedure execution is halted and control returns to the menu bar, or when a layout execution phase completes.

ARRAYS AND POINTERS

CHAPTER 9

ARRAYS AND POINTERS

23rd May 1998 Sat

This chapter covers two topics, arrays and pointers. Both of these topics are more advanced in nature than other topics covered in Part I of this manual.

Arrays

Arrays are used to efficiently store, access, and manage a group of related data. An array can be thought of as a list of numbered variables. Each item in the array is called an *element* of the array.

Arrays can be created for each of the variable data types. All elements in an array are of the same data type. An array is generally of a fixed size, but there are commands to let you resize an array.

Since arrays reside in memory, operations on them are fast. You can copy, sort, search, and otherwise manipulate arrays very quickly. Also, since arrays reside in memory, you should use discretion when creating large arrays to avoid running out of memory. In particular, be careful when using the commands that move records into arrays, since if there are many records, large arrays will be created.

The commands used to create and manage arrays are described in Chapter 18 of this manual.

Using Arrays

You create an array with one of the commands described in the section "Managing Arrays," in Chapter 18 of this manual. The following line creates an array that can store text:

ARRAY TEXT (Name; 7) L355

You reference the elements in an array by using curly braces ({...}). A number is used within the braces. The following lines put seven names into the array called Name:

```
Name{1} := "John"
Name{2} := "Jane"
Name{3} := "Sam"
Name{4} := "Sarah"
Name{5} := "Richard"
Name{6} := "Howard"
Name{7} := "Susan"
```

Figure 9-1 shows the result of these lines.

There are commands that can more efficiently move data into arrays, such as SELECTION TO ARRAY and LIST TO ARRAY.

	Name
Name{1}	John
Name{2}	Jane
Name{3}	Sam
Name{4}	Sarah
Name{5}	Richard
Name{6}	Howard
Name{7}	Susan

Figure 9-1
The Name array filled with data

When you refer to an array element, it acts just like a variable. For example, the following line puts the string "Sam" into the variable My Name:

My Name := Name{3}

Using Two-Dimensional Arrays

Two-dimensional arrays can be created with the array commands. For example, the following line creates three arrays, each with seven elements:

ARRAY TEXT (Names; 3; 7) L355

Figure 9-2 shows the arrays with data in them.

	Name{1}		Name{2}		Name{3}
Name{1}{1}	John	Name{2}{1}	Jim	Name{3}{1}	Brown
Name{1}{2}	Jane	Name{2}{2}	Elizabeth	Name{3}{2}	Smith
Name{1}{3}	Sam	Name{2}{3}		Name{3}{3}	Young
Name{1}{4}	Sarah	Name{2}{4}		Name{3}{4}	Chung
Name{1}{5}	Richard	Name{2}{5}	Owen	Name{3}{5}	Hayter
Name{1}{6}	Howard	Name{2}{6}	Frank	Name{3}{6}	Gonzales
Name{1}{7}	Susan	Name{2}{7}	Mary	Name{3}{7}	Kirby

Figure 9-2
A two-dimensional array

You reference the elements of a two-dimensional array with two sets of curly braces. For example, to reference the seventh element in the second array, you would write

```
Name{2}{7}
```

Using the data in Figure 9-2, this line would return the string "Mary".

It is important to understand that each element of Name is an array. For example, Name{3} refers to the third array—in this case, the one that contains the last names.

One "dimension" of a two-dimensional array can be treated like any other array. For example, to copy the third Name array into another array, Pop Up, you'd use this command:

```
COPY ARRAY (Name{3}; Pop Up)
```

Displaying Arrays—An Example

Arrays are displayed in layouts using pop-up menus and scrollable areas. The following example shows how to fill arrays from lists and manage them when they are displayed. In the example, a pop-up menu allows the user to choose a region of the country. After the user has chosen a region, a scrollable area displays the states in the region, and allows the user to select one of the states.

There are two arrays displayed in this example. One array is called Regions, and is displayed as a pop-up menu. It is filled from a list containing the names of regions of the United States. The other array, States, is displayed as a scrollable area. It is filled with the states for each region. There is also a variable called vState, which displays the state which is selected. Figure 9-3 shows the layout that displays the arrays.

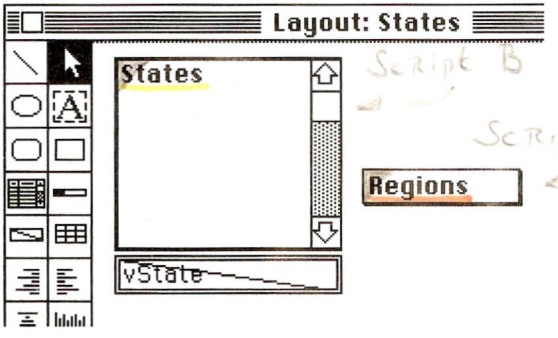


Figure 9-3
Layout containing a scrollable area (States) and a pop-up menu (Regions)

The Regions array is created in the startup procedure, since it never changes.

The following line creates the Regions array and copies a list called Regions into the array.

LIST TO ARRAY ("Regions"; Regions; Links)

The line also creates an array called Links that contains the names of all the linked lists. Each of the items in the Regions list is linked to another list. Figure 9-4 shows the Lists editor and the linked lists. The linked lists are indicated by the small name that follows each of the items.

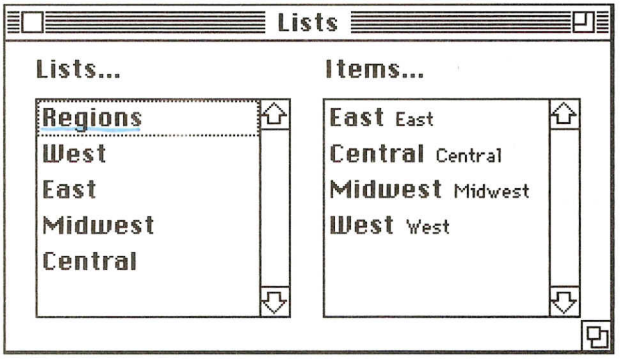


Figure 9-4
The Lists editor with linked lists

The Regions array and the States array both have scripts. The script for the Regions array copies the states for a selected region into the States array. The script is set to always execute ("Only if modified" is unchecked). Here is the script for the Regions array.

```
If (Before | (Regions = 0))                    ` If before display or user chose no item
   Regions := 1                                ` Choose the first item of pop-up menu
End if
LIST TO ARRAY (Links{Regions}; States)         ` Get list of states for the selected region
```

In the script, the If test simply ensures that a pop-up menu item is always chosen.

When the user chooses an item from the Regions pop-up menu, the Regions variable is set to the number of the item. In the last line of the script, the Regions variable is used to reference the name of the list that is stored in the Links array. For example, if the fourth menu item was chosen, then Regions would be set to 4. Figure 9-5 shows the fourth menu item being chosen.

Figure 9-5
Choosing from the Regions pop-up menu

The Links array contains the word *West* as the fourth element. Thus, in this case, the following expression returns West:

```
Links {Regions}
```

Because of this, the line

```
LIST TO ARRAY (Links{Regions}; States)
```

is equivalent to

```
LIST TO ARRAY ("West"; States)
```

and the list named West is copied into the States array. Figure 9-6 shows the result of choosing the fourth menu item.

Figure 9-6
The result of choosing the West menu item

Finally, the script for the States array displays a selected state in the vState variable. If no item in the scrollable area is selected, then vState is set to the empty string. (This is because the zero element of an array contains a null value for that array type.) If an item is selected, vState is set to the name of the state that is selected. Here is the script for the States scrollable area:

```
vState := States{States}                    ` Display the selected state
```

In Figure 9-6, the third item in the States scrollable area was selected. This sets the variable called States to 3. Using this number to reference an element in the States array returns the word *California*, which is assigned to the vState variable for display.

Using Grouped Scrollable Areas

You can group scrollable areas for display in a layout. When they are grouped, they act as if they are one scrollable area. Each scrollable area can have its own font and style.

Figure 9-7 shows three scrollable areas grouped together.

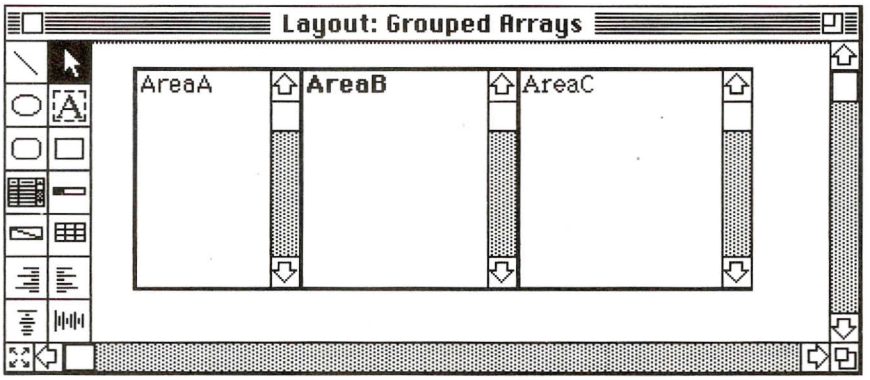


Figure 9-7
Grouped scrollable areas in the Layout editor

Here are some tips on creating a grouped scrollable area:

- Use the same font size for each area.
- Make each area the same height.
- Align the top of each area with the tops of the other areas.
- Make sure the areas are "touching" each other, but do not overlap.
- Move the area that is on the right to the front. The front most area will get the scroll bar.
- To make the areas work as one, select all the areas and then choose the Group menu item from the Object menu.
- Around the group, draw a border that is 1 pixel larger than the group. (As a shortcut, select the group and press Command-1.) Move the right-hand border left 1 pixel for a clean presentation.

The following line fills the three arrays displayed in Figure 9-7. It uses the data in the fields of the [People] file and the [Departments] file. The [Departments] file can be used because it is related to the [People] file. Here is the line:

L361 **SELECTION TO ARRAY** ([People]Last; AreaA; [People]Title; AreaB; [Departments]Name; AreaC)

Figure 9-8 shows the resulting display.

Booker	**Sales Person**	Sales
Warnock	**Director**	Administration
Erickson	**Clerk**	Accounting
Finch	**Sales Person**	Sales
Coggshall	**Technician**	Engineering
Solomon	**Sales Person**	Sales
Hurlow	**Assembler**	Maintenance

Figure 9-8
Grouped arrays being used

Notice that only a single scroll bar is displayed. It is always on the front-most scrollable area, and controls the scrolling of all three arrays as if they were one.

When the user clicks a line, all three areas highlight simultaneously. The variable associated with each scrollable area is set to the number of the line that the user clicks. Only the script for the area that is clicked executes. For example, if the user clicked the name Finch in Figure 9-8, AreaA, AreaB, and AreaC would all be set to 4, but only the script for AreaA would execute.

If you set one of the variables for a scrollable area, the other variables will automatically be set to the same value, and the respective line in the scrollable area will highlight.

The arrays can be sorted with the following line:

SORT ARRAY (AreaB; AreaA; AreaC; >) L357

Figure 9-9 shows the result of the sort.

Hurlow	**Assembler**	Maintenance
Erickson	**Clerk**	Accounting
Warnock	**Director**	Administration
Solomon	**Sales Person**	Sales
Krause	**Sales Person**	Sales
Booker	**Sales Person**	Sales
Finch	**Sales Person**	Sales

Figure 9-9
Grouped arrays sorted

Notice that the arrays were sorted based on the first argument to the SORT ARRAY command. See the section "Managing Arrays," in Chapter 18, for information on this and other array commands. L354

Previous Note

Pointers

Examples See Pointer L370

Using pointers is an advanced method of referring to data. You should thoroughly understand the other concepts presented ~~earlier in Part I~~ HERE before you use pointers. Page 80

When you use the language, you access various objects by name—in particular, files, fields, variables, and arrays. To use one of these, you simply use its name. It is often useful to refer to and access these things without knowing their names. This is what pointers let you do.

The concept behind pointers is not that uncommon in everyday life. You often refer to something without knowing its exact identity. For example, you might say to a friend, "Let's go for a ride in *your car*" instead of "Let's go for a ride in the car with license plate 123ABD." In this case, you are *referencing* the car with license plate 123ABD by using the phrase "your car." The phrase "car with license plate 123ABD" is like the name of an object, and using the phrase "your car" is like using a pointer to reference the object.

Being able to refer to something without knowing its exact identity is very useful. In fact, your friend could get a new car, and the phrase "your car" would still be accurate—it would still be a car and you could still take a ride in it. Pointers work the same way. For example, a pointer could at one time refer to a numeric field called Age, and later refer to a numeric variable called Old Age. In both cases, the pointer is referencing numeric data that could be used in a calculation.

You can use pointers to reference files, fields, variables, arrays, and array elements. Table 9-1 gives an example of each type.

Table 9-1
Examples of pointers

Data Type	To Reference	To Use	To Assign	
File	My File := »[File]	**DEFAULT FILE** (My File»)	n/a	L134
Field	My Field := »[File]Field	**ALERT** (My Field»)	My Field» := "John"	L239
Variable	My Var := »Variable	**ALERT** (My Var»)	My Var» := "John"	L239
Array	My Arr := »Array	**COPY ARRAY** (My Arr»; B)	**COPY ARRAY** (B; My Arr»)	L358
Array element	My Elem := »Array{1}	**ALERT** (My Elem »)	My Elem » := "John"	L239

Using Pointers—An Example

It is easiest to explain the use of pointers by using an example. This example shows how to access a variable through a pointer. We start by creating a variable:

```
My Var := "Hello"
```

My Var is now a variable containing the string "Hello". We can create a pointer to My Var:

```
My Pointer := »My Var
```

The » symbol says "get the pointer to." (Press Option-| to write the » symbol.) In this case, it gets the pointer that references or "points to" My Var. This pointer is assigned to My Pointer with the assignment operator.

My Pointer is now a variable that contains a pointer to My Var. My Pointer *does not* contain "Hello", the value in My Var, but you *can* use My Pointer to get the value contained in My Var. The following expression returns the value in My Var:

```
My Pointer»
```

In this case, it returns the string "Hello". The » symbol, when it follows a pointer, references the object pointed to.

It is important to understand that you can use a pointer followed by the » symbol anywhere that you could have used the object that the pointer points to. This means that you could use the expression My Pointer» anywhere that you could use the original variable My Var.

For example, the following line displays an alert box with the word *Hello* in it:

```
ALERT (My Pointer»)
```

You can also use My Pointer to change the data in My Var. For example, the following statement stores the string "Goodbye" in the variable My Var:

```
My Pointer» := "Goodbye"
```

If you examine the two uses of the expression My Pointer» above, you will see that it acts just as if you had used My Var instead. To summarize: The following two lines perform the same action—both print an alert box containing the current value in the variable My Var.

```
ALERT (My Pointer»)
ALERT (My Var)
```

The following two lines perform the same action: Both assign the string "Goodbye" to My Var.

```
My Pointer» := "Goodbye"
My Var := "Goodbye"
```

Using Pointers to Buttons

This section describes how to use a pointer to reference a button. A button is (from the language's point of view) nothing more than a variable. Although the examples in this section use pointers to reference buttons, the concepts presented apply to the use of all types of pointers.

Let's say that you have a number of buttons in your layouts that need to be enabled or disabled. Each button has a different condition associated with it that is TRUE or FALSE. The condition says whether to disable or enable the button. You could use a test like this one, each time you need to enable or disable the button:

```
If (Condition)                          ` If the condition is TRUE...
   ENABLE BUTTON (My Button)            ` enable the button
Else                                    ` Otherwise...
   DISABLE BUTTON (My Button)           ` disable the button
End if
```

You would need to use a similar test for every button you set, with only the name of the button changing. To be more efficient, you could use a pointer to reference each button and then use a subroutine for the test itself.

You must use pointers if you use a subroutine, because you cannot refer to the button's variables in any other way. For example, here is a subroutine called *Set Button*, which references a button with a pointer:

```
` $1 – Boolean. If TRUE, enable the button. If FALSE, disable the button.
` $2 – Pointer to a button.
If ($1)                                 ` If the condition is TRUE...
   ENABLE BUTTON ($2»)                  ` enable the button
Else                                    ` Otherwise...
   DISABLE BUTTON ($2»)                 ` disable the button
End if
```

You can call the *Set Button* subroutine as follows:

```
Set Button (Test1; »Button1)
Set Button (Test2; »Button2)
```

Using Pointers to Files

Anywhere that the language expects to see a file, you can use a pointer to reference the file. A pointer to a file is primarily used as the first argument to a command that operates on a file.

You create a pointer to a file by using a line like this:

```
FilePtr := »[File1]
```

You can also get a pointer to a file by using the File command. For example,

```
FilePtr := File (1)
```

See the section "Determining the Database Structure," in Chapter 16, for more information on the File command. You can use the referenced file in commands, like this:

```
DEFAULT FILE (FilePtr»)
```

Using Pointers to Fields

Anywhere that the language expects to see a field, you can use a pointer to reference the field.

You create a pointer to a field by using a line like this:

```
FieldPtr := »[File1]Field2
```

You can also get a pointer to a field by using the Field command. For example,

```
FilePtr := Field (1; 2)
```

See the section "Determining the Database Structure," in Chapter 16, for more information on the Field command. You can use the referenced field in commands, like this:

```
SET FONT (FieldPtr»; "Geneva")
```

Using Pointers to Array Elements

You can create a pointer to an array element. For example, the following lines create an array, and assign a pointer to the first array element to a variable called ElemPtr:

```
ARRAY REAL (Arr; 10)          ` Create an array
ElemPtr := »Arr{1}            ` Create a pointer to the array element
```

You could use the pointer to assign a value to the element, like this:

```
ElemPtr» := 8
```

Using Pointers to Arrays

You can create a pointer to an array. For example, the following lines create an array, and assign a pointer to the array to a variable called ArrPtr:

```
ARRAY REAL (Arr; 10)          ` Create an array
ArrPtr := »Arr                ` Create a pointer to the array
```

It is important to understand that the pointer points to the array; it *does not* point to an element of the array. For example, you might use the pointer from the preceding lines like this:

```
SORT ARRAY (ArrPtr»; >)       ` Sort the array
```

If you need to refer to the fourth element in the array by using the pointer, you do this:

```
ArrPtr»{4} := 84
```

This method is different from using an array of pointers. See the next section for a discussion of this technique.

Using an Array of Pointers

It is often useful to have an array of pointers that reference a group of related objects.

One example of such a group of objects is a grid of variables in a layout. Each variable in the grid is sequentially numbered, for example: Var1, Var2,..., Var10. You often need to reference these variables indirectly with a number. If you create an array of pointers, and initialize the pointers to point to each variable, you can then easily reference the variables. For example, to create an array and initialize each element, you could use the following lines:

```
ARRAY POINTER (Vars; 10)      ` Create an array to hold 10 pointers
For ($i; 1; 10)               ` Loop once for each variable
  Vars{$i} := Get pointer ("Var" + String ($i))   ` Initialize the array element
End for
```

The Get pointer function returns a pointer to the named object.

To reference any of the variables, you use the array elements. For example, to fill the variables with the next ten dates (assuming they are variables of the date type), you could use the following lines:

```
For ($i; 1; 10)               ` Loop once for each variable
  Vars{$i}» := Current date + $i – 1   ` Assign the dates
End for
```

Setting a Button Using a Pointer

If you have a series of related radio buttons in a layout, you often need to set them quickly. It is inefficient to directly reference each one of them by name. Figure 9-10 shows five radio buttons named Button1, Button2,..., Button5.

○ **Button1**
○ **Button2**
○ **Button3**
○ **Button4**
○ **Button5**

Figure 9-10
Five radio buttons

In a series of radio buttons, only one radio button is on. The number of the radio button that is on can be stored in a numeric field. For example, if the field called Setting contains 3, then Button3 is set to 1. In your layout procedure, you could use the following code to set the button:

```
If (Before)
  Case of
    : (Setting = 1)                  ` Test Setting which is a field
      Button1 := 1                   ` If Setting is TRUE...
    : (Setting = 2)                  ` turn on the radio button, and so on
      Button2 := 1
    : (Setting = 3)
      Button3 := 1
    : (Setting = 4)
      Button4 := 1
    : (Setting = 5)
      Button5 := 1
  End case
End if
```

For each radio button, a separate case must be tested. This could be a very long procedure if you have many radio buttons in your layout. Fortunately, you can use pointers to solve this problem.

You can use the command Get pointer to return a pointer to a radio button (or any button). The following example uses such a pointer to reference the radio button that needs to be set. Here is the improved code:

```
If (Before)
   $P := Get pointer ("Button" + String (Setting))   ` Get the pointer to the radio button
   $P» := 1                                          ` Turn on the radio button
End if
```

The number of the set radio button needs to be stored in the field called Setting. This is done by a one-line script for each radio button. For example, here is the script for Button3:

```
Setting := 3
```

Passing Pointers to Procedures

You can pass a pointer as a parameter to a procedure. Inside the procedure, you can modify the object referenced by the pointer.

For example, the following procedure, *Take Two*, takes two parameters which are pointers. It changes the object that the first parameter references to uppercase characters, and the object that the second parameter references to lowercase characters. Here is the procedure:

```
` $1 – Pointer to a string. Change this to uppercase.
` $2 – Pointer to a string. Change this to lowercase.
$1» := Uppercase ($1»)
$2» := Lowercase ($2»)
```

Notice that the procedure does not return a value.

The following line uses the *Take Two* procedure to change a field to uppercase characters and a variable to lowercase characters:

```
Take Two (»[My File]My Field; »My Var)
```

If the field, [My File]My Field, contained the string "jones", it would be changed to the string "JONES". If the variable, My Var, contained the string "HELLO", it would be changed to the string "hello".

In the *Take Two* procedure (and, in fact, whenever you use pointers), it is important that the data type of the object being referenced is correct. In the example just given, the pointers must point to an object that contains a string.

Pointers to Pointers

Pointers can reference other pointers. Consider this example:

```
My Var := "Hello"
Point One := »My Var
Point Two := »Point One
ALERT ((Point Two»)»)
```

It displays an alert box with the word *Hello* in it. To begin with, the example illustrates the complexities inherent in the use of pointers. You must be very aware of where and to what a pointer is pointing.

Here is an explanation of each line of the example.

```
My Var := "Hello"
```

This line simply puts the string "Hello" into the variable My Var.

```
Point One := »My Var
```

Point One now contains a pointer to My Var.

```
Point Two:= »Point One
```

Point Two (a new variable) contains a pointer to Point One, which in turn points to My Var (tricky, huh?).

```
Point Two»» := "Goodbye"
```

Now it gets really interesting. Point Two» references the contents of Point One, which in turn references My Var. Point Two»» references the contents of the location referenced by the pointer in Point One. Therefore Point Two»» *simply* references the contents of My Var. So in this case, My Var is assigned "Goodbye".

```
ALERT ((Point Two»)»)
```

Finally, we can access the contents of My Var with the above statement. Here, (Point Two»)» gets the contents of My Var. Notice that this statement uses a slightly different syntax from that used to put information into My Var. This line puts "Hello" into My Var:

```
Point Two»» := "Hello"
```

This line gets "Hello" from My Var and puts it into New Var:

```
New Var := (Point Two»)»
```

PART

LANGUAGE DEFINITION

CHAPTER 10

LANGUAGE DEFINITION

LANGUAGE DEFINITION

This part of the manual formally defines the components that make up the 4th DIMENSION language. It covers

- identifiers
- data types
- constants
- operators
- controlling procedure flow

Identifiers

This section describes the conventions for naming various objects in the 4th DIMENSION language. The names for all objects 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.
- Characters reserved for use as operators, such as * and +, are not allowed.
- 4th DIMENSION will clip any trailing spaces.

Files

You denote a file by placing its name between brackets. A filename can contain up to 15 characters.

Filenames	Filenames in Code	
[Orders]	**DEFAULT FILE** ([Orders])	L134
[Clients]	**INPUT LAYOUT** ([Clients]; "Entry")	L137
[Letters]	**ADD RECORD** ([Letters])	L141

Fields

You denote a field by first specifying the file to which the field belongs. The field's name immediately follows the filename. A field name can contain up to 15 characters.

Field Names	Field Names in Code	
[Orders]Total	[Orders]Total := **Sum** ([Line]Amount)	L347
[Clients]Name	**SEARCH** ([Clients]; [Clients]Name = "Smith")	L194
[Letters]Text	*Capitalize* ([Letters]Text)	

If you are specifying a field in a file procedure, layout procedure, or script of the file, you do not need to specify the filename.

Field Names	Field Names in Code	
Total	Total := **Sum** ([Line]Amount)	L347
Name	**SEARCH** ([Clients]; Name = "Smith")	L194
Text	*Capitalize* (Text)	

Subfiles

You denote a subfile by first specifying the file to which the subfile belongs. The file is the parent file for the subfile. The subfile's name immediately follows the filename. A subfile name can contain up to 15 characters.

Subfile Names	Subfile Names in Code	
[People]Children	**ALL SUBRECORDS** ([People]Children)	L228
[Clients]Phones	**ADD SUBRECORD** ([Clients]Phones; "Add One")	L226
[Letters]Keywords	**NEXT SUBRECORD** ([Letters]Keywords)	L231

A subfile is treated as a type of field; it therefore follows the same rules as a field when used in a layout. If you are specifying a subfile in a file procedure, layout procedure, or script of the parent file, you do not need to specify the parent filename.

Subfile Names	Subfile Names in Code	
Children	**ALL SUBRECORDS** (Children)	L228
Phones	**ADD SUBRECORD** (Phones; "Add One")	L226
Keywords	**NEXT SUBRECORD** (Keywords)	L231

Subfields

You denote a subfield in the same way as a field. You denote the subfield by first specifying the subfile to which the subfield belongs. The subfield's name follows, and is separated from the subfile name by an apostrophe ('). A subfield name can contain up to 15 characters.

Subfield Names	Subfield Names in Code
[People]Child'Name	[People]Child'Name := **Uppercase** ([People]Child'Name)
[Clients]Phones'Number	[Clients]Phones'Number := "408 555–1212"
[Letters]Keywords'Word	*Capitalize* ([Letters]Keywords'Word)

If you are specifying a subfield in a subfile procedure, layout procedure, or script of the subfile, you do not need to specify the subfile name.

Subfield Names	Subfield Names in Code
Name	Name := **Uppercase** (Name)
Number	Number := "408 555–1212"
Word	*Capitalize* (Word)

Global Variables

You denote a variable by using its name. A global variable name can contain up to 11 characters.

Global Variable Names	Global Variable Names in Code
Grand Total	Grand Total := **Sum** ([Account]Amount)
Button1	**If** (Button1 = 1)
My Var	My Var := "Constant String"

Local Variables

You denote a local variable with a dollar sign ($) followed by its name. A local variable name can contain up to 11 characters, not including the $.

Local Variable Names	Local Variable Names in Code
$i	**For** ($i; 1; 100)
$Temp Var	**If** ($Temp Var = "No")
$My String	$My String := "Hello there"

Arrays

You denote an array by using its name. An array name can contain up to 11 characters.

Array Names	Array Names in Code
Items	**ARRAY TEXT** (Items; 20)
Keyword	**SORT ARRAY** (Keyword; >)
Files	**COPY ARRAY** (Files; Scrollable)

You reference an element of an array by using the curly braces ({...}). The element referenced is denoted by a numeric expression.

Array Elements	Array Elements in Code
Items{$i}	Items{$i} := [People]Name
Keyword{3}	**If** (Keyword{3} = "Stop")
Files{$Name}	My File := Files{$Name}

You reference an element of a two-dimensional array by using the curly braces ({...}). The element referenced is denoted by two numeric expressions in two sets of curly braces.

Array Elements	Array Elements in Code
Item{$i}{$j}	Item{$i}{$j} := [People]Name
Keyword{3}{2}	**If** (Keyword{3}{2} = "Stop")
Files{$Name1}{$Name2}	My Field := Files{$Name1}{$Name2}

Layouts

You denote a layout by using a string expression that represents its name. A layout name can contain up to 15 characters.

Layout Names	Layout Names in Code
"Input"	**INPUT LAYOUT** ([People]; "Input")
"Output"	**OUTPUT LAYOUT** ([People]; "Output")
"Note box" + **String** ($i)	**DIALOG** ([Storage]; "Note box" + **String** ($i))

Procedures and Functions

You denote a procedure or function by using its name. A procedure name can contain up to 15 characters.

Procedure Names	Procedure Names in Code
New Client	**If** (*New Client*) L117
Delete Dups	*Delete Dups*
Capitalize	**APPLY TO SELECTION** (*Capitalize*) L186

Procedures and functions can accept parameters (arguments). The parameters are passed to the procedure or function in parentheses, following the procedure or function. Each parameter is separated from the next by a semicolon.

Procedure Names	Parameters Passed to Procedures
Drop Spaces	[People]Name := *Drop Spaces* ([People]Name)
Creator	*Creator* (1; 5; Nice)
Dump	Clone := *Dump* ("is"; "the"; "it")

L24
L56

The parameters are available within the called procedure or function as consecutively numbered local variables: $1, $2,..., $n.

A function returns a value. Inside the function, the local variable $0 contains the value to be returned.

External Procedures, Functions, and Areas

You denote an external procedure, function, or area by using its name. An external procedure name can contain up to 15 characters.

External Procedure Names	External Procedures in Code
4D Word	***4D Word***
Mini Connect	***Mini Connect***
Parse Words	[People]Name := ***Parse Words*** ([People]Name)

Sets

You denote a set by using a string expression that represents its name. A set name can contain up to 80 characters.

Set Names	Sets in Code
"Records to be deleted"	**USE SET** ("Records to be deleted") L277
"Customer Orders"	**CREATE SET** ("Customer Orders") L276
"Selection"+ **String** ($i)	**Records in set** ("Selection"+ **String** ($i)) L282 • L332

Summary of Naming Conventions

Table 10-1 summarizes 4th DIMENSION naming conventions.

Table 10-1
4th DIMENSION naming conventions

Type	Length	Example
File	15	[File1]
Field	15	[File1]Field1
Subfile	15	[File1]Subfile
Subfield	15	[File1]Subfile'Subfield
Global variable	11	Variable
Local variable	11	$Local
Layout	15	"Layout"
Array	11	Array
Procedure	15	*Procedure*
External procedure	15	***External***
Set	80	"Set"

Resolving Naming Conflicts

If a particular object has the same name as another object of a different type (for example, if a field is named Person and a variable is also named Person), 4th DIMENSION uses a priority system to identify the object. It is up to you to ensure that you use unique names for the parts of your database.

4th DIMENSION identifies names used in procedures in the following order:

1. Fields
2. Commands
3. Procedures
4. External procedures
5. Variables

For example, 4th DIMENSION has a built-in function called Date. If you named a procedure *Date*, 4th DIMENSION would recognize it as the built-in function Date, and not as your procedure. This would prevent you from calling your procedure.

Data Types

A variable or expression can be one of seven data types:

- string
- numeric (number)
- date
- time
- Boolean
- picture
- pointer

The first six data types are described in this section. Pointers are described in Chapter 9, in Part I of this manual.

String

- A string expression is abbreviated *string* in the manuals.
- A string is composed of characters.
- Each character can be any of the 256 ASCII characters supported by the Macintosh, although only some of the characters can be displayed. See Appendix D for a table containing the Macintosh ASCII characters.
- A string may contain from 0 to 32,000 characters.
- Strings are also referred to as *text*.
- Strings are converted automatically to the field types Alpha and Text.

Numeric

- A numeric expression is abbreviated *number* in the manuals.
- A number is any number with up to 19 significant digits.
- The value can be between +1e1022 and –1e1022.
- Numbers are stored internally as Macintosh extended reals.
- Numbers are converted automatically to the field types Integer, Long Integer, and Real.

Date

Application (Entry) Designed DD/MM/YY

- A date expression is abbreviated *date* in the manuals.
- A date can be in the range of 1/1/100 to 12/31/32,767. 1st Jan 100 31st Dec 32767
- A date is ordered month/day/year. MM/DD/year
- If a year is given as two digits, it is assumed to be in the 1900's.

Time

- A time expression is abbreviated *time* in the manuals.
- A time can be in the range of 00:00:00 to 596,000:00:00.
- A time is ordered hour:minute:second.
- Times are in 24-hour format.
- A time value can be treated as a number with no conversion. The number returned from a time is the number of seconds that time represents.

hours 596,000
days 24833
weeks 3,547
years 68

Boolean

- A Boolean expression is abbreviated *Boolean* in the manuals.
- A Boolean expression can be either TRUE or FALSE.

Picture

- A picture expression is abbreviated *picture* in the manuals.
- A picture can be any Macintosh picture of type PICT or PICT 2. In general, these types include any picture that can be put on the Clipboard.

Converting Data Types

The language contains functions to convert between data types where such conversions will be meaningful. Table 10-2 lists the data types, the types to convert to, and the commands used.

Table 10-2
Commands that convert data types

Data type	Convert to String	Convert to Number	Convert to Date	Convert to Time
String		Num	Date	Time
Number	String			
Date	String			
Time	String			
Boolean		Num		

Note: Time values can be treated as numbers with no conversion.

Constants

A constant is an expression that has a fixed value. Constants can be of four data types:

- string
- numeric
- date
- time

String Constants

A string constant is enclosed in double, straight quotation marks ("...").

Here are some examples of string constants,

```
"Add Records"
"No records found."
"Invoice"
```

An empty string is specified by two quotation marks with nothing between them ("").

Numeric Constants

A numeric constant is written as a real number.

Here are some examples of numeric constants,

```
27
123.76
.0076
```

Negative numbers are specified with the negation symbol (–). For example:

```
–27
–123.76
–.0076
```

Numbers can be specified with scientific notation, using an *e*, followed optionally by the negation symbol for a negative exponent, and completed with the exponent. For example,

```
2.7e1
1.2376e+2
7.6e–3
```

Date Constants

A date constant is enclosed in exclamation marks (!…!).

A date is ordered month/day/year, with a slash (/) setting off each part.

Displayed as day/month/year

Here are some examples of date constants,

!1/1/76!
!4/4/04!
!12/25/89!

An empty date is specified by !00/00/00!.

A two-digit year is assumed to be in the 1900's.

Time Constants

A time constant is enclosed in time symbols (†…†). (Press Option-t to get the time symbols.)

A time is ordered hour:minute:second, with a colon (:) setting off each part.

Times are specified in 24-hour format.

If the minute or second is omitted, it is assumed to be zero. For example, †1† is equal to †1:00† which is equal to †01:00:00†.

Here are some examples of time constants,

†01:00:00†
†01:01:00†
†13:01:59†

Operators

Operators are symbols used to specify operations to be performed between expressions. Operators perform calculations on numbers, dates, and times. They perform string operations, Boolean operations on logical expressions, and specialized operations on pictures. Operators combine simple expressions to generate new expressions.

Precedence

The order in which an expression is evaluated is called *precedence*. 4th DIMENSION has a strict left-to-right precedence. For example,

3 + 4 * 5

returns 35 because the expression is evaluated as 3 + 4, giving 7, which is then multiplied by 5, with the result 35.

Parentheses can be used to override the left-to-right precedence. For example,

3 + (4 * 5)

returns 23 because the expression (4 * 5) is evaluated first, because of the parentheses. The result is 20, which is then added to 3 for the final result of 23.

Parentheses can be nested inside other sets of parentheses. Be sure that each left parenthesis has a matching right parenthesis.

You must take care to ensure proper evaluation of expressions. Lack of or incorrect use of parentheses can cause either unexpected results or invalid expressions.

The Assignment Operator

The assignment operator (:=) copies the value of the expression to the right of the assignment operator into the variable or field to the left of the operator.

For example, the following line places the value 4 (the number of characters in the word *Acme*) into the variable named MyVar. MyVar is then typed as numeric.

MyVar := **Length** ("Acme") L327

String Operators

Table 10-3 shows the string operators. An expression that uses a string operator returns a string.

Table 10-3
String operators

Operation	Symbol	Syntax	Returns	Example
Concatenation	+	string + string	string	"abc" + "def" → "abcdef"
Repetition	*	string * number	string	"ab" * 3 → "ababab"

Numeric Operators

Table 10-4 shows the numeric operators. An expression that uses a numeric operator returns a number.

The modulo operator (%) divides the first number by the second number and returns a whole number remainder. Here are some examples.

10 % 2 returns 0 because 10 is evenly divided by 2.

10 % 3 returns 1 because the remainder is 1.

10.5 % 2 returns 0 because the remainder is not a whole number.

Table 10-4
Numeric operators

Operation	Symbol	Syntax	Returns	Example
Addition	+	number + number	number	2 + 3 → 5
Subtraction	–	number – number	number	3 – 2 → 1
Multiplication	*	number * number	number	5 * 2 → 10
Division	/	number / number	number	5 / 2 → 2.5
Longint division	\	number \ number	number	5 \ 2 → 2
Modulo	%	number % number	number	5 % 2 → 1
Exponentiation	^	number ^ number	number	2 ^ 3 → 8

Date Operators

Table 10-5 shows the date operators. An expression that uses a date operator returns a date or a number, depending on the operation. All date operations will result in an accurate date, taking into account the change between years and leap years.

Table 10-5
Date operators

Operation	Symbol	Syntax	Returns	Example
Date difference	–	date – date	number	!1/20/90! – !1/1/90! → 19 (days)
Day addition	+	date + number	date	!1/20/90! + 9 → !1/29/90!
Day subtraction	–	date – number	date	!1/20/90! – 9 → !1/11/90!

Time Operators

Table 10-6 shows the time operators. An expression that uses a time operator returns a time or a number, depending on the operation.

Table 10-6
Time operators

Operation	Symbol	Syntax	Returns	Example
Addition	+	time + time	time	†02:03:04† + †01:02:03† → †03:05:07†
Subtraction	–	time – time	time	†02:03:04† – †01:02:03† → †01:01:01†
Addition	+	time + number	number	†02:03:04† + 65 → 7449
Subtraction	–	time – number	number	†02:03:04† – 65 → 7319
Multiplication	*	time * number	number	†02:03:04† * 2 → 14768
Division	/	time / number	number	†02:03:04† / 2 → 3692
Longint division	\	time \ number	number	†02:03:04† \ 2 → 3692
Modulo	%	time % number	number	†02:03:04† % 2 → 0

Comparison Operators

Table 10-7 though Table 10-11 show the comparison operators as they apply to string, numeric, date, time, and pointer expressions. An expression that uses a comparison operator returns a Boolean value, either TRUE or FALSE.

Here are some notes on string comparisons:

- Strings are compared on a character-by-character basis.
- When strings are compared, the case of the characters is ignored; thus, "a"="A" returns TRUE. To test if the case of two characters is different, compare their ASCII codes. For example, the following statement returns FALSE:

  Ascii ("A") = **Ascii** ("a")

- The wildcard character (@) can be used in any string comparison. It will match any number of characters. So, for example, the following expression is TRUE:

  "abcefghij" = "abc@"

  The wildcard must be used in the comparing expression (the expression on the right side). The following expression is FALSE:

  "abc@" = "abcefghij"

Table 10-7
String comparison operators

Operation	Symbol	Syntax	Returns	Example
Equality	=	string = string	Boolean	"abc" = "abc" → TRUE "abc" = "abd" → FALSE
Inequality	#	string # string	Boolean	"abc" # "abd" → TRUE "abc" # "abc" → FALSE
Greater than	>	string > string	Boolean	"abd" > "abc" → TRUE "abc" > "abc" → FALSE
Less than	<	string < string	Boolean	"abc" < "abd" → TRUE "abc" < "abc" → FALSE
Greater than or equal to	>=	string >= string	Boolean	"abd" >= "abc" → TRUE "abc" >= "abd" → FALSE
Less than or equal to	<=	string <= string	Boolean	"abc" <= "abd" → TRUE "abd" <= "abc" → FALSE

Table 10-8
Numeric comparison operators

Operation	Symbol	Syntax	Returns	Example
Equality	=	number = number	Boolean	10 = 10 → TRUE 10 = 11 → FALSE
Inequality	#	number # number	Boolean	10 #11 → TRUE 10 # 10 → FALSE
Greater than	>	number > number	Boolean	11 > 10 → TRUE 10 > 11 → FALSE
Less than	<	number < number	Boolean	10 < 11 → TRUE 11 < 10 → FALSE
Greater than or equal to	>=	number >= number	Boolean	11 >= 10 → TRUE 10 >= 11 → FALSE
Less than or equal to	<=	number <= number	Boolean	10 <= 11 → TRUE 11 <= 10 → FALSE

Table 10-9
Date comparison operators

Operation	Symbol	Syntax	Returns	Example
Equality	=	date = date	Boolean	!1/1/89! = !1/1/89! → TRUE !1/20/89! = !1/1/89! → FALSE
Inequality	#	date # date	Boolean	!1/20/89! # !1/1/89! → TRUE !1/1/89! # !1/1/89! → FALSE
Greater than	>	date > date	Boolean	!1/20/89! > !1/1/89! → TRUE !1/1/89! > !1/1/89! → FALSE
Less than	<	date < date	Boolean	!1/1/89! < !1/20/89! → TRUE !1/1/89! < !1/1/89! → FALSE
Greater than or equal to	>=	date >= date	Boolean	!1/20/89! >= !1/1/89! → TRUE !1/1/89! >= !1/20/89! → FALSE
Less than or equal to	<=	date <= date	Boolean	!1/1/89! <= !1/20/89! → TRUE !1/20/89! <= !1/1/89! → FALSE

Table 10-10
Time comparison operators

Operation	Symbol	Syntax	Returns	Example
Equality	=	time = time	Boolean	†01:02:03† = †01:02:03† → TRUE †01:02:03† = †01:02:04† → FALSE
Inequality	#	time # time	Boolean	†01:02:03† # †01:02:04† → TRUE †01:02:03† # †01:02:03† → FALSE
Greater than	>	time > time	Boolean	†01:02:04† > †01:02:03† → TRUE †01:02:03† > †01:02:03† → FALSE
Less than	<	time < time	Boolean	†01:02:03† < †01:02:04† → TRUE †01:02:03† < †01:02:03† → FALSE
Greater than or equal to	>=	time >= time	Boolean	†01:02:03† >= †01:02:03† → TRUE †01:02:03† >= †01:02:04† → FALSE
Less than or equal to	<=	time <= time	Boolean	†01:02:03† <= †01:02:03† → TRUE †01:02:04† <= †01:02:03† → FALSE

Table 10-11
Pointer comparison operators

Operation	Symbol	Syntax	Returns	Example
Equality	=	pointer = pointer	Boolean	(»Object) = (»Object) → TRUE (»Object1) = (»Object2) → FALSE
Inequality	#	pointer # pointer	Boolean	(»Object1) # (»Object2) → TRUE (»Object) # (»Object) → FALSE

Logical Operators

4th DIMENSION supports two logical operators: conjunction (AND) and disjunction (OR). Both of these operators work on Boolean expressions. A logical AND returns TRUE if both expressions are TRUE. A logical OR returns TRUE if at least one of the expressions is TRUE. See Table 10-12.

Table 10-12
Logical operators

Operation	Symbol	Syntax	Returns	Example
Conjunction (AND)	&	Boolean & Boolean	Boolean	("A" = "A") & (15 # 3) → TRUE
				("A" = "B") & (15 # 3) → FALSE
				("A" = "B") & (15 = 3) → FALSE
Disjunction (OR)	|	Boolean | Boolean	Boolean	("A" = "A") | (15 # 3) → TRUE
				("A" = "B") | (15 # 3) → TRUE
				("A" = "B") | (15 = 3) → FALSE

Figure 10-1 shows the *truth table* for the AND logical operator. The truth table shows the two possible values for either argument to the operator, and the result in each case.

Expr1	*Expr2*	*Expr1 & Expr2*
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

Figure 10-1
Truth table for the AND operator (&)

Figure 10-2 shows the truth table for the OR logical operator.

Expr1	*Expr2*	*Expr1 | Expr2*
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Figure 10-2
Truth table for the OR operator (|)

Picture Operators

Table 10-13 summarizes 4th DIMENSION's picture operators. Table 10-14 shows examples of each of the picture operators. The results are shown for both the Truncated and On Background formats.

Table 10-13
Picture operators

Operation	Symbol	Syntax	Action
Horizontal concatenation	+	pict1 + pict2	Move pict2 to the right of pict1
Vertical concatenation	/	pict1 / pict2	Move pict2 to the bottom of pict1
Exclusive superimposition	&	pict1 & pict2	Perform exclusive OR on pict1 and pict2
Inclusive superimposition	|	pict1 | pict2	Put pict1 on top of pict2
Horizontal move	+	picture + number	Move picture horizontally number pixels
Vertical move	/	picture / number	Move picture vertically number pixels
Resize	*	picture * number	Resize picture by number percent
Horizontal scaling	*+	picture *+ number	Resize picture horizontally by number percent
Vertical scaling	*/	picture */ number	Resize picture vertically by number percent

Table 10-14
Examples of picture operators

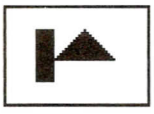
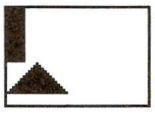
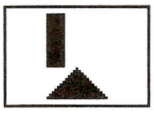
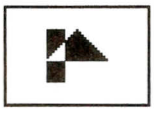
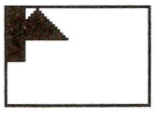
Operation	Example	Picture(s)	On Background	Truncated
Horizontal concatenation	Pict1 + Pict2			
Vertical concatenation	Pict1 / Pict2			
Exclusive superimposition	Pict1 & Pict2			
Inclusive superimposition	Pict1 | Pict2			

Table 10-14 (continued)
Examples of picture operators

Operation	Example	Picture(s)	On Background	Truncated
Horizontal move right	Pict1 + 5			
Horizontal move left	Pict1 + (–5)			
Vertical move down	Pict1 / 5			
Vertical move up	Pict1 / (–5)			
Resize larger	Pict * 2			
Resize smaller	Pict * .5			
Horizontal scale larger	Pict *+ 2			
Horizontal scale smaller	Pict *+ .5			
Vertical scale larger	Pict */ 2			
Vertical scale smaller	Pict */ .5			

Controlling Procec

This section describes statements that control the flow of program execution, including conditional execution and looping statements. See Chapter 11 for a description of the format used to present these statements.

If...Else...End if

L30

If (*Boolean*)
    statement(s)
Else
    statement(s)
End if

Parameter	Type	Description
Boolean	Boolean	Test expression

If...Else...End if is used to control procedure execution based on the result of a test. If *Boolean* is TRUE, the next *statement(s)* are executed until the Else or End if is reached.

The Else and the *statement(s)* following it are optional. If an Else is included, the *statement(s)* following it are executed only if *Boolean* is FALSE.

The End if indicates the end of the If test.

If statements can be nested within If statements, as long as the close of an inner If statement does not appear after the close of an outer If statement. All If statements must begin and end within a given routine. (You cannot distribute parts of an If statement over two or more routines.)

The following example is not realistic, but is used only for illustration. The first line presents the user with a confirmation box. If the user clicks the OK button, then the OK system variable is set to 1 and the ALERT following the If statement is executed. If the user clicks the Cancel button, the OK system variable is set to 0 and the ALERT following the Else statement is executed.

```
CONFIRM ("Press OK or Cancel.")      L240    ` Get a response
If (OK = 1)                                  ` If the user pressed OK
  ALERT ("You pressed OK.")          L239    ` The (OK = 1) was TRUE
Else                                         ` The user pressed Cancel
                                             ` This Else is optional
  ALERT ("You pressed Cancel.")      L239    ` The (OK = 1) was FALSE
End if                                       ` Always need an End if
```

Controlling Procedure Flow

13/9/98

This section describes stater
including conditional execu
description of the format us

if ((field = " ") | (field = "NR"))

Test 1 Test 2

Each text must be enclosed in brackets to function.

ERROR ARISES otherwise

If...Else...End if

If (*Boolean*)
    statement(s)
Else
    statement(s)
End if

Parameter	Type
Boolean	Boolean

If...Else...End if is used to cor
If *Boolean* is TRUE, the nex ed.

The Else and the *statement(s*
the *statement(s)* following it

The End if indicates the end of the If test.

If statements can be nested within If statements, as long as the close of an inner If statement does not appear after the close of an outer If statement. All If statements must begin and end within a given routine. (You cannot distribute parts of an If statement over two or more routines.)

The following example is not realistic, but is used only for illustration. The first line presents the user with a confirmation box. If the user clicks the OK button, then the OK system variable is set to 1 and the ALERT following the If statement is executed. If the user clicks the Cancel button, the OK system variable is set to 0 and the ALERT following the Else statement is executed.

```
CONFIRM ("Press OK or Cancel.")      L240   ` Get a response
If (OK = 1)                                 ` If the user pressed OK
  ALERT ("You pressed OK.")          L239   ` The (OK = 1) was TRUE
Else                                        ` The user pressed Cancel
                                            ` This Else is optional
  ALERT ("You pressed Cancel.")      L239   ` The (OK = 1) was FALSE
End if                                      ` Always need an End if
```

Case of...Else...End case

L30

examples
L145
L148
L154
L178 → L181
L236
L261
L296
L336 (days)
L368 (inuse)

```
Case of
    : (case)
        statement(s)
    : (case)
        statement(s)
:
    Else
        statement(s)
End case
```

Parameter	Type	Description
case	Boolean	Test expression

Case of evaluates a series of *cases*. Case of executes the *statement(s)* belonging to the first and only the first TRUE *case* it encounters, even if a subsequent *case* is TRUE. Procedure execution continues with the statement following End case.

An Else can be included as the last test before the End case. The *statement(s)* following Else are executed only if all the *cases* are FALSE.

The following example is a common way to test for the execution phases of a layout procedure, using Case of...End case to test for each phase.

```
Case of
  : (Before)                    L178      ` Before the layout is displayed
    If (Entry date = !00/00/00!)          ` If it is a new record...
      Entry date := Current date  L338    ` Set the current date
    End if
  : (During)                    L179      ` When the user does something
    Case of                               ` This 'nested' Case of will test user actions
      : (Modified (Field1))     L152      ` If the field was modified...
        Do Stuff
      : (Modified (Field2))     L152
        Do Other Stuff
    End case                              ` End the nested case
  : (After)                     L180      ` When the user accepts the record
    Post Tran                             ` Global to post the transaction
End case
```

While...End while

L 32

While (*Boolean*)
    statement(s)
End while

Parameter	Type	Description
Boolean	Boolean	Test expression

While...End while is a loop that executes the *statement(s)* as long as *Boolean* is TRUE. The value of *Boolean* is tested each time though the loop and is typically set by the *statement(s);* otherwise, the loop will continue forever.

You can nest While statements within While statements, as long as the close of an inner While statement does not appear after the close of an outer While statement.

While loops and Repeat loops are very similar. While loops test the value of *Boolean* at the beginning of the loop, and Repeat loops test the value at the end. Use a While loop if the loop should never be executed (not even once) if *Boolean* is FALSE.

Because *Boolean* must be tested for every cycle of the loop, While loops are necessarily slower than For loops.

The following example lets the user add records to a database. First it presents a confirmation dialog box, asking the user if they want to add records. If the user clicks the OK button, the OK system variable is set to 1 and the While loop is entered. From then on, the loop is executed each time the user accepts a new record, since accepting a record also sets the OK system variable to 1. If the user cancels a record, the OK system variable is set to 0 and the loop ends.

```
L240  CONFIRM ("Do you want to add new records?")     ` Ask the user
      While (OK = 1)    L383                          ` Loop while OK = 1
L191    ADD RECORD
      End while
```

Repeat...Until

L33

Repeat
    statement(s)
Until (*Boolean*)

Parameter	Type	Description
Boolean	Boolean	Test expression

Repeat is a loop that executes the *statement(s)* until *Boolean* is FALSE. The value of *Boolean* is tested each time through the loop and is typically set by the *statement(s);* otherwise, the loop will continue forever.

Repeat differs from While in that it always executes the loop once, whereas if *Boolean* is FALSE, While does not execute the loop at all. Use a Repeat loop when you are depending on one of the statements executed to affect the value of *Boolean*.

Because *Boolean* must be tested for every cycle of the loop, Repeat loops are necessarily slower than For loops.

The following example lets the user add records to a database. The loop is executed each time the user accepts a new record, since accepting a record sets the OK system variable to 1. If the user cancels a record, the OK system variable is set to 0 and the loop ends.

```
Repeat
  ADD RECORD          L141
Until (OK = 0)                    Loop until OK = 0
```

For...End for

L33

For (*counter*; *start value*; *end value*; *{increment}*)
    statement(s)
End for

Parameter	Type	Description
counter	Variable (num)	Variable to use as counter
start value	Number	Value with which to start counter
end value	Number	Value of counter to end loop
increment	Number	Increment amount

For...End for is a loop structure that executes *statement(s)* a specified number of times. The *counter* parameter is used to control the loop, and its value is often used by the statements inside the loop. The *counter* is initialized to *start value* and is incremented after each execution of the loop, by the optional *increment*. If *increment* is not specified, *counter* is incremented by 1. The loop ends when *counter* is greater than *end value*.

If *increment* is specified, and it is a negative number, the *counter* is decremented instead of incremented. In this case, the loop will end when *counter* is less than *end value*.

The *counter* must be a numeric global or local variable. It cannot be an element of an array. The *start value, end value,* and *increment* parameters do not need to be whole numbers.

The *counter* may be modified by statements within the loop.

A For loop is faster than other types of loops.

The following example simply loops from 1 to 100, displaying the current value of the counter $i in a message.

```
For ($i; 1; 100)                          ` Loop 100 times
  Message (String ($i))                   ` Display the counter
End for
```

L243 • L332

The following example loops from 1 to 100, using an increment of .5. Again, the counter is displayed in a message.

```
For ($i; 1; 100; .5)                      ` Loop 200 times
  Message (String ($i))                   ` Display the counter
End for
```

L243 • L332

PART III

THE COMMANDS

PART III - The Commands

Part III of this manual describes the commands in the 4th DIMENSION language. Part III is divided into eight chapters:

Chapter 11—Command Descriptions and Parameters

This chapter describes the format of command descriptions in Part III.

Chapter 12—Setting Defaults

This chapter defines the commands that are used to set the default file and layouts.

Chapter 13—Data Entry and Reporting

This chapter defines the commands that are used for data entry and creating reports. These commands present information to the user both on screen and when printing.

Chapter 14—Managing Data

This chapter defines commands that manage data. Data management includes searching, sorting, importing, exporting, and working with subrecords.

Chapter 15—User Interface

This chapter defines commands used to manage the user interface. The user interface includes messages, windows, menus, and sound.

Chapter 16—Advanced Commands

This chapter defines commands for advanced database design. Advanced design includes managing sets, multi-user databases, transactions, documents, serial communication, and passwords.

Chapter 17—Functions

This chapter defines all math, string, date, and time functions.

Chapter 18—Miscellaneous Commands

This chapter defines commands for working with variables, managing arrays, controlling the execution of procedures, and getting information about data objects.

CHAPTER 11

COMMAND DESCRIPTIONS AND PARAMETERS

COMMAND DESCRIPTIONS AND PARAMETERS

This chapter explains the format used to describe commands in Part III. It also describes the rules you must follow when specifying parameters to commands.

Command Descriptions

Each command description has five parts: "Description Heading"; "Command Syntax"; "Parameters" (if any); "Description"; and "Example." A command description may also have a multi-user description. Figure 11-1 shows the description of a command as it appears in this manual.

