

One Step Up

Development with Omnis 7

Editor:
Hallvard Lærum

Contributing Editors:
Amund Haldorsen
Bjørn Borg Kjølseth

Translated by
Stephen Timmons

© 1994 - AlphaBit a.s
ISBN 82-91465-01-0

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior permission of AlphaBit a.s.

This book was typeset on a Macintosh computer, using Microsoft Word™ for Macintosh (v5.1). The following typefaces were used: Helvetica Narrow, Berkeley Old Style, Nuptial Script, and Wood Type Ornaments 1, all from Adobe. Most of the figures were created in Deneba Canvas™ (v3.0 and v3.5) and copied into Word in EPS format. All graphs were created in Kaleidagraph™ (v2.0) from Abelbeck Software.

Omnis is a registered trademark of Blyth International. Macintosh is a registered trademark of Apple Computer Inc. Canvas is a registered trademark of Deneba Systems Inc.. Kaleidagraph is a registered trademark of Abelbeck Software. Word, MS-DOS and Windows are registered trademarks of Microsoft Corporation.

Printed in Oslo, Norway
by
Trykkeri a.s.

© 1994 AlphaBit a.s
ISBN 82-91465-01-0

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior permission of AlphaBit a/s.

This book was typeset on a Macintosh computer, using Microsoft Word™ for Macintosh (v5.1). The following typefaces were used: Helvetica Narrow, Berkeley Old Style, Nuptial Script, and Wood Type Ornaments 1. Most of the figures were created in Deneba Canvas™ (v3.0 and v3.5) and copied into Word in EPS format. All graphs were created in Blabla Kaleidagraph™ (v2.0)

Omnis is a registered trademark of Blyth International, Macintosh is a registered trademark of Apple Computer. Canvas is a reg. trad. of Deneba, Kaleida of blabla. Word, MS-DOS. Windows are registered trademarks of Microsoft Corporation.

Printed in Oslo, Norway
by
Trykkeri a.s.

Aller siste side, nederst sentrert.:

ISBN 82-91465-01-0

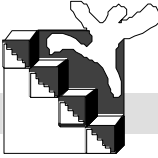


Table of Contents

IntroductionI	Layout & the User Interface
Introduction	A Good Design
Prologue	Effects
How the Book is Put Together	Logical Arrangement of Menus and Windows
How to Use the Table of Contents	
Before You Begin	
How This Book Came About	
Glossary	
General Methods II	The Database EngineIII
The Planning Phase	Data Structure: Memory & Hard Disk
General Considerations	Disk
Evaluation of Data	Some Basic Terminology
Comments and Labeling	Structure of Data on Hard Disk
Working with Procedures	Datafiles
	File Formats
	Records
	Fields
	Indexes
	The Function of the Internal Memory
	How the Internal Memory is Organized
	Variables
Debugging	Field Types & Their Function
General Considerations	Introduction
Window or Procedure?	Text Fields
How the Procedure Window Is Built Up	Number Fields
Finding the Right Procedure	Boolean Fields
Manipulating the Way Procedures Are Run	Date Fields
Viewing and Altering the Values of Variables and Fields	Time Fields
Examining the Sequence of Procedures	Picture Fields
Finding Clutter in the Application	Lists
Debugging in Multi-user Mode	Binary Fields
In Conclusion	Sequence Fields

File Connections	Lists & Tables
In General	What Is a List?
Types of Connections	List Settings
Different Ways of Linking	Manipulating lists
Files	Displaying Lists in Windows
Commands for Editing Data in	Displaying Single List Values
a Datafile	Lists Stored in Data Files
Modifying Contents in a File	Lists Within Lists
of a Higher Level while	Redrawing Lists
Modifying Another File	Binary Search in Lists
	Tables
Elements of an	
Application.....IV	Data Input..... V
Datafiles & Libraries	The Ins & Outs of Enter Data
Application Structure	What Is Enter data?
Opening and Closing Libraries	Entering Enter data
Controlling Datafiles	Tasks for Enter data
CRB and Datafiles	Exiting Enter data
Opening and Closing Datafiles	Canceling Enter data
Example of a Datafile	Enter data on Several Levels
Handling Procedure	Modeless Enter data (v2.x)
Sequence of Procedures	Import & Export
Introduction	Introduction
Field Procedures	File Types
Window Control Procedures	Standard Export Tool
(WCPs)	Export Via Reports
Library Control Procedure	Exporting to Word Processors
(v2.x)	Sequence Numbers
The Timer Procedure	Importing Connected Files
The Jig-Saw Model	The ‘Import field from file’
Windows in the Jig-Saw Model	and ‘Import data’ Commands
Table Fields in the System	Update or Insert New Record?
(v2.x)	
Set Next Action (SNA)	Data Output..... VI
Queue action	Communication VII
Procedure Stack	
Tables of the Jig-Saw Model	3rd Generation
Tables - Macintosh	Programming..... VIII
Tables - Windows	
“Events” as Evoking Factors	External Routines
(Macintosh and Windows)	Introduction