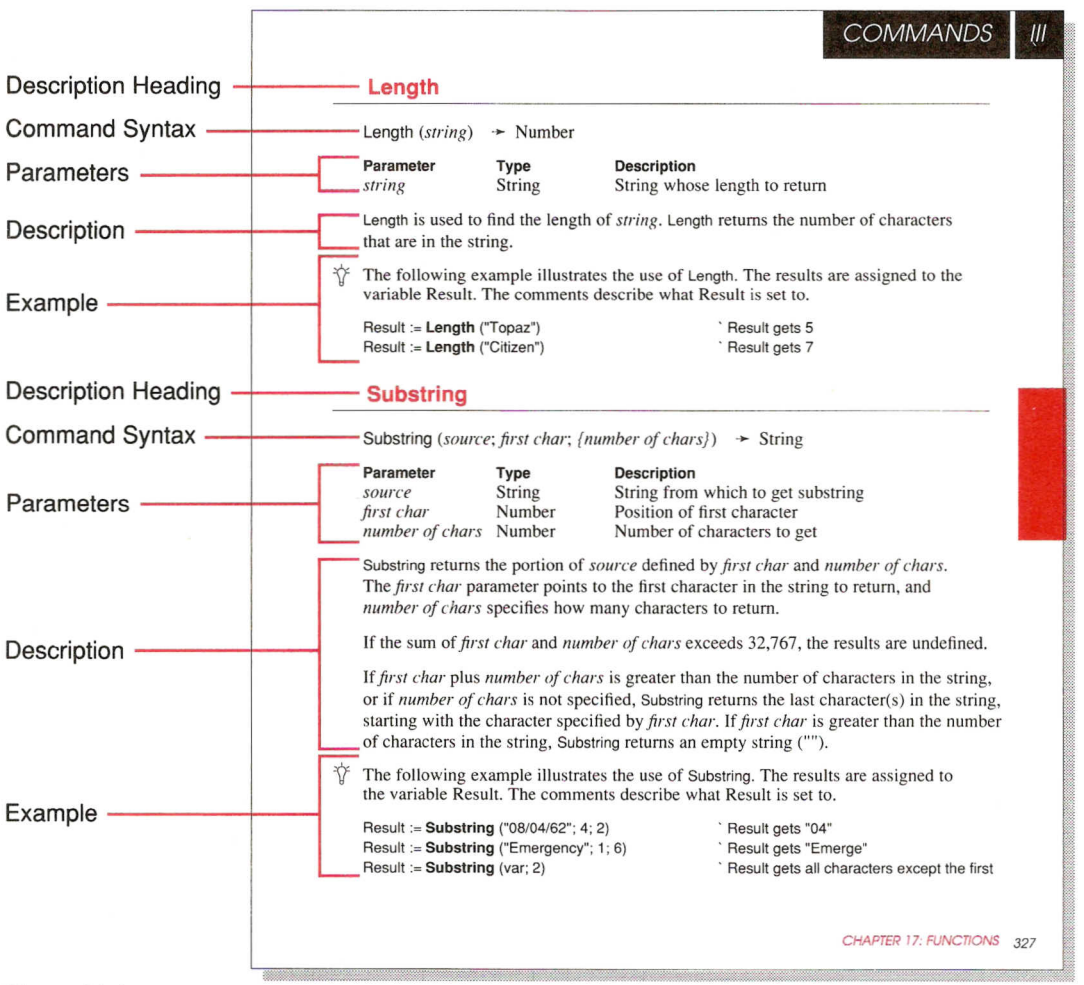


Figure 11-1
Command description as it appears in this manual

The Description Heading

The description heading gives the name of the command name that is described. Similar commands are grouped under one heading. In this case, each command name is given.

The Command Syntax

The command syntax specifies all the possible forms for each command in the description. At the beginning of each command description, each form of the command is shown in a syntax diagram. The name of each command is followed by the command's parameters. If the command is a function, the parameters are followed by an arrow and then the data type of the value that the command returns. Figure 11-2 shows an example of a syntax diagram.

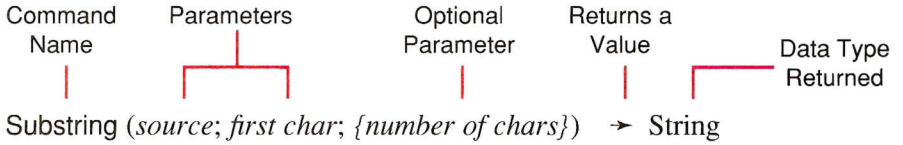


Figure 11-2
A syntax diagram

The Parameters

A parameter is data passed to a command. Parameters are specified with descriptive names printed in italic.

Below the syntax diagram, each parameter is listed in a parameter table, with its type and a short description. If more than one command is described and the parameters to each command are the same, there is only one parameter table. If more than one command is described and the parameters to each command are different, there is a parameter table following each command's syntax diagram. If a command has more than one form, there is a parameter table following each syntax diagram for the command. Figure 11-3 shows how the parameters are described for a command.

Substring (*source*; *first char*; {*number of chars*}) → String

Parameter	Type	Description
source	String	String from which to get substring
first char	Number	Position of first character
number of chars	Number	Number of characters to get

Figure 11-3
Parameters for a command

If a parameter is optional, the parameter name is enclosed in curly braces ({...}) in the command syntax diagram. (See *number of chars* in Figure 11-3.)

If a parameter can be repeated (always optionally), the parameter is followed by an ellipsis (...). The ellipsis is then followed by the parameter name and a number indicating the number of times the parameter can be repeated. If a parameter can be repeated an unlimited number of times, the number is N. For example,

SAVE VARIABLE (*document*; *variable1* {;...; *variableN*})

The Description, Example, and Multi-user Parts

There are three parts which describe how to use the command: the description; the example; and the multi-user description.

The description immediately follows the parameters.

The example is indicated with a marker like the one on this paragraph, and follows the description. Most commands have an example. Some commands have more than one example. If there is more than one example, each example is separately indicated with a marker.

Some commands have special information regarding their use in a multi-user environment. A multi-user description is indicated with a marker like the one on this paragraph. A multi-user description follows the example.

Parameters to Commands

This section gives information about passing parameters to commands.

Specifying Parameters

When you specify parameters to commands, there are a number of rules you must follow. Here is the list of rules:

- Parameters are surrounded by parentheses. For example, if [My File] is a parameter to ADD RECORD, it is specified this way:

  ADD RECORD ([My File]) L141

- If a command has more than one parameter, the parameters are separated by semicolons. This includes parameters that repeat. For example, if [My File] and "Layout In" are parameters to INPUT LAYOUT, they are separated like this:

  INPUT LAYOUT ([My File]; "Layout In") L137

- If an optional parameter is omitted, then any associated semicolon is also omitted. For example, the second parameter for the Request function is optional. Request without the second parameter is written like this:

  Request (String1) L241

  With the second parameter, the two parameters are separated by a semicolon, and it is written like this:

  Request (String1; String2) L241

- If there are no parameters, the parentheses are omitted. This is true both for commands that never take parameters and for commands where the parameters are optional. For example, ADD RECORD with a file parameter is written this way:

  ADD RECORD ([File]) L141

  If the file parameter is omitted, the parentheses are also omitted, and the command is written this way:

  ADD RECORD L141

Parameter Types

Each parameter that is passed to a command has a specific type. Table 11-1 lists the parameter types. It also gives examples, if appropriate, of each type as a constant, an expression, and as a returned value from a command.

Table 11-1
Parameter Types

Symbol	Description	Example of a Constant	Example of an Expression	Example of a Command
Array	Array	n/a	ArrayName	n/a
Boolean	Boolean expression	n/a	x < y	**True** L349
Date	Date expression	!12/25/89!	!12/25/89! + 365	**Current date** L335
Docref	Document reference	n/a	Var	**Open document** ("") L300
Field	Field	n/a	[File1]Field2	**Field** (1; 2)» L321
File	File	n/a	[File1]	**File** (1)» L320
Number	Numeric expression	1.5	Var + 10.5	**Num** ("123.1") L341
Picture	Picture expression	n/a	PictureName + 5	n/a
Pointer	Pointer expression	n/a	»Name	**Get pointer** ("Name") L370
Statement	Logical line of code	Var := 10	n/a	n/a
String	String expression	"Hello"	Var + "abc"	**String** (123) L332
Subfield	Subfield	n/a	[File1]Subfile'Subfield	n/a
Subfile	Subfile	n/a	[File1]Subfile	n/a
Time	Time expression	†12:05:30†	†12:05:30† + †01:00:00†	**Current time** L338
Variable	Variable	n/a	VarName	**Last object**» ?

SETTING DEFAULTS

CHAPTER 12

NEW DATA FILE (document)
OPEN DATA FILE (document)

SETTING DEFAULTS

The commands in this chapter set the default file and layouts that will be used by other commands. Using these commands is equivalent to selecting files and layouts in the User environment.

Setting the Default File

DEFAULT FILE

Many commands in the language require you to specify a file. You can specify the file as the first parameter to the command—or you can set a default file with the DEFAULT FILE command. The examples in this manual alternate between using a default file and specifying the file in the commands.

DEFAULT FILE

L 69

DEFAULT FILE (*file*)

Parameter	Type	Description
file	File	File to set as the default

DEFAULT FILE sets *file* as the default file.

There is no default file until the DEFAULT FILE command is executed. After a default file has been set, any command that omits the file parameter will operate on the default file. For example, consider this command:

INPUT LAYOUT ([File]; "layout")

If the default file is first set to [File], the same command could be written this way:

INPUT LAYOUT ("layout")

Setting a default file has two uses. The first is to simplify and clarify procedures. Consider this example:

SEARCH ([Customers]; [Customers]Name = "Acme")
INPUT LAYOUT ([Customers]; "Add recs")
MODIFY RECORD ([Customers])

Specifying the default file results in a clearer procedure:

DEFAULT FILE ([Customers])
SEARCH ([Customers]Name = "Acme")
INPUT LAYOUT ("Add recs")
MODIFY RECORD

The other reason for setting the default file is to create code that is not file specific. Doing this allows the same code to operate on different files. For example, if you needed a procedure to search, sort, and report on two files, you could write it this way:

```
SEARCH ([File1])
SORT ([File1])
REPORT ([File1]; "")
SEARCH ([File2])
SORT ([File2])
REPORT ([File2]; "")
```

The same routine could be written as a global procedure called *DoReport:*

```
SEARCH
SORT
REPORT ("")
```

The procedure would then be called by the following code:

```
DEFAULT FILE ([File1])
DoReport
DEFAULT FILE ([File2])
DoReport
```

You can also use pointers to files to write code that is not file specific. For more information on this technique, see the section "Determining the Database Structure," in Chapter 16.

DEFAULT FILE does not allow the omission of filenames when referring to fields. For example,

```
[My File]My Field := "A string"
```

could not be written as

```
DEFAULT FILE ([My File])
My Field := "A string"
```

simply because a default file had been set. However, you *can* omit the filename when referring to fields in the file procedure, layout procedures, and scripts that belong to the file.

In 4th DIMENSION, all files are "open" and ready for use. DEFAULT FILE does not "open" a file, set a current file, or prepare the file for input or output. DEFAULT FILE is simply a convenience during programming, to reduce the amount of typing and make the code easier to read.

The following example first shows code without the use of DEFAULT FILE. The example then shows the same code with the use of DEFAULT FILE. The code is a loop commonly used to add new records to a database. The commands INPUT LAYOUT and ADD RECORD both need a file as the first parameter.

```
INPUT LAYOUT ([Customers]; "Add Recs")
Repeat
  ADD RECORD ([Customers])
Until (OK = 0)
```

Specifying the default file results in this code:

```
DEFAULT FILE ([Customers])
INPUT LAYOUT ("Add Recs")
Repeat
  ADD RECORD
Until (OK = 0)
```

Specifying Layouts

INPUT LAYOUT OUTPUT LAYOUT

This section describes the commands used for specifying the input and output layout. Layouts are used extensively in 4th DIMENSION. They are used for data entry, reporting, importing, exporting, and creating a user interface.

Input layouts are associated with commands that display only one record at a time, generally for data entry. Output layouts are associated with commands that display multiple records, usually in a list style, either on screen or to a printer.

The INPUT LAYOUT and OUTPUT LAYOUT commands specify which layouts will be used for each file. Each file has a current input layout and a current output layout—they are used by any command that requires a layout but does not specify one. The layouts are designated in the Design environment by the letter *I* or *O* in the list of layouts. The layouts specified in the Design environment will be used if you do not specify different ones with INPUT LAYOUT or OUTPUT LAYOUT.

Both INPUT LAYOUT and OUTPUT LAYOUT simply designate which layouts to use; they do not actually display the layouts.

INPUT LAYOUT

L 69 3

INPUT LAYOUT ({*file*}; *layout*)

Parameter	Type	Description
file	File	File for which to set the input layout
layout	String	Layout name

INPUT LAYOUT sets the current input layout for *file* to *layout*. Each file has its own input layout. The layout must belong to *file*. (For information on creating layouts, see the *4th DIMENSION Design Reference*.) INPUT LAYOUT does not display the layout; it just designates which layout is displayed or used by another command.

The default input layout is defined in the Design environment and is identified by the letter *I* next to the layout name in the list of layouts. The default layout is used if INPUT LAYOUT does not specify an input layout.

Input layouts are displayed by a number of commands. These commands are generally used to allow the user to enter new data or modify old data.

The following commands all immediately display an input layout:

ADD RECORD	DIALOG	MODIFY RECORD
ADD SUBRECORD	DISPLAY RECORD	MODIFY SUBRECORD

Each of the following commands displays a list of records, using the output layout. Each command then allows the user to double-click on a record, which displays the input layout.

DISPLAY SELECTION	MODIFY SELECTION

An input layout is also displayed if the user double-clicks in an included layout. In this case, you must set the input layout in the Design environment, by assigning the input layout (full-page layout) when creating the included area.

The input layout is also used by the following import commands:

IMPORT DIF	IMPORT SYLK	IMPORT TEXT

The following example shows a typical use of INPUT LAYOUT. Note that although the INPUT LAYOUT command appears immediately before the input layout is used, this is not required, and in fact the command may be executed in a completely different procedure.

```
DEFAULT FILE ([Companies])        ` Set the default file
INPUT LAYOUT ("New Comp")         ` Select the layout for new companies
ADD RECORD                        ` Add a new company
```

OUTPUT LAYOUT L 69.

OUTPUT LAYOUT ({*file*}; *layout*)

Parameter	Type	Description
file	File	File for which to set the output layout
layout	String	Layout name

OUTPUT LAYOUT sets the current output layout for *file* to *layout*. Each file has its own output layout. The layout must belong to *file*. (For information on creating layouts, see the *4th DIMENSION Design Reference*.) OUTPUT LAYOUT does not display the layout; it just designates which layout is printed, displayed, or used by another command.

The default output layout is defined in the Design environment and is identified by the letter *O* next to the layout name in the list of layouts. The default layout is used if OUTPUT LAYOUT does not specify an output layout.

Output layouts are used by three groups of commands. One group displays a list of records on screen, another group generates reports, and the third group exports data.

Each of the following commands displays a list of records, using an output layout:

DISPLAY SELECTION MODIFY SELECTION

An output layout can also be displayed in an included layout. In this case, you must set the output layout in the Design environment, by assigning the output layout (multi-line layout) when creating the included area.

You use the output layout when creating reports with the following commands:

PRINT LABEL PRINT SELECTION

Each of the following export commands also uses the output layout:

EXPORT DIF EXPORT SYLK EXPORT TEXT

The following example shows a typical use of OUTPUT LAYOUT. Note that although the OUTPUT LAYOUT command appears in the example immediately before the output layout is used, this is not required, and in fact the command may be executed in a completely different procedure.

```
INPUT LAYOUT ([Parts]; "Parts In")        ` Select the input layout
OUTPUT LAYOUT ([Parts]; "Parts List")     ` Select the output layout
MODIFY SELECTION ([Parts])                ` This command uses both layouts
```

DATA ENTRY AND REPORTING

CHAPTER 13

DATA ENTRY AND REPORTING

You'll use the commands in this chapter often when creating a custom database. They allow the user to enter data and display it on screen, and to print reports. These commands revolve around layouts. A layout is the primary tool used for entering data and printing reports.

Performing Data Entry and Displaying Records

ADD RECORD	DISPLAY SELECTION	DISPLAY RECORD
MODIFY RECORD	MODIFY SELECTION	

The commands in this section display records on-screen, both for data entry and for viewing in a list.

Data entry is one of the fundamental roles of a database. Data entry is the process by which a user enters data into the database. The first two commands in this section are the most common commands used for data entry. They act just like the New Record and Modify Record menu items in the User environment.

During data entry, input layouts are used to enter information. Input layouts can have multiple pages, with each page displaying different data, or the same data in different ways. For more information on input layouts, see the *4th DIMENSION Design Reference*.

Records in 4th DIMENSION are commonly displayed in a list, using the current output layout. An example is the list of records displayed in the User environment. The list of records allows the user to scroll through the records, examining and selecting them as desired. The two selection commands are used to display the records in this list style.

The other command in this section, DISPLAY RECORD, is used to display a single record. This command uses the input layout to display the record.

Changing the Current Record During Data Entry

The following discussion is of interest to experienced 4th DIMENSION developers.

When a record is displayed for data entry with either the ADD RECORD, MODIFY RECORD, or MODIFY SELECTION command, the displayed record is the current record. If you change to a different current record using a command (such as NEXT RECORD), you must first execute the SAVE RECORD command if you need to save any changes that were made to the displayed record. Note that pressing a Next Record button, or any other automatic action button, saves the record automatically, and you therefore do not need to use the SAVE RECORD command.

Changing to a new current record with a command does not execute a new Before phase for the layout. Moving to a new record with an automatic button *does* execute a new Before phase.

ADD RECORD L69 11
MODIFY RECORD L69 10

ADD RECORD ({*file*}; {*})
MODIFY RECORD ({*file*}; {*})

Parameter	Type	Description
file	File	File to use for data entry
*		Hide scroll bars and size box

ADD RECORD lets the user add a new record to the database. ADD RECORD creates a new record for *file*, makes the new record the current record, and displays the current input layout. After the user has accepted the new record, the new record is the only record in the current selection.

Figure 13-1 shows a typical layout displayed for data entry.

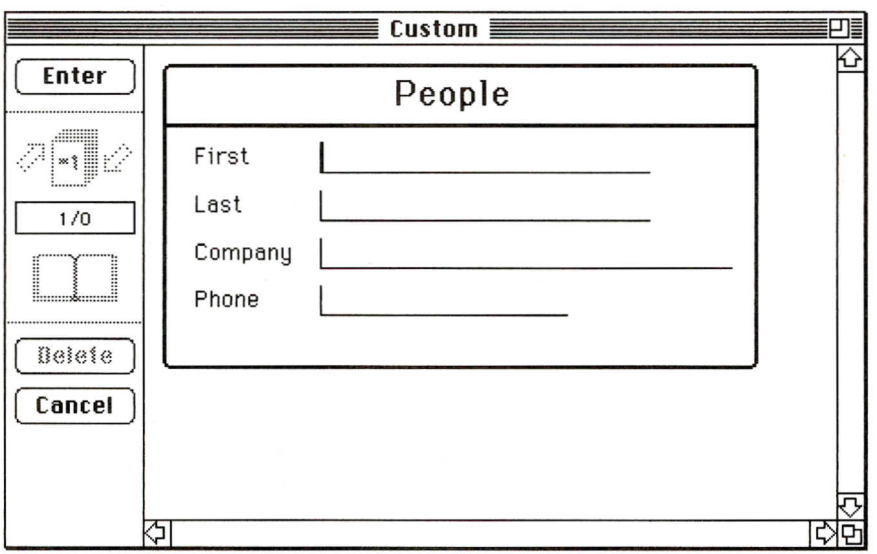


Figure 13-1
An input layout displayed by the ADD RECORD command

The layout is displayed, with either command, in the frontmost window. The window has scroll bars and a size box. Specifying the optional asterisk causes the window to be drawn without scroll bars or a size box.

ADD RECORD displays the layout only until the user accepts or cancels the record. You must execute the command once for each record the user enters.

MODIFY RECORD lets the user modify a record in the input layout. MODIFY RECORD gets the current record of *file* from disk and displays the record in the current input layout. If there is no current record, then MODIFY RECORD does nothing. MODIFY RECORD does not affect the current selection.

If the layout contains buttons for moving within the selection of records, MODIFY RECORD lets the user use them to modify records and move to other records.

With either command, the record is saved (accepted) if the user clicks an Accept button or presses the Enter key, or if the ACCEPT command is executed. Accepting the record sets the OK system variable to 1.

The record is not saved (canceled) if the user clicks a Cancel button or presses the "cancel" key combination (Command-.), or if the CANCEL command is executed. Canceling sets the OK system variable to 0. Even when canceled, the record remains in memory and can be saved if SAVE RECORD is executed before the current record pointer is changed. The OK system variable is set only after the record is accepted or canceled.

If you are using MODIFY RECORD and the user does not change any of the data in the record, the record is not considered modified, and accepting the record does not cause it to be saved again. Actions like changing variables, checking check boxes, and selecting radio buttons do not qualify as modifications; only changing data in a field causes the record to be saved.

The layout procedure execution cycle is started if a layout procedure exists for the layout. Scripts that exist for the layout may also be executed, depending on the user's actions. For more information on the execution cycle, see Chapter 5 in Part I, and "Monitoring the Layout Execution Cycle" in Chapter 13.

The following example is a loop commonly used to add new records to a database.

```
INPUT LAYOUT ([Customers]; "Cust In")      ` Set the input layout for [Customers] file
Repeat                                     ` Loop until the user cancels
  ADD RECORD ([Customers])                 ` Add a new record to the [Customers] file
Until (OK = 0)                             ` Until the user cancels and OK = 0
```

The following example searches the database for a customer. Depending on the results of the search, one of two things may happen. If no customer is found, then the user is allowed to add a new customer with ADD RECORD. If one or more customers are found, the user is presented with each customer's record for modification, with MODIFY RECORD.

```
DEFAULT FILE ([Customers])                    ` Set the default file
INPUT LAYOUT ("Input1")                       ` Set the input layout
OUTPUT LAYOUT ("Output1")                     ` Set the output layout
vNo := Request ("Enter customer number")      ` Get the customer number
SEARCH ([Customers]CustNo = Num (vNo))        ` Search for the customer record
If (Records in selection = 0)                 ` If no customer is found...
  ADD RECORD                                  ` add a new customer
Else
  MODIFY RECORD                               ` Allow the user to modify the record
End case
```

ADD RECORD will perform as described in a multi-user database, except when the new record is accepted. When the new record is accepted, and the After phase is executed, the entire database becomes locked for all other users. Since the database is locked, the other users cannot save any records until the After phase is done. For this reason, it is important that the After phase code of the layout procedure and the scripts be as short as possible.

MODIFY RECORD will not modify a record that is locked. Instead, MODIFY RECORD will display a dialog box informing the user that the record is in use. For more information on using MODIFY RECORD in a multi-user database, see "Managing Multi-user Databases," in Chapter 16.

DISPLAY SELECTION
MODIFY SELECTION

L69 18

DISPLAY SELECTION ({*file*}; {*}*)
MODIFY SELECTION ({*file*}; {*}*)

Parameter	Type	Description
file	File	File to display
*		Use output layout for one record and hide scroll bars in the input layout

DISPLAY SELECTION and MODIFY SELECTION display the current selection of *file*, using the current output layout. The records are displayed in a scrollable list similar to the User environment's output list. If the user double-clicks a record, the record is displayed in the current input layout. The list is displayed in the frontmost window.

Figure 13-2 shows an output layout displayed by the DISPLAY SELECTION or MODIFY SELECTION command.

Employees: 24 of 24

Last Name	First Name	Start Date	Salary	Title
Adler	Frank	4/7/89	$101,586	Engineer
Ambler	Winifred	11/30/87	$91,586	Engineer
Anderson	Nathan	10/19/80	$28,770	Salesperson
Andrews	Michael	4/19/85	$85,864	Designer
Ballard	John	1/28/85	$82,868	Engineer
Bentley	Alice	3/6/79	$29,250	Engineer
Campbell	Arnold	5/13/89	$12,286	Salesperson
Donaldson	Bill	11/3/83	$71,586	Salesperson
Frankheimer	George	7/23/89	$122,870	Salesperson
Franklin	Marsha	5/3/84	$71,986	Salesperson
Johnson	Jasper	7/11/83	$41,986	Engineer
Johnson	Tom	3/15/79	$29,250	Designer
Jones	Samuel	11/11/82	$32,186	Salesperson
Newton	Kendall	6/25/89	$119,870	Salesperson
Ranklin	Anthony	6/8/84	$73,086	Designer

Done

Figure 13-2
A typical record listing using the output layout

After DISPLAY SELECTION or MODIFY SELECTION is executed, there may not be a current record. Use a command such as FIRST RECORD or LAST RECORD to select one.

MODIFY SELECTION allows the user to modify a record when in the input layout; DISPLAY SELECTION does not allow the user to modify a record when in the input layout.

If the selection contains only one record and the optional asterisk is not used, the record appears in the input layout instead of the output layout. If the asterisk is specified, a one-record selection is displayed, using the output layout. If the asterisk is specified and the user displays the record in the input layout, the scroll bars will be hidden.

A button labeled Done is automatically included at the bottom of the list. Clicking this button exits the command. Custom buttons may be used instead; you can put the buttons in the Footer area of the output layout. You can use an Accept or Cancel button to exit.

The user can scroll through the selection and click a record to select the record. If the user clicks a different record, the first record is deselected and the second record is selected. A user can select a group of contiguous records, by clicking the first record and Shift-clicking the last record. To select records that are not adjacent, the user can Command-click each desired record.

After DISPLAY SELECTION or MODIFY SELECTION is executed, the records that the user selected are returned in a set named UserSet. There is only one UserSet for the entire database. The set is associated with the last DISPLAY SELECTION or MODIFY SELECTION command. For more information on the UserSet, see "Managing Sets," in Chapter 16.

If a layout procedure or script exists, the Before phase is executed before the layout is displayed, then the In Header phase is executed, and then the Before and During phases are executed simultaneously, once for each record that is displayed. If the user clicks a button, chooses a menu, or double-clicks a record, a During phase for the output layout procedure is executed.

The following example selects all the records in the [People] file. It then uses the DISPLAY SELECTION command to display the records, and allows the user to select the records that he or she would like to print. Finally, it selects the records with the USE SET command, and prints them with PRINT SELECTION.

```
DEFAULT FILE ([People])          ` Set the default file
ALL RECORDS                      ` Select all records
DISPLAY SELECTION (*)            ` Display the records
USE SET ("UserSet")              ` Use only the records that the user picked
PRINT SELECTION                  ` Print the records that the user picked
```

The following example shows all of the tests needed to completely monitor the execution cycle of a DISPLAY SELECTION or MODIFY SELECTION command. This procedure is the output layout procedure for the displayed layout. The tests must be executed in the order shown.

Note that the last test for During allows you to check the record that the user just double-clicked. To make this test work properly, you must use custom buttons in the Footer area. Otherwise, the default Done button will generate a During phase, and clicking it will only be trapped by the test for the During phase.

In the statements after the test for the During phase, you could change the input layout depending on the information in the record.

```
Case of
  : (Before & During)
    ` Each record is being displayed
  : (Before)
    ` The output list has not yet been displayed
  : (In header)
    ` The header is being displayed
  : (Button = 1)
    ` A button was selected.
    ` You must do this test for each of the buttons in the Footer area.
  : (Menu selected # 0)
    ` A menu was selected
  : (During)
    ` A record was double-clicked.
    ` You may change the input layout here.
    ` You may also cancel the command, and the double-clicked record will be current.
End case
```

MODIFY SELECTION will not modify a record that is locked. Instead, it will display a dialog box informing the user that the record is in use. When displaying records in the list, MODIFY SELECTION automatically sets the file to read-only, to prevent records from being locked for other users. For more information on using MODIFY SELECTION in a multi-user database, see "Managing Multi-User Databases," in Chapter 16.

DISPLAY RECORD

DISPLAY RECORD ({*file*})

Parameter	Type	Description
file	File	File from which to display the record

DISPLAY RECORD displays the current record of *file*, using the current input layout. The record is displayed only until an event redraws the window. Such an event might be the execution of an ADD RECORD command, returning to an input layout, or returning to the menu bar. DISPLAY RECORD does nothing if there is no current record.

DISPLAY RECORD is often used to display custom progress messages. It can also be used to generate a free-running slide show.

If a layout procedure or script exists, the Before phase is executed.

The following example displays a series of records as a slide show. (The records contain pictures.)

```
DEFAULT FILE ([Demo])                      ` Set the default file
ALL RECORDS                                ` Select all of the records
INPUT LAYOUT ("Display")                   ` Set the layout to use for display
For ($i; 1; Records in selection)          ` Loop through all of the records
  DISPLAY RECORD                           ` Display a record
  ` Pause display for 3 seconds.
  $Now := Current time                     ` Get the current time
  While (Abs (Current time – $Now) < 3)    ` Loop for about 3 seconds
  End while
  NEXT RECORD                              ` Move to the next record
End for
```

Managing Layout Pages

FIRST PAGE	NEXT PAGE	GOTO PAGE
LAST PAGE	PREVIOUS PAGE	Layout page

The commands in this section allow you to display different layout pages. There are automatic actions for buttons which perform the same tasks as the FIRST PAGE, LAST PAGE, NEXT PAGE, and PREVIOUS PAGE commands. Whenever appropriate, use the automatic actions button instead of these commands.

Page commands can be used only in an input layout. Output layouts use only the first page. A layout always has at least one page, the first page.

It's important to realize that regardless of the number of pages a layout has, only one layout procedure exists for each layout. You can use the Layout page command to find out which page is being displayed.

FIRST PAGE

FIRST PAGE

FIRST PAGE changes the currently displayed layout page to the first layout page. If a layout is not being displayed, or the first layout page is already displayed, FIRST PAGE does nothing.

The following example is a one-line procedure called from a menu item. It displays the first layout page.

FIRST PAGE

LAST PAGE

LAST PAGE

LAST PAGE changes the currently displayed layout page to the last layout page. If a layout is not being displayed, or the last layout page is already displayed, LAST PAGE does nothing.

The following example is a one-line procedure called from a menu item. It displays the last layout page.

LAST PAGE

NEXT PAGE

NEXT PAGE

NEXT PAGE changes the currently displayed layout page to the next layout page. If a layout is not being displayed, or the last layout page is being displayed, NEXT PAGE does nothing.

The following example is a one-line procedure called from a menu item. It displays the layout page that follows the one currently displayed.

NEXT PAGE

PREVIOUS PAGE

PREVIOUS PAGE

PREVIOUS PAGE changes the currently displayed layout page to the preceding layout page. If a layout is not being displayed, or the first layout page is being displayed, PREVIOUS PAGE does nothing.

The following example is a one-line procedure called from a menu item. It displays the layout page preceding the one currently displayed.

PREVIOUS PAGE

GOTO PAGE

GOTO PAGE (*page number*)

Parameter	Type	Description
page number	Number	Layout page to display

GOTO PAGE changes the currently displayed layout page to the layout page specified by *page number*. If a layout is not being displayed, GOTO PAGE does nothing. If *page number* is greater than the number of pages, the last page is displayed. If *page number* is less than the number of pages, the first page is displayed. You can use GOTO PAGE in a script for a button to take the user to a specific page.

The following example is a script for a button. It displays a specific page, page 3.

GOTO PAGE (3)

Layout page

Layout page → Number

Layout page returns the number of the currently displayed layout page. Since there is only one procedure for the entire layout, this function can be used in input layout procedures to tell which page is currently being displayed.

The following example is a portion of an input layout procedure. It tests for the layout page and calls a global procedure appropriate for that page.

```
Case of
  : (Layout page = 1)
    Page1stuff
  : (Layout page = 2)
    Page2stuff
  : (Layout page = 3)
    Page3stuff
End case
```

Using Data Entry Areas

GET HIGHLIGHT	GOTO AREA	REJECT
HIGHLIGHT TEXT	Last area	
INVERT BACKGROUND	Modified	

The commands in this section affect layout areas used for data entry: fields and variables. These commands work only when a layout is being used for data entry. They allow you to get highlighted text from a data entry area, to highlight text in a data entry area, to move to a specific data entry area, and to test whether a field has been modified.

GET HIGHLIGHT

Note Page 379

GET HIGHLIGHT (*text object*; *first*; *last*)

Parameter	Type	Description
text object	Field or variable	Text object argument
first	Variable	First position of highlight
last	Variable	Last position of highlight

GET HIGHLIGHT is used to find out what text is currently highlighted. The text may be highlighted by the user or by the HIGHLIGHT TEXT command.

The variable *first* is assigned the position of the first highlighted character. The variable *last* is assigned the position of the last highlighted character plus one. If *first* and *last* are equal, the user has not selected any text and the insertion point is before the character specified by the *first* variable.

The following example gets the highlight positions from a field called Comment. The GET HIGHLIGHT command sets two variables, vFirst and vLast. If Comment is highlighted, as in Figure 13-3, then vFirst is set to 9 and vLast is set to 13.

```
GET HIGHLIGHT (Comments; vFirst; vLast)        ` Get the highlight from comments
```

This is text in a text field.

Figure 13-3
Text highlighted in a field

The following example is the same as the first example except that the field does not have any highlighted text, as in Figure 13-4. In this case, vFirst is set to 11 and vLast is set to 11.

```
GET HIGHLIGHT (Comments; vFirst; vLast)          ` Get the highlight from comments
```

```
This is text in a text field.
```

Figure 13-4
Text insertion point in a field

The following example shows how the highlighted text can be extracted with the Substring function.

```
GET HIGHLIGHT (Comments; vFirst; vLast)          ` Get the highlight from comments
` Get the highlighted text using Substring and put it into My Text
MyText := Substring (Comments; vFirst; vLast – vFirst)
```

HIGHLIGHT TEXT

HIGHLIGHT TEXT (*text object*; *first*; *last*)

Parameter	Type	Description
text object	Field or variable	Text object to highlight
first	Number	First position of highlight
last	Number	Last position of highlight

HIGHLIGHT TEXT highlights a section of the text in *text object*. HIGHLIGHT TEXT will go to *text object* if the cursor is not in *text object*.

First is the first character position to be highlighted, and *last* is the last character plus one to be highlighted. If *first* and *last* are the same, the insertion point is positioned before the character specified by *first*, and no characters are highlighted.

If *last* is greater than the number of characters in *text object*, then all characters between *first* and the end of the text are highlighted.

The following example highlights text in a field called Comments, shown in Figure 13-5.

```
HIGHLIGHT TEXT(Comments; 9; 13)                  ` Highlight the text
```

```
This is text in a text field.
```

Figure 13-5
Highlighting text in a field

The following example positions the insertion point in a field called Comments, shown in Figure 13-6.

```
HIGHLIGHT TEXT(Comments; 11; 11)        ` Position the insertion point
```

This is text in a text field.

Figure 13-6
Positioning the insertion point in a field

INVERT BACKGROUND

INVERT BACKGROUND (*text variable*)

Parameter	Type	Description
text variable	Variable	Text variable to invert

INVERT BACKGROUND is used to invert *text variable* in the layout. INVERT BACKGROUND works only for the currently displayed or printed layout and record. You can use INVERT BACKGROUND when displaying on screen or printing to an ImageWriter printer. The LaserWriter printer will not print an inverted background.

The following example is a script for a variable in an output layout. The script tests the value of a field. If the field is positive, the script does nothing. If the field is negative, the script inverts the display of the variable in the layout.

```
vAmount := [Accounts]Amount             ` Put the value of the field in the variable
If (vAmount < 0)                        ` If it is a negative amount...
  INVERT BACKGROUND (vAmount)           ` invert the background
End if
```

GOTO AREA

GOTO AREA (*data entry area*)

Parameter	Type	Description
data entry area	Field or variable	Field or variable to go to

GOTO AREA is used to move the insertion point to *data entry area* in an input layout. It is equivalent to the user's clicking on or tabbing into the field or variable.

The following example is a script for a button. The button is labeled Change ID. When clicked, it first displays the layout page where the ID field can be changed, and then moves to the ID field.

```
GOTO PAGE (2)                           ` Move to the page with the ID field
GOTO AREA (ID)                          ` Move to the ID field
```

Last area

Last area → Pointer

Last area returns a pointer to the last or current enterable area, in other words, the object that the cursor is in or just left. You can use Last area to perform an action on a layout area without having to know which object is currently selected. Be sure to test that the object is the correct data type, using Type, before performing an action on it.

The following example is a script for a button. The script changes the data in the current object to uppercase. The object must be a text or string data type (type 0 or 2).

```
$p := Last area                                  ` Save the pointer to the last area
If ((Type ($p») = 0) | (Type($p») = 2))          ` If it is a string or text area
  $p» := Uppercase ($p»)                         ` Change the area to uppercase
End if
```

Modified

Modified (*field*) → Boolean

Parameter	Type	Description
field	Field	Field to test

Modified returns TRUE if the user has modified *field* during data entry. A field is considered modified when the user changes the data in the field and leaves the field, by pressing Tab or by clicking another field or a button, or in another area (like a scrollable or external area).

It is usually easier to perform operations in scripts than to use Modified in layout procedures. Since a script is executed when a field is modified, the use of a script is equivalent to using Modified in a layout procedure.

Note that tabbing out of a field does not set Modified to TRUE. The field must have been changed for Modified to be TRUE.

The following example tests if either the Quantity field or the Price field has changed. If either has, then the total is recalculated. Note that the same thing could be accomplished by using the second line as the script for the Quantity field and the Price field.

```
If ((Modified (Quantity) | (Modified (Price))    ` If the user changed either field
  Total := Quantity * Price                      ` Recalculate. This line could be a script.
End if
```

REJECT

REJECT

REJECT (*data entry area*)

Parameter	Type	Description
data entry area	Field or variable	Data entry area to reject

This command is rarely used. You should use the built-in data validation tools before using this command.

REJECT has two forms. The first form has no parameters. It rejects the entire data entry and does not accept the record. The second form rejects only the *data entry area*.

The first form of REJECT is used to prevent the user from accepting a record that is not complete. You can achieve the same result without using REJECT by associating the Enter key with a No Action button and using the ACCEPT and CANCEL commands to accept or cancel the record. It is recommended that you use this second technique and do not use the first form of REJECT.

If you use the first form, you execute REJECT to prevent the user from accepting a record, usually because the record is not complete or has inaccurate entries. If the user tries to accept the record, executing REJECT prevents the record from being accepted and the record remains displayed in the layout. The user generally must continue with data entry until the record is acceptable.

The best place to put this form of REJECT is in the script of an Accept button associated with the Enter key. This way, validation occurs only when the record is accepted, and the user cannot bypass the validation by pressing the Enter key.

The second form of REJECT is executed with the *data entry area* parameter. The cursor stays in the data entry area. This form of REJECT forces the user to enter a correct value. This form of the command must be used immediately following a modification to the data entry area. You can test for modification by using the function Modified. You can also use REJECT in the script for the data entry area.

REJECT works only in the During phase of an input layout procedure. You must put either form of the REJECT command in the layout procedure or script for the layout that is being modified. If you are using REJECT for an included layout, put it in the included layout's procedure or script.

You can use HIGHLIGHT TEXT to select the data that is being rejected.

The following example shows the first form of REJECT being used in an Accept button script. The Enter key is set as an equivalent for the button. This means that even if the user presses the Enter key to accept the record, the button's script will be executed. The record is of a bank transaction. If the transaction is a check, then there must be a check number. If there is not a check number, the validation is rejected.

```
Case of
  : ((Trans = "Check") & (Number = ""))        ` If it is a check with no number...
    ALERT ("Please fill in the check number.")  ` Alert the user
    REJECT                                      ` Reject the entry
    GO TO FIELD (Number)                        ` Go to the check number field
End case
```

The following example is part of a script for a Salary field. The script tests whether the Salary field is less than $10,000 and rejects the field if it is. You could perform the same operation by specifying a minimum value for the field in the Layout editor.

```
If (Salary<10000)
  ALERT ("Salary must be greater than $10,000")
  REJECT (Salary)
End if
```

Setting Data Attributes

SET FILTER	SET ENTERABLE
SET CHOICE LIST	SET FORMAT

The commands in this section set data entry attributes for input layouts. These commands perform the same actions as the equivalent areas in the Field dialog box or the Object Definition dialog box in the Layout editor. These commands can be used only on text, numeric, date, or time data entry areas. They are effective only while the layout is displayed on screen. As soon as a new record or layout is displayed, the default settings take effect.

SET FORMAT is an exception, since it can also be used in output layouts, both for printing and for display on the screen.

For more information on setting these attributes, see the *4th DIMENSION Design Reference.*

SET FILTER

SET FILTER (*text object*; *filter*)

Parameter	Type	Description
text object	Field or variable	Field or variable for which to set character filter
filter	String	Character filter to use

SET FILTER changes the character filter for the *text object* displayed in the current layout to *filter*. Using this command is equivalent to entering a character filter for a field or variable in the Layout editor.

The following example sets the character filter for a postal code field. If the address is in the U.S., the filter is set to ZIP codes. Otherwise, it is set to allow for any entry.

```
If (Country = "US")
   ` Set the filter to a ZIP code format
  SET FILTER (Post Code; "&9#####")
Else
   ` Set the filter to accept alpha and numeric and uppercase the alpha
  SET FILTER (Post Code; "~@")
End if
```

SET CHOICE LIST

SET CHOICE LIST (*text object*; *list*)

Parameter	Type	Description
text object	Field or variable	Field or variable to set
list	String	Name of the list to use

SET CHOICE LIST sets the choice list for the *text object* displayed in the current layout to *list*. It is equivalent to selecting a choice list for a field or variable in the Layout editor. The list is displayed during data entry when the user selects the text area.

The following example sets a choice list for a shipping field. If the shipping is overnight, then the choice list is set to shippers who can ship overnight. Otherwise, it is set to the standard shippers.

```
If (Overnight)
  SET CHOICE LIST (Shipper; "Fast Shippers")
Else
  SET CHOICE LIST (Shipper; "Normal Shippers")
End if
```

SET ENTERABLE

SET ENTERABLE (*text object*; *TRUE or FALSE*)

Parameter	Type	Description
text object	Field or variable	Field or variable to set
TRUE or FALSE	Boolean	TRUE for enterable; FALSE for nonenterable

SET ENTERABLE sets the *text object* displayed in the current layout to be either enterable or not. Using this command is equivalent to selecting enterable or nonenterable for a field or variable in the Layout editor.

When the *text object* is enterable (TRUE), the user can move the cursor into the area and enter data. When the *text object* is nonenterable (FALSE), the user cannot move the cursor into the area and cannot enter data.

The following example sets a shipping field, depending on the weight of the shipment. If the shipment is 1 ounce or less, then the shipper is set to US Mail and the field is set to be nonenterable. Otherwise, the field is set to be enterable.

```
If (Weight <= 1)
  Shipper := "US Mail"
  SET ENTERABLE (Shipper; False)
Else
  SET ENTERABLE (Shipper; True)
End if
```

SET FORMAT

extra

SET FORMAT (*text object*; *format*)

Parameter	Type	Description
text object	Field or variable	Field or variable to set
format	String	Format to use

SET FORMAT changes the display format for the *text object* displayed in the current layout to *format*. Using this command is equivalent to entering a format for a field or variable in the Layout editor.

SET FORMAT can be used for both input layouts and output layouts.

The following example changes the format for a ZIP code field, depending on the length of the ZIP code.

```
If (Length (ZIP) = 9)
  SET FORMAT (ZIP; "#####-####")
Else
  SET FORMAT (ZIP; "#####")
End if
```

Special Layout Management

ACCEPT CANCEL REDRAW

The commands in this section allow you to close layouts and redraw portions of a layout.

ACCEPT

Save Record L210

ACCEPT

ACCEPT is used in input layout procedures to accept a new or modified record or subrecord. ACCEPT may also be used to close a layout displayed with the DIALOG command. It performs the same action as a user's pressing the Enter key. The current phase of execution is first completed. One more During phase is executed, and then an After phase, except in dialogs where there is no After phase. L179 L180

ACCEPT is commonly executed as a result of a menu item being chosen.

ACCEPT is also commonly used in the script of a "No Action" button.

After the layout is accepted, the OK system variable is set to 1. ← L383

ACCEPT cannot be queued up. In other words, executing two ACCEPT commands in a row would have the same effect as executing one.

The following example is a one-line procedure called from a procedure associated with a layout menu item. It accepts the current data entry.

```
ACCEPT
```

CANCEL

CANCEL

CANCEL cancels the current input or output layout. CANCEL is equivalent to the user's pressing the "cancel" key combination (usually Command-.) or pressing a Cancel button. In the input layout, CANCEL cancels the record or dialog and exits the layout. In an output layout that is being displayed with a MODIFY SELECTION or DISPLAY SELECTION command, CANCEL cancels the command.

CANCEL is commonly executed as a result of a menu item being chosen.

CANCEL is also commonly used in the script of a "No Action" button. When used during data entry, the script can still save the record with the SAVE RECORD command. After the record is saved, using CANCEL avoids executing the After phase. L180

After the layout is canceled, the OK system variable is set to 0. L383

CANCEL cannot be queued up. In other words, executing two CANCEL commands in a row would have the same effect as executing one.

The following example is a one-line procedure called from a procedure associated with a layout menu item. It cancels the current data entry.

CANCEL

REDRAW

REDRAW (*included file*)

Parameter	Type	Description
included file	File or subfile	Area to redraw

When you use a procedure to change the value of a field or subfield displayed in an included layout, you must execute REDRAW to ensure that the layout is updated.

Printing Reports

REPORT	Subtotal	PAGE SETUP
PRINT SELECTION	Printing page	FORM FEED
BREAK LEVEL	PRINT LAYOUT	PRINT LABEL
ACCUMULATE	PRINT SETTINGS	

Printing reports is often the most important job of a database. The commands in this section allow you to use 4th DIMENSION's flexible reporting capabilities.

You can use the commands PRINT SELECTION, PRINT LAYOUT, and PRINT LABEL to generate reports with layouts created in the Design environment. You can use the REPORT and PRINT LABEL commands to generate reports without using layouts.

A report printed with a layout can have almost any kind of design. The report can include graphics, and the elements making up the report can be arranged in any manner. A report printed with a layout also executes the associated layout procedure, which gives tremendous processing capabilities.

Reports generated with the Quick Report and Label editors are usually simpler in design. The user can design a report from scratch by using these editors. The report designs can be stored on disk. These reports do not use layouts and therefore do not execute layout procedures.

The three primary reporting commands, REPORT, PRINT SELECTION, and PRINT LAYOUT, have varying degrees of flexibility. Generally, the more flexible a command is, the more you as a designer need to do to generate a report.

REPORT is the simplest report generator. It uses the same Quick Report editor that you use in the User environment. The user interface is a simple point, click, and drag interface. The style of a report is a row and column format that can include headers, footers, different fonts and styles, formatting, formulas, breaks, totals, calculations, and multi-column variable-length text. The report can be printed to a high-speed serial device. The report can also be converted to a graph.

PRINT SELECTION is the most commonly used printing command. It uses an output layout. The layout can be of any design. The command allows headers, footers, different fonts and styles, formatting, formulas, breaks, totals, and calculations. Simple reports can be generated without layout procedures, but most reports will include a layout procedure and scripts to process the report.

PRINT LAYOUT is the most flexible of all the printing commands and also the most demanding for the designer. It allows you to mix different layouts on the same page and to include form feeds at any time during the report. PRINT LAYOUT is used only for the most complex printing jobs.

PRINT LABEL can print labels with a layout, providing a high degree of flexibility, or it can use the Label editor as provided in the User environment, letting the user design the label as needed. Since it can print records side-by-side, this command can also be used to generate unusual reports.

All of the print commands print the current selection. It is common to sort the selection before printing.

4th Aug 1998
a s/w
PROCESS

When a user prints a report, he or she may elect to print it on the screen. During printing, 4th DIMENSION displays the current page being printed and the status of the print job. If the user "prints" to the screen, he or she may print the current page by clicking the Print button.

The user may cancel printing by clicking the Stop Printing button. If the user cancels printing either by clicking this button or by canceling a printer dialog box, the OK system variable is set to 0. If the printing is successful, the OK system variable is set to 1.

Activating Break Processing in Layout Reports

Break processing for layout reports can be activated in two ways. The first (a) uses the function Subtotal. The second (b) uses the commands BREAK LEVEL and ACCUMULATE. Both methods can achieve the same results but have different advantages.

Using Subtotal For Break Processing (a)

Example Report
Design page 204

To turn on break processing with the Subtotal function, the function must appear in the layout procedure or a script for the layout. Before printing the report, 4th DIMENSION scans the layout procedure and scripts for the Subtotal function. If it finds it, break processing is activated.

The Subtotal function does not need to be executed for it to turn on break processing. For example, it could be in a script of an object that is below the Footer line and therefore would never be printed or executed. In fact, if the Subtotal function is not executed, the argument to the function does not need to be valid. For example, the following line would turn on break processing:

x := **Subtotal** (x)

When Subtotal is used to activate break processing, you must sort on one more level than you break on. For example, if you wanted two levels of breaks in your report, you would sort on three levels.

4th Aug 1998

Using BREAK LEVEL and ACCUMULATE For Break Processing

You can also use the commands BREAK LEVEL and ACCUMULATE to turn on break processing. In this case, you must execute both of these commands *before* printing a layout report. The Subtotal function is not required when using this method.

When this method is used, you do not need to sort on one extra level. You must, of course, sort on at least as many levels as you need to break on.

Comparing the Two Methods

The primary advantage to using Subtotal to initiate break processing is that you do not need to execute a procedure prior to printing the report. This is especially useful in the User environment. The process to print the report in the User environment is typically like this:

1. Select the records to be printed.
2. Sort the records, sorting on one extra level.
3. Choose Print from the File menu. 4th DIMENSION scans the layout procedure and scripts, finds the Subtotal function, turns on break processing, and prints the report.

There are two disadvantages to using Subtotal to trigger break processing:

- You cannot use Subtotal to activate break processing in compiled procedures.
- You must sort on one extra level. If you have many records this may be time consuming.

Using BREAK LEVEL and ACCUMULATE to activate break processing is the recommended method when using procedures to generate layout reports. The process to print a report using this method is typically like this:

1. Select the records to be printed.
2. Sort the records. Sort on at least the same number of levels as breaks.
3. Execute BREAK LEVEL and ACCUMULATE.
4. Print the report.

You *must* use BREAK LEVEL and ACCUMULATE to activate break processing in compiled procedures.

REPORT

REPORT ({*file*}; *document*; {*})

Parameter	Type	Description
file	File	File to print
document	String	Quick report document
*		Hide printer setup dialog boxes

REPORT prints a report for *file*, using the Quick Report editor shown in Figure 13-7.

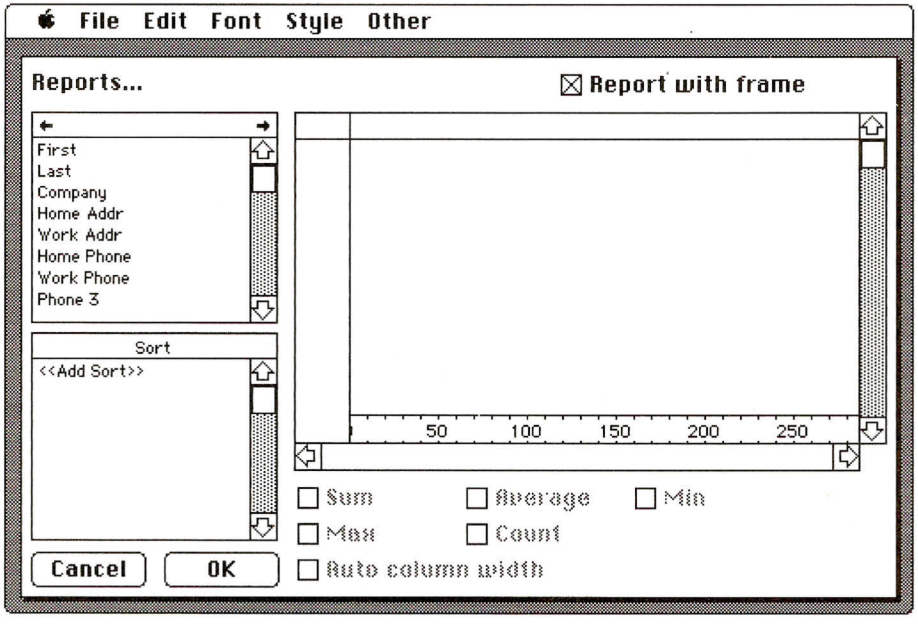


Figure 13-7
The Quick Report editor

The *document* parameter is a report document that was created with the Quick Report editor and saved on disk. You save a report document by choosing Save or "Save as" from the File menu in the Quick Report editor. The document stores the specifications of the report, not the records to be printed. If an empty string ("") is specified for *document*, REPORT displays an open-file dialog box and the user can select the report to print. After a report is selected, the printer setup dialog boxes are displayed, unless the * parameter is specified. If this parameter is specified, the printer dialog boxes are not displayed. The report is then printed.

If the *document* parameter specifies a document which does not exist, the Quick Report editor is displayed. The Quick Report editor allows users to create their own reports. When the Quick Report editor is displayed, the menu bar displays the same five menus that manage the editor in the User environment: File, Edit, Font, Style, and Other. The user has complete control over the editor. See the *4th DIMENSION User Reference* for details on creating reports with the Quick Report editor.

The following example lets the user search the database, and then allows the user to specify the Quick Report document with which to print the report.

```
SEARCH ([People])                    ` Search for records
REPORT ([People]; "")                ` Let the user generate a report
```

List any
L69 12

PRINT SELECTION

PRINT SELECTION ({*file*}; {*}*)

Parameter	Type	Description
file	File	File to print
*		Suppress the printer dialog boxes

PRINT SELECTION prints the current selection of *file*. The records are printed with the current output layout. PRINT SELECTION performs the same action as the Print menu item in the User environment.

By default, PRINT SELECTION displays the printer dialog boxes before printing. You can suppress these dialog boxes by using the optional asterisk parameter. If the user cancels either of the printer dialog boxes, the command is canceled and the report is not printed. Using the optional asterisk causes the report to be printed with the page setup that was in effect when the layout was created, or with the page setup set by the PAGE SETUP command.

During printing, the output layout procedure and the layout's scripts are executed: the Header phase when printing a header, the Before and During phases when printing each record, the Break phase when printing the Break area, and the Footer phase when printing a footer.

You can check whether PRINT SELECTION is printing the first header, by testing Before selection in the Header phase. You can also check for the last footer, by testing End selection in the Footer phase.

To print a sorted selection with subtotals, using PRINT SELECTION, you must first sort the selection. Then, in each Break area of the report, include a variable with a script that assigns the subtotal to the variable. You can also use statistical and arithmetic functions like Sum and Average to assign values to variables.

The following example selects all the records in the [People] file. It then uses the DISPLAY SELECTION command to display the records and allow the user to select the records that he or she would like to print. Finally, it uses the selected records with the USE SET command, and prints them with PRINT SELECTION.

```
DEFAULT FILE ([People])          ` Set the default file
ALL RECORDS                      ` Select all records
DISPLAY SELECTION (*)            ` Display the records
USE SET ("UserSet")              ` Use only the records that the user picked
PRINT SELECTION                  ` Print the records that the user picked
```

BREAK LEVEL

BREAK LEVEL (*level*; *{page break}*)

Parameter	Type	Description
level	Number	Number of break levels
{page break}	Number	Break level for which to do a page break

BREAK LEVEL specifies the number of break levels in a report.

There are two methods used to turn on break processing for layout reports. See "Activating Break Processing," earlier in this section, for information on the two methods.

You must execute BREAK LEVEL and ACCUMULATE before every layout report for which you want to do break processing—they activate break processing for a layout report.

The *level* is the level to which you want to perform break processing. You must have sorted the records with at least that many levels. If you have sorted more levels, those levels will be printed as sorted, but will not be processed for breaks.

Each break level that is generated will print the corresponding Break areas and Header areas in the layout. There should be *level* Break areas in the layout. If there are more Break areas, they will be ignored and will not be printed.

The second (optional) argument, *page break,* is used to cause page breaks during printing.

The following example prints a report with two break levels. The selection is sorted on four levels, but the BREAK LEVEL command specifies to break on only two levels. One field is accumulated with the ACCUMULATE command.

```
DEFAULT FILE ([People])                                  ` Set the default file
` Sort on four levels
SORT SELECTION ([People]Dept; >; [People]Title; >; [People]Last; >; [People]First; >)
` Turn on break processing to 2 levels (Dept and Title)
BREAK LEVEL (2)
ACCUMULATE ([People]Salary)                              ` Accumulate the salaries
OUTPUT LAYOUT ("Dept salary")                            ` Select the report layout
PRINT SELECTION                                          ` Print the report
```

L205

ACCUMULATE

ACCUMULATE (*data1 {;...; dataN}*)

Parameter	Type	Description
data	Field or variable	Numeric field or variable to accumulate

ACCUMULATE specifies the field(s) or variable(s) to be accumulated during a layout report.

There are two methods used to turn on break processing for layout reports. See "Activating Break Processing in Layout Reports," earlier in this section, for information on the two methods. Page 160

You must execute BREAK LEVEL and ACCUMULATE before every layout report for which you want to do break processing—they activate break processing for a layout report.

Use ACCUMULATE when you want to include subtotals for numeric fields or variables in a layout report. ACCUMULATE tells 4th DIMENSION to store subtotals for each of the *data* arguments. The subtotals are accumulated for each break level specified with the BREAK LEVEL command.

Execute ACCUMULATE before printing the report with PRINT SELECTION or before choosing the Print menu item in the User environment. Use the Subtotal function in the layout procedure or a script, to return the subtotal of one of the *data* arguments.

See the example for the BREAK LEVEL command, earlier in this section.

Subtotal

Subtotal (*data*; *{page break}*) → Number

Parameter	Type	Description
data	Field or variable	Numeric field or variable to return subtotal
page break	Number	Break level for which to cause a page break

Subtotal returns the subtotal for *data* for the current or last break level. Subtotal works only when a sorted selection is being printed with PRINT SELECTION or when printing using the Print menu item in the User environment. The *data* parameter must be of type real, integer, or long integer. Assign the result of the Subtotal function to a variable placed in the Break area of the print layout.

There are two methods used to turn on break processing for layout reports. See "Activating Break Processing," earlier in this section, for information on the two methods.

Subtotal initiates break processing when BREAK LEVEL and ACCUMULATE have not been executed. In this case, you must put Subtotal in the layout procedure or a script for the layout. 4th DIMENSION scans the layout procedure and scripts before printing and if Subtotal is present, break processing will be initiated.

If BREAK LEVEL and ACCUMULATE have not been executed, the second (optional) argument to Subtotal is used to cause page breaks during printing. If *break level* is 0, Subtotal does not issue a page break. If *break level* equals 1, Subtotal issues a page break for each level 1 break. If *break level* equals 2, Subtotal issues a page break for each level 1 and level 2 break, and so on.

If BREAK LEVEL and ACCUMULATE have not been executed and you want to have breaks on n sort levels, you must sort the current selection on n + 1 levels. This lets you sort on a last field, so that the field doesn't create unwanted breaks. If you want the last sort field to generate a break, sort the field twice.

The following example is a one-line script in a Break area of a layout (B0, the area above the B0 marker). The variable vSalary is placed in the Break area. The variable is assigned the subtotal of the Salary field for this break level.

vSalary := **Subtotal** (Salary)

Printing page

Printing page → Number

Printing page returns the printing page number. It can be used only when you are printing with PRINT SELECTION or from the Print menu in the User environment.

The following example changes the position of the page numbers on a report, so that the report can be reproduced in a double-sided format. The layout for the report has two variables that display page numbers. A variable in the lower-left corner (Left) will print the even page numbers. A variable in the lower-right corner (Right) will print the odd page numbers. The example tests for even pages and then clears and sets the appropriate variables.

```
If ((Printing page % 2) = 0)          ` If the Modulo is 0 it is an even page
  Left := String (Printing page)      ` Set the left page number
  Right := ""                         ` Clear the right page number
Else                                  ` Otherwise it is an odd page
  Right := String (Printing page)     ` Set the right page number
  Left := ""                          ` Clear the left page number
End if
```

PRINT LAYOUT

PRINT LAYOUT ({*file*}; *layout*)

Parameter	Type	Description
file	File	File to print
layout	String	Layout to print

PRINT LAYOUT should be used only by experienced 4th DIMENSION database designers.

PRINT LAYOUT simply prints *layout* with the current values of fields and variables. It prints only the Detail area (the area between the Header line and the Detail line) of the layout. It is usually used to print very complex reports that require complete control over the process of printing. PRINT LAYOUT does not do any record processing, break processing, form feeds, headers, or footers. These operations are the responsibility of the designer.

Since PRINT LAYOUT doesn't issue a form feed after printing the layout, it is easy to combine different layouts on the same page. Thus, PRINT LAYOUT is perfect for complex printing tasks that involve different files and different layouts. To force a form feed between layouts, use the FORM FEED command.

The printer dialog boxes do not appear when you use PRINT LAYOUT. The report *does not* use the print settings that were assigned in the Design environment when the layout was created. If you want the printer dialog boxes to appear, you must include the PRINT SETTINGS command before any series of PRINT LAYOUT commands.

PRINT LAYOUT builds each printed page in memory. Each page is printed when the page in memory is full. To ensure the printing of the last page after any use of PRINT LAYOUT, you must conclude with the FORM FEED command. Otherwise, the last page stays in memory and is not printed.

Included layouts are not printed with PRINT LAYOUT.

PRINT LAYOUT executes only the Before and During phases of the layout procedure.

The following example simulates a simple PRINT SELECTION command. There is no break processing. The report is for a checkbook register. The report uses one of two different layouts, depending on whether the record is for a check or a deposit.

```
DEFAULT FILE ([Register])                  ` Set the default file
SEARCH( )                                  ` Allow the user to select the records
SORT                                       ` Allow the user to sort the records
PRINT SETTINGS                             ` Allow the user to set up the printer
For ($i; 1; Records in selection)          ` Loop through all the selected records
  If ([Register]Type = "Check")            ` If it is a check...
    PRINT LAYOUT ("Check Out")             ` print the check layout
  Else                                     ` Else it must be a deposit so...
    PRINT LAYOUT ("Deposit Out")           ` print the deposit layout
  End if
  NEXT RECORD                              ` Move to the next record
End for
FORM FEED                                  ` Print the last page
```

PRINT SETTINGS

PRINT SETTINGS

PRINT SETTINGS displays the printer dialog boxes. First it displays the Page Setup dialog box. Then it displays the Print Settings dialog box. If the user clicks OK in both dialog boxes, the OK system variable is set to 1. Otherwise, the OK system variable is set to 0. You should include PRINT SETTINGS or PAGE SETUP before any group of PRINT LAYOUT commands. PRINT SETTINGS has no effect on PRINT SELECTION or PRINT LABEL.

The Print Settings dialog box contains a check box, "Preview on screen," that allows the user to specify to print to the screen.

Figure 13-8 through Figure 13-11 show the Page Setup and Print Settings dialog boxes for the LaserWriter and ImageWriter printers.

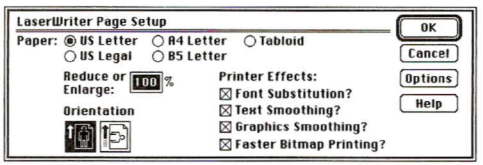


Figure 13-8
The LaserWriter Page Setup dialog box

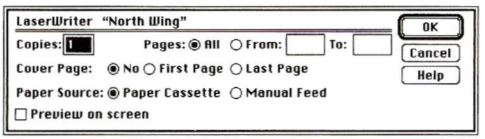


Figure 13-9
The LaserWriter Print Settings dialog box

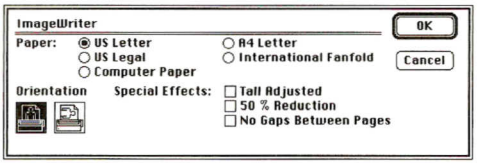


Figure 13-10
The ImageWriter Page Setup dialog box

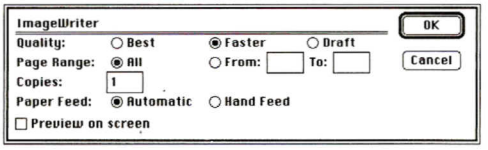


Figure 13-11
The ImageWriter Print Settings dialog box

See the PRINT LAYOUT example, earlier in this section.

PAGE SETUP

PAGE SETUP (*{file}*; *layout*)

Parameter	Type	Description
file	File	File containing layout
layout	String	Layout to use for page setup

PAGE SETUP sets the page setup for the printer to that stored with *layout*. The page setup is stored with the layout when the layout is created in the Design environment. PAGE SETUP can be used before PRINT LAYOUT and PRINT SELECTION.

The following example sets the page setup to that stored with the Sideways layout.

PAGE SETUP ([Global]; "Sideways")

FORM FEED

FORM FEED

FORM FEED prints the data that has been sent to the printer and ejects the page. FORM FEED is used with PRINT LAYOUT to force page breaks and to print the last page. Don't use FORM FEED with the PRINT SELECTION command. Instead, use Subtotal or BREAK LEVEL with the optional parameter to generate page breaks.

See the PRINT LAYOUT example, earlier in this section.

PRINT LABEL L69

PRINT LABEL ({*file*}; {*})

Parameter	Type	Description
file	File	File to print
*		Suppress the printer dialog boxes

PRINT LABEL ({*file*}; *label document*)

Parameter	Type	Description
file	File	File to print
label document	String	Name of disk label document

PRINT LABEL has two forms.

The first form prints the current selection of *file* as labels, using the current output layout. You cannot print subfiles in a label. See the *4th DIMENSION Design Reference* for details on creating layouts for labels.

When using the first form, PRINT LABEL displays the printer dialog boxes before printing. You can suppress these dialog boxes by using the optional asterisk parameter. If the user clicks Cancel in either of the printer dialog boxes, the command is canceled and the labels are not printed. During printing, 4th DIMENSION executes the output layout procedure and the scripts: A Before and During phase occurs when printing each record.

The second form of the command prints labels by using the Label editor. Figure 13-12 shows the Label editor.

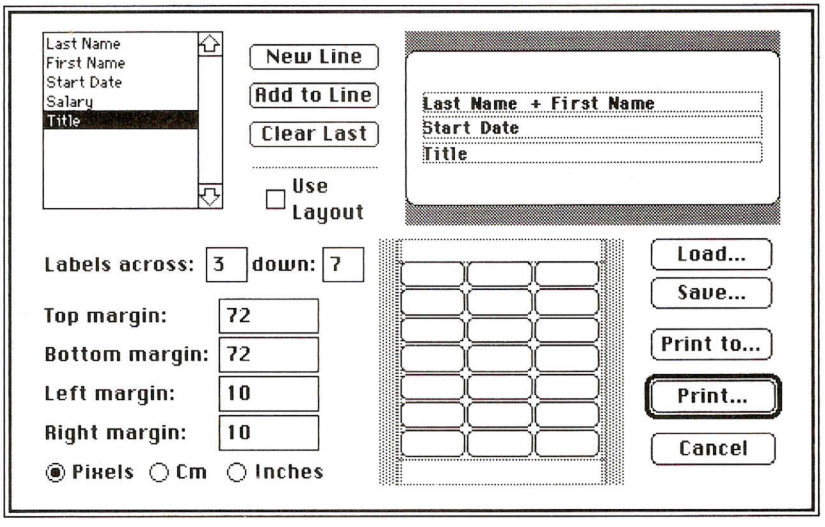


Figure 13-12
The Label editor

If you use the second form of the command, then the labels are printed with the label setup that is defined in *label document*. If *label document* is an empty string ("""), PRINT LABEL will present an open-file dialog box so that the user may specify the file to use for the label setup. If *label document* is the name of a document which does not exist, the Label editor is displayed and the user can define the label setup.

See the *4th DIMENSION User Reference* for details on creating labels with the Label editor.

The following example illustrates the use of the first form of PRINT LABEL. It uses a layout, Label Out, to print the labels. The example uses two procedures. The first is a global procedure that sets the correct output layout and then prints labels.

```
` Global procedure
vCR := Char (13)                          ` Assign the carriage return character
DEFAULT FILE ([Addresses])                ` Set the default file
ALL RECORDS                               ` Select all records
OUTPUT LAYOUT ("Label Out")               ` Select the output layout
PRINT LABEL                               ` Print the labels
OUTPUT LAYOUT ("Output1")                 ` Restore default output layout
```

The second procedure is the layout procedure for the Label Out layout. The layout contains one variable that is used to hold the concatenated fields. If the second address field (Addr2) is blank, it is removed by the procedure. (Note that this task is performed automatically with the Label editor.) The layout procedure does the label creation for each record.

```
` Layout procedure for the Label Out layout
` vLabel is the variable in the layout
` Concatenate names and first address
vLabel := Name1 + " " + Name2 + vCR + Addr1 + vCR
If (Addr2 # "")                           ` If the line is not blank...
   vLabel := vLabel + Addr2 + vCR         ` concatenate Addr2 into vLabel
End if
vLabel := vLabel + City + ", " + St + " " + ZipCode   ` Finally add the rest of the address
```

The following example illustrates the use of the second form of PRINT LABEL. It prints labels using the Label editor setup described in the document called Three Up.

```
PRINT LABEL ([Addresses]; "Three Up")
```

Graphing

GRAPH GRAPH SETTINGS GRAPH FILE

Graphs can be generated in two different ways. Data can be graphed from records (GRAPH FILE) or graphed from subfield or array information (GRAPH). GRAPH FILE uses data from the fields in records to create the graph. It displays the graph in its own window. GRAPH uses information in arrays or subfields, and draws the graph in a Graph area that appears in a layout or dialog box.

The two commands can draw the same eight types of graphs. Table 13-1 shows the graph types and the number associated with each type.

Table 13-1
The eight graph types

Graph Type	Number	Graph Style
Column	1	
Proportional column	2	
Stacked column	3	
Line	4	
Area	5	
Scatter	6	
Pie	7	
Picture	8	Pictures

GRAPH

GRAPH (*graph name*; *graph number*; *x labels*; *y elements1* {;...; *y elements8*})

Parameter	Type	Description
graph name	Variable	Name of the layout Graph area
graph number	Number	Graph type number
x labels	Array or subfield	Labels for the x-axis
y elements	Array or subfield	Data to graph (up to eight allowed)

GRAPH draws a graph for a Graph area in a layout. The data can come from either arrays or subfields.

The *graph name* parameter is the name of the Graph area that displays the graph. The Graph area is created in the Layout editor, using the graph object type. The graph name is the name entered for the variable name. For information on creating a Graph area, see the *4th DIMENSION Design Reference*.

The *graph number* parameter defines the type of graph that will be drawn. It must be a number from 1 to 8. The graph types are listed in Table 13-1, earlier in this section. After a graph has been drawn, you can change the type by changing *graph number* and executing the GRAPH command again.

The *x labels* parameter defines the labels that will be used to label the x-axis (the bottom of the graph). This data can be of string, date, time, or numeric type. There should be the same number of subrecords or array elements in *x labels* as there are subrecords or array elements in each of the *y elements*.

The data specified by *y elements* is the data to graph. The data must be numeric. Up to eight data sets can be graphed, each set off by a semicolon. Pie charts graph only the first *y elements*.

The following example shows how to use variables to create a graph. The code would be inserted in a layout procedure or script. It is not intended to be realistic, since the data is constant. Figure 13-13 shows the resulting graph.

```
ARRAY STRING (4; X; 2)                          ` Create an array for the x-axis
X{1} := "1980"                                  ` X Label #1
X{2} := "1981"                                  ` X Label #2
ARRAY REAL (A; 2)                               ` Create an array for the y-axis
A{1} := 30                                      ` Insert some data
A{2} := 40
ARRAY REAL (B; 2)                               ` Create an array for the y-axis
B{1} := 50                                      ` Insert some data
B{2} := 80
` Set the legends for the graph
GRAPH SETTINGS (vGraph; 0; 0; 0; 0; False; False; True; "Store 1"; "Store 2")
GRAPH (vGraph; 1; X; A; B)                      ` Draw the graph
```

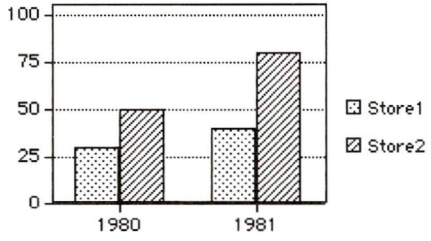


Figure 13-13
Graph from the example

The following example graphs the sales in dollars for sales people in a subfile. The subfile has three fields: Name, Last Year Tot, and This Year Tot. The graph will show the sales for each of the sales people for the last two years.

```
GRAPH (Sales Graph; 1; Sales'Name; Sales'Last Year Tot; Sales'This Year Tot)
```

GRAPH SETTINGS

GRAPH SETTINGS (*g*; *xmin*; *xmax*; *ymin*; *ymax*; *xprop*; *xgrid*; *ygrid*; *title1* {;...; *title8*})

Parameter	Type	Description
g	Variable	Name of the Graph area
xmin	Number or date or time	Minimum x-axis value for proportional graph (line or scatter plot only)
xmax	Number or date or time	Maximum x-axis value for proportional graph (line or scatter plot only)
ymin	Number	Minimum y-axis value
ymax	Number	Maximum y-axis value
xprop	Boolean	TRUE for proportional x-axis; FALSE for normal x-axis (line or scatter plot only)
xgrid	Boolean	TRUE for x-axis grid; FALSE for no x-axis grid (only if *xprop* is TRUE)
ygrid	Boolean	TRUE for y-axis grid; FALSE for no y-axis grid
title	String	Title(s) for graph legend(s)

GRAPH SETTINGS changes the graph settings for the graph *g*. The graph must have already been displayed with the GRAPH command.

These settings are ignored for a pie chart.

The parameters *xmin, xmax, ymin,* and *ymax* all set the minimum and maximum values for their respective axes of the graph. If the value of any pair of these parameters is a null value (0, †00:00:00†, or !00/00/00!, depending on the data type), the default graph values will be used.

The *xprop* parameter turns on proportional plotting for line graphs (type 4) and scatter graphs (type 6). When TRUE, it will plot each point on the x-axis according to the point's value, and then only if the values are numeric, time, or date.

The *xgrid* and *ygrid* parameters display or hide grid lines. A grid for the x-axis will be displayed only when the plot is a proportional scatter or line graph.

The *title* parameter(s) label the legend.

See the GRAPH example, earlier in this section.

GRAPH FILE L 69

GRAPH FILE (*{file}*)

Parameter	Type	Description
file	File	File to graph

GRAPH FILE (*{file}*; *graph number*; *x field*; *y field1 {;...; y field8}*)

Parameter	Type	Description
file	File	File to graph
graph number	Number	Graph type number
x field	Field	Labels for the x-axis
y field	Field	Fields to graph (up to eight allowed)

GRAPH FILE has two forms. The first form displays the Graph window and allows the user to select the fields to be graphed. The second form specifies the fields to be graphed and does not display the Graph window. GRAPH FILE graphs data from a file's fields. Only data from the current selection is graphed.

Using the first form is equivalent to choosing Graph from the Quick menu in the User environment. Figure 13-14 shows the Graph window, which allows the user to define the graph.

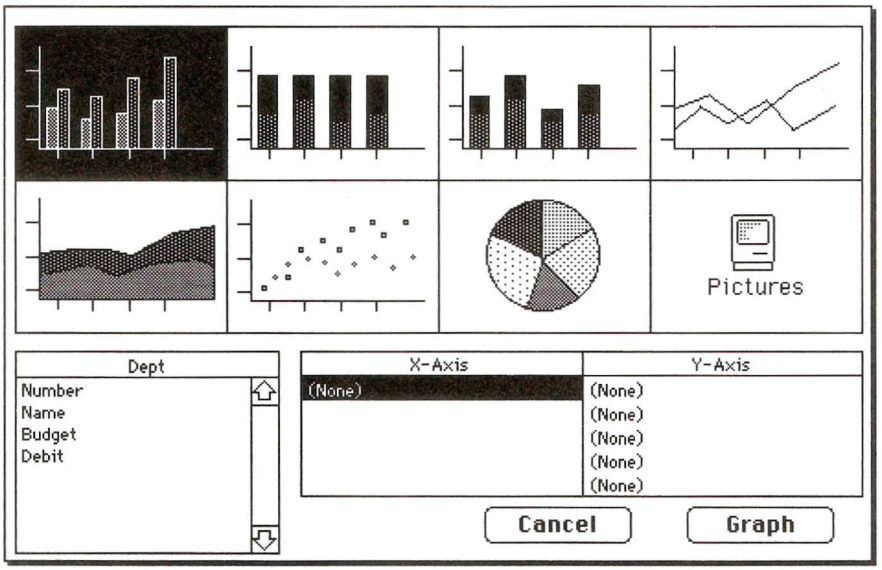


Figure 13-14
Graph window

The second form of the command graphs the fields that are specified for *file*.

The *graph number* defines the type of graph that will be drawn. It must be a number from 1 to 8. The graph types are listed in Table 13-1, earlier in this section. After a graph has been drawn, the user can change the type by choosing from the Graph menu.

The *x field* defines the labels that will be used to label the x-axis (the bottom of the graph). The field type can be Alpha, Integer, Long integer, Real, or Date.

The *y field* is the data to graph. The field type must be Integer, Long integer, or Real. Up to eight *y fields* can be graphed, each set off by a semicolon.

In either form, GRAPH FILE replaces the current menu bar with five menus: File, Edit, Picture, Graph, and Settings. The user can print the currently displayed graph by choosing Print from the File menu. The user can copy the graph from the screen to the Clipboard by using the Edit menu. The Picture menu lets the user change the pictures used in the picture type of graph. The Graph menu allows the user to change the graph type. The Settings menu allows the user to change axis settings.

If the user clicks on a legend for a graph, a dialog box appears. The dialog box allows the user to change the patterns and colors for the data series. The colors are displayed only on a color monitor.

GRAPH FILE graphs only the first 100 columns of a graph. It sums matching x-axis values. For example, if you were graphing all sales per region, the sales for each region would automatically be summed. If you graph data where the x-axis labels are duplicated, the user can choose the Scale menu item to display the Graph Settings dialog box. In the Graph Settings dialog box, the user can deselect the "Group on X-axis" check box to turn off the x-axis grouping.

You can also use the Quick Report editor to generate graphs from field data, by using the "Print to" menu. See the *4th DIMENSION User Reference* for more information on graphing.

The following example illustrates the use of the first form of GRAPH FILE. It presents the Graph window and allows users to select the fields they would like to graph. The code first does a search and a sort to select and arrange the records in the order in which the user would like them graphed.

```
SEARCH ([People])                    ` Search the [People] file
SORT SELECTION ([People])            ` Sort the [People] file
GRAPH FILE([People])                 ` Graph data in the [People] file
```

The following example illustrates the use of the second form of GRAPH FILE. It first searches and sorts the [People] file. It then graphs the salaries of the people.

```
SEARCH ([People]; [People]Title = "Manager")    ` Search the [People] file for managers
SORT SELECTION ([People]; [People]Salary; >)    ` Sort the managers by salary
` Graph the salaries of the managers
GRAPH FILE([People]; 1; [People]Last; [People]Salary)
```

Monitoring the Layout Execution Cycle

Before	In header	Level
During	In break	
After	In footer	

The commands in this section are used to determine what phase of the layout execution cycle is executing. It is recommended that whenever possible, you use scripts instead of testing for the execution cycle in a layout procedure.

See Chapter 5 in Part I of this manual for more information on the execution cycle.

Before

Before → Boolean

Before returns TRUE in a layout procedure before the layout is displayed on screen or printed. The Before phase is usually used to initialize variables and fields before the layout is displayed.

If an input layout contains an included layout, a Before phase is first generated for each included record or subrecord that is displayed. Then the Before phase is generated for the parent record. A Before phase is then generated for any record that is scrolled into view by the user. A Before phase is also generated for any new record or subrecord in an included layout, whether in the multi-line layout or the full-page layout.

When you are printing with PRINT SELECTION or from the Print menu in the User environment, a Before phase is generated before a record is printed. The Before phase for each record or subrecord in an included layout is generated after the parent record's Before phase. This is the opposite order from that of data entry. This order allows a selection of records or subrecords for the included layout to be made in the Before phase of the parent record.

When you are displaying a selection on screen with DISPLAY SELECTION or MODIFY SELECTION, or in the User environment, Before and During are TRUE simultaneously as each record is displayed.

The following example sorts a selection of subrecords before the layout is displayed.

```
Case of
  : (Before)
      ` Sort the children into ascending order
    SORT SUBSELECTION ([Parents]Children; [Parents]Children'First name; >)
:
End case
```

During

L 46 ACCEPT L157

During → Boolean

Input

During returns TRUE in an input layout procedure when any modification is made to an object (field, variable, button, or other active area) and when the layout is accepted. The During phase for input layouts is usually used for data validation, calculations, and updating fields and variables during data entry.

A During phase for an input layout procedure is generated under the following conditions:

- when the user modifies a field or variable and moves from the field or variable
- when the user clicks any button or check box
- when the user clicks in an external object
- when the user chooses from a custom menu, but not from the Apple or Edit menu (except when a user pastes a picture into a field)
- when the user makes a selection from a scrollable area
- within an included record or subrecord, only when the user enters data for that particular record or subrecord
- within an included record or subrecord, when the record or subrecord must be redrawn
- when the user accepts the layout
- when the user cancels the layout

Output Printing

During returns TRUE in an output layout procedure when each record is being printed with PRINT SELECTION or from the User environment Print menu.

During returns TRUE in an output layout procedure when a layout is displayed in a list on the screen with DISPLAY SELECTION or MODIFY SELECTION, or in the User environment. In this case, Before and During are TRUE simultaneously. During also returns TRUE in an output layout procedure when the user has double-clicked a record.

The following example shows all of the tests needed to completely monitor the execution cycle of a DISPLAY SELECTION or MODIFY SELECTION command. This procedure is the output layout procedure for the displayed layout. You must use custom buttons in the Break area for this to work, since the default Done button will only generate a During phase.

The tests must be executed in the order shown. Note that the last test for During allows you to check the record that the user just double-clicked. At this time, you could change the input layout depending on the information in the record.

```
Case of
  : (Before & During)                  ` Each record is being displayed
  : (Before)                           ` The output list has not yet been displayed
  : (In header)                        ` The header is being displayed
  : (Button = 1)                       ` A button was selected
    ` You must do this test for each of the
    ` buttons in the Footer area.
  : (Menu selected # 0)                ` A menu was selected
  : (During)                           ` A record was double-clicked
    ` You may change the input layout here.
    ` You may also cancel the command, and
    ` the double-clicked record will be current.
End case
```

After

After → Boolean

After returns TRUE in an input layout procedure when a new or modified record has been accepted. If there is an included file or included subfile, an After phase is first generated for each record of the included file or subfile.

An After phase is generated only when ADD RECORD, MODIFY RECORD, or MODIFY SELECTION is executed, and then only if a record is accepted. An After phase is also generated if a record is accepted in the User environment. It is not generated for DIALOG or output layout procedures.

If an existing record is not modified and the user accepts the record, the record is not rewritten to disk and an After phase is not generated. In such a situation, you can force an After phase by reassigning a field to itself in the Before or During phases, therefore setting it as modified.

The following example shows the After phase being used to assign the date that the record is modified to a field.

```
Case of
  : (Before)
  :
  : (During)
  :
  : (After)                            ` If the record is being saved...
    Last Modified := Current Date      ` save the date of this modification
End case
```

In header

In header → Boolean

In header returns TRUE in an output layout procedure when a Header area is about to be printed.

The Header area of a layout is the area above the Header marker (marked with an *H*) and below the top of the layout. The header is printed at the top of each page of a report. There can also be Header area for each break level.

You can determine the beginning of a report by using Before selection. Before selection returns TRUE when the first header is about to be printed.

In header also returns TRUE in an output layout procedure when a Header area is displayed on screen.

The following example is a template for a layout procedure. It shows each of the possible reporting phases being tested.

```
Case of
  : (In header)
    Case of
      : (Before selection)
        ` Code for the first header goes here
      : (Level = 1)
        ` Code for a break header level 1 goes here
        ` There would be further tests for more break levels if required
    End case
  : (During)
    ` Code for each record goes here
  : (In break)
    Case of
      : (Level = 0)
        ` Code for a break level 0 goes here
      : (Level = 1)
        ` Code for a break level 1 goes here
        ` There would be further tests for more break levels if required
    End case
  : (In footer)
    If (End selection)
      ` Code for the last footer goes here
    Else
      ` Code for a footer goes here
    End if
End case
```

In break

L46

In break → Boolean

Output

In break returns TRUE in an output layout procedure when a Break area is about to be printed.

In break returns TRUE for each break, that is, when a break level changes.

There are two methods used to turn on break processing for layout reports. See the section "Activating Break Processing," earlier in this chapter, for information on the two methods. L160

See the example for In header, earlier in this section.

Level

L46

Level → Number

Level is used to determine the current break or header level. It returns the level number during an In Break or In Header phase of the execution cycle.

Level 0 is the last level to be printed and is appropriate for printing a grand total. Level returns 1 when 4th DIMENSION prints a break on the first sorted field, 2 when 4th DIMENSION prints a break on the second sorted field, and so on.

See the example for In header, earlier in this section.

In footer

L46

In footer → Boolean

Output

In footer returns TRUE in an output layout procedure when a Footer area is about to be printed.

The Footer area of a layout is above the Footer marker (marked with an *F*) and below the Break marker (marked with a *B*). The footer is printed at the bottom of each page of a report.

You can determine the end of a report by using End selection. End selection returns TRUE when the last footer is about to be printed.

See the example for In header, earlier in this section.

MANAGING DATA

CHAPTER 14

MANAGING DATA

The commands in this chapter help you manage data. They do not display the data—for that purpose, use the commands in Chapter 13, "Data Entry and Reporting."

Managing Selections

ALL RECORDS	DELETE SELECTION	NEXT RECORD
Records in file	MERGE SELECTION	PREVIOUS RECORD
Records in selection	FIRST RECORD	Before selection
APPLY TO SELECTION	LAST RECORD	End selection

These commands help you manage the current selection.

Many of the commands perform an operation on the selection, such as deleting the selection, modifying the selection, or moving within the selection. It is important that you first create the correct selection—generally with the commands described in the section "Searching," in this chapter, and "Managing Sets," in Chapter 16.

ALL RECORDS L 69 17

ALL RECORDS ({*file*})

Parameter	Type	Description
file	File	File for which to select all records

ALL RECORDS selects all the records of *file* as the current selection. ALL RECORDS makes the first record the current record and loads the record from disk. ALL RECORDS returns the records to the default record order.

The following example displays all the records in the database.

```
DEFAULT FILE ([People])      ` Set a default file
ALL RECORDS                  ` Select all the records in the file
DISPLAY SELECTION            ` Display the records in the output layout
```

Records in file

Records in file ({*file*}) ➛ Number

Parameter	Type	Description
file	File	File for which to return number of records

Records in file returns the total number of records in *file*. Records in selection returns the number of records in the current selection only.

The following example displays an alert that shows the number of records in a file. Notice that the number returned by Records in file is converted to a string.

```
ALERT ("There are " + String (Records in file ([People])) + " records in the file.")
```

The following example is a very typical loop that passes through all of the records in a file. Note that this loop performs the same action as APPLY TO SELECTION.

```
ALL RECORDS ([People])                    ` Select all the records
For ($i; 1; Records in file ([People]))   ` Will loop once for each record in the file
  Do Something                            ` Do something that affects the record
  SAVE RECORD                             ` Save the modified record
  NEXT RECORD ([People])                  ` Move to the next record
End for
```

Records in selection

Records in selection ({*file*}) ➛ Number

Parameter	Type	Description
file	File	File for which to return number of records

Records in selection returns the number of records in the current selection of *file*. Records in file returns the total number of records in the file.

The following example shows a loop technique commonly used to move through all the records in a selection. The same action can also be accomplished with the APPLY TO SELECTION command.

```
DEFAULT FILE ([People])                   ` Set the default file
FIRST RECORD                              ` Start at the first record in the selection
For ($i; 1; Records in selection)         ` Loop once for each record
  Do Something                            ` Do something with the record
  NEXT RECORD                             ` Move to the next record
End for
```

APPLY TO SELECTION

L69,

APPLY TO SELECTION ({*file*}; *statement*)

Parameter	Type	Description
file	File	File for which to apply statement
statement	Statement	One line of code or a global procedure

APPLY TO SELECTION applies *statement* to each record in the current selection of *file*. The *statement* can be a statement or a global procedure. If *statement* modifies a record of *file*, the modified record is saved. If *statement* does not modify a record, the record is not saved. If the current selection is empty, APPLY TO SELECTION has no effect. The *statement* can contain a field from a related file if the relation is automatic.

APPLY TO SELECTION can be used to gather information from the selection of records (for example, a total), or to modify a selection (for example, changing the first letter of a field to uppercase).

The progress thermometer is displayed while APPLY TO SELECTION is executing. The MESSAGES OFF command turns off the progress thermometer. If the progress thermometer is displayed, the user can cancel the operation. If the user cancels, the OK system variable is set to 0. Otherwise, the OK system variable is set to 1.

The following example capitalizes all the names in the file. It uses character reference symbols (the ≤ and ≥ characters) to access the first character of the field. For more information on character referencing, see "String Functions," in Chapter 17.

```
APPLY TO SELECTION ([People]; [People]Name≤1≥ := Uppercase ([People]Name≤1≥))
```

If a record is locked during the execution of APPLY TO SELECTION and that record is modified, the record will not be saved. Any locked records are put in a set called LockedSet. After APPLY TO SELECTION has executed, test the LockedSet to see if any records were locked. The following loop will execute until all the records have been modified:

```
Repeat                                          ` Repeat if there are any locked records
  APPLY TO SELECTION ([People]; [People]Name≤1≥ :=
          Uppercase ([People]Name≤1≥))
  USE SET ("LockedSet")                         ` Select only the records that were locked
Until (Records in set ("LockedSet") = 0)        ` Done when there are no locked records
```

DELETE SELECTION

DELETE SELECTION ({*file*})

Parameter	Type	Description
file	File	File from which to delete the current selection

DELETE SELECTION deletes the current selection of records from *file*. If the current selection is empty, DELETE SELECTION has no effect. After the records are deleted, the current selection is empty.

Warning: Deleting a selection of records is a permanent operation, and cannot normally be undone. You can use a transaction if you may need to reverse the deletion. For more information on transactions, see "Using Transactions," in Chapter 16.

The following example displays all the records in the database and allows the user to select which ones to delete. The example has two sections. The first is a global procedure to display the records. The second is a script for a button labeled Delete. Here is the first section:

```
DEFAULT FILE ([People])                    ` Set the default file
ALL RECORDS                                ` Select all records
OUTPUT LAYOUT ("Listing")                  ` Set the layout to list the records
DISPLAY SELECTION                          ` Display all the records
```

Here is the script for the Delete button, which appears in the Footer area of the output layout. The script uses the records the user selected (the UserSet) to delete the selection. (Note that if the user selected no records, DELETE SELECTION has no effect.) Finally, all records are again selected.

```
` Confirm that the user really wants to delete the records
CONFIRM ("You selected " + String (Records in set ("UserSet")) + " people to delete."
          + Char (13) + "Click OK to delete them.")
If (OK = 1)
  USE SET ("UserSet")                      ` Use the records that the user chose
  DELETE SELECTION                         ` Delete the selection of records
End if
ALL RECORDS                                ` Select all records
```

If a record is locked during the execution of DELETE SELECTION, that record is not deleted. Any locked records are represented in a set called LockedSet. After DELETE SELECTION has executed, you can test the LockedSet to see if any records were locked. The following loop will execute until all the records have been deleted:

```
Repeat                                     ` Repeat if there are any locked records
  DELETE SELECTION
  USE SET ("LockedSet")                    ` Select only the records that were locked
Until (Records in set ("LockedSet") = 0)   ` Done when there are no locked records
```

MERGE SELECTION

MERGE SELECTION (*{file}*; *{document type}*)

Parameter	Type	Description
file	File	File to merge
document type	String	Macintosh document type (4 characters)

MERGE SELECTION allows you to merge data with an external document. The document is usually for a word processor, to perform mail merge. Mail merge inserts data from each record into a new copy of the document, thereby generating "customized" documents. MERGE SELECTION requires that a "merge module" be installed. Contact ACIUS or ACI for information on word processors that have merge modules.

FIRST RECORD

FIRST RECORD (*{file}*)

Parameter	Type	Description
file	File	File in which to move to the first record

FIRST RECORD makes the first record of the current selection of *file* the current record, and loads the record from disk. All search, selection, and sorting commands also set the current record to the first record. If the current selection is empty, FIRST RECORD has no effect.

If this command is used during data entry, it acts differently from a "First Record" automatic button action. It will not execute a new Before phase, and the record must be saved with SAVE RECORD.

The following example shows a loop technique commonly used to move through all the records in a selection. You can accomplish the same action by using the APPLY TO SELECTION command.

```
DEFAULT FILE ([People])          ` Set the default file
FIRST RECORD                     ` Start at the first record in the selection
For ($i; 1; Records in selection) ` Loop once for each record
  Do Something                   ` Do something with the record
  NEXT RECORD                    ` Move to the next record
End for
```

LAST RECORD

LAST RECORD ({*file*})

Parameter	Type	Description
file	File	File in which to move to the last record

LAST RECORD makes the last record of the current selection of *file* the current record, and loads the record from disk. If the current selection is empty, LAST RECORD has no effect.

If this command is used during data entry, it acts differently from a "Last Record" automatic button action. It will not execute a new Before phase, and the record must be saved with SAVE RECORD.

The procedure in the following example moves through all the records in a file from the end to the beginning. It displays each record, pausing for three seconds between records.

```
DEFAULT FILE ([People])                  ` Set the default file
OUTPUT LAYOUT ("Pictures")               ` Set the output layout
ALL RECORDS                              ` Select all records to display
LAST RECORD                              ` Move to the end of the file
For ($i; 1; Records in file)             ` Loop once for each record
  DISPLAY RECORD                         ` Display the record
  $T := Current time + 3                 ` Initialize for a timing loop
  While (Current time < $T)              ` Delay for about 3 seconds
  End while
  PREVIOUS RECORD                        ` Move to the previous record
End for
```

NEXT RECORD

NEXT RECORD ({*file*})

Parameter	Type	Description
file	File	File in which to move to the next record

NEXT RECORD makes the next record of the current selection of *file* the current record, and loads the record from disk. If the current selection is empty, or Before selection or End selection is TRUE, NEXT RECORD has no effect.

If NEXT RECORD moves the current record pointer past the end of the current selection, End selection returns TRUE, and there is no current record. If End selection returns TRUE, use FIRST RECORD, LAST RECORD, or GOTO SELECTED RECORD to move the current record pointer back into the current selection.

If this command is used during data entry, it acts differently from a "Next Record" automatic button action. It will not execute a new Before phase, and the record must be saved with SAVE RECORD.

See the FIRST RECORD example, earlier in this section.

PREVIOUS RECORD

PREVIOUS RECORD ({*file*})

Parameter	Type	Description
file	File	File in which to move to the previous record

PREVIOUS RECORD makes the previous record of the current selection of *file* the current record, and loads the record from disk. If the current selection is empty, or Before selection or End selection is TRUE, PREVIOUS RECORD has no effect.

If PREVIOUS RECORD moves the current record pointer before the current selection, Before selection returns TRUE, and there is no current record. If Before selection returns TRUE, use FIRST RECORD, LAST RECORD, or GOTO SELECTED RECORD to move the current record pointer back into the current selection.

If this command is used during data entry, it acts differently from a "Previous Record" automatic button action. It will not execute a new Before phase, and the record must be saved with SAVE RECORD.

See the LAST RECORD example, earlier in this section.

Before selection

Before selection ({*file*}) → Boolean

Parameter	Type	Description
file	File	File for which to test if before selection

Before selection returns TRUE when the current record pointer is before the current selection of *file*. Before selection is commonly used to check whether PREVIOUS RECORD has moved the current record pointer before the first record. If the current selection is empty, Before selection returns TRUE.

Before selection also returns TRUE in the first header when a report is being printed with PRINT SELECTION or from the Print menu. You can use the following statement to test for the first header, and print a special header for the first page:

If (In header & Before selection)

The procedure in the following example is used during the printing of a report. It sets a variable, vTitle, to print in the Header area on the first page.

```
If (In header & Before selection ([Finances]))      ` If the first page of a report, set the title.
  vTitle := "Corporate Report 1988"                 ` Set the title for the first page
Else
  vTitle := ""                                      ` Clear the title for all other pages
End if
```

End selection

End selection (*{file}*) ➛ Boolean

Parameter	Type	Description
file	File	File for which to test if after selection

End selection returns TRUE when the current record pointer is after the end of the current selection of *file*. End selection is commonly used to check whether NEXT RECORD has moved the current record pointer past the last record. If the current selection is empty, End selection returns TRUE.

To move the current record pointer back into the selection, use LAST RECORD, FIRST RECORD, or GOTO SELECTED RECORD. PREVIOUS RECORD does not move the pointer back into the selection.

End selection also returns TRUE in the last footer when a report is being printed with PRINT SELECTION or from the Print menu. You can use the following statement to test for the last footer, and print a special footer for the first page:

```
If (In footer & End selection)
```

The procedure in the following example is used during the printing of a report. It sets a variable, vFooter, to print in the Footer area on the last page.

```
If (In footer & End selection ([Finances]))         `.If the last page of a report, set the footer
  vFooter := "©1988  Acme Corp."                    ` Set the footer for the last page
Else
  vFooter := ""                                     ` Clear the footer for all other pages
End if
```

Searching

SEARCH BY LAYOUT	SEARCH BY FORMULA	SEARCH BY INDEX
SEARCH	SEARCH SELECTION	SEARCH SUBRECORDS

The 4th DIMENSION language contains a number of commands you can use to search for records. They all perform the same basic role—searching through the records of a file, looking for records that match a set of criteria—but each does the task in a different way. When each command has finished executing, it creates a selection of the records that were found.

Searches may be simple or complex. You can search for a single record, such as employee ID number 57. You can search for a selection of records, such as all companies in New York.

A search for records in a file can use a field from a related file if the relation is automatic. See the *4th DIMENSION Design Reference* for information on defining file relations.

Searches that use indexes are generally the fastest. This is especially true when the number of records to search through is large.

It is recommended that you use the SEARCH command whenever possible. The SEARCH command uses the same search techniques as does the Search editor in the User environment. These searches are optimized, using indexes first and then doing a sequential search if needed.

SEARCH BY FORMULA and SEARCH SELECTION are extremely powerful and flexible search commands. They are not restricted to a specific search syntax as are SEARCH and SEARCH BY INDEX. You can use them to do sophisticated searches, such as a search for a substring within a field, or a search based on a calculation. The searches performed by these commands are always sequential searches and therefore slower than an indexed search.

SEARCH SUBRECORDS searches within one record's subrecords. It finds a selection of subrecords, not a selection of records. If you use one of the other search commands to search on a subfield, it will find a selection of records, not a selection of subrecords.

A progress thermometer is displayed while a search is performed. Use MESSAGES OFF to turn off the thermometer. The user can stop the search by clicking either the Stop button in the thermometer or the Cancel button in a search window. After a search, you can test the OK system variable to see if the search was completed. The OK system variable is set to 1 if the search was completed, and to 0 if it wasn't. If you want to display records found by a search command, use DISPLAY SELECTION or MODIFY SELECTION.

SEARCH BY LAYOUT

SEARCH BY LAYOUT ({*file*}; {*layout*})

Parameter	Type	Description
file	File	File for which to return selection of records
layout	String	Search layout

SEARCH BY LAYOUT performs the same action as does the Search by Layout menu item in the User environment. SEARCH BY LAYOUT searches *file* for the data that the user enters into *layout*. The layout must be a layout that belongs to *file*. If *layout* is not specified, the current input layout will be used. The layout must contain the fields that the user is searching for. The search is an intelligent search; indexed fields are automatically used to optimize the search.

If the user clicks an Accept button or presses the Enter key, the OK system variable is set to 1 and the search is performed. If the user clicks a Cancel button or presses the "cancel" key combination, the OK system variable is set to 0 and the search is canceled.

See the *4th DIMENSION User Reference* for information on using the Search by Layout menu item in the User environment.

The procedure in the following example displays the layout named My Search to the user. If the user accepts the layout and performs the search (that is, if the OK system variable is set to 1), the records are displayed.

```
SEARCH BY LAYOUT ([People]; "My Search")    ` Display the layout and perform the search
If (OK = 1)                                  ` If the user performed the search...
  DISPLAY SELECTION ([People])               ` display the records
End if
```

Q44/45
T143-150
U47-63

SEARCH L69,16

SEARCH ({*file*})

Parameter	Type	Description
file	File	File for which to return selection of records

SEARCH ({*file*}; *search argument*; {*})

Parameter	Type	Description
file	File	File for which to return selection of records
search argument		Search argument
*		Continue search flag

SEARCH has two forms. Both forms return a selection of records for *file*. If any indexed fields are specified, the search is optimized: Indexed searches are performed first, resulting in a search that takes the least amount of time possible.

The first form presents the Search editor for *file*. It allows the user to build a search argument within the editor and perform the search.

Figure 14-1 shows the Search editor.

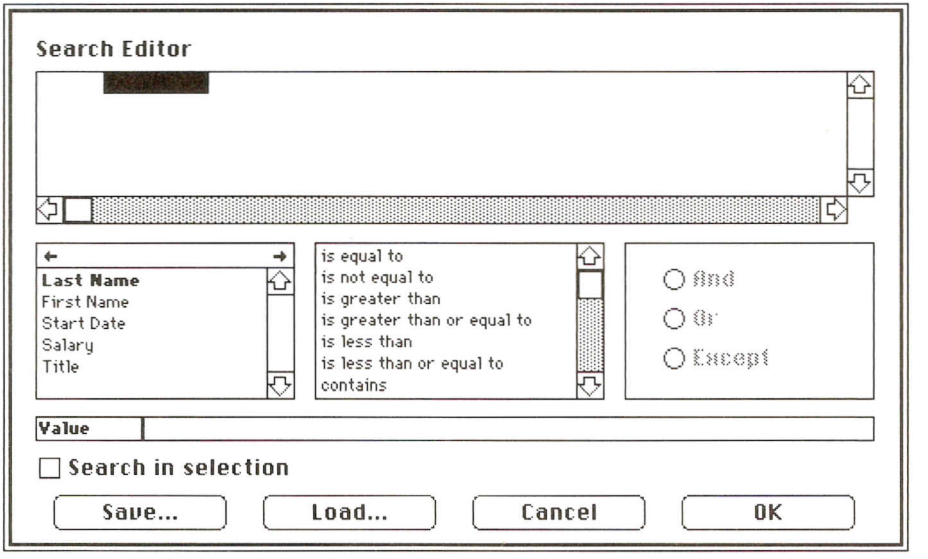


Figure 14-1
The Search editor

If a SEARCH command for *file* has been previously executed with the optional * parameter, then the Search editor is not presented; instead, the search is performed.

The second form of the SEARCH command performs an intelligent search, using *search argument*, and returns a selection of records for *file*.

Complex searches can be "built." You build a search by executing multiple SEARCH commands. You specify a built search by including the optional * parameter at the end of each SEARCH command. The search arguments are then joined together by a conjunction. Built searches are defined later in this command description.

Specifying the Search Argument

The *search argument* uses the format

{conjunction} {;} field {;} comparator {;} value

The *conjunction* is used optionally to join the SEARCH to a previously executed SEARCH.

The *conjunctions* available are the same as those in the User environment Search editor. They are shown in Table 14-1.

Table 14-1
Search conjunctions

Conjunction	Symbol
AND	&
OR	|
Except	#

The *conjunction* is a constant. It cannot be a string.

The *conjunction* is not used if the SEARCH is the first SEARCH command in a built search, or if there is only one SEARCH command executed to perform the search.

The *field* is the field to search. The *field* may belong to another file if it is related by an automatic relation to *file*.

The *comparator* is the comparison that is made between *field* and *value*.

The *comparator* is one of the symbols shown in Table 14-2.

Table 14-2
Search comparison symbols

Comparison	Symbol
Equal to	=
Not equal to	#
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=

The *comparator* can be a constant or a string. If it is a constant, semicolons may optionally surround the *comparator*. If it is a string, semicolons must surround the *comparator*.

The *value* is the data against which *field* will be compared. The *value* can be any expression that evaluates to the same data type as *field*, or it may be a string. If it is a string, it will automatically be converted to the correct data type.

The *value* is evaluated once, at the beginning of the search. The *value* is not evaluated for each record.

wildcard

To search for a string contained in a string (a contains search), use the wildcard symbol (@) in *value*.

Creating Built Searches

You can delay the execution of a search by using a series of SEARCH commands, with the asterisk (*) as the last parameter for each command. This method allows you to build a search with multiple search arguments. To perform the built search, you execute SEARCH without the * parameter. You can also perform the built search by just specifying SEARCH without any parameters.

Here are the rules for built searches:

- The first search argument must not contain a conjunction.
- Each successive search argument *must* begin with a conjunction.
- The first search and every other search, except the last, must use the * parameter.
- To perform the search, do not specify the * parameter in the last SEARCH command. Alternatively, you may execute the SEARCH command without any parameters (the first form of the command).
- Each file maintains its own built search. This means that you can create multiple built searches simultaneously, one for each file. You must use the *file* parameter or set the default file to specify which file to use.

Search Examples

The following command finds the records for all the people named Smith. Remember that 4th DIMENSION is not case sensitive. The Last Name field is indexed. The SEARCH command automatically uses the index for a fast search.

```
SEARCH ([People]; [People]Last Name = "smith")      ` Find every person named Smith
```

The following example finds the records for all the people named John Smith. The Last Name field is indexed. The First Name field is not indexed. When the search is performed, it quickly does an indexed search and reduces the selection of records to those for people named Smith. The search for records with John in the First Name field is then performed sequentially.

Notice that the first SEARCH command includes an asterisk (*) as the last parameter. Including the * prevents the search from happening immediately. The second SEARCH is "built" onto the first with the AND conjunction (&). This causes the search to find the records for all the people whose last name is Smith "and" whose first name is John. The second SEARCH causes the search to be performed, because there is no asterisk (*) parameter.

```
SEARCH ([People]; [People]Last Name = "smith"; *)   ` Find every person named Smith...
SEARCH ([People]; &; [People]First Name = "john")   ` with a first name of John
```

The following search example finds the records for people named Smith or Jones. The Last Name field is indexed. The SEARCH command uses the Last Name index for both searches. The second SEARCH is built onto the first with the OR conjunction (|). This causes the search to find the records for all the people whose last name is Smith "or" whose last name is Jones. Note that the search arguments use strings for the comparators.

```
SEARCH ([People]; [People]Last Name; "="; "smith"; *)      ` Find every person named Smith...
SEARCH ([People]; | [People]Last Name; "="; "jones")       ` or Jones
```

The following example finds the records for people who do not have a company name. It does this by finding entries with empty fields (the empty string).

```
SEARCH ([People]; [People]Company = "")      ` Find every person with no company
```

The following example finds the record for every person whose last name is Smith and who works for a company based in New York. The second search uses a field from another file. This search can be done because the [People] file is related to the [Company] file.

```
SEARCH ([People]; [People]Last Name = "smith"; *)   ` Find every person named Smith...
SEARCH ([People]; & [Company]State = "NY")          ` who works for a company based in NY
```

The following example finds the record for every person whose name falls between *A* (included) and *M* (included).

```
SEARCH ([People]; [People]Name < "n")          ` Find every person from A to M
```

The following example finds the records for all the people living in the San Francisco or Los Angeles areas (ZIP codes beginning with 94 or 90).

```
SEARCH ([People]; [People]ZIP Code = "94@"; *)   ` Find every person in the SF...
SEARCH ([People]; | [People]ZIP Code = "90@")    ` or Los Angeles areas
```

The following example searches an indexed subfield. The search returns a selection of parent records (records for the [People] file). It does not return a selection of subrecords. The result of the search would be the selection of records for all the people who have a child named Sabra.

```
SEARCH ([People]; [People]Children'Name = "Sabra")   ` Find people with child named Sabra
```

The following example finds the record that matches the invoice number entered in the request dialog box.

```
vFind := Request ("Find invoice number:")          ` Get an invoice number from the user
If (OK = 1)                                        ` If the user pressed OK...
  SEARCH ([Invoice]; [Invoice]Number = vFind)      ` find the invoice number that matches vFind
End if
```

The following example finds the records for the invoices entered in 1988. It does this by finding all records that are after 12/31/87 and before 1/1/89. Note that the second search uses a string to represent the date. The SEARCH command automatically converts a string to the correct data type (in this case, a date).

```
SEARCH ([Invoice]; [Invoice]In Date > !12/31/87!; *)   ` Find invoices after 12/31/87...
SEARCH ([Invoice]; & [Invoice]In Date < "1/1/89")      ` and before 1/1/89
```

The following example finds the record for each employee whose salary is between $10,000 and $50,000. The search includes the employees who make $10,000, but excludes those who make $50,000.

```
DEFAULT FILE ([Employee])                          ` Set the default file
SEARCH ([Employee]Salary >= 10000; *)              ` Find employees who make between...
SEARCH (& [Employee]Salary < 50000)                ` $10,000 and $50,000
```

The following example finds the records for the employees in the marketing department who have salaries over $20,000. The Salary field is searched first because it is indexed. Notice that the second search uses a field from another file. It can do this because the [Dept]Name field is related to the [Employee] file with an automatic relation. Although the [Dept]Name field is indexed, this is not an indexed search because the relation must be performed sequentially for each record in the [Employee] file.

```
` Find employees with salaries over $20,000 who are in the marketing department.
SEARCH ([Employee]; [Employee]Salary > 20000; *)
SEARCH ([Employee]; & [Dept]Name = "marketing")
```

The following example finds the records for a group of employees. The employees are found by means of the employee ID. The ID is entered by the user into a request dialog box. For each entry that the user makes, one SEARCH is executed. The first SEARCH is not in the While loop because it does not need the OR conjunction. The SEARCH inside the While loop uses the OR conjunction to join the searches together. When the user presses the Cancel button in the request dialog box, the While loop terminates and the search is executed.

```
DEFAULT FILE ([Employee])                          ` Set the default file
vFind := Request ("Employee ID:")  L241            ` Get the first employee ID
If (OK = 1)                        OK L383         ` If the user did not press Cancel...
  SEARCH ([Employee]ID = vFind; *)                 ` execute the first SEARCH command
End if
` Loop until the user presses the Cancel button
While (OK = 1)
  vFind := Request ("Employee ID:")  L241          ` Get another employee ID
  If (OK = 1)                                      ` If the user did not press Cancel...
    SEARCH ( | [Employee]ID = vFind; *)            ` execute another SEARCH command
  End if
End while
SEARCH                                             ` Perform the search
```

The following example searches for information that was entered into the variable Var. The search could have many different results, depending on the value of Var. For example:

- If Var equals "Copyright@", the file selection contains all laws with texts beginning with *Copyright*.
- If Var equals "@Copyright@", the file selection contains all laws with texts containing at least one occurrence of *Copyright*.
- If Var equals "@Copyright", the file selection contains all laws with texts ending with *Copyright*.

```
SEARCH ([Laws]; [Laws]Text = Var)                  ` Find all laws that "match" the Var variable
```

SEARCH BY FORMULA
SEARCH SELECTION

SEARCH BY FORMULA ({*file*}; {*search formula*})
SEARCH SELECTION ({*file*}; {*search formula*})

Parameter	Type	Description
file	File	File for which to return selection of records
search formula	Boolean	Search formula

SEARCH BY FORMULA and SEARCH SELECTION create a new selection of records for *file*. SEARCH BY FORMULA and SEARCH SELECTION work exactly the same way, except that SEARCH BY FORMULA searches every record in the file and SEARCH SELECTION searches only the records in the current selection.

Both commands apply *search formula* to each record in the file or selection. The *search formula* is a Boolean expression that must evaluate to either TRUE or FALSE. If *search formula* evaluates as TRUE, the record is included in the new selection.

The *search formula* may be simple, perhaps comparing a field to a value; or it may be complex, perhaps performing a calculation or even evaluating information in a related file. The *search formula* can be a 4th DIMENSION function, or a function or expression you have created. You can use wildcards in *search formula* when working with Alpha fields or text.

When the search is complete, the first record of the new selection is loaded from disk and made the current record.

These commands always perform a sequential search, not an indexed search. SEARCH BY FORMULA and SEARCH SELECTION are slower than SEARCH and SEARCH BY INDEX when used on indexed fields. The search time is proportional to the number of records in the file or selection.

Note that the first three examples perform the same searches as the first three examples for SEARCH. The difference is that they are always sequential searches, and therefore slower than the optimized searches performed by SEARCH. They are also searches performed only within the current selection, since they use SEARCH SELECTION.

The following example finds the records in the current selection for the people who are named Smith. Remember that 4th DIMENSION is not case sensitive. The Last Name field is indexed, but the index is ignored since this is a sequential search.

```
` Find the people named Smith
SEARCH SELECTION ([People]; [People]Last Name = "smith")
```

The following example finds the records in the current selection for the people who are named John Smith. Note the use of parentheses to control the evaluation of the Boolean expression.

```
` Find the people named John Smith
SEARCH SELECTION ([People]; ([People]First Name = "john")
          & ([People]Last Name = "smith"))
```

The following example finds the records in the current selection for the people who are named Smith or Jones.

```
` Find the people named Smith or Jones
SEARCH SELECTION ([People]; ([People]Last Name = "smith")
          | ([People]Last Name = "jones"))
```

The following example finds the records for all invoices that were entered in December of any year. It does this by applying the Month of function to each record. This search could not be performed any other way without creating a separate field for the month.

```
` Find the invoices entered in December
SEARCH BY FORMULA ([Invoice]; Month of ([Invoice]Entered) = 12)
```

The following example finds records for all the people who have names with more than ten characters.

```
` Find the people with names longer than ten characters
DEFAULT FILE ([People])                    ` Set the default file
SEARCH BY FORMULA (Length ([People]Name) > 10)
```

SEARCH BY INDEX

U60 Search & Modify

SEARCH BY INDEX (*{search argument1} {;...; search argumentN}*)

Parameter	Type	Description
search argument		Search argument

SEARCH BY INDEX works only on indexed fields. It searches on all records in the file (not just the current selection) that match *search argument*. Although SEARCH gives you greater flexibility, SEARCH BY INDEX is more efficient when you are searching for fields that fall between two values. In all other cases, you should use SEARCH.

If you do not give an argument, SEARCH BY INDEX displays the same dialog box (shown in Figure 14-2) as Search and Modify in the User environment. The dialog box will be displayed for the current default file. You must have set a default file for it to be displayed. This dialog box allows the user to specify the search arguments. If the user accepts the dialog box and performs the search, the OK system variable is set to 1. Otherwise, it is set to 0.

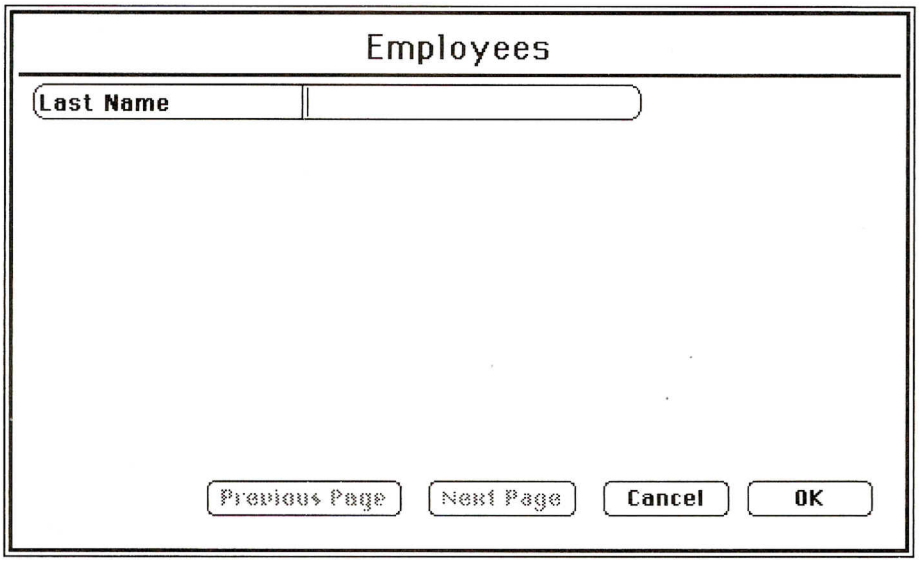


Figure 14-2
The Search by Index dialog box

SEARCH BY INDEX does not need a filename as a parameter. It uses the file prefix from the field to determine what file to search on. Unlike SEARCH, it cannot search on fields from another file.

When the search is complete, SEARCH BY INDEX loads the first record of the new current selection from disk and makes it the current record.

The *search argument* parameter recognizes only two operators: the Equal operator (=) and the Between operator (±). The Equal operator tests for the equality of string, numeric, time, or date values. The Between operator tests for string, numeric, time, or date values that equal or fall between its parameters. (To display the ± character, press the Option-+ keys.)

The *search argument* parameter is constructed in the following manner:

field = *value*

or

field ± *value1*; *value2*

There can be multiple *search argument* parameters, each one set off by a semicolon. 4th DIMENSION automatically performs an AND operation on these search arguments.

The wildcard character (@) works only with string expressions. You can use the wildcard character only at the end of a string expression.

The following example finds the record for every person whose first name starts with *J* and whose last name is Smith.

SEARCH BY INDEX ([People]First Name = "J@"; [People]Last Name = "Smith")

The following example finds all records for invoices with a date of sale falling between January 1, 1985 (included) and the current date (included).

SEARCH BY INDEX ([Invoices]Date of sale ± !1/1/1985!; **Current date**)

The following example finds all records for invoices with a part number falling between 5003 (included) and 5009 (included).

SEARCH BY INDEX ([Parts]Number ± "5003"; "5009")

SEARCH SUBRECORDS

SEARCH SUBRECORDS (*subfile*; *search formula*)

Parameter	Type	Description
subfile	Subfile	Subfile to search
search formula	Boolean	Search formula

SEARCH SUBRECORDS searches *subfile* and creates a new subselection. This is the only command that searches subrecords and returns a selection of subrecords. The *search formula* is applied to each subrecord in *subfile*. If the formula evaluates as TRUE, the subrecord is added to the new subselection. When the search is complete, SEARCH SUBRECORDS makes the first subrecord the current subrecord of *subfile*.

Remember that SEARCH SUBRECORDS searches only the subrecords of the currently selected parent record, and not all the subrecords associated with the several records of the parent file. SEARCH SUBRECORDS does not change the current parent record.

Typically, *search formula* tests a subfield against a variable or a constant, using a relational operator. The *search formula* can contain multiple tests that are joined by AND conjunctions (&) or OR conjunctions (|). The *search formula* can also be or contain a function. The wildcard character (@) works in string arguments.

If neither a current record nor a higher-level subrecord exists, SEARCH SUBRECORDS has no effect.

The following example finds all subrecords containing phone numbers in the 408 area code.

SEARCH SUBRECORDS ([Addresses]Phone; [Addresses]Phone'Number = "408@")

Sorting

SORT BY FORMULA
SORT SELECTION
SORT FILE
SORT SUBSELECTION

Sorting is among the most common of database operations. The commands in this section are often used before a selection of records is displayed or printed. They can be used to sort the records in ascending order, for example from *A* to *Z*, or in descending order, for example from *Z* to *A*. All of the sort commands can sort on more than one level. SORT BY FORMULA can sort on calculated information.

SORT BY FORMULA

U178 , U62 (search)

SORT BY FORMULA (*file*; *expression1*; {*direction1*} {;...; *expressionN*; {*directionN*}})

Parameter	Type	Description
file	File	File to sort
expression	String or number or date or time or Boolean	Expression on which to sort
direction	> or <	> to sort ascending; < to sort descending

SORT BY FORMULA sorts the current selection of *file* according to *expression*. You can sort on multiple expressions within one statement.

Note that you must specify *file*. You cannot use a default file.

SORT BY FORMULA sorts the current selection into ascending or descending order. The *direction* parameter specifies whether to sort the records of *file* in ascending or descending order. If *direction* is the "greater than" symbol (>), the sort is ascending. If *direction* is the "less than" symbol (<), the sort is descending. If *direction* is not specified, the sort is ascending.

Once the sort is completed, the first record of the sorted selection is loaded from disk and is the current record.

During a sort operation, the progress thermometer is displayed, unless you have previously called MESSAGES OFF. After a sort operation, you can test to see whether the sort operation was completed, by checking the OK system variable. The OK system variable is set to 1 if the sort was completed, and to 0 if it wasn't. If the user clicks the Cancel button in the standard sort window, or the Stop button in the standard progress window, the OK system variable is set to 0.

The following example sorts the records of the [People] file into descending order based on the length of each person's last name. The record for the person with the longest last name will be first in the current selection.

SORT BY FORMULA ([People]; **Length** ([People]Last Name); >)

SORT SELECTION

U 80

SORT SELECTION (*{file}*)

Parameter	Type	Description
file	File	File to sort

SORT SELECTION (*{file}*; *field1*; *{direction1}* *{;...; fieldN; {directionN}}*)

Parameter	Type	Description
file	File	File to sort
field	Field	Field on which to sort
direction	> or <	> to sort ascending; < to sort descending

SORT SELECTION has two forms.

The first form displays the Sort dialog box (Figure 14-3) from the User environment, and lets the user specify the sort.

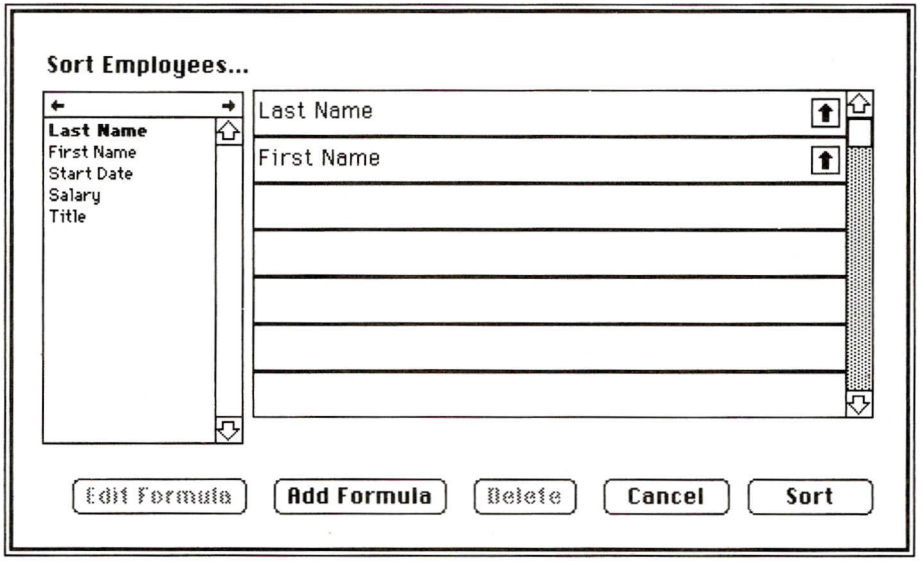


Figure 14-3
The Sort dialog box

The second form sorts the current selection of *file* according to the parameters. If *file* is not specified, SORT SELECTION sorts the current selection of the file containing *field1*. You can sort based on multiple fields within one statement. Once the sort is completed, the first record of the sorted selection is loaded from disk and is the current record.

The *direction* parameter specifies whether to sort *field* in ascending or descending order. If *direction* is the "greater than" symbol (>), the sort is ascending. If *direction* is the "less than" symbol (<), the sort is descending. If *direction* is not specified, the sort is ascending.

If only one field is specified and it is indexed, the index is used for the sort. If the field is not indexed or if there is more than one field, the sort is performed sequentially.

Automatic many-to-one related fields can be used to sort on. The related fields can be specified for all but the first field.

During a sort operation, the progress thermometer is displayed, unless you have previously called MESSAGES OFF. After a sort operation, you can test to see whether the sort operation was completed, by checking the OK system variable. The OK system variable is set to 1 if the sort was completed, and to 0 if it wasn't. If the user clicks the Cancel button in the standard sort window, or the Stop button in the standard progress window, the OK system variable is set to 0.

The following example displays the Sort dialog box for the file [Addresses].

SORT SELECTION ([Addresses])

The following example sorts the current selection of [Addresses] into ascending order, first by ZIP code, and then by last name.

SORT SELECTION ([Addresses]; [Addresses]ZIP; >; [Addresses]Last Name; >)

SORT FILE L 69,19 U 81

SORT FILE (*file*; *field1*; {*direction1*} {;...; *fieldN*; {*directionN*}})

Parameter	Type	Description
file	File	File to sort
field	Field	Field on which to sort
direction	> or <	> to sort ascending; < to sort descending

SORT FILE performs the same action as SORT SELECTION, except that the file is sorted permanently. See the description of SORT SELECTION, earlier in this section, for more information on the command syntax.

Note that you must specify *file*. You cannot use a default file.

When a file is sorted permanently, the records will be displayed and printed in the new order, unless they are again sorted.

SORT FILE must perform three operations to complete its task:

1. The file is sorted.
2. The new sorted order is saved.
3. All indexes for the file are rebuilt.

Since this command restructures the data file and may be time consuming, this command is executed infrequently.

The following example sorts the [Addresses] file permanently into ascending order, first by ZIP code, and then by last name.

SORT FILE ([Addresses]; [Addresses]ZIP; >; [Addresses]Last Name; >)

SORT FILE performs no action when a database is being used in a multi-user environment.

SORT SUBSELECTION L 69 20

SORT SUBSELECTION (*subfile*; *subfield1*; *{direction1}* *{;...; subfieldN; {directionN}}*)

Parameter	Type	Description
subfile	Subfile	Subfile to sort
subfield	Subfield	Subfield on which to sort
direction	> or <	> to sort ascending; < to sort descending

SORT SUBSELECTION sorts the current subselection of *subfile*. It sorts only the subselection of the current parent record.

The *direction* parameter specifies whether to sort *subfield* in ascending or descending order. If *direction* is the "greater than" symbol (>), the sort is ascending. If *direction* is the "less than" symbol (<), the sort is descending.

You can specify more than one level of sort, by including more subfields and sort symbols.

Once the sort is completed, the first subrecord of the sorted subselection is the current subrecord. Sorting subrecords is a dynamic process. Subrecords are never saved in their sorted order. If neither a current record nor a higher-level subrecord exists, SORT SUBSELECTION has no effect.

The following example sorts the [Stats]Sales subfile into ascending order based on the Sales'Bucks subfield.

SORT SUBSELECTION ([Stats]Sales; [Stats]Sales'Bucks; >)

Managing Records

CREATE RECORD	SAVE RECORD
DUPLICATE RECORD	DELETE RECORD

The commands in this section allow you to manage records, creating and adding new ones, and modifying, duplicating, and deleting existing ones. The management of records is one of the most fundamental purposes of a database.

These commands are for managing data without user intervention—they do not display the data or layouts.

CREATE RECORD

CREATE RECORD ({*file*})

Parameter	Type	Description
file	File	File for which to create a new record

CREATE RECORD creates a new empty record for *file*, but does not display the new record. Use ADD RECORD to create a new record and display it for data entry. CREATE RECORD is used instead of ADD RECORD when the data for the record is assigned with the language. The new record becomes the current record and the current selection (a one-record current selection).

If you execute CREATE RECORD during data entry, a new empty record is created and displayed. If the user then accepts the new record, another new record will be created. This will continue until the user cancels a record.

The record exists in memory only until a SAVE RECORD command is executed for the file. If the current record is changed (for example, by a search) before the record is saved, the new record is lost. You may push the new record before it is saved. For information on pushing records and the record stack, see "Using the Record Stack," in Chapter 16.

L271

The following example archives records that are over 30 days old. The example does this task by creating new records in an archive file. When the archiving is finished, the records that were archived are deleted from the [Accounts] file.

```
DEFAULT FILE ([Accounts])                          ` Set the default file
SEARCH ([Accounts]Entered < (Current date – 30))   ` Find records more than 30 days old
For ($i; 1; Records in selection)                  ` Loop once for each record to archive
  CREATE RECORD ([Archive])                        ` Create a new archive record
  [Archive]Number := [Account]Number               ` Copy the fields to the archive record
  [Archive]Entered := [Account]Entered
  [Archive]Amount := [Account]Amount
  SAVE RECORD ([Archive])                          ` Save the archive record
  NEXT RECORD                                      ` Move to the next account record
End for
DELETE SELECTION                                   ` Delete the account records
```

DUPLICATE RECORD

DUPLICATE RECORD (*{file}*)

Parameter	Type	Description
file	File	File for which to duplicate the current record

DUPLICATE RECORD creates a new record for *file* that is a duplicate of the current record. The new record becomes the current record. If there is no current record, then DUPLICATE RECORD does nothing. You must use SAVE RECORD to save the new record.

DUPLICATE RECORD can be executed during data entry. This allows you to create a "clone" of the currently displayed record. Don't forget that you must first execute SAVE RECORD if you want to save any changes made to the original record. You must also execute SAVE RECORD to save the new record.

The following example is a button script for a button without an automatic action. It saves the currently displayed record, duplicates the record, and then saves the duplicate record.

```
SAVE RECORD ([People])        ` Save the current record
DUPLICATE RECORD ([People])   ` Create a "clone"
SAVE RECORD ([People])        ` Save the new record
```

(Modified Record ({file}) > Boolean)
Modified(field) L152

SAVE RECORD

Accept L157

SAVE RECORD ({*file*})

Parameter	Type	Description
file	File	File for which to save the current record

SAVE RECORD saves the current record of *file*. If the record contains any subrecords, they are saved with the record. If there is no current record, then SAVE RECORD is ignored.

You use SAVE RECORD to save a record that you created or modified with code. A record that has been modified by the user in a layout usually does not need to be saved with SAVE RECORD. A record that has been modified by the user in a layout, but has been canceled, can still be saved with SAVE RECORD.

Here are some cases where SAVE RECORD would be required:

- to save a new record created with CREATE RECORD or DUPLICATE RECORD
- to save data from RECEIVE RECORD
- to save a record modified by a procedure
- to save new or modified subrecord data following an ADD SUBRECORD, CREATE SUBRECORD, or MODIFY SUBRECORD command
- to save a record during a transaction
- during data entry to save the displayed record before using a command that changes to a new current record
- during data entry to save the new current record after using a command that changed to the new current record

The following example is part of a procedure that is reading records from a document or from the serial port. The code segment receives a record and then, if it is received properly, saves it.

```
RECEIVE RECORD ([Customers])        ` Receive record from disk or serial port
If (OK = 1)                         ` If the record is received properly...
  SAVE RECORD ([Customers])         ` save it
End if
```

SAVE RECORD will not save a locked record. When using this command in a multi-user environment, you must first be sure that the record is unlocked. You should test if the record is locked at the time that it is loaded. If the record is locked, the command is ignored, the record is not saved, and no error is returned. See the section "Managing Multi-user Databases," in Chapter 16, for more information on locked records.

DELETE RECORD

DELETE RECORD ({*file*})

Parameter	Type	Description
file	File	File for which to delete the current record

DELETE RECORD deletes the current record of *file*. If there is no current record, DELETE RECORD has no effect. In a layout, you can create a Delete Record button instead of using this command.

After the record is deleted, the current selection for *file* is empty. This means that you cannot use DELETE RECORD to go through a selection of records, deleting the ones you select. Instead, you can create a set of the selected records, and use DELETE SELECTION to delete the records.

Warning: Deleting records is a permanent operation and cannot be undone.

The following example deletes an employee record. The example asks the user what employee to delete, searches for the employee's record, and then deletes it.

```
vFind := Request ("Employee ID to delete:")        ` Get an employee ID to search for
If (OK = 1)                                        ` If the user did not cancel...
  SEARCH ([Employee]; [Employee]ID = vFind)        ` find the employee
  DELETE RECORD ([Employee])                       ` Delete the employee
End if
```

DELETE RECORD will not delete a locked record. When using this command in a multi-user environment, you must first be sure that the record is unlocked. If the record is locked, the command is ignored, the record is not deleted, and no error is returned. See the section "Managing Multi-user Databases," in Chapter 16, for more information on locked records.

Importing and Exporting

EXPORT DIF	EXPORT TEXT	IMPORT SYLK
EXPORT SYLK	IMPORT DIF	IMPORT TEXT

The commands in this section import and export data. Both operations are done through a layout.

EXPORT DIF
EXPORT SYLK
EXPORT TEXT

L69₆

EXPORT DIF ({*file*}; *document*)
EXPORT SYLK ({*file*}; *document*)
EXPORT TEXT ({*file*}; *document*)

Parameter	Type	Description
file	File	File from which to export
document	String	Macintosh document to write

The export commands write data from the records of the current selection of *file* to disk. The data is written to *document*, a Macintosh text document.

The export operation is performed through the current output layout. The export operation writes fields and variables based on the entry order of the output layout. Included layouts are ignored. The layout procedure is executed once for each record that is exported. Both the Before and During phases are TRUE.

The *document* parameter can name a new or existing Macintosh document. If *document* is given the same name as an existing document, the existing document is overwritten. The *document* can include a path that includes volume and folder names. See the section, "Communicating With Documents and the Serial Port," in Chapter 16, for information on document paths.

A progress thermometer is displayed during export. The user can cancel the operation by clicking a button labeled Stop. If the export is successful, the OK system variable is set to 1; otherwise, it is set to 0. The thermometer can be hidden with the MESSAGES OFF command.

The export operation uses the current ASCII map. The ASCII map can be used to convert the data for use by other machines or programs. See the *4th DIMENSION User Reference* for more information on the ASCII map.

For EXPORT TEXT, the default field delimiter is the tab character (ASCII 9). The default record delimiter is the carriage return character (ASCII 13). You can change these defaults by assigning values to the two delimiter system variables, FldDelimit and RecDelimit. The user can change the defaults by specifying them in the Export dialog box. See Appendix D for a table of the Macintosh ASCII codes.

Important: Because text fields can contain carriage returns, be careful when using a carriage return as a delimiter if you are exporting text fields.

The following example exports data to a text document. The procedure first sets the output layout so that the data will be exported through the correct layout. It then changes the delimiter system variables.

```
DEFAULT FILE ([People])          ` Set the default file
OUTPUT LAYOUT ("Exporter")       ` Set the layout for export
FldDelimit := 27                 ` Set field delimiter to Escape character
RecDelimit := 10                 ` Set record delimiter to Line Feed
EXPORT TEXT ("My People")        ` Export to the My People document
```

IMPORT DIF
IMPORT SYLK
IMPORT TEXT

IMPORT DIF ({*file*}; *document*)

IMPORT SYLK ({*file*}; *document*)

IMPORT TEXT ({*file*}; *document*)

Parameter	Type	Description
file	File	File into which to import
document	String	Macintosh document to import from

The import commands read data from *document*, a Macintosh text document, into *file*.

The import operation is performed through the current input layout. The import operation reads fields and variables based on the entry order of the input layout. Included layouts are ignored. If the number of fields or variables in the layout does not match the number of fields being imported, the extras are ignored. The layout procedure is executed once for each record that is imported. The After phase is TRUE.

The *document* parameter can include a path that includes volume and folder names. See the section, "Communicating With Documents and the Serial Port," in Chapter 16, for information on document paths.

A progress thermometer is displayed during import. The user can cancel the operation by clicking a button labeled Stop. If the import is successful, the OK system variable is set to 1; otherwise, it is set to 0. The thermometer can be hidden with the MESSAGES OFF command.

The import operation uses the current ASCII map. The ASCII map can be used to convert the data from other machines or programs. See the *4th DIMENSION User Reference* for more information on the ASCII map.

For IMPORT TEXT, the default field delimiter is the tab character (ASCII 9). The default record delimiter is the carriage return character (ASCII 13). You can change these defaults by assigning values to the two delimiter system variables, FldDelimit and RecDelimit. The user can change the defaults by specifying them in the Import dialog box. See Appendix D for a table of the Macintosh ASCII codes.

L384

Important: Because text fields can contain carriage returns, be careful when using a carriage return as a delimiter if you are importing text fields.

The following example imports data from a text document. The procedure first sets the input layout so that the data will be imported through the correct layout. It then changes the delimiter system variables.

```
DEFAULT FILE ([People])                  ` Set the default file
INPUT LAYOUT ("Importer")                ` Set the layout for import
FldDelimit := 27      9      L384        ` Set field delimiter to Escape character
RecDelimit := 10      13     L384        ` Set record delimiter to Line Feed
IMPORT TEXT ("My People")                ` Import from the My People document
```

When importing through a layout, keep any layout procedure as short as possible, since the After phase will lock the database for all other users as each record is saved.

Managing File Relations

RELATE ONE	CREATE RELATED ONE
RELATE MANY	SAVE RELATED ONE

The commands in this section, in particular RELATE ONE and RELATE MANY, establish and manage the relations between files, for both automatic and nonautomatic relations. See the *4th DIMENSION Design Reference* for information on creating relations between files. D49

Using Automatic File Relations With Commands

Two files can be related with automatic file relations. In general, when an automatic file relation is established it loads or selects the related records in a related file. Many operations cause the relation to be established.

These operations include

- data entry
- listing records on the screen in output listings
- reporting
- operations on a selection of records, such as searches, sorts, and applying a formula

To optimize performance, 4th DIMENSION establishes automatic relations only when data from the related records needs to be used. For each of the operations just listed, if a record with an automatic relation is loaded from disk, the related record or records from the related file are selected. If a relation selects only one record of a related file, that record is loaded from disk. If a relation selects more than one record of a related file, a new current selection of records is created for that file, and the first record in the current selection is loaded from disk.

For example, using the file structure in Figure 14-4, if a record for the [People] file is loaded and displayed for data entry, the related record from the [Companies] file is selected and is loaded. Similarly, if a record for the [Companies] file is loaded and displayed for data entry, the related *records* from the [People] file are selected and the first record is loaded.

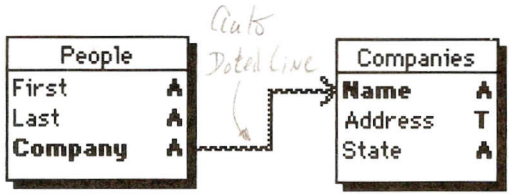


Figure 14-4
Two related files

In Figure 14-4, the [People] file is referred to as the *many file*, and the [Companies] file is referred to as the *one file*. To remember this, think, "There are *many* people related to *one* company." Similarly, the Company field in the [People] file is referred to as the *many field*, and the Name field in the [Companies] file is referred to as the *one field*.

The commands listed in Table 14-3 use automatic relations to load the related records during the operation of the command. All of the commands will establish a many-to-one relation automatically. Only the commands with Yes in the column titled Many Established will automatically establish a one-to-many relation.

Table 14-3
Commands that use automatic relations

Command	Many Established	Command	Many Established
ADD RECORD	Yes	PRINT LABEL	No
ADD SUBRECORD	No	PRINT SELECTION	Yes
APPLY TO SELECTION	No	REPORT	No
DISPLAY SELECTION	No	SEARCH BY FORMULA	Yes
EXPORT DIF	No	SEARCH SELECTION	Yes
EXPORT SYLK	No	SEARCH	Yes
EXPORT TEXT	No	SELECTION TO ARRAY	No
MODIFY RECORD	Yes	SORT BY FORMULA	No
MODIFY SELECTION	Yes (in data entry)	SORT SELECTION	No
MODIFY SUBRECORD	No		

Using Commands to Establish File Relations

Automatic relations don't mean that the related record or records for a file will be selected simply because a command loads a record. After using a command that loads a record, you must explicitly select the related record(s) by using RELATE ONE or RELATE MANY, if you need to access the related data.

Note L 219

Some of the commands listed in Table 14-3 (such as the search commands) load a current record after the completion of the task. In this case, the final record that is loaded *does not* automatically select the record(s) related to it. Again, you must explicitly select the related record(s) by using RELATE ONE or RELATE MANY, if you need to access the related data.

The commands listed in Table 14-4 load a current record. They do not automatically select the related record(s).

Table 14-4
Commands that load a record

Command	Command	Command
ALL RECORDS	NEXT RECORD	SEARCH SELECTION
CREATE RECORD	ONE RECORD SELECT	SORT BY INDEX
FIRST RECORD	PREVIOUS RECORD	SORT FILE
GOTO RECORD	SEARCH	SORT SELECTION
GOTO SELECTED RECORD	SEARCH BY FORMULA	USE SET
LAST RECORD	SEARCH BY INDEX	
LOAD RECORD	SEARCH BY LAYOUT	

AUTOMATIC RELATIONS (one; many)

RELATE ONE

D52, D53

RELATE ONE ({*file*})

Parameter	Type	Description
file	File	File for which to establish all automatic relations

RELATE ONE (*field*; {*choice field*})

Parameter	Type	Description
field	Field	Many field
choice field	Field	Choice field from the one file

RELATE ONE has two forms.

The first form of the command establishes all automatic many-to-one relations for *file*. This means that for each field in *file* that has an automatic many-to-one relation, the command will select the related record in each related file.

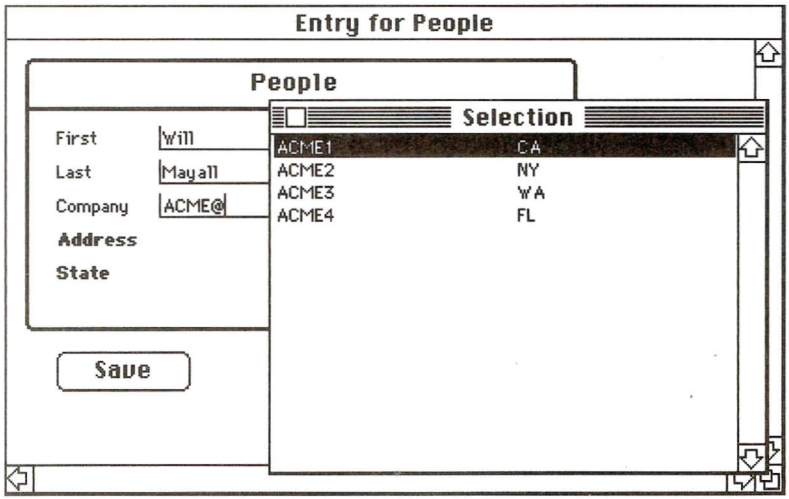
The second form of RELATE ONE selects the record related to *field*. The relation does not need to be automatic. RELATE ONE loads the related record into memory, making it the current record and current selection for its file.

The optional *choice field* can be specified only if *field* is an Alpha field. The *choice field* must be a field in the related file. The *choice field* must be an Alpha field or of a numeric data type.

If *choice field* is specified and more than one record is found in the related file, RELATE ONE displays a selection list of records that match the value in *field*. In the list, the left column displays related field values, and the right column displays *choice field* values.

More than one record might be found if *field* ends with the wildcard character (@). If there is only one match, the relation is to that match, and the list does not appear.

Figure 14-5 shows a record being entered and a selection list displayed in front of the record.



D 54

Figure 14-5
A selection list for a related file

In Figure 14-5, the following command caused the selection list to appear:

Many One

RELATE ONE ([People]Name; [Company]State)

Note L217

The user entered "ACME@" to see a list of all companies whose names begin with ACME, along with each company's state.

Specifying *choice field* is the same as specifying a Wildcard Choice when establishing the file relation. See the *4th DIMENSION Design Reference* for information on specifying a Wildcard Choice.

RELATE ONE works with relations to subfiles, but you must have a relation to the parent file and to the subfile's related field in order for the relation to be properly established. When using a relation to a subrecord, you must first use RELATE ONE to load the related record into memory, then use a second RELATE ONE command for the subfile.

In the following example, the [Invoice] file is related to the [Customers] file with two nonautomatic relations. See Figure 14-6 for the file structure.

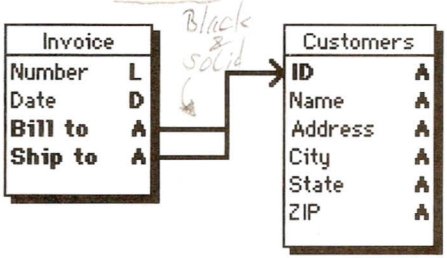


Figure 14-6
Invoice file related to Customers file with nonautomatic relations

One relation is from "[Invoice]Bill to" to [Customers]ID, and the other relation is from "[Invoice]Ship to" to [Customers]ID. Figure 14-7 shows the [Invoice] file layout that displays the [Invoice] file's "bill to" and "ship to" information.

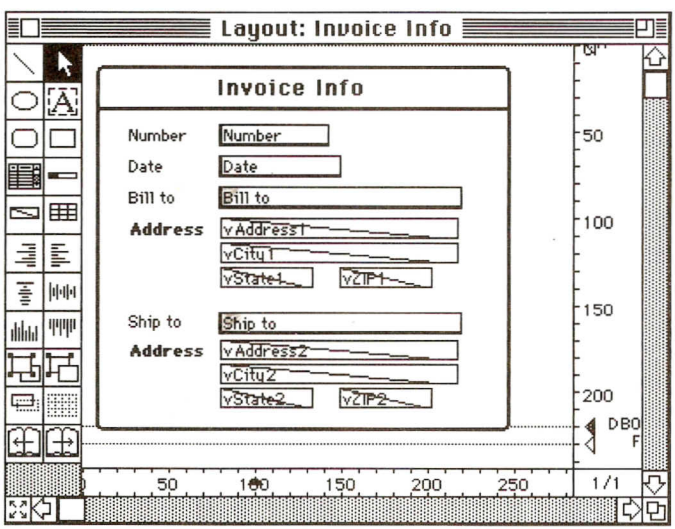


Figure 14-7
Layout to display related information

Since both relations are to the same file, [Customers], the information they get must be displayed in variables. If the [Customers] fields were displayed instead, only the data from the second relation would be displayed.

The following two procedures are the scripts for the "[Invoice]Bill to" and "[Invoice]Ship to" fields. Here is the script for the "[Invoice]Bill to" field:

```
RELATE ONE (Bill to; [Customers]Address)
vAddress1 := [Customers]Address
vCity1 := [Customers]City
vState1 := [Customers]State
vZIP1 := [Customers]ZIP
```

Here is the script for the "[Invoice]Ship to" field:

```
RELATE ONE (Ship to; [Customers]Address)
vAddress2 := [Customers]Address
vCity2 := [Customers]City
vState2 := [Customers]State
vZIP2 := [Customers]ZIP
```

RELATE MANY

RELATE MANY (*{file}*)

Parameter	Type	Description
file	File	File to establish all one-to-many relations

RELATE MANY (*field*)

Parameter	Type	Description
field	Field	One field

RELATE MANY has two forms.

The first form establishes all automatic one-to-many relations for *file*. It changes the current selection for each file that has an automatic one-to-many relation to *file*. The new current selections reflect the current value for each related field in the related file (the One file).

The second form establishes the one-to-many relation for *field*. It changes the current selection for only those files that have relations with *field*. This means that the related records in the selecting file become the current selection for that file.

In the following example, three files are related with automatic relations. The file structure is shown in Figure 14-8.

Figure 14-8
Three related files

Both the [People] file and the [Parts] file have a many-to-one relation to the [Companies] file. This also means that the [Companies] file has a one-to-many relation to both the [People] file and the [Parts] file. Figure 14-9 shows a layout for the [Companies] file that will display related records from both the [People] file and the [Parts] file.

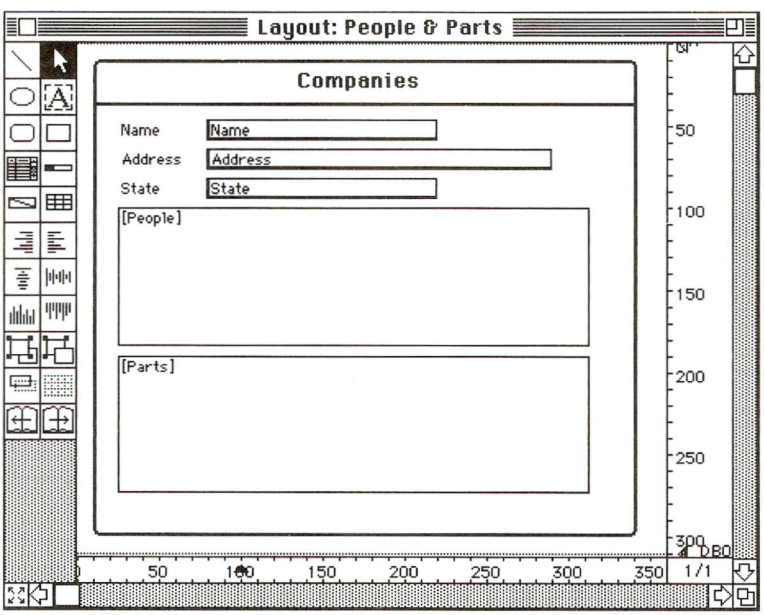


Figure 14-9
Layout that shows related records for two files

When the People & Parts layout in Figure 14-9 is displayed, the related records for both the [People] file and the [Parts] file are loaded. The related records are not loaded if a record for the [Companies] file is selected with code. In this case, you must use the RELATE MANY command.

For example, the following procedure moves through each record of the [Companies] file. For each company, an alert box is displayed. The alert box shows the number of people in the company (the number of related [People] records), and the number of parts they supply (the number of related [Parts] records). (In the example, the argument to the ALERT command is printed on multiple lines for clarity.) Note that the RELATE MANY command is needed, even though the relations are automatic.

```
ALL RECORDS ([Companies])                                  ` Select all records in the file
SORT SELECTION ([Companies]; [Companies]Name)              ` Sort in alphabetical order
For ($i; 1; Records in file ([Companies])                  ` Loop once for each record
  RELATE MANY ([Companies]Name)                            ` Select the related records
  ALERT ("Company: " + [Companies]Name + Char (13) +
         "People in company: " +
         String (Records in selection ([People])) + Char (13) +
         "Number of parts they supply: " +
         String (Records in selection ([Parts])))
  NEXT RECORD ([Companies])                                ` Move to the next record
End for
```

CREATE RELATED ONE

CREATE RELATED ONE (*field*)

Parameter	Type	Description
field	Field	Many field

CREATE RELATED ONE performs two actions. If a related record does not exist for *field* (that is, if a match is not found for the current value of *field*), CREATE RELATED ONE creates a new related record. If a related record exists, CREATE RELATED ONE acts just like RELATE ONE and loads the related record into memory. To save the new or modified record, execute SAVE RELATED ONE.

CREATE RELATED ONE acts differently from CREATE RECORD in two ways. First, the input layout procedure for the current input layout is executed, and second, the blank record is saved even if SAVE RELATED ONE is not executed.

SAVE RELATED ONE

SAVE RELATED ONE (*field*)

Parameter	Type	Description
field	Field	Many field

SAVE RELATED ONE saves the record related to *field*. You must execute a SAVE RELATED ONE command to save any record created with CREATE RELATED ONE, or when you want to save modifications to a record loaded with RELATE ONE. SAVE RELATED ONE does not apply to subfiles, because saving the parent record automatically saves the subrecords.

SAVE RELATED ONE will not save a locked record. When using this command in a multi-user environment, you must first be sure that the record is unlocked. If the record is locked, the command is ignored, the record is not saved, and no error is returned. See the section "Managing Multi-user Databases," in Chapter 16, for more information on locked records.

Managing Old Data

Old	SAVE OLD RELATED ONE
OLD RELATED ONE	OLD RELATED MANY

When a record is loaded, 4th DIMENSION makes a copy of the loaded record. When the record is modified, the changes are made to the copy. The copy is not written to disk until the record is accepted or saved. Until the copy is saved, you can use the commands in this section to access the old data (data from before the modification).

Old

Old (*field*) → String, number, date, or time

Parameter	Type	Description
field	Field	Field for which to return old value

For the current record, Old returns the value *field* held before *field* was modified. In other words, it returns the value of the field as it is stored on disk. Old works on the field whether the field has been modified by a procedure or by the user.

If a record is new, Old returns an empty value for the field. For example, if the record is new and the field is an Alpha field, Old returns an empty string. If the field is a numeric field, Old returns zero (0). If the field is a date field, Old returns !00/00/00!

Old may not be applied to Text or Picture fields. It may be applied to all other field types, including subfields, but has no meaning when applied to a complete subfile.

8 July 98
Example
Check Book
(Level III)
Transactions
(Check Number
Script)

OLD RELATED ONE

OLD RELATED ONE (*field*)

Parameter	Type	Description
field	Field	Many field

OLD RELATED ONE operates the same way as RELATE ONE does, except that OLD RELATED ONE uses the old value of *field* to establish the relation.

OLD RELATED ONE loads the record previously related to the current record. The fields in that record can then be accessed. If you want to modify this old related record and save it, you must execute SAVE OLD RELATED ONE.

SAVE OLD RELATED ONE

SAVE OLD RELATED ONE (*field*)

Parameter	Type	Description
field	Field	Many field

SAVE OLD RELATED ONE operates the same way as SAVE RELATED ONE does, but uses the old relation to the field, to save the old related record. Before you use SAVE OLD RELATED ONE, you must load the record with OLD RELATED ONE. Use SAVE OLD RELATED ONE when you want to save modifications to a record loaded with OLD RELATED ONE.

SAVE OLD RELATED ONE will not save a locked record. When using this command in a multi-user environment, you must first be sure that the record is unlocked. If the record is locked, the command is ignored, the record is not saved, and no error is returned. See the section "Managing Multi-user Databases," in Chapter 16, for more information on locked records.

OLD RELATED MANY

OLD RELATED MANY (*field*)

Parameter	Type	Description
field	Field	One field

OLD RELATED MANY establishes the one-to-many relation for *field,* based on the value of *field* before it was modified (the saved value). The related records in the related file are loaded into the selection for that file.

Working With Subrecords

ADD SUBRECORD	Records in subselection	PREVIOUS SUBRECORD
MODIFY SUBRECORD	APPLY TO SUBSELECTION	Before subselection
CREATE SUBRECORD	FIRST SUBRECORD	End subselection
DELETE SUBRECORD	LAST SUBRECORD	
ALL SUBRECORDS	NEXT SUBRECORD	

The commands in this section allow you to perform tasks with subrecords similar to those you perform with records—you can add and modify subrecords, apply a formula, and delete subrecords.

You can also move within a selection of subrecords with the commands FIRST SUBRECORD, LAST SUBRECORD, NEXT SUBRECORD, and PREVIOUS SUBRECORD. It is important that you create the correct subselection before using these commands. If there are no records in the current subselection, the commands do nothing.

When a layout is displayed, the movement commands all have equivalent actions that can be assigned to buttons without any programming.

ADD SUBRECORD
MODIFY SUBRECORD

ADD SUBRECORD (*subfile*; *layout*; *{*}*)
MODIFY SUBRECORD (*subfile*; *layout*; *{*}*)

Parameter	Type	Description
subfile	Subfile	Subfile to use for data entry
layout	String	Layout to display
*		Hide scroll bars and size box

ADD SUBRECORD lets the user add a new subrecord to *subfile*, using *layout*. ADD SUBRECORD creates a new subrecord in memory, makes it the current subrecord, and displays *layout*. There must be a current record for the parent file. If a current record does not exist, ADD SUBRECORD has no effect. The *layout* must belong to *subfile*.

MODIFY SUBRECORD acts exactly like ADD SUBRECORD, except that MODIFY SUBRECORD displays the current subrecord for modification. If there is not a current subrecord, then MODIFY SUBRECORD does nothing.

The subrecord is kept in memory (accepted) if the user clicks an Accept button or presses the Enter key, or if the ACCEPT command is executed. Accepting the subrecord sets the OK system variable to 1. The new subrecord is not saved to disk until the parent record is saved.

The subrecord is not saved if the user clicks a Cancel button or presses the "cancel" key combination (Command-.), or if the CANCEL command is executed. Canceling sets the OK system variable to 0.

Subrecords are always added to the current parent record. If a subfile is within a subfile, make sure that the proper parent record (subrecord) is first selected.

The *layout* is displayed in the window with scroll bars and a size box. Specifying the optional asterisk (*) causes the window to be drawn without scroll bars or a size box.

The layout procedure execution cycle is started if a layout procedure exists for *layout*. Scripts that exist for *layout* may also be executed, depending on the user's actions. For more information on the execution cycle, see the Chapter 5 in Part I of this manual.

The following example is part of a global procedure. It adds a subrecord for a new child to an employee's record. The data for the children is stored in a subfile named [Employees]Children. Notice that the [Employees] record must be saved in order for the new subrecord to be saved.

```
ADD SUBRECORD ([Employees]Children; "Add Child")
If (OK = 1)                                  ` If the user accepted the record...
  SAVE RECORD ([Employees])                  ` save the employee's record
End if
```

CREATE SUBRECORD

CREATE SUBRECORD (*subfile*)

Parameter	Type	Description
subfile	Subfile	Subfile for which to create a new subrecord

CREATE SUBRECORD creates a new subrecord for *subfile* and makes the new subrecord the current subrecord. The new subrecord is saved only when the parent record is saved. The parent record may be saved by a command such as SAVE RECORD or by the user's accepting the record. If there is no current record, CREATE SUBRECORD has no effect. To add a new subrecord through a subrecord input layout, use ADD SUBRECORD.

You can create many subrecords without having to save each one individually. All changes to a subfile (additions, modifications, and deletions) are saved only when the parent record is saved.

The following example is a button script. When the script is executed (that is, when the button is pressed), it creates new subrecords for children. The Repeat loop lets the user add children until he or she clicks Cancel. The layout displays the children in an included layout, but will not allow direct data entry into the subfile because the enterable option has been turned off.

```
Repeat                                          ` Repeat until the user clicks Cancel
  vChild := Request("Name (cancel when done):")  ` Get the child's name
  If (OK = 1)                                    ` If the user clicked OK...
    CREATE SUBRECORD(Children)                   ` add a new subrecord for a child
    Children'Name := vChild                      ` Assign the child's name to the subfield
  End if
Until (OK = 0)
```

DELETE SUBRECORD

DELETE SUBRECORD (*subfile*)

Parameter	Type	Description
subfile	Subfile	Subfile from which to delete the current subrecord

DELETE SUBRECORD deletes the current subrecord of *subfile*. If there is no current subrecord, DELETE SUBRECORD has no effect. After the subrecord is deleted, the current subselection for *subfile* is empty. As a result, DELETE SUBRECORD can't be used to scan through a subselection and delete selected subrecords.

The deletion of subrecords is not permanent until the parent record is saved. Deleting a parent record automatically deletes all its subrecords.

To delete a subselection, first select the subrecords to delete, and then use APPLY TO SUBSELECTION. For example, the following line will delete the current subrecords for [File]Subfile:

```
APPLY TO SUBSELECTION ([File]Subfile; DELETE SUBRECORD ([File]Subfile))
```

ALL SUBRECORDS

ALL SUBRECORDS (*subfile*)

Parameter	Type	Description
subfile	Subfile	Subfile in which to select all subrecords

ALL SUBRECORDS makes all the subrecords of *subfile* the current subselection. If a current parent record does not exist, ALL SUBRECORDS has no effect. When a parent record is first loaded, the subselection contains all subrecords. A subselection may not contain all subrecords after ADD SUBRECORD, SEARCH SUBRECORDS, or DELETE SUBRECORD is executed.

The following example selects all subrecords, to be sure they are included in the sum.

```
ALL SUBRECORDS ([Stats]Sales)
Total Sales := Sum ([Stats]Sales'Dollars)
```

Records in subselection

Records in subselection (*subfile*) → Number

Parameter	Type	Description
subfile	Subfile	Subfile for which to count number of subrecords

Records in subselection returns the number of subrecords in the current subselection of *subfile*. Records in subselection applies only to subrecords in the current record. It is the subrecord equivalent of Records in selection. The result is undefined if no parent record exists.

The following example selects all subrecords, then displays the number of children for the parent record.

```
ALL SUBRECORDS ([People]Children)          ` Select all children, then display how many
ALERT ("Number of children: " + String (Records in subselection ([People]Children)))
```

APPLY TO SUBSELECTION

APPLY TO SUBSELECTION (*subfile*; *statement*)

Parameter	Type	Description
subfile	Subfile	Subfile to which to apply the formula
statement	Statement	One line of code or a global procedure

APPLY TO SUBSELECTION applies *statement* to each subrecord in the current subselection of *subfile*. The *statement* may be a statement or a global procedure. If *statement* modifies a subrecord, the modified subrecord is written to disk only when the parent record is written. If the subselection is empty, APPLY TO SUBSELECTION has no effect.

APPLY TO SUBSELECTION can be used to gather information from the subselection or to modify the subselection.

Since subrecords reside in memory, APPLY TO SUBSELECTION is faster than APPLY TO SELECTION.

The following example calculates the total sale price for each invoice line from the number of units, and the unit price.

```
ALL SUBRECORDS ([Invoice]Line)          ` Select all subrecords
APPLY TO SUBSELECTION ([Invoice]Line;
        [Invoice]Line'Total := [Invoice]Line'Price * [Invoice]Line'Units)
```

FIRST SUBRECORD

FIRST SUBRECORD (*subfile*)

Parameter	Type	Description
subfile	Subfile	Subfile in which to move to the first subrecord

FIRST SUBRECORD makes the first subrecord of the current subselection of *subfile* the current subrecord. All search, selection, and sort commands also set the current subrecord to the first subrecord. If the current subselection is empty, FIRST SUBRECORD has no effect.

The following example concatenates the first and last names of children stored in a subfile. It copies the names into an array, called Names.

```
` Create an array to hold the names
ARRAY TEXT (Names; Records in subselection ([People]Children))
FIRST SUBRECORD ([People]Children)                ` Start at the first subrecord
` Loop once for each child
For ($i; 1; Records in subselection ([People]Children))
  Names{$i} := [People]Children'First + " " + [People]Children'Last
  NEXT SUBRECORD ([People]Children)
End for
```

LAST SUBRECORD

LAST SUBRECORD (*subfile*)

Parameter	Type	Description
subfile	Subfile	Subfile in which to move to the last subrecord

LAST SUBRECORD makes the last subrecord of the current subselection of *subfile* the current subrecord. If the current subselection is empty, LAST SUBRECORD has no effect.

The following example concatenates the first and last names of children stored in a subfile. It copies the names into an array, called Names. It is the same as the example for FIRST SUBRECORD except that it moves through the subrecords from last to first.

```
` Create an array to hold the names
ARRAY TEXT (Names; Records in subselection ([People]Children))
LAST SUBRECORD ([People]Children)                 ` Start at the last subrecord
` Loop once for each child
For ($i; 1; Records in subselection ([People]Children))
  Names{$i} := [People]Children'First + " " + [People]Children'Last
  PREVIOUS SUBRECORD ([People]Children)
End for
```

NEXT SUBRECORD

NEXT SUBRECORD (*subfile*)

Parameter	Type	Description
subfile	Subfile	Subfile in which to move to the next subrecord

NEXT SUBRECORD moves the current subrecord pointer to the next subrecord in the current subselection of *subfile*. If NEXT SUBRECORD moves the pointer past the last subrecord, End subselection returns TRUE, and there is no current subrecord. If End subselection returns TRUE, use FIRST SUBRECORD or LAST SUBRECORD to move the pointer back into the current subselection. If the current subselection is empty, or Before subselection returns TRUE, NEXT SUBRECORD has no effect.

See the example for FIRST SUBRECORD, earlier in this section.

PREVIOUS SUBRECORD

PREVIOUS SUBRECORD (*subfile*)

Parameter	Type	Description
subfile	Subfile	Subfile in which to move to the previous subrecord

PREVIOUS SUBRECORD moves the current subrecord pointer to the previous subrecord in the current subselection of *subfile*. If PREVIOUS SUBRECORD moves the pointer before the first subrecord, Before subselection returns TRUE, and there is no current subrecord. If Before subselection returns TRUE, use FIRST SUBRECORD or LAST SUBRECORD to move the pointer back into the current subselection. If the current subselection is empty, or End subselection returns TRUE, PREVIOUS SUBRECORD has no effect.

See the example for LAST SUBRECORD, earlier in this section.

Before subselection

Before subselection (*subfile*) → Boolean

Parameter	Type	Description
subfile	Subfile	Subfile for which to test if pointer is before selection

Before subselection returns TRUE when the current subrecord pointer is before the first subrecord of *subfile*. Before subselection is used to check whether PREVIOUS SUBRECORD has moved the pointer before the first subrecord. If the current subselection is empty, Before subselection returns TRUE.

The following example is a script for a button. When the button is clicked, the pointer moves to the previous subrecord. If the pointer is before the first subrecord, it moves to the last subrecord.

```
PREVIOUS SUBRECORD ([People]Children)        ` Move to the previous subrecord
If (Before subselection ([People]Children)   ` If we have gone too far...
  LAST SUBRECORD ([People]Children)          ` move to the last subrecord
End if
```

End subselection

End subselection (*subfile*) → Boolean

Parameter	Type	Description
subfile	Subfile	Subfile for which to test if pointer is after selection

End subselection returns TRUE when the current subrecord pointer is after the end of the current subselection of *subfile*. End subselection is used to check whether NEXT SUBRECORD has moved the pointer after the last subrecord. If the current subselection is empty, End subselection returns TRUE.

The following example is a script for a button. When the button is clicked, the pointer moves to the next subrecord. If the pointer is after the last subrecord, it moves to the first subrecord.

```
NEXT SUBRECORD ([People]Children)            ` Move to the next subrecord
If (End subselection ([People]Children)      ` If we have gone too far...
  FIRST SUBRECORD ([People]Children)         ` move to the first subrecord
End if
```

CHAPTER 15

USER INTERFACE

USER INTERFACE

The commands in this chapter manage the user interface. They primarily affect what the user will see on the screen.

Layout Object Management

BUTTON TEXT	FONT	SET COLOR
ENABLE BUTTON	FONT SIZE	
DISABLE BUTTON	FONT STYLE	

The commands in this section affect the way that layout objects appear. Layout objects include fields, variables, buttons, check boxes, radio buttons, scrollable areas, pop-up menus, thermometers, rulers, and dials.

The changes that these commands make to a layout are effective only for the layout that is currently being displayed or printed. The layout reverts to its default display, when a new layout or a new record is displayed. These commands should be used in layout procedures or scripts.

The font, font size, and font style in an input layout can be changed for fields, variables, buttons, check boxes, radio buttons, scrollable areas, and pop-up menus. Fonts for fields and variables can be changed in both input and output layouts.

BUTTON TEXT

BUTTON TEXT (*button*; *button text*)

Parameter	Type	Description
button	Variable	Layout button variable
button text	String	Text to display in the button

BUTTON TEXT changes the text inside *button* to *button text*. BUTTON TEXT affects only buttons that display text: plain buttons, check boxes, and radio buttons. The new button text is used only for the currently displayed layout. The text must be set each time the layout is displayed. The button area must be large enough to accommodate the text; if it is not, the text is truncated.

See the ENABLE BUTTON example, next in this section.

ENABLE BUTTON
DISABLE BUTTON

ENABLE BUTTON (*button*)
DISABLE BUTTON (*button*)

Parameter	Type	Description
button	Variable	Layout button variable

These commands control whether a button is active or not. Buttons include plain buttons, invisible buttons, highlight buttons, check boxes, and radio buttons. Do not use these commands on buttons that are controlled by automatic actions except for the Delete Record action.

ENABLE BUTTON enables *button*, a button that was previously disabled with DISABLE BUTTON. Figure 15-1 shows enabled buttons: a check box, two radio buttons, and a plain button.

DISABLE BUTTON disables (dims) *button*. You use DISABLE BUTTON to prevent a button from being used. A button should be disabled when the action that it causes would be inappropriate. Figure 15-2 shows disabled buttons: a check box, two radio buttons, and a plain button.

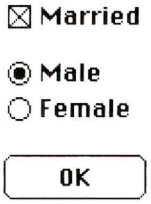


Figure 15-1
Enabled buttons

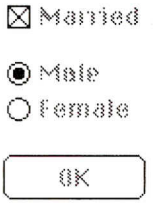


Figure 15-2
Disabled buttons

Buttons are by default enabled. A button can be disabled only with DISABLE BUTTON or by 4th DIMENSION when using automatic button actions. A button is disabled only while the layout is displayed; it must be disabled each time the layout is displayed.

The following example searches a file and enables or disables a button labeled Delete, depending on the results of the search.

```
DEFAULT FILE ([People])                          ` Set the default file
SEARCH BY INDEX ([People]Name = vName)           ` Search for people to delete
Case of
  : (Records in selection = 0)                   ` No people found
    BUTTON TEXT (bDelete; "Can't Delete")
    DISABLE BUTTON (bDelete)
  : (Records in selection = 1)                   ` One person found
    BUTTON TEXT (bDelete; "Delete Person")
    ENABLE BUTTON (bDelete)
  : (Records in selection > 1)                   ` Many people found
    BUTTON TEXT (bDelete; "Delete People")
    ENABLE BUTTON (bDelete)
End case
```

SET COLOR

SET COLOR (*object*; *color*)

Parameter	Type	Description
object	Field or variable	Object for which to set color
color	Number	Color of object

SET COLOR sets the foreground and background colors for *object*.

The *color* parameter specifies both the foreground and background colors. The *color* is calculated as follows:

```
Color := – (Foreground + (256 * Background))
```

The *color* is always a negative number. For example, if the foreground color is to be 20 and the background color is to be 10, then *color* is – (20 + (256 * 10)) or –2580.

Each color, foreground and background, is represented by a number between 0 and 255. The color that is displayed is dependent on the colors in the current color palette.

The following example sets the color for a button named My Button. The color is set to the values of the two variables named Foreground and Background.

```
SET COLOR (My Button; – (Foreground + (256 * Background)))      ` Set the My Button color
```

FONT

FONT (*object*; *font name*)

Parameter	Type	Description
object	Field or variable	Object for which to set font
font name	String	Name of the font

FONT changes the font in which *object* is displayed to *font name*. The *font name* is the Macintosh name of the font. The effect of FONT is the same as that of selecting an object in the Layout editor and choosing a font from the Font menu.

The following example sets the font for a button named My Button. The font is set to the Geneva font, a system font.

FONT (My Button; "Geneva") ` Set the My Button font

FONT SIZE

FONT SIZE (*object*; *size*)

Parameter	Type	Description
object	Field or variable	Object for which to set font size
size	Number	Size of the font

FONT SIZE sets the font size for *object*. The *size* is any integer between 1 and 127. The effect of FONT SIZE is the same as that of selecting an object in the Layout editor and choosing a font size from the Font menu. If the exact font size doesn't exist, characters are scaled. If the size is 0, the font size reverts to the size originally defined in the layout.

The area for the object, as defined in the layout, must be large enough to display the data in the new size; otherwise, the text may be truncated or not displayed at all.

The following example sets the font size for a button named My Button. The font size is set to 14.

FONT SIZE (My Button; 14) Set the My Button font size

FONT STYLE

FONT STYLE (*object*; *style number*)

Parameter	Type	Description
object	Field or variable	Object for which to change font style
style number	Number	Style of the font

FONT STYLE sets the font style for *object*. The effect of FONT STYLE is the same as that of selecting an object in the Layout editor and choosing a font style from the Font menu. The *style number* is a Macintosh font style code. By adding codes together, you can created combined styles.

The numeric codes for FONT STYLE are presented in Table 15-1.

Table 15-1
Font styles

Style	Number	Style	Number
Plain	0	Outline	8
Bold	1	Shadow	16
Italic	2	Condensed	32
Underline	4	Extended	64

The following example sets the font style for a button named My Button. The font style is set to bold italic.

```
FONT STYLE (My Button; 3)          ` Set the My Button font style
```

Displaying Messages to the User

ALERT	DIALOG	ERASE WINDOW
CONFIRM	MESSAGE	MESSAGES ON
Request	GOTO XY	MESSAGES OFF

The commands in this section let you display messages to the user. Messages include standard Macintosh dialog boxes, such as alerts, and custom messages, such as the message window and progress thermometers.

There are three standard Macintosh dialog boxes: alerts, confirmation dialog boxes, and requests. All three types should be used only when the user must be informed of something important. These dialog boxes are modal, meaning that the user must dismiss the dialog box by clicking a button or pressing Enter before he or she can continue.

ALERT should be used simply to inform the user. CONFIRM should be used to inform the user and obtain confirmation before performing an action. Request should be used when text information is also required from the user.

ALERT

ALERT (*message*)

Parameter	Type	Description
message	String	Message to display in the alert

ALERT displays a "Note" type alert box. The alert displays *message* and contains an OK button. The alert box can display up to 255 characters, depending on the widths of characters.

Alerts are used to provide information (such as an error message) to the user without requiring any information to be returned. They are also useful during development, for displaying status information (such as variable values) to the designer.

The following example displays an alert showing information about a company. Notice that the string that is displayed contains carriage returns, which cause the string to wrap to the next line.

```
CR := Char (13)
ALERT ("Company: " + [Companies]Name + CR +
        "People in company: " + String (Records in selection ([People])) + CR +
        "Number of parts they supply: " + String (Records in selection ([Parts])))
```

Figure 15-3 shows the alert box that is displayed.

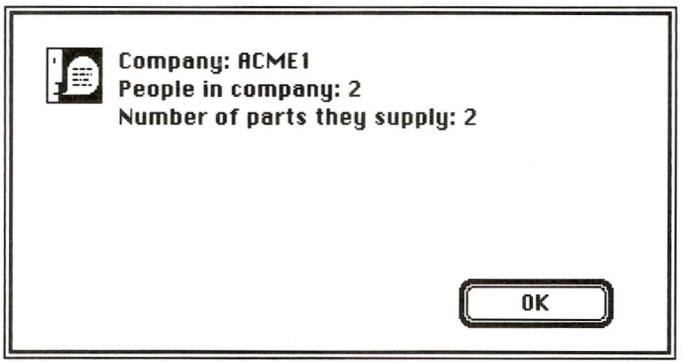


Figure 15-3
Alert box

CONFIRM

CONFIRM (*message*)

Parameter	Type	Description
message	String	Message to display in the confirmation dialog box

CONFIRM displays a "Caution" type dialog box with *message* and two buttons: OK and Cancel. The OK button is the default button. The user can click the OK button or press Enter to accept the dialog box, setting the OK system variable to 1. The user can click the Cancel button to cancel the dialog box, setting the OK system variable to 0. The dialog box can display up to 80 characters.

The following example displays a confirmation dialog box asking the user to confirm an operation. The If test uses alerts to show how the OK variable is set.

```
CONFIRM ("Complete the operation?")
If (OK = 1)
  ALERT ("The user pressed the OK button.")
Else
  ALERT ("The user pressed the Cancel button.")
End if
```

Figure 15-4 shows the confirmation dialog box that is displayed.

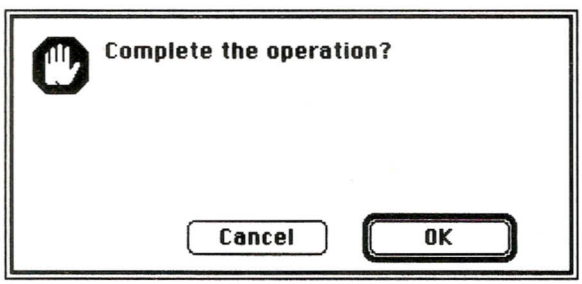


Figure 15-4
Confirmation dialog box

Request

Request (*message*; *{default response}*) → String

Parameter	Type	Description
message	String	Message to display in the request dialog box
default response	String	Default data entered in the text area

Request displays a dialog box with a prompt, *message;* a text input area with an optional default value specified by *default response;* and two buttons (OK and Cancel). The user can click the OK button or press Enter to accept the dialog box, setting the OK system variable to 1. The user can click the Cancel button to cancel the dialog box, setting the OK system variable to 0.

The user can enter text into the text input area. If the user clicks OK, Request returns the text. If the user clicks cancel, Request returns an empty string ("").

If the response should be a numeric or a date value, convert the string returned by Request to the proper type with the Num or Date function.

A request dialog box can display about 30 characters, depending on the width of the characters. Any message that is too long is truncated.

If you need to get several pieces of information from the user, design a layout and present it with DIALOG, rather than presenting a succession of Request dialog boxes.

The following example displays the request dialog box shown in Figure 15-5. The information that the user enters is stored in the vReturn variable. The example then displays one of two different alert boxes, depending on which button the user clicked.

```
vReturn := Request ("Enter the information:"; "Default")
If (OK = 1)
  ALERT ("You entered " + vReturn + " and you pressed the OK button.")
Else
  ALERT ("You pressed the Cancel button.")
End if
```

Figure 15-5 shows the request dialog box that is displayed.

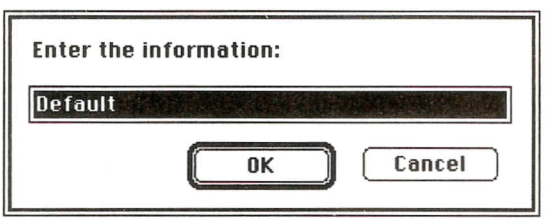


Figure 15-5
Request dialog box

DIALOG

DIALOG ({*file*}; *layout*)

Parameter	Type	Description
file	File	File containing the layout
layout	String	Layout to display as dialog

DIALOG presents *layout* to the user. This command is often used to get information from the user through variables, or to present information to the user.

Any fields that are displayed with DIALOG are the fields from the current record of *file* and are nonenterable. DIALOG does not automatically save a record as does ADD RECORD or MODIFY RECORD. If any fields must be saved, use SAVE RECORD to save the record.

It is normal to display the layout inside a type 1 window (a modal window), created with the OPEN WINDOW command. (See the section "Managing Windows," later in this chapter, for more information on window types.)

DIALOG is used instead of ALERT, CONFIRM, or Request when the information that must be presented or gathered is more complex than those commands can manage.

Unlike ADD RECORD or MODIFY RECORD, DIALOG does not use the current input layout, since the command specifies the layout. Also, the default button panel is not used if buttons are omitted. Instead, two buttons, OK and Cancel, are automatically created. Adding any custom buttons removes the OK and Cancel buttons.

Clicking an Accept button or pressing the "accept" key (usually the Enter key) sets the OK system variable to 1. Clicking a Cancel button or pressing the "cancel" key combination (usually Command-.) sets the OK system variable to 0. The OK system variable is not set until the dialog is closed.

If a layout procedure exists, the Before and During phases are executed. If any scripts exist, they are executed when appropriate.

The following example shows the use of DIALOG to specify search criteria. Note that this example duplicates the functionality of the SEARCH BY LAYOUT command. A custom layout is displayed so that the user can enter the search criteria. The buttons that are displayed are the default buttons.

```
DEFAULT FILE ([Company])                    ` Set the default file
OPEN WINDOW (10; 40; 370; 220; 1)           ` Open a modal window
DIALOG ("Search Layout")                    ` Display the search dialog
CLOSE WINDOW                                ` Always close the window
If (OK = 1)                                 ` If the user accepted the dialog...
  SEARCH ([Company]Name = vName; *)         ` search for the company in...
  SEARCH (& [Company]State = vState)        ` the specified state
End if
```

Figure 15-6 shows the resulting custom dialog box.

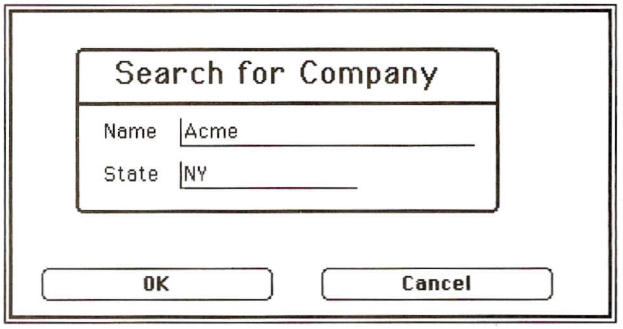


Figure 15-6
Custom search dialog box

MESSAGE

MESSAGE (*message*)

Parameter	Type	Description
message	String	Message to display

MESSAGE displays *message* on the screen in a special message window. The message is temporary and is erased as soon as a layout is displayed or the procedure stops executing. If another MESSAGE is executed, the old message is erased.

MESSAGE is usually used to inform the user of some activity.

If a window is opened with OPEN WINDOW, the open window behaves like a terminal. The message text is displayed in 9-point Monaco. The Monaco font is monospaced (uses fixed-width characters) and can therefore be used to accurately position messages in the window.

Successive messages do not erase previous messages when displayed in a window opened with OPEN WINDOW. Instead, they are concatenated onto existing messages. If a message is wider than the window, 4th DIMENSION automatically performs text wrap. If you want to control line breaks, concatenate carriage returns into your message by using Char (13). If the message has more lines than the window, 4th DIMENSION automatically scrolls the message.

You can use ERASE WINDOW and GOTO XY to position messages in an open window. A character from a new message overwrites and erases a character already displayed in the same position. The window does not display a cursor.

The following example displays the default message window.

```
MESSAGE ("The current status is OK")
```

Figure 15-7 shows the resulting message window.

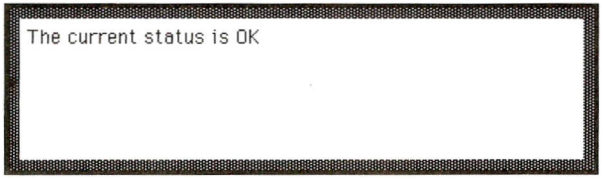


Figure 15-7
Default message window

The following example shows how messages appear in an open window. The example also shows how an open window can be used as a dumb terminal, displaying characters that are typed at the keyboard or even received through the serial port.

The procedure begins by opening the window. The first message appears at the upper-left corner of the window. Then the message position is changed, using the GOTO XY command, and a message is printed. The next line is too long and wraps around to the subsequent line. Note that the text is not broken between words. The final message has a carriage return in it, demonstrating how a carriage return moves the cursor to the beginning of the next line.

Finally, an event procedure is installed. It is a simple one-line procedure that echoes anything that is typed on the keyboard. (For more information on event procedures, see "ON EVENT CALL," in the section "Controlling the Execution of Procedures," in Chapter 18.) After the event procedure is installed, a While loop is entered. The While loop continues until the user presses the Q key. If the user presses the E key, the If test within the While loop erases the window.

The last line in the window was typed from the keyboard and written to the window by the event procedure.

If text reaches the bottom of the window, the window automatically scrolls up, so that any text that is on the first line is lost.

```
` Open a custom window
OPEN WINDOW (10; 45; 500; 330; 0; "My Window")
` Display the first message
MESSAGE ("This is at position 0,0 in the window.")
` Position the cursor
GOTO XY (30; 5)
MESSAGE ("This is at position 30,5 in the window.")
```

```
GOTO XY (50; 10)
MESSAGE ("This message is too long and wraps on to the next line.")
GOTO XY (0; 15)
MESSAGE ("This message has a" + Char (13) + "carriage return in it (Char (13))")
GOTO XY (0; 20)
KeyCode := 0                                  ` Preset the KeyCode system variable
ON EVENT CALL ("Key Proc")
While (Char (KeyCode) # "q")                  ` Loop until "q" is pressed
  If (KeyCode = Ascii ("E"))                  ` If the user pressed "E"...
    ERASE WINDOW                              ` Erase the window
    KeyCode := 0                              ` Reset the KeyCode system variable
  End if
End while
ON EVENT CALL ("")                            ` Remove the event procedure
CLOSE WINDOW                                  ` Close the custom window
```

The following one-line procedure is *Key Proc*, the event procedure installed by ON EVENT CALL in the procedure just given. The procedure simply echoes to the screen whatever is typed at the keyboard.

```
MESSAGE (Char (KeyCode))
```

Figure 15-8 shows the result of the example.

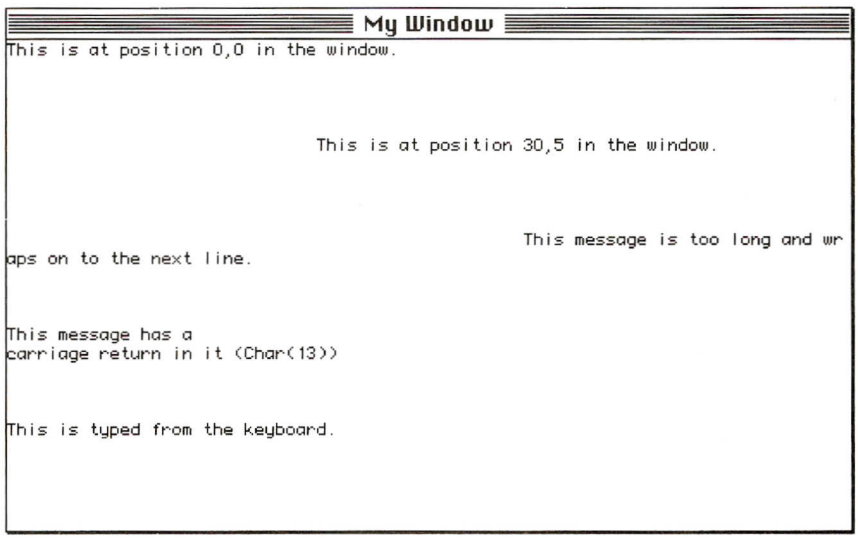


Figure 15-8
Window showing messages

GOTO XY

GOTO XY (*x*; *y*)

Parameter	Type	Description
x	Number	x (horizontal) position of cursor
y	Number	y (vertical) position of cursor

GOTO XY positions the cursor (an invisible cursor) in a window opened by OPEN WINDOW. GOTO XY works only when a layout is not being displayed.

The upper-left corner is position 0,0. The cursor is automatically placed at 0,0 when a window is opened, and after ERASE WINDOW is executed.

After GOTO XY positions the cursor, MESSAGE can be used to print characters in the window. You can use OPEN WINDOW, GOTO XY, and MESSAGE to emulate a character-based terminal. OPEN WINDOW opens the window on screen that displays the text; GOTO XY positions the cursor so that the text is written in the correct position; and MESSAGE writes the data at the cursor position.

GOTO XY positions the cursor properly because the Monaco font is used in the custom window. The Monaco font is monospaced, meaning that all of its characters are the same width.

See the second MESSAGE example, earlier in this section.

ERASE WINDOW

ERASE WINDOW

ERASE WINDOW clears the contents of the window created by OPEN WINDOW and moves the cursor to the upper-left corner of the window, the GOTO XY (0; 0) position. Don't confuse ERASE WINDOW, which clears the contents of a window, with CLOSE WINDOW, which removes the window from the screen.

See the second MESSAGE example, earlier in this section.

MESSAGES ON
MESSAGES OFF

MESSAGES ON
MESSAGES OFF

MESSAGES ON and MESSAGES OFF turn on and off the progress thermometers that 4th DIMENSION displays while executing time-consuming processes. By default, messages are on. Table 15-2 shows User environment menu items that display the progress thermometer.

Table 15-2
User environment menu items that display the progress thermometer

Menu Items	Menu Items	Menu Items
Apply Formula	Report	Sort Selection
Export Data	Search by Layout	Sort File
Graph	Search by Formula	
Import Data	Search Editor	

Table 15-3 shows commands that display the progress thermometer.

Table 15-3
Commands that display the progress thermometer

Commands	Commands	Commands
APPLY TO SELECTION	IMPORT DIF	SEARCH
EXPORT DIF	IMPORT SYLK	SORT BY FORMULA
EXPORT SYLK	IMPORT TEXT	SORT FILE
EXPORT TEXT	SEARCH BY LAYOUT	SORT SELECTION
GRAPH FILE	SEARCH BY FORMULA	
REPORT	SEARCH SELECTION	

The following example turns off the progress thermometer before doing a sort, and then turns it back on after completing the sort.

MESSAGES OFF
SORT SELECTION ([Addresses]; [Addresses]ZIP; >; [Addresses]Name2; >)
MESSAGES ON

Managing Windows

OPEN WINDOW	Screen height	SET WINDOW TITLE
CLOSE WINDOW	Screen width	

The commands in this section let you manage windows. Managing windows includes opening and closing custom windows, determining the screen size, and changing a window's title.

About Windows

Windows are used to display information to the user. There are three main uses: to enter data, to display data, and to inform the user.

There is always at least one window open. This window is a standard window with a title bar and a size box. Scroll bars are added when needed, to let the user scroll to hidden areas.

In the User environment, this window displays either the record list (output layout) or the data entry screen (input layout). In the Runtime environment, this window displays a splash screen (a custom graphic). The splash screen is immediately erased and replaced with data by commands that display layouts. When the commands finish executing, the splash screen is again displayed.

Custom windows can be opened with the OPEN WINDOW command. The custom windows can be any Macintosh style of window. Any data that is displayed will be displayed in these new windows. The custom windows will remain open only until control returns to the splash screen menu bar. Custom windows should be closed with the CLOSE WINDOW command when no longer needed.

Some commands open their own windows. Commands such as GRAPH FILE, REPORT, and PRINT LABEL open a window that becomes the frontmost window.

The Different Window Types

There are five basic types of windows. There are a number of variations on these types, including zoom boxes, scroll bars, and size boxes. It is the designer's responsibility to ensure that the window is appropriate for the type of information that is displayed in it.

Figures 15-9 through 15-20 below show each of the five window types, both without scroll bars and with scroll bars as appropriate.

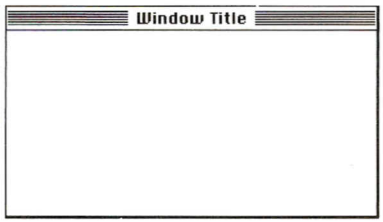


Figure 15-9
Type 0 window

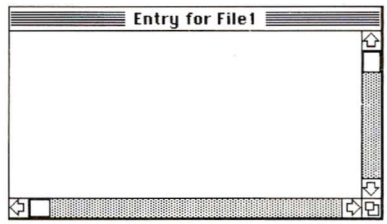


Figure 15-10
Type 0 window with scroll bars

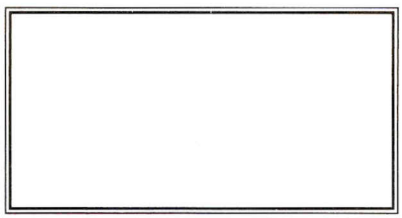


Figure 15-11
Type 1 window

Figure 15-12
Type 2 window

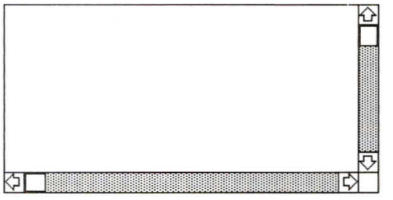


Figure 15-13
Type 2 window with scroll bars

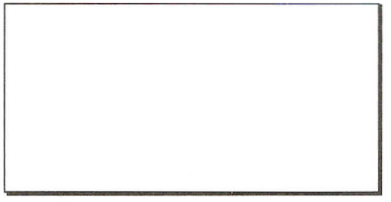


Figure 15-14
Type 3 window

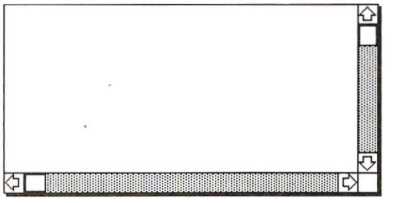


Figure 15-15
Type 3 window with scroll bars

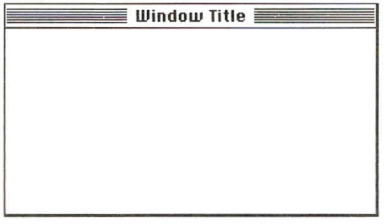


Figure 15-16
Type 4 window

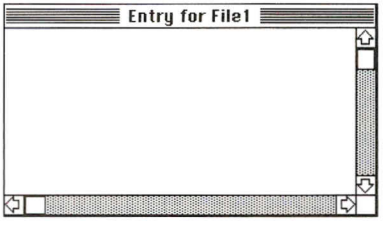


Figure 15-17
Type 4 window with scroll bars

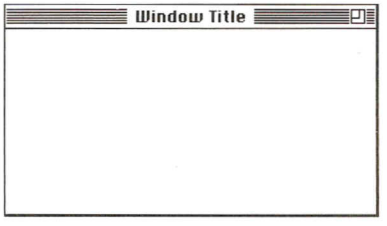


Figure 15-18
Type 8 window

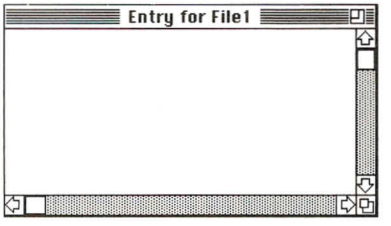


Figure 15-19
Type 8 window with scroll bars

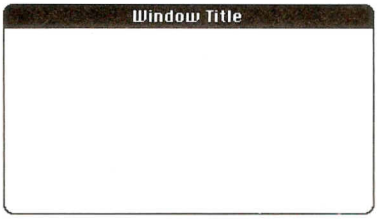


Figure 15-20
Type 16 window

The Modal Window

Window type 1 is a modal window. A modal window does not allow the the user to choose commands from the menus. A modal window should be used only when it is required that the user immediately finish an action before proceeding.

Positioning Windows and Window Borders

Every Macintosh screen has a menu bar at the top. Normally, the menu bar is 20 pixels high. You must take the menu bar into account when positioning a window.

Every window has a border around it. You must also take the border into account when specifying the size of a window.

Figure 15-21 shows a type 4 window. The measurements of the window are shown.

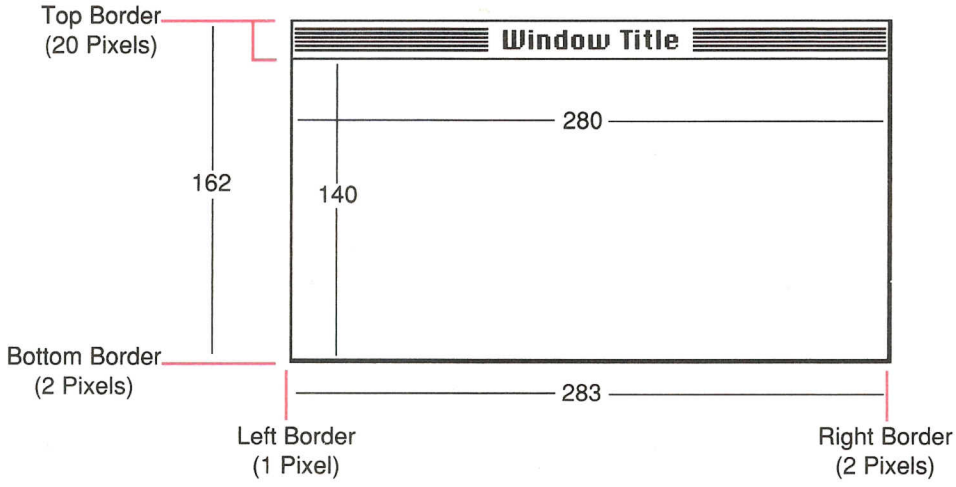


Figure 15-21
Measurements of a window

The inside area of the window, 280 pixels by 140 pixels, is the area specified for the window size. Notice that the total area of the window is larger than the area specified. This is true of all window types.

Table 15-4 lists the sizes of the window borders for all window types.

Table 15-4
Window border sizes

Type	Top	Left	Bottom	Right
0	20	1	2	2
1	8	8	8	8
2	1	1	1	1
3	1	1	3	3
4	20	1	2	2
8	20	1	2	2
16	20	1	2	2

Table 15-5 lists the dimensions of a full-size window of each type, on a 9-inch Macintosh screen (for example, the Macintosh SE screen).

Table 15-5
Window sizes to open on a 9-inch screen

Type	Top	Left	Bottom	Right
0	40	2	340	510
1	29	9	333	503
2	22	2	340	510
3	22	2	338	508
4	40	2	340	510
8	40	2	340	510
16	40	2	340	510

Scroll Bars, the Size Box, and the Zoom Box

Scroll bars in a window allow the user to scroll to parts of the window that are not displayed. Scroll bars use 16 pixels of the usable area at the right side and bottom of a window.

Scroll bars are controlled by the command that is displaying in the window, not by the window type. The two commands that display record lists—DISPLAY SELECTION and MODIFY SELECTION—both add scroll bars to the window. The commands that allow data entry—ADD RECORD, MODIFY RECORD, ADD SUBRECORD, and MODIFY SUBRECORD—all let you control whether scroll bars are displayed, by using an optional parameter, the asterisk (*). The DIALOG command never displays scroll bars.

You should never put anything that displays scroll bars in a type 1 or type 16 window.

Windows can be resized by means of a size box in the lower-right corner of the window. (See Figure 15-22.) Only a type 0 or a type 8 window can be resized. The size box is displayed only if scroll bars are also displayed. Dialog boxes never display scroll bars or a size box.

Figure 15-22
A size box

Even when the size box is not displayed, a type 0 or type 8 window can still be resized.

A zoom box is a small box at the right side of a window's title bar. (See Figure 15-23.) Clicking the zoom box zooms the window to full-screen size. Clicking the zoom box again zooms the window back to its previous size. A zoom box appears only in a window of type 8.

Figure 15-23
A zoom box

Setting Window Titles

Window types 0, 4, 8, and 16 allow a title to be displayed at the top of the window. The window title can be changed with the SET WINDOW TITLE command. Windows that do not allow a window title simply ignore the SET WINDOW TITLE command.

Commands that display information in the window set the window title to something appropriate for the command. For example, ADD RECORD changes the title to "Entry for *File*," where *File* is the name of the file to which a record is being added. You can change the default window title, by executing SET WINDOW TITLE in the layout procedure.

External window (left; top; right; bottom; type; area) > Longint

OPEN WINDOW

OPEN WINDOW (*left*; *top*; *right*; *bottom*; *{type}*; *{window title}*)

Parameter	Type	Description
left	Number	Pixels from left side of screen to left edge
top	Number	Pixels from top of screen to top edge
right	Number	Pixels from left side of screen to right edge
bottom	Number	Pixels from top of screen to bottom edge
type	Number	Window type
window title	String	Title of window

OPEN WINDOW opens a new window with the dimensions given by the first four parameters:

- *left* is the distance in pixels from the left edge of the screen to the left internal edge of the window.
- *top* is the distance in pixels from the top of the screen to the top internal edge of the window. The top of the menu bar is the top pixel.
- *right* is the distance in pixels from the left edge of the screen to the right internal edge of the window.
- *bottom* is the distance in pixels from the top of the screen to the bottom internal edge of the window.

The *type* parameter is optional. It represents the type of window you want to display, and corresponds to the seven windows shown in Figures 15-9 through 15-20 earlier in this section.

The *title* parameter is the optional title for the window.

If the last two parameters are omitted, OPEN WINDOW draws a type 1 window (a modal window).

If more than one window is open, the last window opened is the active (frontmost) window. Only information within the active window can be modified. Any other windows can be viewed. When the user types, the active window will always come to the front.

Layouts are displayed inside an open window. Text from the MESSAGE command also appears in the window.

To make your windows independent of display size, you can use Screen height and Screen width to calculate the upper-left and lower-right corners of the window. See the example for an illustration of this technique.

The following example demonstrates the use of OPEN WINDOW. The example prompts the user for the window height and width. It then uses that information to create a centered window. The first code segment is a global procedure that calls the second procedure. Here is the first procedure.

```
Repeat
  $Height := Num (Request ("Height (click cancel if done):"))
  If (OK = 1)
    $Width := Num (Request ("Width:"))
    $Type := Num (Request ("Window type:"))
    If (OK = 0)                                  ` If the user did not enter a window type...
      Center Window ($Height; $Width)            ` center the window with two parameters
    Else
      $Title := Request ("Window title:"))
      If (OK = 0)                                ` If the user did not enter a window title...
        Center Window ($Height; $Width; $Type)   ` center the window with three parameters
      Else                                       ` Otherwise, use all the parameters
        Center Window ($Height; $Width; $Type; $Title)
      End if
    End if
    $Stop := Current time + 10                   ` Pause for 10 seconds
    While (Current time < $Stop)
    End while
    CLOSE WINDOW                                 ` Close the window
  End if
Until ($Height = 0)                              ` Loop until user cancels the height request
```

The following code is the procedure, *Center Window,* that opens a centered window. It is called by the first procedure. Notice that it can accept two, three, or four parameters.

```
` Global procedure: Center Window
` $1 – Window width
` $2 – Window height
` $3 – Window type (optional)
` $4 – Window title (optional)
$sw := Screen width / 2                          ` Find center of screen (width)
$sh := Screen height / 2 + 10                    ` Find center of screen – menu bar (height)
$ww := $1 / 2                                    ` Half of requested window width
$wh := $2 / 2                                    ` Half of requested window height
Case of
  : (Count parameters = 2)
    OPEN WINDOW ($sw – $ww; $sh – $wh; $sw + $ww; $sh + $wh)
  : (Count parameters = 3)
    OPEN WINDOW ($sw – $ww; $sh – $wh; $sw + $ww; $sh + $wh; $3)
  : (Count parameters = 4)
    OPEN WINDOW ($sw – $ww; $sh – $wh; $sw + $ww; $sh + $wh; $3; $4)
End case
```

CLOSE WINDOW

CLOSE WINDOW

CLOSE WINDOW closes the window opened by an OPEN WINDOW command. CLOSE WINDOW has no effect if a custom window isn't open; it does not close the standard window. CLOSE WINDOW also has no effect if called while a layout is active in the window. You must call CLOSE WINDOW when you are done using a window opened by OPEN WINDOW.

The following example opens a window and adds new records with the ADD RECORD command. When the records have been added, the window is closed with CLOSE WINDOW.

```
OPEN WINDOW (5; 40; 250; 300; 0; "New Employee")
Repeat                                  ` Loop until the user cancels
  ADD RECORD ([Employees])              ` Add a new employee record
Until (OK = 0)
CLOSE WINDOW                            ` Close the custom window
```

Screen height
Screen width

Screen height → Number
Screen width → Number

Screen height and Screen width return the height and width of the screen, in pixels. These commands can be used to determine the type of Macintosh screen in use. They will work with any type of screen.

If there are multiple display devices attached to the Macintosh, these commands return the size of the screen where the menu bar is displayed.

Table 15-6 lists the standard Macintosh screen sizes in pixels.

Table 15-6
Macintosh screen sizes

Macintosh	Height	Width
Macintosh Plus	342	512
Macintosh SE	342	512
Macintosh II (standard monitors)	480	640

See the OPEN WINDOW example, earlier in this section.

SET WINDOW TITLE

SET WINDOW TITLE (*title*)

Parameter	Type	Description
title	String	Window title

SET WINDOW TITLE changes the title of the current window to that specified by *title*. The current window may be the 4th DIMENSION standard window or the custom window, opened with OPEN WINDOW. The custom title remains in the window until you change it with another SET WINDOW TITLE command. If you create a custom window with a title bar, you can specify the title in the OPEN WINDOW command. You can also change it with SET WINDOW TITLE.

Titles in the User environment can be set with SET WINDOW TITLE, but remember that 4th DIMENSION automatically changes the title of its window when you choose from User environment menus. For example, when you select a menu item like New Record in the Entry menu, 4th DIMENSION sets the title to "Entry for *File*."

The following example sets the window title to whatever the user enters into the Request box.

```
` Set the title to the Request
SET WINDOW TITLE (Request ("Window title:"; "Custom Title"))
```

Managing Menus

MENU BAR	DISABLE ITEM	Menu selected
CHECK ITEM	ENABLE ITEM	

The commands in this section allow you to switch to different menu bars, check menu items, and enable and disable menu items.

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 15-24 shows these components.

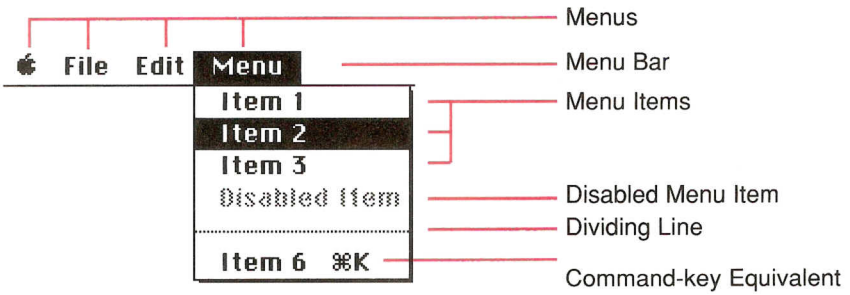


Figure 15-24
Menu components

You create menu bars in the Design environment's Menu editor. Menu bars are identified by number, rather than by name. The first menu bar is Menu Bar #1. It is also the default menu bar. If you wish to open an application with a menu bar other than Menu Bar #1, you must force it with the MENU BAR command in a startup procedure.

Each menu item can have one global procedure attached to it. This procedure is called a *master procedure*. You associate a procedure with a menu item by typing the name of a global procedure in the Procedures column of the Menu editor window. The user executes the procedure by choosing the menu item to which the procedure belongs. If you don't assign a procedure to a menu item, choosing that menu item causes 4th DIMENSION to quit the menu system. If the user is using the 4th DIMENSION Runtime version, this means quitting to the Finder.

Using the Menu editor, you can create a provisional menu system before you write the global procedures that will activate the menu items. There is no requirement that procedures exist when you work in the Menu editor. However, your menu items will not carry out their intended purposes until you associate the appropriate procedures with each menu item.

Every 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 standard editing commands. The Apple and Edit menus cannot be modified. The file menu has only one menu item—Quit. Notice that Quit has no procedure associated with it. That's how it causes 4th DIMENSION to quit the Runtime environment. You can rename the File menu, add menu items to it, or keep it as is. If it is renamed, it will no longer appear to the left of the Edit menu. It is recommended that you always keep Quit as the last item in the File menu.

Like menu bars, menus are numbered. Because they cannot be modified, the Apple and Edit menus are not included in the count. Instead, File is menu 1. Thereafter, menus are numbered sequentially from left to right (2, 3, 4, and so on). Menu numbering is important when you are working with the Menu selected function.

The items within each menu are numbered sequentially from the top of the menu to the bottom. The topmost item is item 1.

Custom Menus

Using 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. You can use it to create menus and Command-key combinations with which the user can choose menu items without using the mouse. You can password-protect menu items, associate menu bars with layouts, and enable, disable, and check items by means of procedures.

There are two types of menu bars: splash screen menu bars and layout menu bars.

The splash screen menu bar is used when a splash screen is displayed in the Runtime environment. Choosing an item from one of the menus in this menu bar executes the procedure that is attached to the menu item.

A layout menu bar is displayed when a layout is displayed. You associate a menu bar with a layout by using the "Menu Bar" menu item from the Layout menu in the Layout editor. The menus on a layout menu bar are appended to the current menu bar when the layout is displayed. The menus are appended for input layouts in both the User and Runtime environments and also for output layouts in the Runtime environment. Menu items in a layout menu bar always execute any procedures that are attached to them.

Layout menu bars are specified by a menu bar number. If the number of the displayed splash screen menu bar is the same as the number of the appended layout menu bar, the layout menu bar is not appended.

If you specify a negative number for a layout menu bar, 4th DIMENSION uses the absolute value of the menu bar. For example, if you specify –3 as the menu bar, Menu Bar 3 is used. When a layout menu bar has been specified with a negative number, the menu items for all the menus in the menu bar (splash screen and layout) will execute the procedures that are attached to them. This is the recommended method of associating a menu bar that will be used in the Runtime environment.

If you do not specify a negative number for a layout menu bar, choosing a menu item from a splash screen menu *will not* execute its procedure; instead, the layout procedure will be executed and you can use Menu selected to test for the selected menu. This is more difficult than using a negative number for the layout menu bar, and should generally not be used in the Runtime environment.

MENU BAR

MENU BAR (*menu bar number*)

Parameter	Type	Description
menu bar number	Number	Number of the menu bar

MENU BAR replaces the current menu bar with the menu bar specified by *menu bar number*. All menu items revert to the way they were defined in the Menu editor (either enabled or disabled). All menu items are displayed without check marks.

It is common to define multiple menus that are identical except that different menu items are enabled or disabled. MENU BAR is then used to switch between the menus, to enable and disable the menu items. Using this trick is often simpler than using ENABLE ITEM and DISABLE ITEM.

When a user enters the Runtime environment, the first menu bar is displayed (Menu Bar #1). You can change this menu bar, when opening a database, in the global startup procedure (*Startup*), or in the startup procedure for an individual password.

The following example changes the current menu bar to Menu Bar #3.

MENU BAR (3)

CHECK ITEM

CHECK ITEM (*menu*; *menu item*; *mark*)

Parameter	Type	Description
menu	Number	Menu number
menu item	Number	Menu item number
mark	String	Mark character

CHECK ITEM is used to clear or put a check mark next to the menu item specified by *menu* and *menu item*. By default, all menu items are unchecked. The *mark* parameter should be either a check mark (ASCII 18) or a space. A space erases the mark on the specified menu item. All marks are erased when a MENU BAR command is executed. MENU BAR can be used to return all menu items to their default, unchecked state.

The Edit and Apple menus are built in and are not a part of the menu count. The File menu is generally menu number 1.

The following example checks or unchecks a menu item. The example checks the menu item if the variable Check It is true; otherwise, it unchecks the menu item.

```
If (Check It)                          ` If Check It is true...
  CHECK ITEM (2; 1; Char (18))         ` Check the menu item
Else
  CHECK ITEM (2; 1; " ")               ` Uncheck the menu item
End if
```

DISABLE ITEM
ENABLE ITEM

DISABLE ITEM (*menu*; *menu item*)

ENABLE ITEM (*menu*; *menu item*)

Parameter	Type	Description
menu	Number	Menu number
menu item	Number	Menu item number

DISABLE ITEM disables (dims) the menu item specified by *menu* and *menu item*. If *menu item* is 0, then the entire menu is disabled.

ENABLE ITEM enables the menu item specified by *menu* and *menu item*. If *menu item* is 0, menu items are returned to the state defined in the Menu editor.

A menu item is enabled or disabled only until the menu bar is updated with a MENU BAR command. MENU BAR can be used to return all items to their default state.

As a general rule, if you find yourself disabling a particular item a lot, set it as disabled in the Menu editor. If you need to enable or disable several menu items at the same time, you may find that it is more efficient to switch menu bars instead.

The Edit and Apple menus are built in and are not a part of the menu count. The File menu is menu number 1, and the first menu to the right of the Edit menu is menu 2.

The following example assumes that the fourth menu item in the second menu (not counting the Edit menu) is to delete records. The menu item should be enabled only when there are records to delete. The example enables or disables the menu item appropriately.

```
If (Records in selection ([People]) # 0)   ` If people were found...
  ENABLE ITEM (2; 4)                       ` Enable the delete menu item
Else                                       ` Otherwise, if no people were found...
  DISABLE ITEM (2; 4)                      ` Disable the delete menu item
End if
```

Menu selected

Menu selected → Number

Menu selected is used only when input layouts or output layouts are displayed. It detects which menu item has been chosen from a menu.

Whenever possible, it is recommended that you use procedures associated with menu items in an associated menu bar instead of using Menu selected. Associated menu bars are easier to manage, since it is not necessary to test for their selection.

Menu selected returns the Macintosh menu-selected number, a long integer. Menu selected returns 0 if no menu item was selected.

To find the menu number, divide Menu selected by 65,536 and convert the result to an integer. To find the menu item number, calculate the modulo of Menu selected with the modulus 65,536. Use the following formulas to calculate the menu number and menu item number:

Menu := **Menu selected** \ 65536
Menu Item := **Menu selected** % 65536

The Edit and Apple menus are built in and aren't part of the menu count. The File menu is menu number 1, and the first menu to the right of the Edit menu is menu 2.

The following example uses Menu selected to supply the menu and item arguments to CHECK ITEM.

```
Case of
  : (During)
⋮
  : (Menu selected # 0)
    CHECK ITEM (Menu selected \ 65536 ; Menu selected % 65536; Char (18))
End case
```

Playing Sound

BEEP PLAY

The commands in this section make sounds through the Macintosh speaker.

BEEP

BEEP

BEEP causes the Macintosh to generate a beep. The Macintosh may emit a sound other than a beep, depending on how the user has set the Control Panel for sound.

The following example causes a beep (or other sound).

BEEP

PLAY

PLAY (*sound name*; {*channel*})

Parameter	Type	Description
sound name	String	Sound name
channel	Number	Synthesizer channel

PLAY plays the sound resource named by *sound name*.

The *channel* parameter specifies the Macintosh synthesizer channel. If *channel* is not specified, the channel is for simple digitized sounds and is synchronous. Synchronous means that all processing stops until the sound has finished. If *channel* is 1, the channel is for simple digitized sounds and is asynchronous. Asynchronous means that processing does not stop and the sound plays in the background. Table 15-7 lists the possible values for *channel*.

Table 15-7
Values for the *channel* parameter

Channel Type	*Channel*
Note Synthesizer	1
Wave Table Synthesizer	3
Sampled Sound Synthesizer	5
MIDI Synthesizer In	7
MIDI Synthesizer Out	9

The following example is in a startup procedure. It welcomes the user with a sound called Welcome Sound.

PLAY ("Welcome Sound") ` Play the Welcome Sound

CHAPTER 16

ADVANCED COMMANDS

ADVANCED COMMANDS

This chapter defines commands for advanced database design. Advanced design includes using record numbers and managing sets, multi-user databases, transactions, documents, serial communication, and passwords.

Using Numbers Associated With Records

Record number	Selected record number	Sequence number
GOTO RECORD	GOTO SELECTED RECORD	

The commands in this section allow you to manage records by referencing them directly with numbers. These numbers are associated with each record in a file or selection.

There are three numbers that are associated with a record:

- the record number
- the selected record number
- the sequence number

The record number is the absolute record number for a record. The record number is automatically assigned to each new record and remains constant for the record until the record is deleted or the file is permanently sorted. Record numbers are reused if records are deleted.

The selected record number is the position of the record in the current selection. The selected record number is completely dependent on the current selection. If the selection is changed or sorted, the selected record number may change.

The sequence number is a unique nonrepeating number that may be assigned to a record. The sequence number is not automatically stored with each record. It starts at 1 and is incremented for each new record that is saved.

Record Number Examples

The tables that follow illustrate the numbers that are associated with records. Here is a description of each table:

- Each line represents information about a record.
- The order of the lines is the order in which the records would be displayed in an output list.
- The Data column is the data from a field in each record. It contains a person's name.
- The Record Number column is the record's absolute record number. This is the number returned by the Record number function.
- The Selected Record Number column is the record's position in the current selection. This is the number returned by the Selected record number function.
- The Sequence Number column is the record's unique sequence number. This is the number returned by the Sequence number function when the record was created. It is stored in a field.

Table 16-1 shows the records after they are entered. The records are not sorted. The default order for the records is by record number. The records are in the default order after any command changes the current selection without sorting it; for example, after the Show All menu item is chosen in the User environment, or after the ALL RECORDS command is executed. The record number starts at 0. The selected record number and the sequence number start at 1. The sequence number is stored with each record in a field.

Table 16-1
Records and their numbers when first entered

Data	Record Number	Selected Record Number	Sequence Number
Tess	0	1	1
Terri	1	2	2
Sabra	2	3	3
Sam	3	4	4
Lisa	4	5	5

Table 16-2 shows the same records sorted by name. The same record number remains associated with each record. The selected record number reflects the record's new position in the sorted selection. The sequence number never changes, since it was assigned when each record was created and is stored in the record.

Table 16-2
Records after being sorted by name

Data	Record Number	Selected Record Number	Sequence Number
Sabra	2	1	3
Lisa	4	2	5
Sam	3	3	4
Terri	1	4	2
Tess	0	5	1

Table 16-3 shows the records after Sam is deleted. Only the selected record numbers have changed. Remember that the selected record numbers are the order in which the records are displayed.

Table 16-3
Records and their numbers after a record is deleted

Data	Record Number	Selected Record Number	Sequence Number
Tess	0	1	1
Terri	1	2	2
Sabra	2	3	3
Lisa	4	4	5

Table 16-4 shows the records after a new record has been added for Liz. A new record is added to the end of the current selection until the list is redisplayed with a command such as Show All in the User environment. Notice that Sam's record number is reused for the new record. Also notice that the sequence number continues to increment.

Table 16-4
Records and their numbers after a new record is added

Data	Record Number	Selected Record Number	Sequence Number
Tess	0	1	1
Terri	1	2	2
Sabra	2	3	3
Lisa	4	4	5
Liz	3	5	6

Table 16-5 shows the records after three records were selected and then sorted. Only the selected record number associated with each record changes.

Table 16-5
Records and their numbers after a selection and sort

Data	Record Number	Selected Record Number	Sequence Number
Sabra	2	1	3
Liz	3	2	6
Terri	1	3	2

Use special care when using these numbers in multi-user databases. The record number should generally not be used, since another user may delete the record and then save a new record in its place. See the section "Managing Multi-user Databases," later in this chapter, for more information.

Record number

Record number ({*file*}) → Number

Parameter	Type	Description
file	File	File for which to return the current record number

Record number returns the absolute record number for the current record of *file*. If there is no current record, such as when the record pointer is before or after the current selection, Record number returns –1. If the record is a new record that has not been saved, Record number returns –3.

The following example saves the current record number and then does a search to see if any other records have the same data.

```
$Rec Num := Record number ([People])                    ` Get the record number
SEARCH ([People]; [People]Last = [People]Last)          ` Anyone else with the last name?
` Display an alert with the number of people with the same last name
ALERT ("There are " + String (Records in selection ([People]) + " with that name.")
GOTO RECORD ([People]; $Rec Num)                        ` Go back to the same record
```

GOTO RECORD

GOTO RECORD (*{file}*; *record*)

Parameter	Type	Description
file	File	File in which to go to the record
record	Number	Number returned by Record number

GOTO RECORD loads and selects the specified record of *file*. The *record* parameter is the number returned by the Record number function. It is not the same number as the one returned by the Selected record number function. After executing this command, the record is the only record in the selection.

See the example for Record number, earlier in this section.

Selected record number

Selected record number (*{file}*) → Number

Parameter	Type	Description
file	File	File for which to return the selected record number

Selected record number returns the position of the current record within the current selection of *file*. The selected record number is not the same number as the number returned by Record number. (Record number returns the absolute record number in the file.)

If there is no current record, such as when the record pointer is before or after the current selection, Selected record number returns –1. If the record is a new record that has not been saved, Selected record number returns –3.

The following example saves the current selected record number.

```
Cur Rec Num := Selected record number ([People]) ` Get the selected record number
```

GOTO SELECTED RECORD

GOTO SELECTED RECORD (*{file}*; *record*)

Parameter	Type	Description
file	File	File in which to go to the selected record
record	Number	Position of record in the selection

GOTO SELECTED RECORD moves to the specified record in the current selection of *file* and makes that record the current record. The current selection does not change. The *record* parameter is not the same as the number returned by Record number; it represents the record's position in the current selection. The record's position is dependent on how the selection is made and whether the selection is sorted.

If there are no records in the current selection, or the number is not in the selection, then GOTO SELECTED RECORD does nothing.

The following example loads data from fields in a selection of records into an array, called Names. An array of integers, called RecNum, is filled with numbers that will represent the selected record numbers. Both arrays are then sorted. The resulting arrays can be used to reference the records in the selection.

```
` Copy the names into an array
SELECTION TO ARRAY ([People]Last Name; Names)
` Create an array for the selected record numbers
ARRAY INTEGER (RecNum; Size of array (Names))
For ($i; 1; Size of array (Names))                    ` Fill the array with numbers
  RecNum{$i} := $i
End for
SORT ARRAY (Names; RecNum; >)                         ` Sort both arrays
```

If the array, Names, is displayed in a scrollable area, the user can click one of the items. When the user clicks an item, the name of the array is set to the number of that item. For example, in Figure 16-1, the third item is selected, and therefore Names is set to 3.

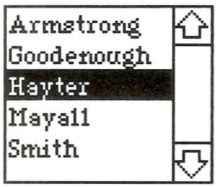


Figure 16-1
Selected name in a scrollable area

The value in Names can be used to load the associated record in the selection. The value in Names is used to access an element in the RecNum array. The value in the RecNum element is the selected record number of the record corresponding to the item clicked in the scrollable area. The next procedure is the script for the Names scrollable area. It uses GOTO SELECTED RECORD to load the record for the name that the user clicked.

```
GOTO SELECTED RECORD (RecNum{Names})
```

Sequence number

Sequence number ({*file*}) → Number

Parameter	Type	Description
file	File	File for which to return the sequence number

Sequence number returns the next sequence number for *file*. The sequence number is the same number assigned by using the #N symbol as the default value for a field in a layout. (See the *4th DIMENSION Design Reference* for information on assigning default values.) The sequence number is unique for each file. It is a nonrepeating number that is incremented for each new record added to the file. The numbering starts at 1. The number will not be lost when you delete records. The sequence number is incremented when a new record is saved, whether the number is used or not.

D156

There are four primary reasons for using Sequence number instead of the #N symbol:

- Records were created by using procedures instead of layouts.
- The number needs to start at a number other than 1.
- The number needs an increment greater than 1.
- The sequence number is part of a code, for example a part number code.

To store the sequence number by means of a procedure, create a long integer field in the file and assign the sequence number to the field.

If the sequence number needs to start at a number other than 1, simply add the difference to Sequence number. For example, if the sequence number needed to start at 1000, you would use the following statement to assign the number:

```
Seq Field := Sequence number ([File]) + 999
```

The following example is part of a layout procedure. It tests to see if this is a new record (if the invoice number is an empty string). If it is a new record, the example procedure assigns an invoice number. The invoice number is formed from two pieces of information: the sequence number, and the operator's ID, which was entered when the database was opened. The sequence number is formatted as a 5-character string.

```
If (Invoice No = "")                                  ` If this is a new part number...
   ` Create a new invoice number.
   ` The invoice number is a string that ends with Operator ID.
  Invoice No := String (Sequence number; "00000") + Operator ID
End if
```

([File])

In a multi-user database, the sequence number is updated each time a user saves a new record. When a new record is saved, other users cannot save a new record while the After phase is active. To use Sequence number in a multi-user database, you must assign the sequence number in the After phase. This ensures that the sequence number is unique and is not being used by another user.

If you need to assign the sequence number to a new record created by means of a procedure in a multi-user database, you need to use START TRANSACTION to ensure that no one else is saving a record at the same time.

Using the Record Stack

PUSH RECORD POP RECORD ONE RECORD SELECT

The commands in this section allow you to put records (push them) onto the record stack, and to remove them (pop them) from the stack.

Each file has its own record stack. 4th DIMENSION maintains the record stacks for you. Each record stack is a last-in-first-out (LIFO) stack. Stack capacity is limited by memory.

PUSH RECORD and POP RECORD should be used with discretion. Each record that is pushed uses part of free memory. Pushing too many records can cause an out-of-memory condition.

4th DIMENSION clears the stack of any unpopped records when you return to the menu at the end of the execution of your procedure.

PUSH RECORD and POP RECORD are useful when you want to examine records in the same file during data entry. To do this, you push the record, search and examine records in the file (copy fields into variables, for example), and finally pop the record to restore the record.

PUSH RECORD

PUSH RECORD (*{file}*)

Parameter	Type	Description
file	File	File from which to push record

PUSH RECORD pushes the current record of *file* (and its subrecords, if any) onto the file's record stack. PUSH RECORD may be executed before a record is saved.

The following example pushes the record for the customer onto the record stack.

```
PUSH RECORD ([Customer])                    ` Push the customer's record onto the stack
```

POP RECORD

POP RECORD ({*file*})

Parameter	Type	Description
file	File	File for which to pop record

POP RECORD pops a record (and its subrecords, if any) belonging to *file* from the file's record stack, and makes the record the current record.

If you push a record, change the selection so as not to include the pushed record, and then pop the record, the current record is not in the current selection. If you want to designate the popped record as the current selection, use ONE RECORD SELECT. If you use any commands that move the record pointer before saving the record, you will lose the copy in memory.

The following example pops the record for the customer off the record stack.

```
POP RECORD ([Customer])                    ` Pop the customer's record off the stack
ONE RECORD SELECT ([Customer])             ` Make sure the record is in the selection
```

ONE RECORD SELECT

ONE RECORD SELECT ({*file*})

Parameter	Type	Description
file	File	File for which to select record

ONE RECORD SELECT reduces the current selection of *file* to the current record. If no current record exists, ONE RECORD SELECT has no effect.

See the POP RECORD example, earlier in this section.

Managing Sets

CREATE EMPTY SET	CLEAR SET	Is in set
CREATE SET	DIFFERENCE	Records in set
USE SET	INTERSECTION	SAVE SET
ADD TO SET	UNION	LOAD SET

Sets offer you a powerful, swift means for manipulating record selections. Besides the ability to create sets, relate them to the current selection, and store, load, and clear sets, 4th DIMENSION offers three standard set operations:

- Intersection
- Union
- Difference

Sets and the Current Selection

A set is a compact representation of a selection of records. The idea of sets is closely bound to the idea of the current selection.

Sets are generally used for the following purposes:

- to work with more than one selection
- to save and later restore a selection
- to access the selection a user made on screen (the UserSet)
- to perform a logical operation between selections

The current selection is a list or table that points to each record that is currently selected. The list exists in memory. Only the records that are currently selected are in the list. A selection doesn't actually contain the records, but only a list of pointers to the records. Each pointer to a record takes 32 bits (4 bytes) in memory. When you work on a file, you always work with the records in the current selection. When a selection is sorted, only the list of pointers is rearranged. There is only one current selection for each file.

Like a current selection, a set represents a selection of records. A set does this by using a very compact representation for each record. Each record is represented by 1 bit (⅛ of a byte). Operations using sets are very fast, because computers can perform operations on bits very quickly. A set contains 1 bit for every record in the file, whether the record is included in the set or not.

The size of a set, in bytes, is always equal to the total number of records in the file divided by 8. For example, if you create a set for a file containing 10,000 records, the set takes up 1250 bytes, which is about 1.2K in RAM. Sets are very economical in terms of RAM and disk space.

There can be many sets for each file; in fact, sets can be saved to disk separately from the database. A set is never directly used to access records. The current selection must first be changed to reflect the set.

A set is never in a sorted order—the records are simply indicated as belonging to the set or not.

A set "remembers" which record was the current record at the time the set was created.

Table 16-6 compares the concepts of the current selection and of sets.

Table 16-6
Current selection and sets concepts compared

Comparison	Current Selection	Sets
Number per file	1	0 to many
Sortable	Yes	No
Can be saved on disk	No	Yes
RAM per record	32 bits (4 bytes)	1 bit (⅛ of a byte)
Combinable	No	Yes
Contains current record	Yes	Yes, as of the time the set was created

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

Important: Sets are independent from the data. This means that after changes are made to a file, a set may no longer be accurate. There are many operations that can cause a set to be inaccurate. If you created a set of all the people from New York, and then changed the data in one of those records to "New Jersey," the set would not change, since the set is simply a representation of a selection of records. Deleting records and then adding new records can also include records in a set that were not originally included. Sets can be guaranteed to be accurate only as long as the data in the corresponding selection has not been changed.

Set Example

The example below deletes duplicate records from a file. The file contains information about people. A For loop moves through all the records, comparing the current record to the previous record. If the first name and the last name are the same, then the record is added to a set. At the end of the loop, the set is made the current selection and the current selection is deleted.

```
DEFAULT FILE ([People])                          ` Set the default file
CREATE EMPTY SET ("Duplicates")                  ` Create an empty set for duplicate records
ALL RECORDS                                      ` Select all records
` Sort the records by ZIP, address, and name so
` that the duplicates will be next to each other
SORT SELECTION ([Addresses]ZIP; >; [Addresses]Address; >; [Addresses]Name; >)
$Name := [People]Name                            ` Initialize variables that hold the
$Address := [People]Address                      ` fields from the previous record
$ZIP := [People]ZIP
NEXT RECORD                                      ` Go to second record to compare to first
```

```
For ($i; 2; Records in file)                    ` Loop through all records starting at #2
  ` If the name, address, and ZIP are the same as the
  ` previous record then it is a duplicate record.
  If (([People]Name = $Name) & ([People]Address = $Address) & ([People]ZIP = $ZIP))
    ADD TO SET ("Duplicates")                   ` Add current record (the duplicate) to set
  Else
    $Name := [People]Name                       ` Save this record's name, address, and ZIP
    $Address := [People]Address                 ` for comparison with the next record
    $ZIP := [People]ZIP
  End if
  NEXT RECORD                                   ` Move to the next record
End for
USE SET ("Duplicates")                          ` Use the duplicate records that were found
DELETE SELECTION                                ` Delete the duplicate records
CLEAR SET ("Duplicates")                        ` Remove the set from memory
```

As an alternative to immediately deleting the records at the end of the procedure, you could display them on screen or print them, so that a more detailed comparison could be made.

The UserSet System Set

4th DIMENSION maintains a system set named UserSet. UserSet automatically stores the most recent selection of records selected on screen by the user. Thus, you can display a group of records with MODIFY SELECTION or DISPLAY SELECTION, 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. There is only one UserSet for a database. Each file *does not* have its own UserSet. UserSet becomes "owned" by a file when a selection of records is displayed for the file. The following procedure illustrates how you can display records, allow the user to select some, and then use UserSet to display the selected records.

UserSet

LockedSet

NextPage

```
` Display all records and allow user to select any number of them.
` Then display this selection by using UserSet to change the current selection.
DEFAULT FILE ([People])                         ` Set the default file
OUTPUT LAYOUT ("Display")                       ` Set the output layout
ALL RECORDS                                     ` Select all the people
ALERT ("Press Command and Click to select the people required.")
DISPLAY SELECTION                               ` Display the people
USE SET ("UserSet")                             ` Use the people that were selected
ALERT ("You chose the following people.")
DISPLAY SELECTION                               ` Display the selected people
```

The LockedSet System Set

LockedSet

UserSet

Previous Page

The APPLY TO SELECTION and DELETE SELECTION commands create a set named LockedSet when used in a multi-user environment. LockedSet indicates which records were locked during the operation of the command. For more information, see the section "Managing Multi-user Databases," later in this chapter, and the sections on the APPLY TO SELECTION and DELETE SELECTION commands, in Chapter 14.

CREATE EMPTY SET

CREATE EMPTY SET (*{file}*; *set*)

Parameter	Type	Description
file	File	File for which to create an empty set
set	String	Name of the new empty set

CREATE EMPTY SET creates a new empty set, *set,* for *file*. You can add to this set with the ADD TO SET command. If a set with the same name already exists, the existing set is cleared by the new set.

The following example creates a new set and then "merges" the UserSet with it (with the UNION command), so that the UserSet can be saved.

```
CREATE EMPTY SET ([People]; "Save Set")      ` Create a new set
UNION ("UserSet"; "Save Set"; "Save Set")    ` Merge the two sets together
```

CREATE SET

CREATE SET (*{file}*; *set*)

Parameter	Type	Description
file	File	File for which to create a set from the selection
set	String	Name of the new set

CREATE SET creates a new set, *set,* for *file,* and places the current selection in *set*. The current record pointer for the file is saved with *set*. If *set* is used with USE SET, the current selection and current record are restored. As with all sets, there is no sorted order, and when *set* is used the default order is used. If a set already exists with the same name, the existing set is cleared by the new set.

The following example creates a set after doing a search so that the set can be saved to disk.

```
SEARCH ([People])                            ` Let the user do a search
CREATE SET ([People]; "Save Set")            ` Create a new set
SAVE SET ("Save Set"; "My Search")           ` Save the set on disk
```

USE SET

USE SET (*set*)

Parameter	Type	Description
set	String	Name of the set to use

USE SET makes the records in *set* the current selection for the file to which the set belongs.

When you create a set, the current record is "remembered" by the set. USE SET retrieves the position of this record and makes the record the new current record. If you delete this record before you execute USE SET, 4th DIMENSION selects the first record in the set as the current record. Also, if you form a set that does not contain the position of the current record, USE SET selects the first record in the set as the current record. The set commands INTERSECTION, UNION, DIFFERENCE, and ADD TO SET reset the current record.

Caution: Remember that a set is a representation of a selection of records at the moment that the set is created. If the records that the set represents change, the set may no longer be accurate. Therefore, a set saved to disk should normally represent a group of records that does not change frequently. A number of things can make a set invalid: modifying a record of the set, deleting a record of the set, or changing the criteria that determined a set.

The following example uses LOAD SET to load a set of the Acme locations in New York. It then uses USE SET to make the loaded set the current selection.

```
` Load the set into memory
LOAD SET ([Companies]; "NY Acme"; "NY Acme Set")
USE SET ("NY Acme")                    ` Change the current selection to NY Acme
CLEAR SET ("NY Acme")                  ` Clear the set from memory
```

ADD TO SET

ADD TO SET ({*file*}; *set*)

Parameter	Type	Description
file	File	File from which to add current record
set	String	Name of the set to which to add the record

ADD TO SET adds the current record of *file* to *set*. The set must already exist; if it does not, an error occurs. If a current record does not exist for *file*, ADD TO SET has no effect.

The following example adds the currently displayed record to a set. The first section of code is a global procedure that displays a selection. The procedure creates a new set, displays the records, and then creates a current selection from the records that the user selected.

```
DEFAULT FILE ([Invoices])          ` Set the default file to Invoices
CREATE EMPTY SET ("Selected")      ` Create a new set for the file
MODIFY SELECTION                   ` Display the records
USE SET ("Selected")               ` Use the records that the user selected
```

The next section of code is a script for a button in the input layout. It simply adds the current record (the record that the user is viewing) to the existing set. When the user is done viewing the records, the records in the set are the ones that the user selected.

```
ADD TO SET ("Selected")            ` Add the current record to the set
```

CLEAR SET

CLEAR SET (*set*)

Parameter	Type	Description
set	String	Name of the set to clear from memory

CLEAR SET clears *set* from memory and frees the memory used by *set*. CLEAR SET does not affect files, selections, or records. To save a set before clearing it, use the SAVE SET command. Since sets use memory, it is good practice to clear sets when they are no longer needed.

The following example creates a set, saves it to disk, and then clears the set.

```
DEFAULT FILE ([People])            ` Set the default file
SEARCH                             ` Let the user do a search
CREATE SET ("Save Set")            ` Create a new set
SAVE SET ("Save Set"; "My Search") ` Save the set on disk
CLEAR SET ("Save Set")             ` Clear the set from memory
```

DIFFERENCE

NOT IMP

DIFFERENCE (*set1*; *set2*; *result set*)

Parameter	Type	Description
set1	String	Original set
set2	String	Set to "exclude"
result set	String	Resulting set

DIFFERENCE compares *set1* and *set2* and excludes all records that are in *set2* from the *result set*. In other words, a record is included in the *result set* only if it is in *set1*, but not in *set2*. Table 16-7 shows all possible results of a set Difference operation.

Table 16-7
Results of a set Difference operation

Set1	Set2	Result Set
Yes	No	Yes
Yes	Yes	No
No	Yes	No
No	No	No

Figure 16-2 shows the result of a Difference operation graphically. The shaded area is the result set.

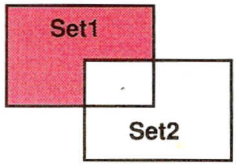


Figure 16-2
The result set of a Difference operation

The *result set* is created by DIFFERENCE. The *result set* replaces any set that already exists with the same name, including *set1* and *set2*. Both *set1* and *set2* must be from the same file. The *result set* belongs to the same file as *set1* and *set2*.

The following example excludes the records that a user selects from a displayed selection. The records are displayed on screen with the following line:

```
DISPLAY SELECTION ([Customers])          ` Display the customers in a list
```

At the bottom of the list of records is a button with a script. The script excludes the records that the user has selected (the UserSet), and displays the new set.

```
CREATE SET ([Customers]; "Current")                    ` Create a set of the current selection
DIFFERENCE ("Current";"UserSet";"Current")             ` Exclude records that the user selected
USE SET ("Current")                                    ` Use the new set
CLEAR SET ("Current")                                  ` Clear the set
```

INTERSECTION

INTERSECTION (*set1*; *set2*; *result set*)

Parameter	Type	Description
set1	String	First set
set2	String	Second set
result set	String	Resulting set

INTERSECTION compares *set1* and *set2* and selects only the records that are in both *set1* and *set2*. Table 16-8 shows all possible results of a set Intersection operation.

Table 16-8
Results of a set Intersection operation

Set1	Set2	Result Set
Yes	No	No
Yes	Yes	Yes
No	Yes	No
No	No	No

Figure 16-3 shows the result of an Intersection operation graphically. The shaded area is the result set.

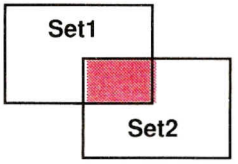


Figure 16-3
The result set of an Intersection operation

The *result set* is created by INTERSECTION. The *result set* replaces any set that already exists with the same name, including *set1* and *set2*. Both *set1* and *set2* must be from the same file. The *result set* belongs to the same file as *set1* and *set2*.

The following example finds the customers that are served by two sales representatives, Joe and Abby. Each sales representative has a set that represents his or her customers. The customers that are in both sets are represented by both Joe and Abby.

```
INTERSECTION ("Joe"; "Abby"; "Both")      ` Put the customers in both sets in Both
USE SET ("Both")                          ` Use the set
CLEAR SET ("Both")                        ` Clear this set but save the others
DISPLAY SELECTION ([Customers])           ` Display the customers served by both
```

UNION

OR

UNION (*set1*; *set2*; *result set*)

Parameter	Type	Description
set1	String	First set
set2	String	Second set
result set	String	Resulting set

UNION creates a set that contains all records from *set1* and *set2*. Table 16-9 shows all possible results of a set Union operation.

Table 16-9
Results of a set Union operation

Set1	Set2	Result Set
Yes	No	Yes
Yes	Yes	Yes
No	Yes	Yes
No	No	No

Figure 16-4 shows the result of a Union operation graphically. The shaded area is the result set.

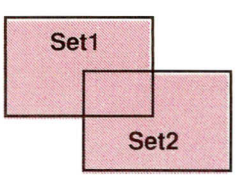


Figure 16-4
The result set of a Union operation

The *result set* is created by UNION. The *result set* replaces any set that already exists with the same name, including *set1* and *set2*. Both *set1* and *set2* must be from the same file. The *result set* belongs to the same file as *set1* and *set2*.

The following example adds records to a set of best customers. The records are displayed on screen with the first line. After the records are displayed, a set of the best customers is loaded from disk, and any records that the user selected (the UserSet) are added to the set. Finally, the new set is saved to disk.

```
DEFAULT FILE ([Customers])                  ` Set the default file
ALL RECORDS                                 ` Select all the customers
DISPLAY SELECTION                           ` Display all the customers in a list
LOAD SET ("Best"; "Saved Best")             ` Load the set of best customers
UNION ("Best"; "UserSet"; "Best")           ` Add any selected to the set
SAVE SET ("Best"; "Saved Best")             ` Save the set of best customers
```

Is in set

Is in set (*set*) → Boolean

Parameter	Type	Description
set	String	Set to test

Is in set tests whether the current record for the file that *set* belongs to is in *set*. Is in set returns TRUE if the current record of the file is in *set*, and returns FALSE if the current record of the file is not in *set*.

The following example is a button script. It tests to see whether the record currently displayed is in the set of best customers.

```
If (Is in set ("Best"))                     ` Check if it is a good customer
  ALERT ("They are one of our best customers.")
Else
  ALERT ("They are not one of our best customers.")
End if
```

Records in set

Records in set (*set*) → Number

Parameter	Type	Description
set	String	Set to test

Records in set returns the number of records in *set*. If *set* does not exist, or if there are no records in *set*, Records in set returns 0.

The following example displays an alert saying what percentage of the customers are rated as the best.

```
` First calculate the percentage
$Percent := (Records in set ("Best") / Records in file ([Customers])) * 100
` Display an alert with the percentage
ALERT (String ($Percent; "##0%") + " of our customers are the best.")
```

SAVE SET

SAVE SET (*set*; *document*)

Parameter	Type	Description
set	String	Name of the set to save
document	String	Name of the disk file to which to save

SAVE SET saves *set* to *document*, a document on disk.

The *document* need not have the same name as the set. If you supply an empty string for *document*, a create-file dialog box appears, so that the user can enter the name of the file. You can load a saved set with the LOAD SET command.

If the user clicks Cancel in the create-file dialog box, or there is an error during the save operation, the OK system variable is set to 0. Otherwise, it is set to 1.

SAVE SET is often used to save to disk the results of a time-consuming search.

Caution: Remember that a set is a representation of a selection of records at the moment that the set is created. If the records that the set represents change, the set may no longer be accurate. Therefore, a set saved to disk should normally represent a group of records that does not change frequently. A number of things can make a set invalid: modifying a record of the set, deleting a record of the set, or changing the criteria that determined a set.

The following example uses SAVE SET to save the result of a sequential search. The search is for all of the Acme locations in New York. The resulting set is saved to a user-specified set.

```
DEFAULT FILE ([Companies])                    ` Set the default file
SEARCH ([Companies]Name = "Acme@"; *)         ` First part of the search...
SEARCH (& [Companies]State = "NY")            ` second part of the search
CREATE SET ([Companies]; "NY Acme")           ` Makes the current selection a set
SAVE SET ("NY Acme"; "")                      ` Saves to a user-named set
CLEAR SET ("NY Acme")                         ` Clear the set from memory
```

Using SAVE SET in a multi-user database is discouraged, since the data that the set represents may be changed by other users, therefore making the set invalid.

LOAD SET

LOAD SET (*{file}*; *set*; *document*)

Parameter	Type	Description
file	File	File to which the set belongs
set	String	Set to be created in memory
document	String	Document holding the set

LOAD SET loads from *document* a set that was saved with the SAVE SET command.

The set that is stored in *document* must be from *file*. The set created in memory is overwritten if it already exists.

The *document* parameter is the name of the Macintosh document that contains the set. The document need not have the same name as the set. If you supply an empty string for *document*, an open-file dialog box appears, so that the user can choose the set to load.

If the user clicks Cancel in the open-file dialog box, or there is an error during the load operation, the OK system variable is set to 0. Otherwise, it is set to 1.

Caution: Remember that a set is a representation of a selection of records at the moment that the set is created. If the records that the set represents change, the set may no longer be accurate. Therefore, a set loaded from disk should normally represent a group of records that does not change frequently. A number of things can make a set invalid: modifying a record of the set, deleting a record of the set, or changing the criteria that determined a set.

The following example uses LOAD SET to load a set of the Acme locations in New York.

```
` Load the set into memory
LOAD SET ([Companies]; "NY Acme"; "NY Acme Set")
USE SET ("NY Acme")                    ` Change the current selection to NY Acme
CLEAR SET ("NY Acme")                  ` Clear the set from memory
```

Using LOAD SET in a multi-user database is discouraged, since the data that the set represents may be changed by other users, therefore making the set invalid.

Managing Multi-user Databases

Locked	READ WRITE	Semaphore
LOAD RECORD	READ ONLY	CLEAR SEMAPHORE
UNLOAD RECORD		

4th DIMENSION automatically manages simple multi-user databases by allowing only one user at a time to modify a record. There are three primary reasons for using the multi-user commands in this section:

- You are modifying records by using the language.
- You want to use a custom user interface for multi-user operations.
- You need to optimize network activities.

The multi-user commands are specific to multi-user operations. These commands have no effect in a single-user database.

There are three important concepts to be aware of when using commands in a multi-user database:

- Each file is in either a read-only or a read-write state.
- Records become locked or unlocked when they are loaded.
- A locked record cannot be modified.

In the sections that follow, the person performing an operation on the multi-user database is the *local user*. Other people using the database are referred to as the *other users*. The discussion is from the perspective of the local user.

Locked Records

A locked record cannot be modified by the local user. A locked record can be loaded, but cannot be modified. A record is locked when one of the other users has successfully loaded the record for modification. Only the user who is modifying the record sees that record as unlocked. All other users see the record as locked, and therefore unavailable for modification.

A file must be in a read-write state for a record to be loaded unlocked.

Read-Only and Read-Write States

Each file in a database is in either a read-write or a read-only state for each user of the database. *Read-only* means that records for the file can be loaded but not modified. In other words, they are always locked and unmodifiable for the local user. *Read-write* means that records for the file can be loaded and modified if no other user has locked the record first.

A file is set to read-write with the command READ WRITE. Read-write is the default state for all files when a database is opened. When a file is read-write and a record is loaded, the record will become unlocked if no other user has locked the record first. If the record is locked by another user, the record is loaded, but it is locked, and the local user is not able to save modifications. A file must be set to read-write and the record loaded for it to become unlocked and thus modifiable.

A file is set to read-only with the command READ ONLY. When a file is read-only and a record is loaded, the record is always locked. In other words, the record can be displayed, printed, and otherwise used, but it cannot be modified.

Each user has his or her own local state (read-only or read-write) for each file in the database. The current record for each file is loaded according to the current state. When you use the commands READ WRITE or READ ONLY to change to a different state, only the records that are subsequently loaded are affected.

4th DIMENSION automatically sets a file to read-only for commands that do not require write access to records. Table 16-10 lists the commands that set a file to read-only.

Table 16-10
Commands that set a file to read-only

Command	Command	Command
DISPLAY SELECTION	GRAPH FILE	PRINT LABELS
EXPORT DIF	MERGE SELECTION	PRINT SELECTION
EXPORT SYLK	SELECTION TO ARRAY	REPORT
EXPORT TEXT		
MODIFY SELECTION (except when a record is double-clicked)		

Before executing any of the commands in Table 16-10, 4th DIMENSION saves the current state (read-only or read-write) for the file. After the command has executed, the state is restored. If you need to modify records during the operation performed by any of these commands, you can use 4D Customizer to override this feature. See the section "4D Customizer" in the *4th DIMENSION Utilities Guide* for more information on setting this feature.

Loading, Modifying, and Unloading Records

Before the local user can modify a record, the file must be in the read-write state, and the record must be loaded and unlocked.

The commands in Table 16-11 load a record.

Table 16-11
Commands that load a record

Command	Command	Command
ALL RECORDS	MODIFY RECORD	SEARCH
APPLY TO SELECTION	NEXT RECORD	SEARCH BY FORMULA
CREATE LINKED ONE	OLD RELATED MANY	SEARCH BY INDEX
CREATE RECORD	OLD RELATED ONE	SEARCH BY LAYOUT
FIRST RECORD	ONE RECORD SELECT	SEARCH SELECTION
GOTO RECORD	PREVIOUS RECORD	SORT BY INDEX
GOTO SELECTED RECORD	RELATE MANY	SORT FILE
LAST RECORD	RELATE ONE	SORT SELECTION
LOAD RECORD	USE SET	

Any of the commands in Table 16-11 loads the current record (if there is one) and sets the record as locked or unlocked. The record is loaded according to the current state of its file (read-only or read-write).

A record may also be loaded for a related file by any of the commands that cause an automatic relation to be established. See the section "Managing File Relations," in Chapter 14, for a list of these commands.

If a file is in the read-only state, then a record that is loaded from that file is locked. A *locked* record *cannot* be saved or deleted. Read-only is the preferred state, since it allows other users to load, modify, and then save the record.

If a file is in the read-write state, then a record that is loaded from that file is unlocked only if no other users have locked the record first. An *unlocked* record *can* be saved. A file should be put into the read-write state only immediately before a record needs to be loaded, modified, and then saved.

You use the Locked command to test whether a record is locked by another user. If a record is locked (Locked is TRUE), load the record with the LOAD RECORD command and again test whether the record is locked. This sequence must be continued until the record is unlocked (Locked is FALSE), if the record is to be modified.

A record must be released (and therefore unlocked for the other users) with UNLOAD RECORD. If a record is not unloaded, it will remain locked for all other users until a different current record is selected.

Loops to Load Unlocked Records

The following example shows the simplest loop with which to load an unlocked record.

```
READ WRITE                                  ` Set the file's state to read-write
Repeat                                      ` Loop until the record is unlocked
  LOAD RECORD                               ` Load the record and set the locked status
Until (Not (Locked))
` Do something to the record here
READ ONLY                                   ` Set the file's state to read-only
```

The loop continues indefinitely until the record is unlocked.

A loop like this is used only if the record is unlikely to be locked by anyone else, since the user would have to wait for the loop to terminate. Thus, it is unlikely that the loop would be used as is unless the record could only be modified by means of a procedure.

The following procedure uses the loop to load an unlocked record and modify the record:

```
DEFAULT FILE (Inventory])                   ` Set the default file
READ WRITE
Repeat                                      ` Loop until the record is unlocked
  LOAD RECORD                               ` Load the record and set the locked status
Until (Not (Locked))
[Inventory]Part Qty := [Inventory]Part Qty – 1   ` Modify the record
SAVE RECORD                                 ` Save the record
UNLOAD RECORD                               ` Unload the record so others can modify it
READ ONLY
```

The MODIFY RECORD command automatically notifies the user if a record is locked, and prevents the record from being modified. The following example avoids the automatic notification by first testing the record with the Locked command. If the record is locked, the code allows the user to cancel.

The example first installs an event-trapping procedure so that the While loop can be terminated. It then loads the current record for the default file and tests whether the record is locked by another user. If the record is locked, a message is displayed and the loop continues. If the record is unlocked, then the If code is executed and the user can modify the record. After the record is modified, it is unloaded so that other users can modify it.

```
Loop := True                                ` Initialize the loop variable
READ WRITE                                  ` Set the file's state to read-write
LOAD RECORD                                 ` Load the record and set the locked status
ON EVENT CALL ("Trap")                      ` Install the event-trapping procedure
```

```
While (Loop & Locked)                          ` Loop while the record is locked
  MESSAGE ("The record is locked by another user. Press Q to cancel.")
  LOAD RECORD                                  ` Load the record and set the locked status
End while
ON EVENT CALL ("")                             ` Remove event trapping
If (Not (Locked))                              ` If the record is unlocked...
  MODIFY RECORD                                ` let the user modify the record
  UNLOAD RECORD                                ` Unload the record so others can modify it
End if
READ ONLY                                      ` Set the file's state to read-only
```

The following is the *Trap* procedure installed by the ON EVENT CALL command. The *Trap* procedure simply sets Loop to FALSE when the user presses the Q key. When Loop is set to FALSE, the loop ends, the record is not unlocked, and MODIFY RECORD is not executed.

```
If (Char (KeyCode) = "q")
  Loop := False
End if
```

Using Commands in a Multi-user Database

A number of commands in the language perform specific actions when they encounter a record that is locked. They behave normally if they do not encounter a locked record. Here is a list of those commands, showing the actions of each when it encounters a locked record. For more information on each of the commands, see its description in Part III.

- MODIFY RECORD—This command displays a dialog box stating that the record is in use. The record is not displayed and therefore the user cannot modify the record.
- MODIFY SELECTION—This command behaves normally except when the user double-clicks a record to modify it. MODIFY SELECTION then acts like MODIFY RECORD and displays a dialog box stating that the record is in use.
- APPLY TO SELECTION—This command loads a locked record, but does not modify it. APPLY TO SELECTION can be used to read information from the file without special care. If the command encounters a locked record, the locked record is put into a system set called LockedSet.
- DELETE SELECTION—This command does not delete any locked records. It skips the locked records. If it encounters a locked record, the locked record is put into a system set called LockedSet.
- DELETE RECORD—This command is ignored if the record is locked. No error is returned. You must test that the record is unlocked before executing this command.

- SAVE RECORD—This command is ignored if the record is locked. No error is returned. You must test that the record is unlocked before executing this command.
- ARRAY TO SELECTION—This command does not save any records that are locked.

The GOTO RECORD command and the set commands need special attention if they are used in a multi-user database:

- GOTO RECORD—Records in a multi-user database may be deleted and added by other users. Therefore the record numbers may change. Use caution when directly referencing a record by number in a multi-user database.
- Set Commands—Special care needs to be taken with sets since the information that the set was based on may be changed by another user.

Locked

Locked ({*file*}) → Boolean

Parameter	Type	Description
file	File	File to check for record locked

Locked tests whether the current record of *file* is locked.

If Locked returns TRUE, then the record is locked by another user and cannot be saved. In this case, use LOAD RECORD to load the record until Locked returns FALSE.

If Locked returns FALSE, then the record is unlocked, meaning that the record *is* locked for all other users. The local user (and only the local user) can modify and save the record. A file must be in a read-write state for Locked to return FALSE.

If another user has deleted the record you loaded, Locked returns FALSE and an empty record is in memory. This prevents an infinite loop from occurring if you are trying to load a deleted record.

Use this function to find out whether the record is locked; then take appropriate action—such as giving the user a choice of waiting for the record to be free or skipping the operation.

During transaction processing, LOAD RECORD and Locked are often used to test records. If a record is locked, it is common to cancel the transaction.

See the examples in "Loops to Load Unlocked Records," earlier in this section.

LOAD RECORD

LOAD RECORD ({*file*})

Parameter	Type	Description
file	File	File from which to load record

LOAD RECORD loads the current record of *file*. The status of the record can then be tested with the Locked command. If there is no current record, LOAD RECORD has no effect.

UNLOAD RECORD is used to release (unlock) the record for other users.

See the examples in "Loops to Load Unlocked Records," earlier in this section.

UNLOAD RECORD

UNLOAD RECORD ({*file*})

Parameter	Type	Description
file	File	File for which to unload record

UNLOAD RECORD unloads the current record of *file*. If the record is unlocked for the local user (locked for the other users), UNLOAD RECORD unlocks the record for the other users.

Although UNLOAD RECORD unloads the record from memory, it remains the current record.

When another record is made the current record, the previous current record is automatically unloaded and therefore unlocked for other users.

Always execute this command when you are done modifying a record and want to make it available to other users, yet retain the record as your current record.

See the examples in "Loops to Load Unlocked Records," earlier in this section.

READ WRITE

READ WRITE ({*file*})

Parameter	Type	Description
file	File	File for which to set multi-user state

READ WRITE changes the state of *file* to read-write. When a record is loaded, it is unlocked if no other user has locked the record. This command does not change the status of the currently loaded record, only that of subsequently loaded records.

The default state for all files is read-write.

Use READ WRITE when you must modify a record and save the changes. Also use READ WRITE when you must lock a record for other users, even if you are not making any changes.

See the examples in "Loops to Load Unlocked Records," earlier in this section.

READ ONLY

READ ONLY ({*file*})

Parameter	Type	Description
file	File	File for which to set multi-user state

READ ONLY changes the state of *file* to read-only. All subsequent records that are loaded are locked, and the user cannot save any changes made to them.

Use READ ONLY when the loaded record does not need to be unlocked—in other words, when you do not need to modify the record.

See the examples in "Loops to Load Unlocked Records," earlier in this section.

Semaphore

Semaphore (*semaphore*) → Boolean

Parameter	Type	Description
semaphore	String	Semaphore to set

A semaphore is a simple message between workstations (each user's computer) in a multi-user database. A semaphore simply exists or does not exist. The procedures that each user is running can test for the existence of a semaphore. By creating and testing semaphores, procedures can communicate between workstations.

Semaphore is a function that returns TRUE if *semaphore* exists. If *semaphore* does not exist, Semaphore creates the semaphore and returns FALSE. Only one user at a time can create a semaphore.

Semaphores *are not* needed for multi-user operation. This command performs no action but to set the semaphore. Semaphores are simply a messaging mechanism between workstations in a multi-user environment.

Semaphores are stored in the Flags file. If a semaphore is accidently left set, you can clear it and all other semaphores by throwing away the Flags file while no one is using the database.

The semaphore names "1" through "64" are for internal use by 4th DIMENSION, and should not be used as semaphores.

The following example checks a semaphore. In the example, the semaphore is set when someone wants to prevent any access to a specific file. Note that this *does not* lock the file; the procedures that use the semaphore simply recognize its meaning.

```
` Did someone set a semaphore called File Lock?
If (Semaphore ("File Lock"))
  ALERT ("Someone has prevented access to that file.")
Else
  Do Access                                  ` The semaphore is set. Do something.
  CLEAR SEMAPHORE ("File Lock")              ` Clear the semaphore for other users
End if
```

CLEAR SEMAPHORE

CLEAR SEMAPHORE (*semaphore*)

Parameter	Type	Description
semaphore	String	Semaphore to clear

CLEAR SEMAPHORE erases *semaphore*.

The semaphore names "1" through "64" are for internal use by 4th DIMENSION, and should not be cleared.

See the example for Semaphore, earlier in this section.

Using Transactions

START TRANSACTION CANCEL TRANSACTION VALIDATE TRANSACTION

Transactions are a series of related data modifications that are made to a database. A transaction is not saved permanently to a database until the transaction is validated. If a transaction is not completed, either because it is canceled or because of some outside event, the modifications are not saved.

A transaction is started with the command START TRANSACTION. The database immediately becomes locked to all other users in a multi-user database. There can be only one transaction at a time. For this reason, it is critical that all transactions be as short as possible, and that there be no user intervention.

If a transaction is in process, transactions started on all other workstations will wait until the current transaction is complete. The pending transactions are started randomly; the transactions are not queued in order.

During a transaction, all changes to the data of a database are stored locally in a temporary buffer. If the transaction is accepted with ACCEPT TRANSACTION, the changes are saved permanently. If the transaction is canceled with CANCEL TRANSACTION, the changes are not saved.

Transaction Example

The example in this section is based on the database structure shown in Figure 16-5. The database is a simple invoicing system. The invoice lines are stored in a subfile called [Invoice]Lines, and the inventory is stored in a file called [Parts].

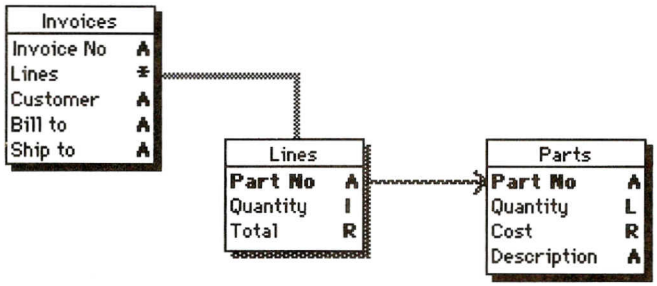
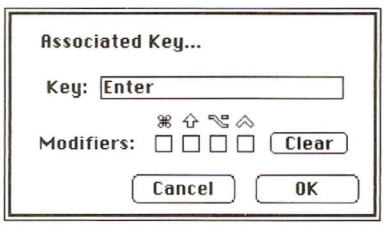


Figure 16-5
An invoice database

The user enters the invoice and invoice lines. During data entry, the inventory is checked, and the user is alerted if an item is out of stock.

When the user is done, he or she clicks a button to save the invoice. The example procedure is the script for that button. The button has no action. The Enter key is associated with the button. This ensures that the script is executed even if the user presses the Enter key to accept the record. Figure 16-6 shows the Enter key being associated with the button.



⌘ ⇧ ⌥ ^
Command, Shift, Option CtrL
D 168

Figure 16-6
The Enter key associated with a button

Although the inventory has been checked during data entry, it is possible that another user has saved an invoice that depleted inventory. If the inventory is not sufficient, a transaction allows all the changes that are made to the database to be canceled.

The script starts a transaction. It then reduces the number of parts in inventory by the number of parts ordered in the invoice line. If there are not enough parts in inventory, or the record is locked, the transaction is canceled and the user is alerted. Otherwise, the invoice record is saved and the transaction is validated.

It is important that the invoice record is saved as part of the transaction, since if there is an error (such as a power failure or a disk error) while the transaction is in progress, the complete transaction, including the record, will be canceled.

```
Trans Error := 0                                    ` Use this to check the type of error
ALL SUBRECORDS ([Invoices]Lines)                    ` Select all the invoice lines
ON ERR CALL ("Cancel Trans")                        ` This traps errors such as disk full
START TRANSACTION                                   ` Start transaction before any data changes

` Loop once for each record and only as long as there are no errors
While (Not (End subselection ([Invoices]Lines)) & (Trans Error = 0))
  RELATE ONE ([Invoices]Lines'Part No)              ` Get the part record & remove the quantity
  [Parts]Quantity := [Parts]Quantity – [Invoices]Lines'Quantity
  Case of                                           ` Check for errors
    : ([Parts]Quantity < 0)
      Trans Error := 10000                          ` Use error codes that 4D does not use
    : (Locked ([Parts]))
      Trans Error := 10001
    Else                                            ` No errors and enough in stock...
      SAVE RECORD ([Parts])                         ` so save the record and...
      NEXT SUBRECORD ([Invoices]Lines)              ` go to the next invoice line
  End case
End while

If (Trans Error = 0)                                ` This next step could cause an error...
  SAVE RECORD([Invoices])                           ` and must be part of the transaction
End if

If (Trans Error = 0)                                ` Finally, if there were no errors...
  VALIDATE TRANSACTION                              ` save everything
Else                                                ` Otherwise, cancel the transaction
  CANCEL TRANSACTION
End if

Case of
  : (Trans Error = 0)                               ` There were no errors so...
    CANCEL                                          ` leave data entry
  : (Trans Error = 10000)                           ` There was an error, so tell the user
    ALERT ("Part no: " + [Invoices]Lines'Part No + Char (13) + "is out of stock.")
  : (Trans Error = 10001)
    ALERT ("Part no: " + [Invoices]Lines'Part No + Char(13) + "is in use.")
  Else                                              ` For all other errors
    ALERT ("Error #" + String (Trans Error) + " occurred.")
End case

ON ERR CALL ("")                                    ` Reset error trapping
```

The example procedure installs *Cancel Trans* as an ON ERR CALL procedure in the third line. The *Cancel Trans* procedure traps any unexpected errors and sets the Trans Error variable, therefore canceling the transaction if an error occurs. The following line is the complete *Cancel Trans* procedure:

```
Trans Error := Error
```

It is important to note that the example is for use when entering a new invoice. If an invoice needed to be modified, the Old command would be used to update the inventory.

START TRANSACTION

extra

START TRANSACTION

START TRANSACTION starts a transaction. All changes to the database will be stored temporarily until the transaction is accepted (validated) or canceled.

See the example earlier in this section.

Only one user at a time can have a transaction active. If another user has started a transaction, this command will continue checking until the other user's transaction has completed. It will then start the transaction.

CANCEL TRANSACTION

CANCEL TRANSACTION

CANCEL TRANSACTION cancels the transaction that was started with START TRANSACTION. CANCEL TRANSACTION returns the data in the database to the condition it was in before the start of the transaction.

See the example earlier in this section.

VALIDATE TRANSACTION

VALIDATE TRANSACTION

VALIDATE TRANSACTION accepts the transaction that was started with START TRANSACTION. VALIDATE TRANSACTION saves the changes to the database that occurred during the transaction.

See the example earlier in this section.

Communicating With Documents and the Serial Port

Create document	SEND PACKET	RECEIVE BUFFER
Open document	RECEIVE PACKET	SEND RECORD
Append document	SET CHANNEL	RECEIVE RECORD
CLOSE DOCUMENT	ON SERIAL PORT CALL	SEND VARIABLE
DELETE DOCUMENT	SET TIMEOUT	RECEIVE VARIABLE
USE ASCII MAP		

The commands in this section allow you to send and receive data to and from both documents and the serial port. Two commands in particular are used for communications: SEND PACKET and RECEIVE PACKET. These commands send data as packets. A *packet* is just a piece of data, generally a string of characters.

Working With Documents

L 383

The document commands create, open, close, and delete Macintosh documents (disk files). Documents can be read from and written to using the commands RECEIVE PACKET and SEND PACKET. The documents may be used to store database information, such as variables, sets, and copies of records. The documents may also have been created with another application.

You can open multiple documents with the document commands. The number of open documents is limited only by the Macintosh file system. However, you should close all documents that do not need to be open, to avoid having too many Macintosh files open at the same time.

The document commands use document names. A document name can contain a path—the description of the location of a document in a directory. A path to a document is constructed as follows:

volume:folder1:...:folder*n*:document

For example, to access a document named Sales Table, contained in a folder named Sales Folder, in a folder named Business, saved on a disk (a volume) named Office, the path would be

Office:Business:Sales Folder:Sales Table.

If a document is specified without a path, it is assumed to be in the folder that contains the database data file.

4th DIMENSION maintains a system variable called Document. The Document system variable contains the name of and path to the document that was last accessed.

Note on loose sheet

Docref

Commands that open a document return a *document reference.* You use the document reference to access the document. You should never modify the document reference. When a document command returns a document reference, save it in a variable and use the variable to refer to the document.

You can use a local variable to store the document reference, but be careful. If the procedure ends and you have not closed the document, you will not be able to, because the local variable will have been cleared.

Create document

Create document (*document*; *{type}*) → Docref

Parameter	Type	Description
document	String	Document name
type	String	Document type (4 characters)

Create document creates a new document with the name *document,* and returns a document reference to the document. If *document* already exists on the disk, it is overwritten. The document is opened for writing.

If *document* is an empty string ("""), a Macintosh create-file dialog box is presented, and the user may specify a new document name. See Figure 16-7.

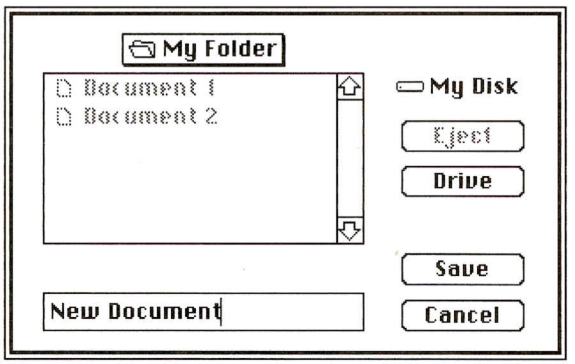


Figure 16-7
The create-file dialog box

A document type may be specified with *type*, a 4-character string. The most common document type is TEXT. If a document type is not specified, then a TEXT-type document is created.

If the user creates a document, the OK system variable is set to 1 and the Document system variable is set to the name of the opened document. Otherwise, the OK system variable is set to 0.

Page L299→ 23 Oct 1999

Debug

```
TRACE
yy:=Create document("Pat")
x:=document
SEND PACKET(yy,"asas")
CLOSE DOCUMENT(yy)
DELETE DOCUMENT(x)
```

yy : 00:50:10 Doc Ref Page 299
document : Pat
x : Pat

Abort | No Trace | Step | View | Edit

Debug

```
TRACE
yy:=Create document("Pat")
x:=document
SEND PACKET(yy,"asas")
CLOSE DOCUMENT(yy)
DELETE DOCUMENT(document)
```

yy : 00:50:10 Doc Ref Page 299
document : Pat
x : Pat

Abort | No Trace | Step | View | Edit

document "path name"

The following example creates and opens a new document called Note, writes the string "Hello" into it, and closes the document.

```
Doc := Create document ("Note")          ` Create a new document called Note
SEND PACKET (Doc; "Hello")               ` Write one word into the document
CLOSE DOCUMENT (Doc)                     ` Close the document
```

Open document
Append document

Doc Ref page 299

Open document (*document*; *{type}*) → Docref

Append document (*document*; *{type}*) → Docref

Parameter	Type	Description
document	String	Document name
type	String	Document type (4 characters)

Open document opens *document*, an existing Macintosh document, for reading or writing. Data written to the document is written at the beginning of the document and overwrites any existing data.

Append document opens *document*, an existing Macintosh document, for writing. Data written to the document is appended to the end of the document.

With both commands, if *document* is an empty string ("""), a Macintosh open-file dialog box is presented, and the user may specify the document name. See Figure 16-8.

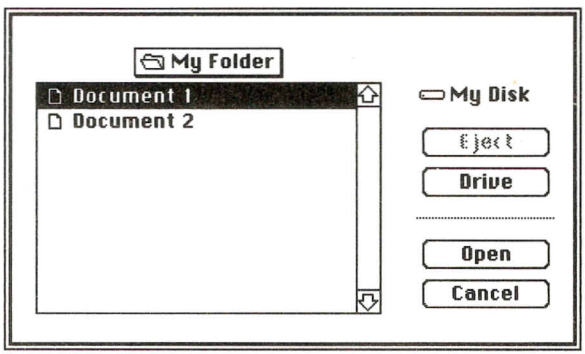


Figure 16-8
The open-file dialog box

If you use an empty string for *document* (that is, if the open-file dialog box is presented to the user), you may specify the document type with *type*, a 4-character string. The open-file dialog box then displays only files of that type. The most common document type is TEXT. If you do not specify *type*, documents of all types can be opened.

If the user opens a document, the OK system variable is set to 1 and the Document system variable is set to the name of the opened document. Otherwise, the OK system variable is set to 0.

The following example opens an existing document called Note, writes the string "Goodbye" into it, and closes the document. If the document already contained the string "Hello", the string would be overwritten.

```
Doc := Open document ("Note")          ` Open a document called Note
SEND PACKET (Doc; "Goodbye")           ` Write one word into the document
CLOSE DOCUMENT (Doc)                   ` Close the document
```

The following example opens an existing document called Note, appends the string " and so long" and a carriage return onto the end of the document, and closes the document. If the document already contained the string "Goodbye", the document would now contain the string "Goodbye and so long", followed by a carriage return.

```
Doc := Append document ("Note")        ` Create a new document called Note
SEND PACKET (Doc; " and so long" + Char (13))  ` Append a string onto the document
CLOSE DOCUMENT (Doc)                   ` Close the document
```

CLOSE DOCUMENT

CLOSE DOCUMENT (*document ref*) Page 288

Parameter	Type	Description
document ref	Docref	Document reference

CLOSE DOCUMENT closes the document specified by *document ref*.

Closing a document is the only way to ensure that the data written to a file is saved. You must close all documents to ensure that they are properly saved.

The following example lets the user create a new document, writes the string "Hello" into it, and closes the document.

```
Doc := Create document ("")            ` Create a new document called Note
SEND PACKET (Doc; "Hello")             ` Write one word into the document
CLOSE DOCUMENT (Doc)                   ` Close the document
```

Note

Doc contains Document reference
Docref

DELETE DOCUMENT

DELETE DOCUMENT (*document*)

Parameter	Type	Description
document	String	Document to delete

DELETE DOCUMENT deletes *document*. Deleting a document sets the OK system variable to 1. If DELETE DOCUMENT can't delete the document, the OK system variable is set to 0. DELETE DOCUMENT does not work on open documents.

DELETE DOCUMENT doesn't accept an empty string argument for *document*. If an empty string is used, the open-file dialog box is not displayed and an error is generated.

Warning: DELETE DOCUMENT can delete *any* file on a disk. This includes documents created with other applications as well as the applications themselves. DELETE DOCUMENT should be used with extreme caution. Deleting a document is a permanent operation and cannot be undone.

The following example deletes the document named Note.

DELETE DOCUMENT ("Note")	Delete the document

SEND PACKET

SEND PACKET ({*document ref*}; *packet*)

Parameter	Type	Description
document ref	Docref	Document reference
packet	String	Packet to send

SEND PACKET sends *packet* to the serial port or to a document. If *document ref* is specified, the packet is written to the Macintosh document referenced by *document ref*. If *document ref* is not specified, the packet is written to the serial port or document previously opened by the SET CHANNEL command.

Before you use SEND PACKET, you must open a serial port with SET CHANNEL, or a document with one of the document commands.

When writing to a document, the first SEND PACKET begins writing at the beginning of the document unless the document was opened with Append document. Until the document is closed, each subsequent packet is appended to any previously sent packets.

Important: SEND PACKET writes Macintosh ASCII data. Macintosh ASCII uses 8 bits. Standard ASCII uses only the lower 7 bits. Many devices do not use the 8th bit in the same way as does the Macintosh. Such devices include computers that use PC-DOS or MS-DOS, and the ImageWriter printer. If the string to be sent contains data that uses the 8th bit, be sure to create an ASCII map to translate the ASCII characters, and execute USE ASCII MAP before using SEND PACKET.

The following example writes data from fields to a document. It writes the fields as fixed-length fields. Fixed-length fields are always of a specific length. If a field is shorter than the specified length, the field is padded with spaces. (That is, spaces are added to make up the specified length.) Although the use of fixed-length fields is an inefficient method of storing data, some computer systems and applications still use them.

```
DEFAULT FILE ([People])                          ` Set the default file
Doc := Create document ("")                      ` Create a TEXT document
If (OK = 1)                                      ` If the user opened a new document
  For ($i; 1; Records in selection)              ` Loop once for each record
    ` Send a packet. Create the packet from
    ` a string of 15 spaces and the first name field.
    SEND PACKET (Doc; Change string ("           "; [People]First; 1))
    ` Send a second packet. Create the packet from
    ` a string of 15 spaces and the first name field.
    ` This could be in the first SEND PACKET,
    ` but is separated for clarity.
    SEND PACKET (Doc; Change string ("           "; [People]Last; 1))
    NEXT RECORD
  End for
  ` Send a Control-z (SUB) which is used
  ` as an end-of-file marker for some computers.
  SEND PACKET (Doc; Char (26))
  CLOSE DOCUMENT (Doc)                           ` Close the document
End if
```

RECEIVE PACKET

RECEIVE PACKET ({*document ref*}; *receive var*; *number of char*)

Parameter	Type	Description
document ref	Docref	Document reference
receive var	Variable	Variable to receive data
number of char	Number	Number of characters to receive

RECEIVE PACKET ({*document ref*}; *receive var*; *stop char*)

Parameter	Type	Description
document ref	Docref	Document reference
receive var	Variable	Variable to receive data
stop char	String	Character at which to stop receiving

RECEIVE PACKET has two forms.

The first form specifies the number of characters (*number of char*) to receive. RECEIVE PACKET transfers the number of characters specified into *receive var*.

The second form reads data until a specified character (*stop char*) is read. RECEIVE PACKET transfers characters into *receive var* until it encounters the first *stop char*. RECEIVE PACKET skips the *stop char* and does not return it in *receive var*.

With either form, RECEIVE PACKET reads data from the serial port or a document. Before using RECEIVE PACKET, you must open a serial port or document with SET CHANNEL, or open a document with one of the document commands. If *document ref* is specified, the data is read from a Macintosh document. If *document ref* is not specified, the data is read from the serial port or document opened by the SET CHANNEL command.

When reading a document, RECEIVE PACKET begins reading at the beginning of the document. The reading of each subsequent packet begins at the character following the last character read.

The OK system variable will be set to 1 if the packet is received without error.

When attempting to read past the end of a file, RECEIVE PACKET will return with the data read up to that point. The Error system variable will be set to 0 if this or any other error occurs.

If RECEIVE PACKET is reading from a serial port, the user can interrupt RECEIVE PACKET by pressing Option-Space unless an ON ERR CALL procedure has been installed. RECEIVE PACKET can also be interrupted by the SET TIMEOUT command. In either case, the OK system variable is set to 0 if the RECEIVE PACKET command is canceled.

The following example reads 20 characters from the serial port into the variable Get Twenty.

RECEIVE PACKET (Get Twenty; 20)

The following example reads data from the document referenced by the variable My Doc into the variable vData. It reads until it encounters a carriage return (Char (13)).

Return L385

RECEIVE PACKET (My Doc; vData; **Char** (13))

The following example reads data from a document into fields. The data is stored as fixed-length fields. The procedure calls a subroutine to strip any trailing spaces (spaces at the end of the string). The subroutine follows the procedure.

```
DEFAULT FILE ([People])                      ` Set the default file
Doc := Open document ("";"TEXT")             ` Open a TEXT document
If (OK = 1)                                  ` If the document was opened...
  While (OK = 1)                             ` Loop until no more data
    RECEIVE PACKET (Doc; $Var1; 15)          ` Read 15 characters
    RECEIVE PACKET (Doc; $Var2; 15)          ` Do the same as above for the second field
    If (OK = 1)                              ` If we are not at the end of the document...
      CREATE RECORD                          ` Create a new record
      [People]First := Strip ($Var1)         ` Save the first name
      [People]Last := Strip ($Var2)          ` Save the last name
      SAVE RECORD                            ` Save the record
    End if
  End while
  CLOSE DOCUMENT (Doc)                       ` Close the document
End if
```

OK L383

The spaces at the end of the data are stripped by the following subroutine, called *Strip*:

```
For ($i; Length ($1); 1; –1)                 ` Loop from end of string to beginning
  If ($1≤$i≥ # " ")                          ` If it is not a space...
    $i := –$i                                ` Force the loop to end
  End if
End for
$0 := Delete string ($1; –$i; Length ($1))   ` Delete the spaces
```

SET CHANNEL

SET CHANNEL (*port*; *setup*)

Parameter	Type	Description
port	Number	Port to use
setup	Number	Port setup

SET CHANNEL (*operation*; {*document*})

Parameter	Type	Description
operation	Number	File operation code
document	String	Document on which to perform operation

The SET CHANNEL command has two forms. The first form opens a serial port. The second form opens a document. You can open only one serial port or one document at a time with this command.

1st

The first form of the SET CHANNEL command opens a serial port, setting the protocol and other port information. Data can be sent with SEND PACKET, SEND RECORD, or SEND VARIABLE, and received with RECEIVE BUFFER, RECEIVE PACKET, RECEIVE RECORD, or RECEIVE VARIABLE.

SET CHANNEL opens one serial port. The serial port is the one used by ON SERIAL PORT CALL and all subsequent serial operations.

The first parameter, *port*, selects the port and the protocol. You determine the value for *port* by adding together the port and protocol as listed in Table 16-12. For example, to use XON/XOFF with the modem port, you would add 1 + 20 = 21. You would then use 21 as the value of the *port* parameter.

Table 16-12
Values for the *port* parameter

Object	*Port*	Setting
Port	0	Printer
	1	Modem
Protocol	0	None
	20	XON/XOFF
	30	DTR

2nd The second parameter, *setup*, sets the speed, number of data bits, number of stop bits, and parity. You determine the value for *setup* by adding together the speed, data bits, stop bits, and parity as listed in Table 16-13. For example, to set 1200 baud, 8 data bits, 1 stop bit, and no parity, you would add 94 + 3072 + 16384 + 8192 = 27742. You would then use 27742 as the value of the *setup* parameter.

Table 16-13
Values for the *setup* parameter

Control	*Setup*	Setting
Speed (in baud)	380	300
	189	600
	94	1200
	62	1800
	46	2400
	30	3600
	22	4800
	14	7200
	10	9600
	4	19200
	0	57000
Data bits	0	5
	1024	6
	2048	7
	3072	8
Stop bits	16384	1
	–32768	1.5
	–16384	2
Parity	8192	None
	4096	Odd
	12288	Even

The second form of the SET CHANNEL command allows you to create, open, and close a document. Unlike the document commands, it can open only one document at a time. The document can be read from or written to.

The first parameter, *operation,* specifies the operation to be performed on the document specified with *document*. Table 16-14 lists the values of *operation* and the resulting operation with different values for *document*. The first column lists the allowed values for the operation parameter. The second column lists the allowed values for the document parameter. The third column lists the resulting operation. For example, to display an open-file dialog box to open a text file, you would use the following line:

```
SET CHANNEL (13; "")
```

Table 16-14
Values for the *operation* and *document* parameters

Operation	*Document*	**Result**
10	String	Opens the document specified by String. If the document doesn't exist, the document is opened and created.
10	"" (empty string)	Displays the open-file dialog box to open a file. All file types are displayed.
11	none	Closes an open file.
12	"" (empty string)	Displays the create-file dialog box to create a new file.
13	"" (empty string)	Displays the open-file dialog box to open a file. Only text file types are displayed.

All of the operations in Table 16-14 set the Document system variable if appropriate. They also set the OK system variable to 1 if the operation was successful. Otherwise, the OK system variable is set to 0.

You should use the document commands for all normal document operations. You should use SET CHANNEL for documents only when you need to use one of these commands: SEND RECORD, RECEIVE RECORD, SEND VARIABLE, or RECEIVE VARIABLE. The document commands do not operate with these commands.

The following example opens a printer port connected to an ImageWriter II.

```
SET CHANNEL (0; 10 + 3072 + 16384 + 8192)        ` Printer port, ImageWriter
```

See the example for SEND RECORD, later in this section, for an example of the second form of SET CHANNEL.

ON SERIAL PORT CALL

ON SERIAL PORT CALL (*serial procedure*)

Parameter	Type	Description
serial procedure	String	Procedure to call

ON SERIAL PORT CALL installs *serial procedure* as an interrupt procedure for managing serial port events. The interrupt procedure is automatically called by 4th DIMENSION when a character enters the serial port buffer. Giving an empty string for *serial procedure* turns off serial port event handling.

4th DIMENSION suspends the operation that is running when port activity occurs, and does not return to the operation until it has executed the interrupt procedure. 4th DIMENSION will call the interrupt procedure in the User or Runtime environment any time a character enters the serial port.

4th DIMENSION automatically calls the interrupt procedure when the serial port buffer contains one or more characters. If you decide to do nothing with the buffer contents, don't forget to clear the buffer contents by calling RECEIVE BUFFER. If you don't clear the buffer, 4th DIMENSION calls your installed procedure again.

The following line installs an interrupt procedure called *Interruption*:

```
ON SERIAL PORT CALL ("Interruption")
```

The *Interruption* procedure takes whatever is in the serial buffer and concatenates it onto a variable called Got It. The variable, Got It, can then be read later by other parts of the application. Here is the *Interruption* procedure:

```
RECEIVE BUFFER (v)                    ` Read the serial port buffer
Got It := Got It + v                  ` Save the data
```

The following line removes the interrupt procedure:

```
ON SERIAL PORT CALL ("")
```

SET TIMEOUT

SET TIMEOUT (*seconds*)

Parameter	Type	Description
seconds	Number	Seconds until the timeout

SET TIMEOUT specifies how much time a serial port command has in which to complete. If the serial port command does not complete within the specified time, *seconds,* the serial port command is canceled, and the OK system variable is set to 0. Note that the time is the total time allowed for the command to execute, not the time between characters received.

To cancel a previous setting and stop monitoring serial port communication, use a setting of 0 for *seconds*.

Table 16-15 lists the commands that are monitored.

Table 16-15
Commands monitored by SET TIMEOUT

Command	Command
RECEIVE PACKET	SEND PACKET
RECEIVE RECORD	SEND RECORD
RECEIVE VARIABLE	SEND VARIABLE

The following example sets the serial port to receive data. It then sets a timeout. The data is read with RECEIVE PACKET. If the data is not received in time, an error occurs.

```
` Set for: Modem; 9600 baud; 8 data bits; 1 stop bit
SET CHANNEL (1; 10 + 8192 + 3072 + 16384)
SET TIMEOUT (10)                          ` Set the timeout for 10 seconds
RECEIVE PACKET (v; Char (13))             ` Read until a carriage return
If (OK = 0)                               ` If there was an error...
  Alert ("Error receiving data.")         ` tell the user
Else                                      ` Otherwise...
  [People]Name := v                       ` save the data
End if
```

26" July 1992
Goff Application
"Buffer" Reception H11

RECEIVE BUFFER

RECEIVE BUFFER (*receive var*)

Parameter	Type	Description
receive var	Variable	Variable to receive data

RECEIVE BUFFER reads the serial port that was previously initialized with SET CHANNEL. The serial port has a buffer that fills with characters until a command reads from the buffer. RECEIVE BUFFER puts the characters in the buffer into *receive var* and clears the buffer. If there are no characters in the buffer, then *receive var* will contain nothing.

The Macintosh serial port buffer is 64 characters in size. This means that the buffer can hold 64 characters before it overflows. When it is full and new characters are received, the new characters replace the oldest characters. The old characters are lost; therefore it is essential that the buffer is read quickly when new characters are received.

RECEIVE BUFFER is often used in a procedure installed by ON SERIAL PORT CALL. When the procedure installed by ON SERIAL PORT CALL is called, RECEIVE BUFFER is used to read whatever is in the serial port buffer.

RECEIVE BUFFER is different from RECEIVE PACKET in that it takes whatever is in the buffer and then immediately returns. RECEIVE PACKET, on the other hand, waits until either it finds a specific character or a certain number of characters are in the buffer.

See the example for ON SERIAL PORT CALL, earlier in this section.

SEND RECORD

SEND RECORD ({*file*})

Parameter	Type	Description
file	File	File from which to send the current record

SEND RECORD sends the current record of *file* to the serial port or document opened by the SET CHANNEL command. The record is sent with a special internal format that can be read only by RECEIVE RECORD. If no current record exists, SEND RECORD has no effect.

The complete record is sent. This means that all subrecords and pictures that are associated with the record are also sent.

Caution: When records are being sent and received, the sending file's structure and the receiving file's structure must be the same. This means the files must have same number of fields, type of fields, and order of fields.

The following example sends all the records in a file.

```
SET CHANNEL (10; "Archive")                  ` Open a file
ALL RECORDS ([My File])                      ` Select all the records
For ($i ; 1; Records in file ([My File]))    ` Loop through each record
  SEND RECORD ([My File])                    ` Send the record
  NEXT RECORD ([My File])                    ` Move to the next record
End for
SET CHANNEL (11)                             ` Close the file
```

RECEIVE RECORD

RECEIVE RECORD (*{file}*)

Parameter	Type	Description
file	File	File into which to receive the record

RECEIVE RECORD receives a record into *file* from the serial port or document opened by the SET CHANNEL command. The record must be sent with SEND RECORD. Before you execute RECEIVE RECORD, you must create a new record for *file* with the CREATE RECORD command. If the record is received correctly, you must then use SAVE RECORD to save the new record.

The complete record is received. This means that all subrecords and pictures that were sent with the record are also received.

During the execution of RECEIVE RECORD, the user can interrupt by pressing Option-Space unless an ON ERR CALL procedure has been installed. To check for interrupts, you can test the OK system variable. The OK system variable is set to 1 if the record is received. Otherwise, the OK system variable is set to 0.

Caution: When records are being sent and received, the sending file's structure and the receiving file's structure must be the same. This means the files must have the same number of fields, type of fields, and order of fields.

The following example receives all the records in a document.

```
SET CHANNEL (10; "Archive")                  ` Open a file
CREATE RECORD ([My File])                    ` Create a new record
RECEIVE RECORD ([My File])                   ` Receive the record
While (OK = 1)                               ` Receive the records
  SAVE RECORD ([My File])                    ` Save the last received record
  CREATE RECORD ([My File])                  ` Create a new record
  RECEIVE RECORD ([My File])                 ` Receive the record
End while
SET CHANNEL (11)                             ` Close the file
```

SEND VARIABLE

SEND VARIABLE (*variable*)

Parameter	Type	Description
variable	Variable	Variable to send

SEND VARIABLE sends *variable* to the document or serial port previously opened by SET CHANNEL. The variable is sent with a special internal format that can be read only by RECEIVE VARIABLE. SEND VARIABLE sends the complete variable (including its type and value). Don't confuse SEND VARIABLE with SAVE VARIABLE.

The following example sends a variable to the serial port.

SET CHANNEL (1; 10 + 8192 + 3072 + 16384)
SEND VARIABLE (My Variable)

RECEIVE VARIABLE

RECEIVE VARIABLE (*variable*)

Parameter	Type	Description
variable	Variable	Variable into which to receive

RECEIVE VARIABLE receives *variable*, a variable sent by SEND VARIABLE, from the document or serial port previously opened by SET CHANNEL. The variable will be created with the correct type. If the variable already exists, it will be overwritten.

During the execution of RECEIVE VARIABLE, the user can interrupt by pressing Option-Space unless an ON ERR CALL procedure has been installed. To check for interrupts, you can test the OK system variable. The OK system variable is set to 1 if the record is received. Otherwise, the OK system variable is set to 0.

The following example receives a variable from the serial port.

SET CHANNEL (1; 10 + 8192 + 3072 + 16384)
SET TIMEOUT (20) ` To force an interrupt if there is an error
RECEIVE VARIABLE (My Variable)

USE ASCII MAP L69s

USE ASCII MAP (*mapname*; *I/O*)

Parameter	Type	Description
mapname	String	Document name of the map to use
I/O	Number	1 for Output map; 2 for Input map

USE ASCII MAP (*; *I/O*)

Parameter	Type	Description
*		Use to reset to default ASCII map
I/O	Number	1 for Output map; 2 for Input map

USE ASCII MAP has two forms. The first form loads the ASCII map named *mapname* from disk and uses that ASCII map. If *I/O* is 1, the map is loaded as the output map. If *I/O* is 2, the map is loaded as the input map.

The ASCII map must have been previously created with the ASCII map dialog box in the User environment. Once an ASCII map is loaded, 4th DIMENSION uses the map during transfer of data between the database and a document or a serial port. Transfer operations include the import and export of text (ASCII), DIF, and SYLK files. An ASCII map also works on data transferred with SEND PACKET, RECEIVE PACKET, and RECEIVE BUFFER. It has no effect on transfers of data done with SEND RECORD, SEND VARIABLE, RECEIVE RECORD, and RECEIVE VARIABLE.

If you give an empty string for *mapname*, USE ASCII MAP displays a standard open-file dialog box so that the user can specify an ASCII map document. Whenever you execute USE ASCII MAP, the OK system variable is set to 1 if the map is successfully loaded, and to 0 if it is not.

The second form of USE ASCII MAP, with the asterisk (*) parameter instead of *mapname*, restores the default ASCII map. If *I/O* is 1, the map is reset for output. If *I/O* is 2, the map is reset for input. The default ASCII map has no translation between characters.

The following example loads a special ASCII map from disk. It then exports data. Finally, the default ASCII map is restored.

```
USE ASCII MAP ("My Chars"; 1)          ` Load and use an alternative ASCII map
EXPORT TEXT ([My File]; "My Text")     ` Export data through the map
USE ASCII MAP (*; 1)                   ` Restore the default map
```

Managing Access Privileges

EDIT ACCESS	CHANGE PASSWORD
CHANGE ACCESS	Current user

The commands in this section let you change passwords and change access privileges for the database. See Chapter 8 in the *4th DIMENSION Design Reference* for more information on the password access system.

EDIT ACCESS

EDIT ACCESS

EDIT ACCESS allows the user to edit the password system. The Password Access editor from the Design environment is used to edit the access.

Groups can be edited by the Designer and the Administrator, and by group owners. The Designer and the Administrator can edit any group. Group owners can edit only the groups they own. Users can be put into and removed from the groups.

The Designer and Administrator can add new users as well as assign them to groups.

The following example displays the Password Access editor to the user.

EDIT ACCESS

CHANGE ACCESS

CHANGE ACCESS allows the user to change to a different access level without leaving the database. The same password dialog box that the user entered the database through is presented and the user can enter as a different user. If the user clicks Cancel, the user enters as the Guest.

The following example displays the password dialog box to the user.

CHANGE ACCESS

CHANGE PASSWORD

CHANGE PASSWORD (*password*)

Parameter	Type	Description
password	String	New password

CHANGE PASSWORD changes the password of the current user. This command replaces the current password with the new password, *password*.

See the example for Current user, next.

Current user

Current user → String

Current user returns the user name of the current user. If the user enters as the Guest, Current user returns an empty string.

The following example allows the user to change his or her password. It first presents the password dialog with the CHANGE ACCESS command. This forces the user to select his or her user name and enter the password. If the user enters as other than a guest, a request dialog allows them to change the password.

```
CHANGE ACCESS                                  ` Present user with the password dialog
  If the user cancelled or entered as Guest, the current user = ""
If (Current user # "")
  $pw1 := Request ("New password:")            ` Ask for the new password
  If (OK = 1)                                  ` If the user clicked OK
    $pw2 := Request ("Enter password again:")  ` Confirm the new password
    If ((OK = 1) & ($pw1 = $pw2))              ` If user clicked OK & validated password
      CHANGE PASSWORD ($pw1)                   ` Set the new password
    End if
  End if
End if
```

Determining the Database Structure

Count files	Filename	File
Count fields	Fieldname	Field
FIELD ATTRIBUTES		

The commands in this section return a complete description of the structure of a database. They return the number of files and number of fields in each file, the names of the files and fields, and the field types and attributes.

Determining the database structure is extremely useful when you are developing and using modules of procedures and layouts that can be copied into different databases. The ability to read the database structure allows you to develop and use portable code.

Storing the Database Structure in Arrays

It is often very useful to have the filenames and field names in arrays. Arrays allow you to quickly access the names without having to read each one individually. Using arrays, you can immediately change a scrollable area to display the current file's fields, by simply copying the field array to the displayed array.

The following code shows you how to create the arrays:

```
` The string below is used to hold 1 character for each possible field type
$TList := "ARTPD B*IL H"
` The array, Files, will contain the names of the files
ARRAY STRING (15; Files; Count files)
` The 2-dimensional array, Fields, will contain the names of the fields for all files
` It begins with zero elements for each file
ARRAY STRING (15; Fields; Count files; 0)
` The 2-dimensional array, Types, will contain the types of the fields
ARRAY STRING (1; Types; Count files; 0)
For ($i; 1; Count files)                          ` Loop for each file
  Files{$i} := Filename ($i)                      ` Get the filenames
  ` Resize the the arrays to the number of fields in the file
  ARRAY STRING (15; Fields{$i}; Count fields ($i))
  ARRAY STRING (1; Types{$i}; Count fields ($i))
  For ($j; 1; Count fields ($i))                  ` Loop for each field
    Fields{$i}{$j} := Fieldname ($i; $j)          ` Get the field name
    FIELD ATTRIBUTES ($i; $j; $x; $y; $z)         ` Get the field attributes
    Types{$i}{$j} := $TList≤$x + 1≥               ` Save only the field type
  End for
End for
```

Using this code, you could immediately refer to the name of any file or field. For example, the second filename is returned by this expression:

```
Files {2}
```

And the name of the third field of the second file is returned by this expression:

```
Fields {2}{3}
```

You can refer to files and fields through reference numbers. Thus, you can create code that is portable, and refer to the structure of a database without knowing the file and field names. Be careful to check that you are operating on the correct field type before performing an operation on the field. For example, if you are performing a search, restrict the field types to those other than Subfile and Picture.

You can reference files and fields through pointers. You use the File and Field commands to access the pointers. For example, if Customers was field number 2 in file number 1, you could assign a new value to the field with the following statement:

```
Field (1; 2)» := "Acme, Co."
```

File and field pointers are transient and may not be the same each time the database is opened. Conversely, assuming the database structure has not changed, file numbers and field numbers do stay the same.

Count files

Count files → Number

Count files returns the number of files in the database. Files are numbered in the order in which they are created.

The following example sets the NumFiles variable to the number of files in the database.

NumFiles := **Count files**

Count fields

Count fields (*file number*) → Number

Parameter	Type	Description
file number	Number	File number

Count fields (*file pointer*) → Number

Parameter	Type	Description
file pointer	Pointer	Pointer to a file

Count fields has two forms.

Count fields returns the number of fields in the file specified by *file number* or *file pointer*. Fields are numbered in the order in which they are created.

The following example sets the NumFields variable to the number of fields in the third file.

NumFields := **Count fields** (3)

The following example sets the NumFields variable to the number of fields in the file named [File1].

NumFields := **Count fields** (»[File1])

Filename

Filename (*file number*) ➛ String

Parameter	Type	Description
file number	Number	File number

Filename (*file pointer*) ➛ String

Parameter	Type	Description
file pointer	Pointer	Pointer to a file

Filename has two forms.

Filename returns the name of the file that corresponds to *file number* or *file pointer*.

The following example sets the first element of the array, FileArray, to the name of the first file.

FileArray{1} := **Filename** (1)

The following example sets the first element of the array, FileArray, to the name of file, [MyFile]. This is useful because if you change the name of [MyFile], the new name will be returned.

FileArray{1} := **Filename** (»[MyFile])

Fieldname

Fieldname (*file number*; *field number*) ➛ String

Parameter	Type	Description
file number	Number	File number
field number	Number	Field number

Fieldname (*field pointer*) ➛ String

Parameter	Type	Description
field pointer	Pointer	Pointer to a field

Fieldname has two forms.

Fieldname returns the name of the field that corresponds to *file number* and *field number*, or to *field pointer*.

The following example sets the second element of the array, FieldArray{1}, to the name of the second field in the first file.

FieldArray{1}{2} := **Fieldname** (1; 2)

The following example sets the second element of the array, FieldArray{1}, to the name of the field, [MyFile]MyField. This is useful because if you change the name of [MyFile]MyField, the new name will be returned.

```
FieldArray{1}{2} := Fieldname (»[MyFile]MyField)
```

File

File (*file number*) → Pointer

Parameter	Type	Description
file number	Number	File number

File (*file pointer*) → Number

Parameter	Type	Description
file pointer	Pointer	Pointer to a file

File (*field pointer*) → Number

Parameter	Type	Description
field pointer	Pointer	Pointer to a field

File has three different forms.

If passed *file number*, File returns a pointer to the file.

If passed *file pointer*, File returns the file number of the file.

If passed *field pointer*, File returns the file number of the field. This form is used with the second form of Field to get the file number and field number of a field by using only a field pointer.

The following example sets the FilePtr variable to a pointer to the third file.

```
FilePtr := File (3)
```

Passing FilePtr to the second form of File returns the number 3. For example, the following line sets FileNum to 3:

```
FileNum := File (FilePtr)
```

The following example sets the FileNum variable to the file number of [File3].

```
FileNum := File (»[File3])
```

The following example sets the FileNum variable to the file number of the file to which the field named [File3]Field1 belongs.

```
FileNum := File (»[File3]Field1)
```

Field

Field (*file number*; *field number*) ➛ Pointer

Parameter	Type	Description
file number	Number	File number
field number	Number	Field number

Field (*field pointer*) ➛ Number

Parameter	Type	Description
field pointer	Pointer	Pointer to a field

Field has two different forms.

If passed *file number* and *field number*, Field returns a pointer to the field.

If passed *field pointer*, Field returns the field number of the field.

The following example sets the FieldPtr variable to a pointer to the second field in the third file.

```
FieldPtr := Field (3; 2)
```

Using the second form of the Field command on FieldPtr returns the number 2. For example, the following line sets FieldNum to 2:

```
FieldNum := Field (FieldPtr)
```

The following example sets the FieldNum variable to the field number of [File3]Field2.

```
FieldNum := Field (»[File3]Field2)
```

FIELD ATTRIBUTES

FIELD ATTRIBUTES (*file number*; *field number*; *type*; *{length}*; *{index}*)

Parameter	Type	Description
file number	Number	File number
field number	Number	Field number
type	Variable (number data type)	Field type
length	Variable (number data type)	Alpha field length
index	Variable (Boolean data type)	FALSE if not indexed; TRUE if indexed

FIELD ATTRIBUTES (*field pointer*; *type*; *{length}*; *{index}*)

Parameter	Type	Description
field pointer	Pointer	Pointer to a field
type	Variable (number data type)	Field type
length	Variable (number data type)	Alpha field length
index	Variable (Boolean data type)	FALSE if not indexed; TRUE if indexed

FIELD ATTRIBUTES has two different forms.

FIELD ATTRIBUTES assigns to the variables *type*, *length*, and *index* information about the field specified by *file number* and *field number*, or *field pointer*.

The *type* parameter is set to a numeric value for one of the ten field types shown in Table 16-16.

Table 16-16
Field types and their numbers

Field Type	Number
Alpha	0
Real	1
Text	2
Picture	3
Date	4
Boolean	6
Subfile	7
Integer	8
Longint	9
Time	11

Information returned in *length* is meaningful only for Alpha fields. It is set to the defined length of the field.

Information returned in *index* is meaningful only for Alpha, Integer, Long Integer, Real, Date, Time, and Boolean fields. It is set to FALSE if there is no index for the field, and TRUE if there is an index for the field.

The following example sets the variables vType, vLength, and vIndex to the attributes for the third field of the first file.

FIELD ATTRIBUTES (1; 3; vType; vLength; vIndex)

The following example sets the variables vType, vLength, and vIndex to the attributes for the field named [File3]Field2.

FIELD ATTRIBUTES (»[File3]Field2; vType; vLength; vIndex)

Controlling Data Flushing

FLUSH BUFFERS

The command in this section flushes the data buffers.

FLUSH BUFFERS

FLUSH BUFFERS

Executing FLUSH BUFFERS immediately saves the data buffers to disk. All changes that have been made to the database are stored on disk. This command is ignored in multi-user environments since data is not cached while in a multi-user environment. The preference setting in the Design environment that specifies how often to save is normally used to control buffer flushing.

FUNCTIONS

CHAPTER 17

FUNCTIONS

The functions in this chapter perform date, time, string, and numeric operations.

String Functions

Length	Insert string	Uppercase
Substring	Delete string	String
Position	Replace string	Ascii
Change string	Lowercase	Char

This section describes functions that work on strings. It also describes the character reference symbols. None of the string functions alters the string expressions used as parameters.

Character Reference Symbols

Tutorial 91

string ≤ *position* ≥ → String (1 character)

Parameter	Type	Description
string	String	String whose character to return
position	Number	Position of character to return

The character reference symbols (≤...≥) are used to refer to a single character within *string*. The position of the character, *position,* is specified between the character reference symbols.

The character at *position* is returned.

If the character reference symbols appear on the left side of the assignment operator, a character is assigned to the referenced position in the string. For example, the following line sets the first character of Name to uppercase:

Name≤1≥ := **Uppercase** (Name≤1≥)

If you refer to characters that are beyond the length of the string, the results are undefined.

The following lines show the use of the character reference symbols.

```
Result := "abcd"≤3≥                  ` Result gets "c"
Result := Last Name≤1≥               ` Result gets first character in Last Name
Result := City≤$i≥                   ` Result gets the $i'th character in City
```

The following subroutine is a function that capitalizes the first letter of each word in a string and returns the resulting string.

```
$0 := $1                                  ` Copy the string to the returned value
$0≤1≥ := Uppercase ($0≤1≥)                ` Always capitalize the first letter
For ($i; 1; Length ($0) – 1)              ` Loop for all characters except the first
  If ($0≤$i≥ < "a") | ($0≤$i≥ > "z")      ` If the character is not a letter...
    $0≤$i+1≥ := Uppercase ($0≤$i + 1≥)    ` Uppercase the next letter
  End if
End for
```

Length

Length (*string*) ➛ Number

Parameter	Type	Description
string	String	String whose length to return

Length is used to find the length of *string*. Length returns the number of characters that are in the string.

The following example illustrates the use of Length. The results are assigned to the variable Result. The comments describe what Result is set to.

```
Result := Length ("Topaz")                ` Result gets 5
Result := Length ("Citizen")              ` Result gets 7
```

Substring

Substring (*source*; *first char*; {*number of chars*}) ➛ String

Parameter	Type	Description
source	String	String from which to get substring
first char	Number	Position of first character
number of chars	Number	Number of characters to get

Substring returns the portion of *source* defined by *first char* and *number of chars*. The *first char* parameter points to the first character in the string to return, and *number of chars* specifies how many characters to return.

If the sum of *first char* and *number of chars* exceeds 32,767, the results are undefined.

If *first char* plus *number of chars* is greater than the number of characters in the string, or if *number of chars* is not specified, Substring returns the last character(s) in the string, starting with the character specified by *first char*. If *first char* is greater than the number of characters in the string, Substring returns an empty string ("").

The following example illustrates the use of Substring. The results are assigned to the variable Result. The comments describe what Result is set to.

```
Result := Substring ("08/04/62"; 4; 2)        ` Result gets "04"
Result := Substring ("Emergency"; 1; 6)       ` Result gets "Emerge"
Result := Substring (var; 2)                  ` Result gets all characters except the first
```

Position

Position (*find*; *string*) → Number

Parameter	Type	Description
find	String	String to find
string	String	String in which to search

Position returns the position of the first occurrence of *find* in *string*.

If Position fails to find the string *find*, it returns a zero (0). If Position finds an occurrence of *find*, it returns the position of the first character of the occurrence in *string*. If you ask for the position of an empty string within an empty string, Position returns one (1).

The following example illustrates the use of Position. The results are assigned to the variable Result. The comments describe what Result is set to.

```
Result := Position ("ll"; "Willow")           ` Result gets 3
Result := Position (var1; var2)               ` Returns the first occurrence of var1 in var2
```

Change string

Change string (*source*; *what*; *where*) → String

Parameter	Type	Description
source	String	Original string
what	String	New characters
where	Number	Where to start the changes

Change string changes a group of characters in *source* and returns the resulting string. Change string overlays *source*, with the characters in *what*, at the character described by *where*.

If *what* is an empty string ("""), Change string returns *source* unchanged. Change string always returns a string of the same length as *source*.

Change string is different from Insert string in that it overwrites characters instead of inserting them.

The following example illustrates the use of Change string. The results are assigned to the variable Result. The comments describe what Result is set to.

```
Result := Change string ("Macintosh SE"; "II"; 11)    ` Result gets "Macintosh II"
Result := Change string ("Acme"; "CME"; 2)            ` Result gets "ACME"
Result := Change string ("November";"Dec"; 1)         ` Result gets "December"
```

also L303

Insert string

Insert string (*source*; *what*; *where*) → String

Parameter	Type	Description
source	String	String into which to insert
what	String	String to insert
where	Number	Where to insert

Insert string inserts a string into *source* and returns the resulting string. Insert string inserts the string *what* before the character described by *where*.

If *what* is an empty string ("""), Insert string returns *source* unchanged.

If *where* is greater than the length of *source*, then *what* is appended to *source*. If *where* is less than zero (0), then *what* is inserted in front of *source*.

Insert string is different from Change string in that it inserts characters instead of overwriting them.

The following example illustrates the use of Insert string. The results are assigned to the variable Result. The comments describe what Result is set to.

```
Result := Insert string ("The tree"; " green"; 4)    ` Result gets "The green tree"
Result := Insert string ("Shut"; "o"; 3)             ` Result gets "Shout"
Result := Insert string ("Indention"; "ta"; 6)       ` Result gets "Indentation"
```

Delete string

Delete string (*source*; *where*; *number of chars*) → String

Parameter	Type	Description
source	String	String from which to delete
where	Number	First character to delete
number of chars	Number	Number of characters to delete

Delete string deletes *number of chars* from *source*, starting at *where*, and returns the resulting string. Delete string does not modify *source*.

Delete string returns the same string as *source* in a number of cases:

- if *source* is an empty string
- if *where* is zero (0) or less than zero
- if *where* is greater than the length of *source*
- if *number of chars* is zero (0)

If *where* plus *number of chars* is equal to or greater than the length of *source*, the characters are deleted to the end of *source*.

If *number of chars* is negative, the characters that would have been deleted are inserted.

The following example illustrates the use of Delete string. The results are assigned to the variable Result. The comments describe what Result is set to.

```
Result := Delete string ("Lamborghini"; 6; 5)      ` Result gets "Lambo"
Result := Delete string (var; 3; 32000)            ` Result gets the first two characters of var
Result := Delete string ("Indentation"; 6; 2)      ` Result gets "Indention"
```

Replace string

Replace string (*source*; *old string*; *new string*; *{how many}*) → String

Parameter	Type	Description
source	String	Original string
old string	String	Character(s) to replace
new string	String	String to replace with
how many	Number	How many times to replace

Replace string replaces every occurrence of *old string* in *source* with *new string*.

If *new string* is an empty string ("""), Replace string deletes each occurrence of *old string* in *source*.

If *how many* is specified, Replace string will replace only the number of occurrences of *old string* specified, starting at the first character of *source*.

If *old string* is an empty string, Replace string returns an empty string.

The following example illustrates the use of Replace string. The results are assigned to the variable Result. The comments describe what Result is set to.

```
Result := Replace string ("Willow"; " ll"; "d")      ` Result gets "Widow"
Result := Replace string ("Shout"; "o "; "")         ` Result gets "Shut"
Result := Replace string (var; Char (9); ",")        ` Replaces all tabs in var with commas
```

Lowercase

Lowercase (*string*) ➛ String

Parameter	Type	Description
string	String	String to convert to lowercase

Lowercase takes *string* and returns the string with all alphabetic characters in lowercase. The original string is not affected. Lowercase affects only the characters *A* through *Z*.

The following example is a function called *Caps,* which capitalizes the first character of the string passed to it. For example, Name := *Caps* ("john") would set Name = "John". The example uses the character reference symbols, ≤ and ≥.

```
` Function: Caps (string)
$0 := Lowercase ($1)
$0≤1≥ := Uppercase ($1≤1≥)
```

Uppercase

Uppercase (*string*) ➛ String

Parameter	Type	Description
string	String	String to convert to uppercase

Uppercase takes *string* and returns the string with all alphabetic characters in uppercase. The original string is not affected. Uppercase affects only the characters *a* through *z*.

See the example for Lowercase, earlier in this section.

String

Num L341

String (*number*; {*format*}) ➛ String

Parameter	Type	Description
number	Number	Number to convert to string
format	String	Format to use for conversion

String (*date*; {*format*}) ➛ String

Date L335

Parameter	Type	Description
date	Date	Date to convert to string
format	Number	Format: 1, 2, 3, or 4

String (*time*; {*format*}) ➛ String

Time L338

Parameter	Type	Description
time	Time	Time to convert to string
format	Number	Format: 1, 2, 3, 4, or 5

String has three forms. Each form takes data that is not a string and returns the data as a string.

The first form of String returns *number* as a string, optionally formatting *number* with *format*. The format is the same as that used to format numbers in layouts. See the *4th DIMENSION Design Reference* for more information on formatting numbers.

The second form of String returns *date* as a string, using the *MM/DD/YY* format. If *format* is specified, the date is formatted according to the formats shown in Table 17-1.

Table 17-1
Format parameters for date strings

Quick Report D135

Format	Name	Example
1	Short	1/16/89
2	Abbreviated	Mon, Jan 16, 1989
3	Long	Monday, January 16, 1989
4	Short 2	01/16/1989

The third form of String returns *time* as a string, using the *HH:MM:SS* format. If *format* is specified, the time is formatted according to the formats shown in Table 17-2.

Creating Style D13

Table 17-2
Format parameters for time strings

Format	Name	Example
1	HH:MM:SS	01:02:03
2	HH:MM	01:02
3	hour min sec	1 hour 2 minutes 3 seconds
4	hour min	1 hour 2 minutes
5	H:MM AM/PM	1:02 AM

The following example returns a number, vNum, converted to a string and formatted with a dollar-style format.

Result := **String** (vNum; "$###,##0.00")

The following example displays an alert box with the current date.

ALERT ("Today's date is " + **String** (**Current date**))

The following example displays an alert box with the current time in AM/PM format.

ALERT ("The time is " + **String** (**Current time**; 5))

Ascii

Ascii (*character*) ➛ Number

Parameter	Type	Description
character	String	Character to return as an ASCII code

Ascii returns the Macintosh ASCII code of *character*.

If there is more than one character in the string, Ascii returns only the code for the first character.

The Char function is the counterpart of Ascii, returning the character that an ASCII code represents. See Appendix D for the Macintosh ASCII codes.

Because uppercase and lowercase characters are treated as equal, you can use Ascii to test for case. For example, this line returns FALSE:

(**Ascii** ("A") = **Ascii** ("a"))

This line, however, returns TRUE:

("A" = "a")

The following example returns the ASCII value of the first character of the string, *A*.

GetAsc := **Ascii** ("ABC") ` GetAsc gets 65

Char

Char (*ASCII code*) → String (1 character)

Parameter	Type	Description
ASCII code	Number	ASCII code from 0 to 255

Char returns the character that *ASCII code* represents.

Char is commonly used to add to a procedure characters that cannot be entered from the keyboard. Table 17-3 lists some of those characters. (The characters for codes 16–20 exist only in the Chicago font.)

Table 17-3
Chicago font special characters

Code	Control Key	ASCII Character
9	Tab	TAB
13	Return	CR
16	p	
17	q	⌘
18	r	✓
19	s	◆
20	t	

The following example uses Char to assign the carriage return character to a variable. The example then sets the default file and displays an alert. The carriage return is used in the alert to display a second line of information.

```
CR := Char (13)                                    ` Set CR to the carriage return character
DEFAULT FILE ([Employees])                         ` Set the default file
ALERT ("Employees: " + String (Records in file) + CR + "Press OK to continue.")
```

Date Functions

Creating Stacks D 157

Current date	Day number	Month of
Date	Day of	Year of

This section describes date functions.

Current date

Current date ➛ Date

Current date returns the current date as kept by the Macintosh system clock.

The following example displays an alert box with the current date in it.

ALERT ("The date is " + **String** (**Current date**))

Date

Design Manual Page XXVI

Note re: DD/MM/YYYY

Date (*date string*) ➛ Date

Parameter	Type	Description
date string	String	String representing the date to be returned

Date evaluates *date string* and returns a date.

The *date string* parameter must follow the normal rules for the format of a date. The date must be in the order *MM/DD/YY* (month, day, year). The month and day may be one or two digits. The year may be two or four digits. If the year is two digits, then Date adds *19* to the beginning of the year. The following characters are valid date separators: slash (/), space, period (.), and hyphen (-).

If *date string* is invalid, the date returned is undefined. Date does not evaluate an alphabetic date like "Jan 1, 1990."

String L 332

The following example prompts the user for a date, using a request box. The string the user enters is converted to a date and stored in the ReqDate variable.

ReqDate := **Date** (**Request** ("Enter date"; **String** (**Current date**)))

While studying Scheduler Procedure CalcCalendar

Day number

Day number (*date*) → Number

Parameter	Type	Description
date	Date	Date for which to return the day number

Day number returns a number representing the weekday on which *date* falls. Table 17-4 lists the day numbers. Day number returns 1 for null dates.

Table 17-4
Day numbers

Day	Number
Sunday	1
Monday	2
Tuesday	3
Wednesday	4
Thursday	5
Friday	6
Saturday	7

The following example is a function that returns the current day as a string.

```
$Day := Day number (Current date)          ` $Day gets the current day number
Case of
  : ($Day = 1)
    $0 := "Sunday"
  : ($Day = 2)
    $0 := "Monday"
  : ($Day = 3)
    $0 := "Tuesday"
  : ($Day = 4)
    $0 := "Wednesday"
  : ($Day = 5)
    $0 := "Thursday"
  : ($Day = 6)
    $0 := "Friday"
  : ($Day = 7)
    $0 := "Saturday"
End case
```

Day of

Day of (*date*) → Number

Parameter	Type	Description
date	Date	Date for which to return the day

Day of returns the day of the month of *date*.

The following example illustrates the use of Day of. The results are assigned to the variable Result. The comments describe what Result is set to.

```
Result := Day of (!12/25/88!)          ` Result gets 25
Result := Day of (Current date)        ` Result gets the day of the current date
```

Month of

Month of (*date*) → Number

Parameter	Type	Description
date	Date	Date for which to return the month

Month of returns a number indicating the month of *date*.

The following example illustrates the use of Month of. The results are assigned to the variable Result. The comments describe what Result is set to.

```
Result := Month of (!12/25/88!)        ` Result gets 12
Result := Month of (Current date)      ` Result gets the month of the current date
```

Year of

Year of (*date*) → Number

Parameter	Type	Description
date	Date	Date for which to return the year

Year of returns a number indicating the year of *date*.

The following example illustrates the use of Year of. The results are assigned to the variable Result. The comments describe what Result is set to.

```
Result := Year of (!12/25/88!)         ` Result gets 1988
Result := Year of (Current date)       ` Result gets the year of the current date
```

Time Functions

Current time | Time | Time string

This section describes time functions. Times can be treated like a number when performing calculations. You can use the following statements to calculate the hours, minutes, and seconds of a time:

```
Hours := Time Var \ 3600          ` Returns the number of hours
Minutes := Time Var \ 60 % 60     ` Returns the number of minutes
Seconds := Time Var % 60          ` Returns the number of seconds
```

Current time

Current time ➛ Time

Current time returns the current time from the Macintosh system clock. The current time is always between †00:00:00† and †23:59:59†, inclusive. String can be used to convert the time into a string.

The following example shows a method you can use to time the length of an operation. In the example, *LongOperation* is a global procedure that needs to be timed.

```
It Took := Current time                          ` Save the start time
LongOperation                                    ` Perform the operation to be timed
It Took := Current time – It Took                ` Calculate how long it took
Alert ("The operation took " + String (It Took; 4))  ` Display how long it took
```

Time

Time (*time string*) ➛ Time

Parameter	Type	Description
time string	String	Time for which to return number of seconds

Time returns the time specified by *time string*. The *time string* parameter must follow the *HH:MM:SS* format and be in 24-hour time.

The following example displays an alert box with the message, "1:00 P.M. = 13 hours 0 minute."

```
ALERT ("1:00 P.M. = " + String (Time ("13:00:00"); 4))
```

Time string

Time string (*seconds*) ➛ String

Parameter	Type	Description
seconds	Number	Seconds from midnight

Time string takes *seconds*, the number of seconds since midnight, and returns the time as a string in 24-hour format. The string is in the *HH:MM:SS* format.

If you go beyond the number of seconds in a day (86,400), Time string continues to add hours, minutes, and seconds. For example, Time string (86401) returns 24:00:01.

String is different from Time. The String command formats a number as a number and the Time command formats it as a time.

The following example displays an alert box with the message, "46800 seconds is 13:00:00."

ALERT ("46800 seconds is " + **Time string** (46800))

Mathematical Functions

Abs	Int	Random
Dec	Log	Round
Exp	Num	Trunc

This section describes the standard math functions. Each of these functions returns a numeric value.

Note: 4th DIMENSION uses SANE (the Standard Apple Numeric Environment) for all calculations and with all numeric functions. The accuracy of numeric operations and functions is therefore dependent on SANE. SANE packages are available from vendors other than Apple Computer, Inc. Different SANE packages are also used for different hardware configurations. The use of different SANE packages may cause slightly different results for the same operations.

Abs

Abs (*number*) ➛ Number

Parameter	Type	Description
number	Number	Number of which to return the absolute value

Abs returns the absolute (unsigned, positive) value of *number*.

If *number* is negative, it is returned as positive. If *number* is positive, it is unchanged.

The following example returns the absolute value of –10.3, which is 10.3.

```
vVector := Abs (–10.3)          ` vVector gets 10.3
```

Dec

Dec (*number*) → Number

Parameter	Type	Description
number	Number	Number of which to return the decimal part

Dec returns the decimal (fractional) part of *number*. The value returned is always positive or zero.

The following example takes a monetary value expressed as a real number, and extracts the dollar part and cents part. If Amount were 7.31, then Dollars would be set to 7 and Cents would be set to 31.

```
Dollars := Int (Amount)          ` Get the dollars
Cents := Dec (Amount) * 100      ` Get the fractional part
```

Exp

Exp (*number*) → Number

Parameter	Type	Description
number	Number	Number to evaluate

Exp raises the natural log base (e = 2.71828182845904524) by the power of *number*. Exp is the inverse function of Log.

The following example assigns the exponential of 2 to v. (The log of v is 2.)

```
v := Exp (2)
```

Int

Int (*number*) → Number

Parameter	Type	Description
number	Number	Number of which to return the integer portion

Int returns the integer portion of *number* without rounding. Int truncates a negative *number* toward zero.

The following example illustrates how Int works for both a positive and a negative number. Note that the decimal portion of the number is removed.

```
x := Int (123.4)          ` x gets 123
y := Int (–123.4)         ` y gets –123
```

Log

Log (*number*) → Number

Parameter	Type	Description
number	Number	Number of which to return the log

Log returns the natural (Napierian) log of *number*. Log is the inverse function of Exp. A natural log has a base of 2.71828182845904524 (*e*), and a common log has a base of 10.

To convert to common log (log10), multiply the log by 0.434294481903251828.

To convert a common log to a natural log, multiply the common log by 2.30258509279404568.

The following example assigns the natural log of 2 to LogE, and then converts this number to the common log of 2.

```
LogE := Log (2)
Log10 := LogE * 0.434294481903251828
```

Num

String L332

Num (*string*) → Number

Parameter	Type	Description
string	String	String to be converted to a number

Num (*Boolean*) → Number (0 or 1)

Parameter	Type	Description
Boolean	Boolean	Boolean value to be converted to 0 or 1

The Num function has two forms.

The first form of Num converts *string* into a numeric value.

If *string* consists only of one or more alphabetic characters, Num returns a zero. If *string* includes alphabetic characters mixed in with numeric characters, Num ignores the alphabetic characters. Thus, Num transforms the string "a1b2c3" into the number 123.

There are three reserved characters that Num treats specially. They are the period (.), the hyphen (-), and *e* (or *E*). They are interpreted as numeric format characters.

The period is interpreted as a decimal place and must appear embedded in a numeric string.

The hyphen causes the number or an exponent to be negative. The hyphen must appear before any numeric characters or after the *e* for an exponent. If a hyphen is embedded in a numeric string, all numbers to the right are ignored, so Num ("123–456") returns 123.

The *e* or *E* causes any numeric characters to its right to be interpreted as an exponent. The *e* must be embedded in a numeric string. Thus, Num ("123e–2") returns 1.23.

The second form of Num evaluates *Boolean* and returns 0 or 1. If *Boolean* is FALSE, Num returns 0. If *Boolean* is TRUE, Num returns 1.

The following example illustrates how Num works when passed a numeric argument. Each line assigns a number to the Result variable. The comments describe the results.

```
Result := Num ("ABCD")          ` Result gets 0
Result := Num ("A1B2C3")        ` Result gets 123
Result := Num ("123")           ` Result gets 123
Result := Num ("123.4")         ` Result gets 123.4
Result := Num ("–123")          ` Result gets –123
Result := Num ("–123e2")        ` Result gets –12300
```

In the following example, Num of the customer debits returns either 0 or 1. Using the asterisk (*) as a string repetition operator, the customer comment is then stored in a field called [Client]Risk.

```
` If client owes less than 1000, a good risk.
` If client owes more than 1000, a bad risk.
[Client]Risk := ("Good" * Num ([Client]Debt < 1000)) + ("Bad" * Num ([Client]Debt >= 1000))
```

Random

Random → Number

Random returns a random integer value between 0 and 32,767 (inclusive).

To define a range of integers, use this formula:

Random % (End – Start + 1) + Start

Start is the first number in the range, and End the last.

The following example assigns a random integer between 10 and 30 to the Result variable.

```
Result := Random % 21 + 10
```

Round

Round (*number*; *places*) → Number

Parameter	Type	Description
number	Number	Number to be rounded
places	Number	Number of decimal places to round to

Round returns *number* rounded to the number of decimal places given by *places*.

If *places* is positive, *number* is rounded to *places* decimal places. If *places* is negative, *number* is rounded on the left of the decimal point.

If the digit following *places* is 5 though 9, Round rounds toward positive infinity for a positive number, and toward negative infinity for a negative number. If the digit following *places* is 0 through 4, Round rounds toward zero.

The following example illustrates how Round works with different arguments. Each line assigns a number to the Result variable. The comments describe the results.

```
Result := Round (16.857; 2)          ` Result gets 16.86
Result := Round (32345.67; –3)       ` Result gets 32000
Result := Round (29.8725; 3)         ` Result gets 29.873
Result := Round (–1.5; 0)            ` Result gets –2
```

Trunc

Trunc (*number*; *places*) → Number

Parameter	Type	Description
number	Number	Number to truncate
places	Number	Decimal places to truncate to

Trunc returns *number* with its decimal part truncated by the number of decimals specified by *places*. Trunc always truncates toward negative infinity.

If *places* is positive, *number* is truncated to *places* decimal places. If *places* is negative, number is truncated on the left of the decimal point.

The following example illustrates how Trunc works with different arguments. Each line assigns a number to the Result variable. The comments describe the results.

```
Result := Trunc (216.897; 1)         ` Result gets 216.8
Result := Trunc (216.897; –1)        ` Result gets 210
Result := Trunc (–216.897; 1)        ` Result gets –216.9
Result := Trunc (–216.897; –1)       ` Result gets –220
```

Trigonometric Functions

Arctan	Sin
Cos	Tan

This section describes the trigonometric functions. The functions all operate on radians. One degree equals 0.0174532925199432958 radians.

Note: 4th DIMENSION uses SANE (the Standard Apple Numeric Environment) for all calculations and with all numeric functions. The accuracy of numeric operations and functions is therefore dependent on SANE. SANE packages are available from vendors other than Apple Computer, Inc. Different SANE packages are also used for different hardware configurations. The use of different SANE packages may cause slightly different results for the same operations.

Arctan

Arctan (*number*) → Number

Parameter	Type	Description
number	Number	Tangent to be returned in radians

Arctan returns the arctangent in radians of *number*, where *number* is a tangent.

Cos

Cos (*number*) → Number

Parameter	Type	Description
number	Number	Number, in radians, of which to return the cosine

Cos returns the cosine of *number*, where *number* is expressed in radians.

Sin

Sin (*number*) → Number

Parameter	Type	Description
number	Number	Number, in radians, of which to return the sine

Sin returns the sine of *number*, where *number* is expressed in radians.

Tan

Tan (*number*) → Number

Parameter	Type	Description
number	Number	Number, in radians, for which to return the tangent

Tan returns the tangent of *number*, where *number* is given in radians.

Statistical Functions

Average	Sum	Std deviation
Max	Sum squares	Variance
Min		

These functions perform calculations on a series of values. The values for the Average, Max, Min, and Sum functions can be fields from a selection of records, or they can be subrecords. The values for the Sum squares, Std deviation, and Variance functions can be fields when used in a report, or they can be subrecords.

These functions work on numeric data only. Each of these functions returns a numeric value.

Using a Field

When Average, Max, Min, or Sum is used on a field, it must load each record in the current selection to calculate the result. If there are many records, this process may take some time.

When these functions are used in a report, they behave differently than at other times. This is because the report itself must load each record. Use these functions in a layout procedure or script when you are printing with the PRINT SELECTION command or when you are printing in the User environment by choosing the Print menu item from the File menu.

When you use these functions in a report, the values that are returned are meaningful only in a footer or break and at break level 0. This means that they are meaningful only at the end of a report, after all the records have been processed. You typically use the functions in a script for a nonenterable area that is included in the B0 Break area. The script assigns the value returned to the variable associated with the area.

Average

Average (*series*) ➛ Number

Parameter	Type	Description
series	Field or subfield	Data for which to return the average

Average returns the arithmetic mean (average) of *series*.

The following example sets a variable that is in the B0 Break area of an output layout. The line of code is the script for the variable. The script is not executed until the level 0 break.

vAverage := **Average** ([Employees] Sales)

The following example finds the average age of an employee's children from subfile data.

vAvg Age := **Average** ([Employees]Children'Age)

Max

Max (*series*) ➛ Number

Parameter	Type	Description
series	Field or subfield	Data for which to return the maximum value

Max returns the maximum value in *series*.

The following example is a script for a variable, vMax, placed in the break 0 portion of the layout. The variable is printed at the end of the report. The script assigns the maximum value of the field to the variable, which is then printed in the last break of the report.

vMax := **Max** ([People]Age)

The following example finds the maximum sales of an employee, and displays the result in an alert box. The sales amounts are stored in a subfield, [Employees]Sales'Dollars.

Alert ("The maximum sale was " + **String** (**Max** ([Employees]Sales'Dollars)))

Min

Min (*series*) → Number

Parameter	Type	Description
series	Field or subfield	Data for which to return the minimum value

Min returns the minimum value in *series*.

The following example is a script for a variable, vMin, placed in the break 0 portion of the layout. The variable is printed at the end of the report. The script assigns the minimum value of the field to the variable, which is then printed in the last break of the report.

vMin := **Min** ([People]Age)

The following example finds the minimum sales of an employee, and displays the result in an alert box. The sales amounts are stored in a subfield, [Employees]Sales'Dollars.

Alert ("The minimum sale was " + **String** (**Min** ([Employees]Sales'Dollars)))

Sum

Sum (*series*) → Number

Parameter	Type	Description
series	Field or subfield	Data for which to return the sum

Sum returns the sum (total of all values) for *series*.

The following example is a script for a variable, vTotal, placed in a layout. The script assigns the sum of all the lines in an invoice to the variable. The invoice lines are stored in a subfile called [Invoice]Lines.

vTotal := **Sum** ([Invoice]Lines'Line Total)

Sum squares

Sum squares (*series*) → Number

Parameter	Type	Description
series	Subfield or field	Data for which to return the sum of squares

Sum squares returns the sum of squares of *series*. You can only use a field with this function when printing a report.

The following example is a script for a variable called Squares. The script assigns the sum of squares for a data series to Squares.

Squares := **Sum squares** ([File]Data'Series)

Std deviation

Std deviation (*series*) → Number

Parameter	Type	Description
series	Subfield or field	Data for which to return the standard deviation

Std deviation returns the standard deviation of *series*. You can only use a field with this function when printing a report.

The following example is a script for a variable called Deviate. The script assigns the standard deviation for a data series to Deviate.

Deviate := **Std deviation** ([File]Data'Series)

Variance

Variance (*series*) → Number

Parameter	Type	Description
series	Subfield or field	Data for which to return the variance

Variance returns the variance for *series*. You can only use a field with this function when printing a report.

The following example is a script for a variable called Var. The script assigns the sum of squares for a data series to Var.

Var := **Variance** ([File]Data'Series)

Logical Functions

True False Not

This section describes logical functions.

True

True → Boolean (TRUE)

True returns the Boolean value TRUE.

The following example sets the variable, My Var, to TRUE.

My Var := **True**

False

False → Boolean (FALSE)

False returns the Boolean value FALSE.

The following example sets the variable, My Var, to FALSE.

My Var := **False**

Not

Not (*Boolean*) → Boolean

Parameter	Type	Description
Boolean	Boolean	Boolean value to negate

The Not function returns the negation of *Boolean*, changing a TRUE to FALSE or a FALSE to TRUE.

The following example first assigns TRUE to a variable. The example then changes the variable's value to FALSE, and then back to TRUE.

```
Result:= True                    ` Result is set to TRUE
Result:= Not (Result)            ` Result is set to FALSE
Result:= Not (Result)            ` Result is set to TRUE
```

MISCELLANEOUS COMMANDS

CHAPTER 18

MISCELLANEOUS COMMANDS

The commands described in this chapter allow you to work with variables, arrays, and parameters. They also allow you to control the execution of procedures for special purposes such as debugging.

Working With Variables

SAVE VARIABLE	CLEAR VARIABLE
LOAD VARIABLE	Undefined

This section describes commands that manage variables.

SAVE VARIABLE

SAVE VARIABLE (*document*; *variable1 {;...; variableN}*)

Parameter	Type	Description
document	String	Document to which to save the variables
variable	Variable	Variable to save

SAVE VARIABLE saves *variable* to *document*, a disk document.

The variables do not need to be of the same type (text, numeric, date, time, Boolean, or picture).

The document need not have the same name as the variable. If you supply an empty string ("") for *document*, a standard create-file dialog box appears, so that the user can choose the document to create.

The OK system variable is set to 1 if the variable is saved properly, and to 0 if it is not.

When you write variables to documents with SAVE VARIABLE, 4th DIMENSION uses an internal data format. You can retrieve the variables only with the LOAD VARIABLE command. Don't use RECEIVE VARIABLE or RECEIVE PACKET to read a document created by SAVE VARIABLE.

The following example saves three variables to a file called Disk File.

SAVE VARIABLE ("Disk File"; My String; My Number; My Picture)

LOAD VARIABLE

LOAD VARIABLE (*document*; *variable1* {;...; *variableN*})

Parameter	Type	Description
document	String	Document containing the variables
variable	Variable	Variable into which to load

LOAD VARIABLE loads *variable* from *document*, a document that was created with the SAVE VARIABLE command.

The variable is created, or overwritten if it already exists.

The document need not have the same name as the variable. If you supply an empty string ("") for *document*, a standard open-file dialog box appears, so that the user can choose the document to open.

The OK system variable is set to 1 if the variable is loaded properly, and to 0 if it is not.

The following example loads three variables from a document named Disk File.

LOAD VARIABLE ("Disk File"; My String; My Number; My Picture)

CLEAR VARIABLE

CLEAR VARIABLE (*variable*)

Parameter	Type	Description
variable	Variable	Variable to clear

CLEAR VARIABLE erases *variable* from memory. CLEAR VARIABLE sets the variable to undefined.

CLEAR VARIABLE is used primarily for clearing large variables, such as pictures, from memory.

Local variables, that is, variables preceded with a dollar sign ($), cannot be cleared with CLEAR VARIABLE. They are automatically cleared when the procedure they are in completes execution.

The following example clears the variable My Var.

CLEAR VARIABLE (My Var) ` Clear My Var

Undefined

Undefined (*variable*) → Boolean

Parameter	Type	Description
variable	Variable	Variable to test

Undefined returns TRUE if *variable* has not been defined, and FALSE if *variable* has been defined. A variable is defined if a value has been assigned to it. A variable is undefined if it has not had a value assigned to it, or it has been cleared with CLEAR VARIABLE.

The following example tests whether the variable Exec is undefined. If it is undefined, the variable is created.

```
If (Undefined (Exec))
  Exec := ""
End if
```

Managing Arrays

ARRAY BOOLEAN	ARRAY REAL	Size of array
ARRAY DATE	ARRAY TEXT	LIST TO ARRAY
ARRAY STRING	SORT ARRAY	ARRAY TO LIST
ARRAY INTEGER	COPY ARRAY	SELECTION TO ARRAY
ARRAY LONGINT	INSERT ELEMENT	ARRAY TO SELECTION
ARRAY PICTURE	DELETE ELEMENT	
ARRAY POINTER	Find in array	

The commands in this section manage arrays. They allow you to create arrays, sort arrays, find elements within arrays, move data to and from files, and perform other operations. Arrays are commonly displayed as scrollable areas and pop-up menus.

Array list L361

ARRAY BOOLEAN
ARRAY DATE
+ **ARRAY STRING** Yes
ARRAY INTEGER No
ARRAY LONGINT No
ARRAY PICTURE
ARRAY POINTER
ARRAY REAL
* **ARRAY TEXT** Yes L80-86

ARRAY BOOLEAN (*array name*; *size1*; *{size2}*)
ARRAY DATE (*array name*; *size1*; *{size2}*)
ARRAY INTEGER (*array name*; *size1*; *{size2}*)
ARRAY LONGINT (*array name*; *size1*; *{size2}*)
ARRAY PICTURE (*array name*; *size1*; *{size2}*)
ARRAY POINTER (*array name*; *size1*; *{size2}*)
ARRAY REAL (*array name*; *size1*; *{size2}*)
* ARRAY TEXT (*array name*; *size1*; *{size2}*)

Parameter	Type	Description
array name	Array	Name of the new array
size1	Number	Number of elements in the array, or number of arrays if *size2* is specified
size2	Number	Number of elements in a 2-dimensional array

+ ARRAY STRING (*string length*; *array name*; *size1*; *{size2}*)

Parameter	Type	Description
string length	Number	Length of strings
array name	Array	Name of the new array
size1	Number	Number of elements in the array, or number of arrays if *size2* is specified
size2	Number	Number of elements in a 2-dimensional array

All of the array commands create an array of elements in memory.

The *array name* parameter is the name of the new array.

The *size1* parameter is the number of elements in the array.

The *size2* parameter is optional; it creates a two-dimensional array. In this case, *size1* specifies the number of arrays and *size2* specifies the number of elements in each array.

Each array in a two-dimensional array can be treated like an element. This means that you can insert and delete entire arrays in a two-dimensional array with the other commands in this section.

The *string length* parameter is specified only for string arrays. It specifies the number of characters that each array element in a string array can contain. Operations performed on a string array are faster than operations performed on a text array.

Table 18-1 shows the formulas used to calculate the amount of memory used for each array type.

Table 18-1
Memory used by arrays

Array Type	Formula For Determining Memory Usage in Bytes
Boolean	8 + (Number of elements / 8)
Date	8 + (Number of elements * 6)
String	8 + (Number of elements * Length defined for the elements)
Integer	8 + (Number of elements * 2)
Long Integer	8 + (Number of elements * 4)
Picture	8 + (Number of elements * 4) + Sum of the size of each picture
Pointer	8 + (Number of elements * 16)
Real	8 + (Number of elements * 10)
Text	8 + (Number of elements * 4) + Sum of the size of each text element

When an array is first created, its elements are empty values: 0 for numeric arrays; "" for string and text arrays; !00/00/00! for date arrays; and FALSE for Boolean arrays.

You refer to the elements by using the indirection braces. For example, My Array{2} refers to the second element in My Array. You can not refer to an element simply by appending a number to the array name.

You refer to elements in a two-dimensional array by two sets of indirection braces. For example, My Array{3}{5} refers to the fifth element in the third array.

The commands in this section are also used to resize existing arrays. If you use one of these commands on an existing array, the command will add or delete elements. For example, if there are four elements in a Text array called Mine, the following line would remove two elements:

ARRAY TEXT (Mine; 2)

The two elements would be erased from the array. The following line would add four elements without disturbing the existing elements:

ARRAY TEXT (Mine; 6)

The following line would delete all elements (except the 0 element) but leave the array defined:

ARRAY TEXT (Mine; 0)

An element 0 (*array name*{0}) is always created for an array, and is set to a null value of the array type. Use Size of array to find the size of the array.

The following example creates a string array and moves information from a subfile into the array.

```
ALL SUBRECORDS ([People]Children)
ARRAY STRING (15; Names; Records in subselection ([People]Children))
For ($i; 1; Size of array (Names))
  Names{$i} := [People]Children'Name
  NEXT SUBRECORD ([People]Children)
End for
```

SORT ARRAY

L 86

SORT ARRAY (*array1 {;...; arrayN}; {direction}*)

Parameter	Type	Description
array	Array	Array to sort
direction	> or <	> to sort ascending; < to sort descending

SORT ARRAY sorts one or more arrays into ascending or descending order. The type of *array* can be any type except pointer or picture.

The *direction* parameter specifies whether to sort *array* in ascending or descending order. If *direction* is the "greater than" symbol (>), the sort is ascending. If *direction* is the "less than" symbol (<), the sort is descending. If *direction* is not specified, then the sort is ascending.

If more than one array is specified, the arrays are sorted following the sort order of the first array; they are not sorted independently. This feature is especially useful with grouped arrays.

The following example creates three arrays and then sorts them by name without having to sort the records the arrays are based on.

```
ALL RECORDS ([People])
SELECTION TO ARRAY ([People]Name; N; [People]Company; C; [Company]Address; A)
SORT ARRAY (C; N; A; >)
```

COPY ARRAY

L 82

COPY ARRAY (*from*; *to*)

Parameter	Type	Description
from	Array	Array from which to copy
to	Array	Array to which to copy

COPY ARRAY is used to duplicate an existing array. It creates the array *to* with the exact contents, size, and type of the array *from*. If the array *to* already exists, it is replaced with the newly created one.

The following example fills an array named C. It then creates a new array, named D, the same size as C and with the same contents.

```
ALL RECORDS ([People])                          ` Select all records in People
SELECTION TO ARRAY ([People]Company; C)         ` Move the company field data into array C
COPY ARRAY (C; D)                               ` Copy the array C to the array D
```

INSERT ELEMENT

INSERT ELEMENT (*array*; *where*; *{num of elements}*)

Parameter	Type	Description
array	Array	Name of the array
where	Number	Where to insert the element(s)
num of elements	Number	Number of elements to insert

INSERT ELEMENT inserts one or more array elements into *array*. The new element(s) are inserted "above" the element specified by *where*, and initialized to the empty value for the array type. All elements beyond *where* are moved "down" by *num of elements*.

If *where* is greater than the size of the array, the elements are added to the end of the array.

The *num of elements* parameter is the number of elements to insert. If *num of elements* is not specified, then one element is inserted. The size of the array grows dynamically by *num of elements*.

The following example inserts five new elements, starting at element 10.

```
INSERT ELEMENT (List; 10; 5)
```

DELETE ELEMENT

DELETE ELEMENT (*array*; *where*; {*num of elements*})

Parameter	Type	Description
array	Array	Array to delete elements from
where	Number	Element to begin delete from
num of elements	Number	Number of elements to delete

DELETE ELEMENT deletes one or more elements from *array*. The elements are deleted starting at the element specified by *where*.

The *num of elements* parameter is the number of elements to delete. If *num of elements* is not specified, then one element is deleted. The size of the array shrinks dynamically by *num of elements*.

The following example deletes three elements, starting at element 5.

DELETE ELEMENT (List; 5; 3)

Find in array

Find in array (*array*; *value*; {*start*}) → Number

Parameter	Type	Description
array	Array	Array to search
value	String or number or date or Boolean	Value to search for
start	Number	Element at which to start search

Find in array returns the number of the first element in *array* that matches *value*. Please note that Find in array is a function and not a command.

Find in array works for text, string, numeric, date, and Boolean arrays. The array and value must be of the same type.

If no match is found, Find in array returns –1.

If *start* is specified, the command starts searching at the element number specified by *start*.

In the following example an array named C is created from a selection of records. The GOTO SELECTED RECORD command uses Find in array to determine which record has "Acme" in the Company field, and then makes that record the current record.

```
ALL RECORDS ([People])                          ` Select all records in People
SELECTION TO ARRAY ([People]Company; C)         ` Move the company field data into array C
GOTO SELECTED RECORD ([People]; Find in array (C; "Acme"))
```

Size of array

Size of array (*array*) → Number

Parameter	Type	Description
array	Array	Array of which to return the size

Size of array returns the number of elements in *array*. If *array* is a two-dimensional array, Size of array returns the number of arrays.

The following example returns the size of the array My Array.

```
vSize := Size of array (My Array)                ` vSize gets the size of My Array
```

LIST TO ARRAY

L 83 /84

L 83

LIST TO ARRAY (*list*; *array*; *{linked array}*)

Parameter	Type	Description
list	String	List from which to copy
array	Array	Array to which to copy the list
linked array	Array	Array to which to copy the linked list

LIST TO ARRAY creates *array* from *list*. It copies the data from *list* into *array*. The array is overwritten if it already exists. The created array is always a text array unless you have previously defined the array as string.

The array, *linked array*, is filled with the names of any linked lists. If an element of the list has a linked list, the name of the linked list is put into the array element with the same number as the list element. If there is no linked list, then the element is the empty string. The *linked array* is the same size as *list*. You can use the names in the linked array to access the linked lists.

The following example copies the items of a list called Regions into an array called aRegions. The names of the linked lists are copied into an array called Links.

```
LIST TO ARRAY ("Regions"; aRegions; Links)
```

ARRAY TO LIST

L 83/84

(TEXT STRING < ARRAY
Field data to List
ie Sapphire Code Name)

ARRAY TO LIST (*array*; *list*; {*linked array*})

Parameter	Type	Description
array	Array	Array to copy to the list
list	String	List to which to copy the array
linked array	Array	Names of linked lists

ARRAY TO LIST copies *array* to *list*. If *list* does not exist, it is created.

The array, *linked array*, is used to link lists to each element in *array*. If an element of the *linked array* is not the empty string, then the name in the element is used to link a list to the corresponding item in the list. If that item is not a valid list name, then the link is not established.

The following example copies an array called aRegions into a list called Regions. An array called Links is used to specify the linked lists.

ARRAY TO LIST (aRegions; "Regions"; Links)

You cannot modify a list while working in a multi-user environment.

SELECTION TO ARRAY

L 85

SELECTION TO ARRAY (*field1*; *array1* {;...; *fieldN*; *arrayN*})

Parameter	Type	Description
field	Field	Field to use for data
array	Array	Array to receive the field data

SELECTION TO ARRAY creates one or more arrays and copies data from the field or fields of the current selection into the array or arrays. The current selection of the file specified by the first field is used. A field from another file can be included if an automatic relation exists between the files. The size of the array(s) is equal to the number of records in the selection.

The new arrays are typed according to the field type. An exception is if a Text field is copied into a string array. In this case, the array will remain a string array. Another exception is Time fields, which are put into Long Integer arrays.

Important: SELECTION TO ARRAY can create large arrays, depending on the size of the selection. Since arrays reside in memory, you must be sure there is enough memory to hold the array.

In the following example, the [People] file has an automatic relation to the [Company] file. The three arrays F, C, and A are sized according to the number of records selected in the [People] file, and will contain information from both files.

SELECTION TO ARRAY ([People]First; F; [People]Company; C; [Company]Address; A)

ARRAY TO SELECTION

L85

ARRAY TO SELECTION (*array1*; *field1* {;...; *arrayN*; *fieldN*})

Parameter	Type	Description
array	Array	Array to copy to the selection
field	Field	Field to receive the array data

ARRAY TO SELECTION copies one or more arrays into a selection of records. All fields listed must belong to the same file.

If a selection already exists, the elements of the array are put into the records, based on the order of the array and the order of the records. If there are more elements than there are records, then new records are created. The records, whether they are new or existing, are automatically saved.

If the arrays are of different sizes, the first array is used to determine how many elements to copy. Any additional arrays are also moved into the field that follows each array name.

This command is the complement of SELECTION TO ARRAY. The ARRAY TO SELECTION command does not allow fields from related files even when an automatic relation exists.

Warning: ARRAY TO SELECTION overwrites information in existing records, and so should be used with caution.

In the following example, the two arrays F and C put data into the [People] file. The F array is put in the First field, and the C array is put in the Company field.

ARRAY TO SELECTION (F; [People]First; C; [People]Company)

If any records are locked, they are skipped.

SET ABOUT (item; procedure)

Controlling the Execution of Procedures

ABORT	TRACE	ON ERR CALL
QUIT 4D	NO TRACE	ON EVENT CALL
EXECUTE		

The commands in this section let you stop the execution of procedures, monitor procedure execution, and manage errors and events.

ABORT

ABORT

ABORT stops procedure execution. Using ABORT is equivalent to clicking the Abort button in the Syntax Error window or in the Debugger.

ABORT is rarely used, and should normally not be used in a finished database application. ABORT is usually used during development to handle unexpected errors.

ABORT must be executed in a procedure installed by ON ERR CALL. The ON ERR CALL procedure should handle the error and then execute ABORT to return control to the menus.

If a layout is being displayed, ABORT stops procedure execution and returns control to the layout. However, if the layout is in the After phase, ABORT stops execution of the layout procedure, leaves the layout, and saves the record. ABORT does not affect the OK system variable; it simply stops execution of the layout procedure.

If a layout is not being displayed, ABORT stops procedure execution and returns control to the menus.

Use ABORT with discretion. It may be more appropriate to return to the procedure in which the error occurred. For example, if the error is in reading a file with RECEIVE PACKET, it may be necessary to return and close the open file.

QUIT 4D

QUIT 4D

D251
T188

QUIT 4D quits 4th DIMENSION and returns to the Finder. If the user is performing data entry, the record will be canceled and not saved. QUIT 4D may be used at any time.

Quit to User
Quit to Finder

Example: Scheduler
(Level 3 Advanced)

EXECUTE

EXECUTE (*statement*)

Parameter	Type	Description
statement	String	Code to be executed

EXECUTE executes *statement* as a line of code. The statement string must be one line. If *statement* is an empty string, EXECUTE does nothing.

The rule of thumb is that if the statement can be executed as a one-line global procedure, then it will execute properly. EXECUTE should be used sparingly, as it slows down execution speed.

The statement can include

- global procedures
- commands
- assignments
- global variables

The statement cannot contain control of flow statements.

The following example is a procedure that executes any statement entered. The procedure aborts at the end so that it can be executed within the Debugger without generating an error.

```
$Command := ""                                  ` Initialize the command string
Repeat
  $Command := Request ("Execute:"; $Command)    ` Get statement from the user
  If (OK = 1)                                   ` If the user clicked OK
    EXECUTE ($Command)                          ` Execute the command
  End if
Until (OK = 0)                                  ` Until the user clicks Cancel
ABORT
```

TRACE
NO TRACE

ERROR. This seems to be an endless loop (after Output listing) (on compilation of Output listing)

TRACE
NO TRACE

You use TRACE and NO TRACE during development of a database to trace procedures.

The TRACE command turns on the 4th DIMENSION Debugger. The Debug window is displayed before the next line of code is executed, and continues to be displayed for each line of code that is executed. You can also turn on the Debugger by holding down the Option key and the mouse button while code is executing.

NO TRACE turns off the Debugger engaged by TRACE, by an error, or by the user. Using NO TRACE has the same effect as clicking the No Trace button in the Debugger.

ON ERR CALL

ON ERR CALL (*error procedure*)

Parameter	Type	Description
error procedure	String	Error procedure to be called

ON ERR CALL installs the procedure named by *error procedure* as the procedure for managing errors. If *error procedure* is an empty string, error handling returns to 4th DIMENSION. After installation, 4th DIMENSION calls the procedure named by *error procedure* when an error occurs.

You can identify errors by reading the Error system variable. The error-handling procedure should normally present an error message to the user. 4th DIMENSION error codes are listed in Appendix E. Errors may also be generated by the Macintosh; some of the most common ones are listed in Appendix E.

The ABORT command may be used to terminate processing. If you don't call ABORT in the installed procedure, 4th DIMENSION returns to the interrupted procedure.

If an error occurs in the procedure that is installed with ON ERR CALL, 4th DIMENSION takes over error handling. Therefore, you should make sure that the installed error-handling procedure cannot generate an error. Also, you cannot use ON ERR CALL inside the error-handling procedure.

ON ERR CALL is commonly used in the startup procedure for a custom database to manage error handling for that database application. ON ERR CALL may also be placed at the beginning of a procedure to handle errors specific to that procedure.

When an ON ERR CALL procedure is installed, it is not possible to trace a procedure by using Option-click. This is because Option-click generates an error code that immediately activates the ON ERR CALL procedure.

The following example shows the installation of the error-handling procedure.

```
ON ERR CALL ("DoError")
```

The procedure below is the *DoError* procedure, installed by the example command. The procedure displays a confirmation dialog box, which displays the error. If the user clicks the OK button, the procedure executes ABORT, which stops all procedure execution.

```
` DoError; called by ON ERR CALL
CONFIRM ("Error #" + String (Error) + ". Do you want to stop?")
If (OK = 1)
  ABORT                                    ` End procedure, return to menus
End if
```

ON EVENT CALL

example L-245

ON EVENT CALL (*event procedure*)

Parameter	Type	Description
event procedure	String	Event procedure to be called

ON EVENT CALL installs the procedure named by *event procedure* as the procedure for managing events. If *event procedure* is an empty string, event handling returns to 4th DIMENSION. After installation, 4th DIMENSION calls the procedure named by *event procedure* when an event occurs. An event can be either a mouse click or a keystroke.

A procedure must be executing for an event to be recognized. This means that ON EVENT CALL is usually appropriate only for procedures that are executing for more than a few seconds. Since layout procedures are executed only during an execution phase (Before, During, After, and so on), it is usually inappropriate to install an ON EVENT CALL procedure when displaying a layout.

In the event-handling procedure, you can read three system variables—MouseDown, KeyCode, and Modifiers. L 389

The MouseDown system variable is set to 1 if the event is the user's clicking the mouse button, and to 0 if it is not.

The KeyCode system variable is set to the ASCII code for a keystroke. Appendix D lists the ASCII codes for the Macintosh.

The Modifier system variable contains the Macintosh keyboard modifier value. The Modifier system variable indicates whether any of the following modifier keys were down when the event occurred: Command, Shift, Caps Lock, Option, or Control. The modifier keys do not generate an event on their own; a key or the mouse button must also be pressed.

The following code evaluates the Modifier system variable and sets five variables (Command, Shift, Caps Lock, Option, and Control) to 1 if the key is pressed, and to 0 if it is not.

```
$M := Modifiers \ 256                    ` $M can be any variable name
Command := $M %2
$M := $M \ 2
Shift := $M % 2
$M := $M \ 2
CapsLock := $M % 2
$M := $M \ 2
Option := $M % 2
$M := $M \ 2
Control := $M % 2
```

D168
L295

Important: The system variables, MouseDown, KeyCode, and Modifiers, contain significant values only within an ON EVENT CALL procedure.

ON EVENT CALL is rarely used. It is normally used before a section of code that must monitor all events, and then cleared by an empty string argument immediately following that section. At the end of the installed procedure, control returns to the interrupted procedure.

In the following example, the *Infinite Loop* procedure installs the *Event* procedure to trap events. *Infinite Loop* then enters an infinite loop. It can exit the loop only when the *Event* procedure changes the value of the Loop variable. After the procedure has exited the loop, the empty string is used to reset ON EVENT CALL.

```
` Global procedure: Infinite loop
ON EVENT CALL ("Event")                    ` Install the Event procedure
Loop := True                               ` Loop is set by Event procedure
While (Loop)                               ` Loop forever (until Event sets Loop)
End while
ON EVENT CALL ("")                         ` Reset ON EVENT CALL
```

The following procedure is the *Event* procedure installed by the procedure just given. It creates a string variable, $Text, that describes the event. The procedure then displays an alert describing the event. If the user presses the Q key, the global variable Loop is set to FALSE, and the *Infinite Loop* procedure terminates.

```
` Global procedure: Event
$Text := ""                                ` $Text contains a string describing the event
Case of                                    ` Either the mouse or a key was pressed
  : (MouseDown = 1)                        ` The mouse was pressed
    $Text := "The Mouse was pressed"
  : (KeyCode # 0)                          ` A key was pressed
    $Text := "KeyCode = " + String (KeyCode)
End case
` The following code tests which modifiers were pressed
` and changes the $Text string appropriately.
$Text := $Text + Char (13) + "Modifiers "
$M := Modifiers \ 256
$Text := $Text + (": Command" * ($M % 2))
$M := $M \ 2
$Text := $Text + (": Shift" * ($M % 2))
$M := $M \ 2
$Text := $Text + (": CapsLock" * ($M % 2))
$M := $M \ 2
$Text := $Text + (": Option" * ($M % 2))
$M := $M \ 2
$Text := $Text + (": Control" * ($M % 2))
ALERT ($Text)                              ` Display the event information
If (Char (KeyCode) = "q")                  ` If the user pressed Q then reset Loop
  Loop := False
End if
```

Getting Information About Data Objects

Count parameters	Get pointer
Is a variable	Type

The functions in this section return information about data objects.

Count parameters

Count parameters → Number

Count parameters returns the number of parameters passed to a procedure. Count parameters is meaningful only in a global procedure that has been called by another procedure (a subroutine). In all other cases Count parameters returns 0.

The following example is a function that returns a string. The string is the concatenation of whatever strings are passed to it.

```
$0 := ""                                  ` Initialize the return value to null
For ($i; 1; Count parameters)             ` Loop for each parameter
  $0 := $0 + ${$i}                        ` Concatenate the parameters
End for
```

Is a variable

Is a variable (*parameter*) → Boolean

Parameter	Type	Description
parameter	Pointer	Data object to test

This function is usually used to test whether a parameter that was passed to a procedure was a variable. Is a variable returns TRUE if *parameter* is a pointer to a variable. Is a variable returns FALSE if *parameter* is a pointer to a field.

The following example is a function named *SumIt* that returns a sum of an array or a field. If the parameter is a field, it uses the Sum function. If the parameter is an array, it calculates the sum. Notice that it uses the Type function to check that the parameter is the correct data type.

```
$0 := 0                                          ` Initialize the total
If (Is a variable ($1))                          ` The parameter is a variable
   ` Is it a numeric array?
  If ((Type ($1») = 14) | (Type ($1») = 15) | (Type ($1») = 16))
    For ($i; 1; Size of array ($1»))             ` Loop once for each element in the array
      $0 := $0 + $1»{$i}                         ` Add each element to the sum
    End for
  End if
Else                                             ` Do this for fields
   ` Is it a numeric field?
  If ((Type ($1») = 1) | (Type ($1») = 8) | (Type ($1») = 9))
    $0 := Sum ($1»)                              ` Sum the field
  End if
End if
```

If you had a numeric field called [People]Salary, you could sum it using the following line:

SumIt (»[People]Salary)

If you had a numeric array called Line Totals, you could sum it using the following line:

SumIt (»Line Totals)

Get pointer

Pointers L 87/L94

Examples of Pointers
L152 Last errors
L318 Count fields
L319 File name
L320 File
L321 Field
L371 Type

Get pointer (*name*) ➛ Pointer

Parameter	Type	Description
name	String	Name of a variable

Get pointer returns a pointer to *name*. The *name* is the name of a variable. The variable does not need to exist before Get pointer is executed.

The following example sets a series of variables named CB1, CB2,..., CB20 to 1.

```
For ($i; 1; 20)
  $p := Get pointer ("CB" + String ($i))         ` Get a pointer to the next variable
  $p» := 1                                       ` Set the variable
End for
```

Set > Pointer
Nil (variable) > Boolean

Type

Type (*parameter*) → Number

Parameter	Type	Description
parameter	Field or variable	Data object for which to return the type

Type returns the data type of *parameter*. Type is usually used to test whether a parameter is the correct data type. Table 18-2 lists the numbers returned for the data types.

Table 18-2
Data type numbers

Data Type	Number	Data Type	Number
String	0	Real array	14
Real	1	Integer array	15
Text	2	Long integer array	16
Picture	3	Date array	17
Date	4	Text array	18
Undefined	5	Picture array	19
Boolean	6	Pointer array	20
Subfile	7	String array	21
Integer	8	Boolean array	22
Long integer	9	Pointer	23
Time	11	Fixed String	24

different variables

Outline separator

The following example is a script for a button. The script changes the data in the current object to uppercase. The object must be a text or string data type (type 0 or 2).

```
$p := Last object                          ` Save the pointer to the last area
If ((Type ($p») = 0) | (Type($p») = 2))    ` If it is a string or text area
   $p» := Uppercase ($p»)                  ` Change the area to uppercase
End if
```

See the example for the Is a variable function, earlier in this section, for another example that uses the Type function.

APPENDIXES

APPENDIX A: Compatibility With Version 1.0

This appendix describes compatibility issues to consider when updating procedures created in version 1.0 of 4th DIMENSION.

Obsolete and Changed Functionalities

This section describes how version 2.0 differs from version 1.0 in the use of the language.

File Relations—Links

In version 1.0, relations between files were managed by links. Links used record pointers stored with each record to maintain file relations. Version 2.0 does not store record pointers to maintain file relations. All version 1.0 commands used for linking have been renamed to reflect the new functionality in file relations.

Version 2.0 supports "dynamic" file relations. This means that a relation is dependent only on the data that is stored in a record. The relation is not dependent on record pointers (links). Thus it is possible to perform such advanced operations as searching and sorting based on data in another file. 4th DIMENSION can also update and maintain dynamic relations more quickly.

In version 1.0, the use of the second argument to LOAD LINKED RECORD was sometimes used to link the same data to two different records. This practice was discouraged, since rebuilding the file would lose the link. The use of the second argument to select identical data is not supported in version 2.0. You must use unique data (a key field) to establish relations in version 2.0.

Variable Indirection

The two methods of variable indirection used in version 1.0, alpha indirection and numeric indirection, are not supported by the Compiler and will not be supported in future versions of 4th DIMENSION. Databases that are converted from version 1.0 will support these methods, but it is recommended that you use pointers.

Pointers and true arrays are used to support the tasks that these methods performed. The use of pointers and arrays gives you more power and flexibility in the management of variables. See Chapter 9, "Arrays and Pointers," for more information on these topics.

Numeric Indirection

Version 1.0 allowed the programmer to reference variables by using a number inside curly braces ({...}). This method of indirection was commonly used to reference variable tables (pseudo-arrays; for example, v1, v2,..., vN).
In version 2.0, you should use the array commands to create true arrays.
For example, in version 1.0, you might have used a loop like this to create a pseudo-array containing numbers:

```
$i := 1
While ($i < 11)
  My Array{$i} := 0
  $i := $i + 1
End while
```

In version 2.0, you simply write a line like this:

Language 90

```
ARRAY REAL (My Array; 10)
```

The indirection method used in version 1.0 was also commonly used to reference a group of radio buttons or check boxes. You should now use the Get pointer command to indirectly reference radio buttons and check boxes. For example, in version 1.0, if you had a group of check boxes with variables named CB1, CB2,...,CB10, you might have used a loop like this one using variable indirection:

```
$i := 1
While ($i < 11)
  If (CB{$i} = 1)                          ` If the check box is checked
    ` Do something here
  End if
  $i := $i + 1
End while
```

In version 2.0, you use a loop like this:

```
For ($i; 1; 10)
  If (Get pointer ("CB" + String ($i))» = 1)      ` If the check box is checked
    ` Do something here
  End if
End for
```

Alpha Indirection

Although version 2.0 supports variable indirection using the section symbol (§) in converted databases, its use is discouraged. It is recommended that you use pointers and the Get pointer function instead of this method.

For example, you could refer to a variable like this:

```
§("My Var") := 10
```

This line puts 10 into the variable named My Var.

Using the Get pointer function and a pointer, you would write this:

Language 91

```
$p := Get pointer ("My Var")
$p» := 10
```

Setting Graph Legends

Version 1.0 used the name of a variable table (a pseudo-array; for example, v1, v2,..., vN) to label a graph legend. This method is no longer supported. Use the GRAPH SETTINGS command to set the graph legends.

Size of Arrays

In version 2.0, you cannot refer to the size of the array by examining element 0 as you could in version 1.0. Use the command Size of array for this purpose.

Matching Parentheses

If you inadvertently left off matching parentheses, version 1.0 did not alert you. Version 2.0 will now catch this problem and generate a syntax error.

The Flush System Variable

4th DIMENSION version 2.0 uses optimized data caching, making the Flush system variable obsolete. This variable is not supported in version 2.0.

Changes in Commands

This section describes how commands in version 2.0 differ from those in version 1.0.

Changed Command Names

Table A-1 shows the old and new names of commands. It is important to note that procedures in databases converted from version 1.0 are automatically updated to use the new names.

Table A-1
Changed Command Names

Old Name	New Name
CREATE LINKED RECORD	CREATE RELATED ONE
Current password	Current user
GET HIGHLIGHTED TEXT	GET HIGHLIGHT
GO TO FIELD	GOTO AREA
GO TO XY	GOTO XY
LOAD LINKED RECORD	RELATE ONE
LOAD OLD LINKED RECORD	OLD RELATED ONE
SAVE LINKED RECORD	SAVE RELATED ONE
SAVE OLD LINKED RECORD	SAVE OLD RELATED ONE
SEARCH	SEARCH BY FORMULA
Squares sum	Sum squares

Obsolete Commands

This section describes commands that are obsolete. They remain in the language for compatibility with version 1.0. These commands may not be supported in future versions of 4th DIMENSION. Although they still execute properly, you should change any procedures that use these commands to use the new features of version 2.0.

- ACTIVATE LINK—You should use RELATE ONE instead of ACTIVATE LINK.
- Mod—You should use the Modulo operator (%), which performs the same operation as and executes faster than the Mod function.
- SORT BY INDEX—You should use SORT SELECTION instead of SORT BY INDEX.

Changed Command Operations

This section describes commands whose operations have changed.

- BEEP—The Macintosh system no longer supports a beep length.
- CLEAR VARIABLE—In version 2.0, this command takes a variable name as the argument, instead of a string. For compatibility, you can use a string, but this usage is discouraged and is not supported by the Compiler. It may not be supported by future versions of 4th DIMENSION.
- Current password (now Current user)—In version 1.0, this function returned the current password. In version 2.0, this function returns the the current user name.
- Current time and Time—In version 2.0, these functions return a time. If you display the data on screen, the data will be formatted as a time.
- FONT—Apple Computer, Inc. recommends using only a font name to refer to a font. The FONT command still supports the use of a number to refer to a font, but this method is discouraged and may not be supported in future versions.
- GET HIGHLIGHTED TEXT (now GET HIGHLIGHT)—In version 1.0, this command used the first parameter to get the highlight from the specified object. In version 2.0, GET HIGHLIGHT *always* gets the highlighted area from the last object and ignores the first parameter. Page 149
- GO TO FIELD (now GOTO AREA)—In version 2.0, this command goes to variables as well as fields.
- HIGHLIGHT TEXT—In version 2.0, this command goes to the object as well as highlighting it.
- In header—In version 2.0, this function returns TRUE in break headers in addition to returning TRUE in page headers.
- Level—In version 2.0, this function returns the level in a break header and in breaks.
- ON SERIAL PORT CALL—In version 2.0, the installed procedure is called any time there is serial port activity. In version 1.0, a procedure had to be running for the procedure to be called.
- OPEN WINDOW—In version 2.0, this command opens multiple windows. If you call OPEN WINDOW without CLOSE WINDOW in your databases, multiple windows will be opened.
- REDRAW—In version 2.0, this command is not needed after a sort of data in an included layout.
- SEND RECORD and RECEIVE RECORD—Records saved by using these commands are not compatible between version 1.0 and version 2.0.
- String—In version 2.0, this function formats string, date, and time.
- USE ASCII MAP—In version 2.0, this command requires a second argument to specify whether the map is to be used for input or output.

APPENDIX B: Preparing Code For the Compiler

This appendix gives you information that is useful in preparing your procedures for compilation.

General Compiler Rules

- Variable indirection is not allowed. You cannot use alpha indirection, with the section symbol (§), to indirectly reference variables. Nor can you use numeric indirection, with the curly braces ({...}), for this purpose.
- You can't change the data type of a global variable or array.
- You can't change a one-dimensional array to a two-dimensional array, or change a two-dimensional array to a one-dimensional array.
- You can't change the length of string variables or elements in string arrays.
- You should specify the data type of a variable by using the Compiler directives where the data type is ambiguous.
- Wherever possible, use variables of a long integer data type for maximum performance. This rule applies especially to any variable used as a counter.
- To clear a variable (initialize it to null), use CLEAR VARIABLE with the name of the variable. Do not use a string to represent the name of the variable in the CLEAR VARIABLE command.
- The Undefined function will always return FALSE. Variables are always defined.

Commands and Compiler Compatibility

- LOAD VARIABLE—The variables must be of the same type as those loaded from disk.
- CLEAR VARIABLE—This command sets the variable to a null value but does not set it to undefined. Variables can never be undefined in compiled code. This command is used to clear large variables, such as pictures, from memory.
- TRACE, NO TRACE—These commands have no effect in compiled procedures.
- Undefined—Since variables are always "defined" in compiled code, Undefined always returns FALSE when used in compiled procedures.

Report Break Processing

The Subtotal function will not initiate break processing in compiled procedures. You must use the BREAK LEVEL command to initiate break processing, and use the ACCUMULATE command to specify what to accumulate for subtotals.

IDLE

Compiler Directives

The commands in this section are used to declare the variables used by your procedures. If you will be compiling your procedures, you should declare all variables by using these commands. These commands do not need to appear in an executed procedure. For example, you could put them in a procedure called *Compiler* that is never executed.

C_BOOLEAN
C_DATE
C_INTEGER extra
C_LONGINT
C_PICTURE
C_POINTER
C_REAL
C_TEXT
C_TIME
C_STRING extra

C_BOOLEAN (*variable1 {;...; variableN}*)
C_DATE (*variable1 {;...; variableN}*)
C_INTEGER (*variable1 {;...; variableN}*)
C_LONGINT (*variable1 {;...; variableN}*)
C_PICTURE (*variable1 {;...; variableN}*)
C_POINTER (*variable1 {;...; variableN}*)
C_REAL (*variable1 {;...; variableN}*)
C_TEXT (*variable1 {;...; variableN}*)
C_TIME (*variable1 {;...; variableN}*)

Parameter	Type	Description
variable	Variable	Name of variable(s) to pre-declare

C_STRING (*size*; *variable1 {;...; variableN}*)

Parameter	Type	Description
size	Number	Size of the string
variable	Variable	Name of variable(s) to pre-declare

These commands have no effect on the normal operation of 4th DIMENSION. They affect only compiled procedures.

These commands pre-declare variables and cast the variables as a specified data type. Pre-declaring variables resolves ambiguities concerning a variable's data type. If a variable is not pre-declared with one of these commands, the Compiler will attempt to determine a variable's data type. The data type of a variable used in a layout is often difficult for the Compiler to determine. Therefore it is especially important that you use these commands to pre-declare a variable used in a layout.

- C_BOOLEAN casts each specified variable as a Boolean variable.
- C_DATE casts each specified variable as a date variable.
- C_INTEGER casts each specified variable as an integer variable.
- C_LONGINT casts each specified variable as a long integer variable.
- C_PICTURE casts each specified variable as a picture variable. You can perform picture operations only on variables that have been declared as pictures.
- C_POINTER casts each specified variable as a pointer variable.
- C_REAL casts each specified variable as a real variable.
- C_TEXT casts each specified variable as a text variable.
- C_TIME casts each specified variable as a time variable.
- C_STRING casts each specified variable as a string variable. The *size* parameter specifies the length of the string the variable can contain. A string variable is faster in use than a text variable.

Numeric operations on long integer and integer variables are usually much faster than operations on the default numeric type (extended).

APPENDIX C: 4th DIMENSION System Variables

This appendix summarizes the 4th DIMENSION system variables.

OK

The OK system variable is the most commonly used of all system variables. In general, it is set to 1 after an operation has completed successfully, and it is set to 0 if an operation does not complete successfully. The following commands set the OK system variable.

example
L187
L193
L198
L199
L228
L240
L241
L242
*L254
L255
L303
L305
L310
L312
L316
L364

ADD RECORD
ADD SUBRECORD
Append document
APPLY TO SELECTION
APPLY TO SUBSELECTION
ARRAY TO LIST
ARRAY TO SELECTION
CLOSE DOCUMENT
CONFIRM
CREATE EMPTY SET
Create document
DELETE DOCUMENT
DIALOG
DISPLAY SELECTION
EXPORT DIF
EXPORT SYLK
EXPORT TEXT
IMPORT DIF
IMPORT SYLK
IMPORT TEXT
LOAD SET
LOAD VARIABLE
MERGE SELECTION
MODIFY RECORD
MODIFY SELECTION
MODIFY SUBRECORD
Open document
PRINT LABEL
PRINT SETTINGS
RECEIVE PACKET
RECEIVE RECORD
RECEIVE VARIABLE
REPORT
Request
SAVE SET
SAVE VARIABLE
SEARCH
SEARCH BY FORMULA
SEARCH BY INDEX
SEARCH BY LAYOUT
SEARCH SELECTION
SEND PACKET
SEND RECORD
SEND VARIABLE
SET CHANNEL
SORT BY FORMULA
SORT FILE
SORT SELECTION
USE ASCII MAP
USE SETTINGS
VALIDATE TRANSACTION

CANCEL L158
OK set to 0
ACCEPT L157
OK set to 1

Document

L278

The Document system variable contains the name of the Macintosh disk file that was last opened or created with one of the following commands.

Append document
Create document
EXPORT DIF
EXPORT SYLK
EXPORT TEXT
IMPORT DIF
IMPORT SYLK
IMPORT TEXT
LOAD SET
LOAD VARIABLE
MERGE SELECTION
Open document
PRINT LABEL
REPORT
SAVE SET
SAVE VARIABLE
SET CHANNEL
USE ASCII MAP

example

FldDelimit

L214

The FldDelimit system variable contains the ASCII code of the character to use as the field delimiter when importing or exporting text. By default, this value is 9, the ASCII code for the tab character. Change the value to set a new field delimiter.

RecDelimit

L214

The RecDelimit system variable contains the ASCII code of the character to use as the record delimiter when importing or exporting text. By default, this value is 13, the ASCII code for the carriage return character. Change the value to set a new record delimiter.

Error

The Error system variable is valid only in a procedure installed by ON ERR CALL. L365 This variable contains the code for the error. Appendix E lists 4th DIMENSION and Macintosh error codes.

examples L368 L368

MouseDown, KeyCode, and Modifiers

These system variables are valid only in a procedure installed by ON EVENT CALL. L366

The MouseDown system variable is set to 1 if the mouse button was pressed. Otherwise, it is set to 0.

The KeyCode system variable contains the ASCII code of the key that was pressed.

The Modifiers system variable contains the Macintosh keyboard modifier codes. See the description of the ON EVENT CALL command for more information on the Modifiers system variable. L366

APPENDIX D: ASCII Codes

This appendix consists of two tables. Table D-1 presents the standard ASCII codes. Table D-2 presents the extended Macintosh character set for the Times font.

Table D-1
Standard ASCII codes

	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
Null	NUL	0	0	0	SPACE sp	32	40	20	@	64	100	40	‘	96	140	60
	SOH	1	1	1	!	33	41	21	A	65	101	41	a	97	141	61
	STX	2	2	2	“	34	42	22	B	66	102	42	b	98	142	62
	ETX	3	3	3	#	35	43	23	C	67	103	43	c	99	143	63
	EOT	4	4	4	$	36	44	24	D	68	104	44	d	100	144	64
	ENQ	5	5	5	%	37	45	25	E	69	105	45	e	101	145	65
	ACK	6	6	6	&	38	46	26	F	70	106	46	f	102	146	66
Bell	BEL	7	7	7	‘	39	47	27	G	71	107	47	g	103	147	67
	BS	8	10	8	(	40	50	28	H	72	110	48	h	104	150	68
Horz Tab	HT	9	11	9	)	41	51	29	I	73	111	49	i	105	151	69
Line Feed	LF	10	12	A	*	42	52	2A	J	74	112	4A	j	106	152	6A
Vert Tab	VT	11	13	B	+	43	53	2B	K	75	113	4B	k	107	153	6B
Form Feed	FF	12	14	C	,	44	54	2C	L	76	114	4C	l	108	154	6C
Return	CR	13	15	D	-	45	55	2D	M	77	115	4D	m	109	155	6D
	SO	14	16	E	.	46	56	2E	N	78	116	4E	n	110	156	6E
	SI	15	17	F	/	47	57	2F	O	79	117	4F	o	111	157	6F
	DLE	16	20	10	0	48	60	30	P	80	120	50	p	112	160	70
	DC1	17	21	11	1	49	61	31	Q	81	121	51	q	113	161	71
	DC2	18	22	12	2	50	62	32	R	82	122	52	r	114	162	72
	DC3	19	23	13	3	51	63	33	S	83	123	53	s	115	163	73
	DC4	20	24	14	4	52	64	34	T	84	124	54	t	116	164	74
	NAK	21	25	15	5	53	65	35	U	85	125	55	u	117	165	75
	SYN	22	26	16	6	54	66	36	V	86	126	56	v	118	166	76
	ETB	23	27	17	7	55	67	37	W	87	127	57	w	119	167	77
	CAN	24	30	18	8	56	70	38	X	88	130	58	x	120	170	78
	EM	25	31	19	9	57	71	39	Y	89	131	59	y	121	171	79
	SUB	26	32	1A	:	58	72	3A	Z	90	132	5A	z	122	172	7A
Escape	ESC	27	33	1B	;	59	73	3B	[	91	133	5B	{	123	173	7B
	FS	28	34	1C	<	60	74	3C	\	92	134	5C	|	124	174	7C
	GS	29	35	1D	=	61	75	3D	]	93	135	5D	}	125	175	7D
	RS	30	36	1E	>	62	76	3E	^	94	136	5E	~	126	176	7E
	US	31	37	1F	?	63	77	3F	_	95	137	5F	DEL	127	177	7F

Table D-2
Extended Macintosh character set (Times)

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
Ä	128	200	80	†	160	240	A0	¿	192	300	C0	‡	224	340	E0
Å	129	201	81	°	161	241	A1	¡	193	301	C1	·	225	341	E1
Ç	130	202	82	¢	162	242	A2	¬	194	302	C2	‚	226	342	E2
É	131	203	83	£	163	243	A3	√	195	303	C3	„	227	343	E3
Ñ	132	204	84	§	164	244	A4	ƒ	196	304	C4	‰	228	344	E4
Ö	133	205	85	•	165	245	A5	≈	197	305	C5	Â	229	345	E5
Ü	134	206	86	¶	166	246	A6	Δ	198	306	C6	Ê	230	346	E6
á	135	207	87	ß	167	247	A7	«	199	307	C7	Á	231	347	E7
à	136	210	88	®	168	250	A8	»	200	310	C8	Ë	232	350	E8
â	137	211	89	©	169	251	A9	…	201	311	C9	È	233	351	E9
ä	138	212	8A	™	170	252	AA		202	312	CA	Í	234	352	EA
ã	139	213	8B	´	171	253	AB	À	203	313	CB	Î	235	353	EB
å	140	214	8C	¨	172	254	AC	Ã	204	314	CC	Ï	236	354	EC
ç	141	215	8D	≠	173	255	AD	Õ	205	315	CD	Ì	237	355	ED
é	142	216	8E	Æ	174	256	AE	Œ	206	316	CE	Ó	238	356	EE
è	143	217	8F	Ø	175	257	AF	œ	207	317	CF	Ô	239	357	EF
ê	144	220	90	∞	176	260	B0	–	208	320	D0		240	360	F0
ë	145	221	91	±	177	261	B1	—	209	321	D1	Ò	241	361	F1
í	146	222	92	≤	178	262	B2	“	210	322	D2	Ú	242	362	F2
ì	147	223	93	≥	179	263	B3	”	211	323	D3	Û	243	363	F3
î	148	224	94	¥	180	264	B4	‘	212	324	D4	Ù	244	364	F4
ï	149	225	95	µ	181	265	B5	’	213	325	D5	ı	245	365	F5
ñ	150	226	96	∂	182	266	B6	÷	214	326	D6	ˆ	246	366	F6
ó	151	227	97	∑	183	267	B7	◊	215	327	D7	˜	247	367	F7
ò	152	230	98	∏	184	270	B8	ÿ	216	330	D8	¯	248	370	F8
ô	153	231	99	π	185	271	B9	Ÿ	217	331	D9	˘	249	371	F9
ö	154	232	9A	∫	186	272	BA	⁄	218	332	DA	˙	250	372	FA
õ	155	233	9B	ª	187	273	BB	¤	219	333	DB	°	251	373	FB
ú	156	234	9C	º	188	274	BC	‹	220	334	DC	˚	252	374	FC
ù	157	235	9D	Ω	189	275	BD	›	221	335	DD	˝	253	375	FD
û	158	236	9E	æ	190	276	BE	fi	222	336	DE	˛	254	376	FE
ü	159	237	9F	ø	191	277	BF	fl	223	337	DF	ˇ	255	377	FF

APPENDIX E: 4th DIMENSION and Macintosh Error Messages

This appendix lists the error codes that may occur during the use of 4th DIMENSION. Error codes generated by the Macintosh and returned by 4th DIMENSION are also listed.

You can trap the error codes by using the ON ERR CALL command. If you use this command, the Error system variable contains the error code.

Table E-1 lists codes returned primarily because of syntax errors in procedures. These are errors in the design of a procedure.

Table E-1
4th DIMENSION procedure error codes

Code	Reason
1	"(" expected.
2	Field expected.
3	The command may be executed only on a field in a subfile.
4	Parameters in the list must all be of the same type.
5	There is no file to apply the command to.
6	The command may only be executed on a field of type: 'Subfile'.
7	A numeric argument was expected.
8	An alphanumeric argument was expected.
9	The result of a conditional test was expected.
10	The command can't be applied to this field type.
11	The command can't be applied between two conditional tests.
12	The command can't be applied between two numeric arguments.
13	The command can't be applied between two alphanumeric arguments.
14	The command can't applied between two date arguments.
15	The operation is not compatible with the two arguments.
16	The field has no relation.
17	A file was expected.
18	Field types are incompatible.
19	The field is not indexed.
20	An "=" was expected.
21	The procedure does not exist.
22	The fields must belong to the same file or subfile for a sort or graph.
23	"<" or ">" expected.
24	";" expected.
25	There are too many fields for a sort.
26	The field type must be alpha, date, or numeric.
27	The field must be prefixed by its file's name.

Table E-1 (continued)
4th DIMENSION procedure error codes

Code	Reason
28	The field type must be numeric.
29	The value must be 1 or 0.
30	A variable was expected.
31	There is no menu bar with this number.
32	A date was expected.
33	Unimplemented command or function.
35	The sets are from different files.
36	The filename is bad.
37	":=" expected.
39	The set does not exist.
40	This is a function, not a procedure.
41	A variable or field belonging to a subfile was expected.
42	The record can't be pushed onto the stack.
43	The function can't be found.
44	The procedure can't be found.
45	Field or variable expected.
46	A numeric or alphanumeric argument was expected.
47	The field type must be alpha.
48	Syntax error.
49	This operator can't be used here.
50	These operators can't be used together.
51	Unimplemented module.
54	Argument types are incompatible.
55	A Boolean argument was expected.
56	Field, variable, or file expected.
57	An operator was expected.
58	")" expected.
59	This kind of argument was not expected here.

Table E-2 lists the code returned if there are too many procedure calls or too many records pushed on the stack.

Table E-2
4th DIMENSION stack error code

Code	Reason
–9996	The stack is full.

Table E-3 lists codes returned because the user has caused an interruption or accessed something to which he or she does not have password privileges.

Table E-3
4th DIMENSION user error codes

Code	Reason
1006	Program interrupted by user. User pressed Option-click.
–9991	Privilege error.
–9992	Wrong password.
–9994	Serial interruption generated by the user.

Table E-4 lists codes returned because of problems in input or output. This includes errors in serial communications and errors when accessing the disk.

Table E-4
4th DIMENSION I/O error codes

Code	Reason
52	Serial port timeout.
–9990	Serial port timeout.
–9994	Serial communication interrupted by user. User pressed Option-space.
–9995	The limit of the demonstration version has been reached.
–9997	The maximum number of records has been reached.
–9998	The index key already exists. The entry is not unique.
–9999	Not enough space on disk to save the record.

Table E-5 lists codes returned because of problems due to damage detected in the database. These are serious errors. The user should be alerted to back up the database and repair it by using 4D Tools.

Table E-5
4th DIMENSION error codes for damaged database

Code	Reason
53	Index out of range.
–9989	Invalid structure.
–9993	Damaged menu bar.
–10000	Invalid data address.
–10001	Invalid index structure.
–10002	Invalid record structure.
–10003	Record # is out of range.
–10004	Index block # is out of range.

Table E-6 lists codes returned by the Macintosh File Manager. These codes can be returned when you are using the document commands.

Table E-6
Macintosh File Manager error codes

Code	Reason
–33	File directory full.
–34	All allocation blocks on the volume are full.
–35	Specified volume doesn't exist.
–36	I/O error.
–37	Bad filename or volume name.
–38	File not open.
–39	Logical end-of-file reached during read operation.
–40	Attempt to position before start of file.
–42	Too many files open.
–43	File not found.
–44	Volume is locked by a hardware setting.
–45	File is locked.
–46	Volume is locked by a software flag.
–47	File is busy.
–48	File with specified name and version number already exists.
–49	File already open.
–53	Volume not on line.
–54	Attempt to open locked file for writing.
–61	Read/write permission doesn't allow writing.

Table E-7 lists codes returned by the Macintosh Printing Manager. These codes can be returned during printing.

Table E-7
Macintosh Printing Manager error codes

Code	Reason
–1	Problem saving file to be printed.
-17	Module cannot be implemented.
-27	Problem opening or closing connection with printer.
-128	Printing interrupted by the user.
-4100	Printer connection has been interrupted.
-4101	Printer busy or not connected.
-8150	A LaserWriter is not selected.
-8151	The printer has been initialized with a different driver version.

Table E-8 lists the code returned by the Macintosh Memory Manager. This code could be returned if you are running low on memory during printing.

Table E-8
Macintosh Memory Manager error code

Code	Reason
–108	Not enough room in heap zone.

Table E-9 lists the codes returned by the Macintosh Resource Manager. These codes could be returned if you try to load a 'SND ' resource that does not exist.

Table E-9
Macintosh Resource Manager error codes

Code	Reason
–192	Resource not found.
–193	Resource file not found.

Table E-10 lists the NaN codes returned by the Macintosh. NaN stands for "Not a Number." It is a Standard Apple Numeric Environment (SANE) representation and appears when an operation produces a result that is beyond SANE's scope.

Table E-10
Macintosh SANE NaN messages

NaN code	Reason
1	Invalid square root.
2	Invalid addition.
4	Invalid division.
8	Invalid multiplication.
9	Invalid remainder.
17	Converting an invalid ASCII string.
20	Converting a Comp type number to floating-point.
21	Creating a NaN with a zero code.
33	Invalid argument to a trig function.
34	Invalid argument to an inverse trig function.
36	Invalid argument to a log function.
37	Invalid argument to an xi or xy function.
38	Invalid argument to a financial function.
255	Uninitialized storage.

INDEX

⌘ Command
⇧ Shift
⌥ Option
⌃ Control
Ctrl

Keys
D168

Symbols
L334

≤ ≥ 326-327 Character Reference

$0 57 Function (Return Value)

Index

Cast of Characters

A

>> Pointers Option Shift 1 88
<< Option 1 88

B

C

D

E

F

M

N

O

P

Q

R

S

T

U

V

W

wild card @ 196

Y

Z

$Ø Pago 57

Index to the Commands

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Upgrade Addendum

Y

218 AUTOMATIC RELATIONS (one; many)
381 IDLE
134 NEWDATA FILE (document)
134 OPENDATA FILE (document)
371 Self > Pointer
253 External window (left; top; right; bottom; type; area;) > Longint
210 Modified Record ({file}) > Boolean
371 Nil (variable) > Boolean
363 SET ABOUT (item; procedure)

Errors

Date	Error	Comment
18th June 1998 Thurs	This seems to be a endless loop.	arises when 2nd 4th Dim Document Doubled Clicked from Outside, ie in Finder
Last Week	Using (4D Tools™) Major (Serious Data Damage)	Importing Text file from Excel ?
18 June 1998	What seems to work. Import Text from Excel Export as Text from 4th Dim & Re Import.	
18/19 June 1998 Night	4th Dim Tools Serious Damage to ? ? (after? Compact (before)	Seems to arise when Lay Outs etc Deleted from/during Application Development Unneeded items
	Application seems OK to compact Once items ie Lay Outs are not Deleted	
10th July 98 (b)(c)	4th Dim Tools on workbench on MacTest (a) Endless loop, Trace, DWS Trace (Crash)	
9th Mar 01	(Open Data Base) (Open Data file)	+ over used, ie opened/used a Number of times with in Running of Application. Application Crashes