- What Is an External Routine?
- Examples of External Routines
- Can I Create an External Routine?
- Where Do External Routines Have to Be before I Can Have Access to Them?
- When Should I Use External Routines?
- Which Functions in Omnis Are Used in Connection With External Routines?
- What Functions May I Use in an External Routine?
- What Does the Code in an External Routine Look Like?
- The Stack Problem
- What If I Want to Know More?
- Difficult Words
- An Example with Source Code

Introduction to Notation

- What is Notation?
- The Branched System
- How to Write Notational Expressions
- Syntax and Debugging
- Windows and Notation

Special Topics..... IX

Keyboard Shortcuts

- Using Keyboard Shortcuts
- Increasing the Number of Potential Hotkeys (v2.x only)
- Standard Hotkeys
- Word Processing Techniques
- Hot keys in v3.0

Beyond the Tricky Bit ...X



Introduction

Prologue.....	2
Frustration: Act I	
Frustration: Act II	
The work of a developer involves more than code	
How the Book is Put Together.....	4
Form and content	
Structure	
Layout and design	
How to Use the Table of Contents.....	6
Before You Begin.....	7
How This Book Came About.....	8

Prologue

This book is meant for you, an Omnis developer. Its aim is to explain important principles, help you when you get stuck, and perhaps be what it takes for all the pieces to come together when Omnis appears hopelessly complicated.

Frustration: Act I

Programming in Omnis isn't always that easy. Everything looks so simple and tidy to begin with. For many of us, our first encounter with Omnis is one of sheer enchantment. We're easily won over by the enormous possibilities that this developer's tool represents, and our creative urges are so difficult to harness that we can kiss good-bye to a good night's sleep for a while. Windows, reports, menus – everything is there, waiting to be used. Nor is there any reason to hide the fact that Omnis *truly is* a unique development environment. Nevertheless, rough times lie ahead. Maybe it's the initial enthusiasm, or maybe it's just our unfamiliarity with the product, but sooner or later we're sure to run up against a brick wall: at some point we won't be able to get a certain procedure or window to work the way we want it to. This is the moment of frustration. Our infatuation begins to cool and our mood turns to gloom. The workday once again becomes a bleak prospect, and Omnis gets all the blame!

Frustration: Act II

Hopefully, this is where this book can come to your rescue. Its aim is to help you get past the steepest part of the learning curve and get a firm grip on the many aspects of a developer's work as rapidly as possible. Once you clear this first, intimidating hurdle, everything will begin to go a lot more smoothly.

Maturing as a developer is more than just mastering various functions and commands; it's also about understanding what goes on *inside* the program, about what makes Omnis tick. So this book is geared more toward imparting a basic understanding of the principles

of Omnis, and less toward dispensing a few highly technical procedures.

The work of a developer involves more than code

Because we wanted to cover every aspect of a developer's work, this book contains chapters on planning and debugging. These subjects may seem like side issues, but in their own way they are every bit as important as the other topics that are dealt with. Moreover, database development is a great deal more than merely transferring data to and from various storage media. The ultimate goal of any application is to provide a forum where information may be stored and retrieved in some rational way. The interplay between application and user is an inescapable link in this chain. Yes, the application must function well here, but quite a different set of factors determines how successful the *user interface* will be. For this reason I've also chosen to deal with these more "artistic" aspects, in a separate chapter entitled "Layout and the User Interface."

How the Book is Put Together

Form and content

“One Step Up” is for the more advanced developer, who will already have plowed through the instruction manuals that came with Omnis. Further, it is meant as a supplement to the handbooks; selected topics are given detailed treatment, and from time to time information will be provided that you won’t find anywhere in the manuals. To understand the descriptions and follow the thrust of the discussion, you’ll need a working knowledge of the most important commands and their functions. Many of the paragraphs are illustrated with explanatory procedures and figures. You are free to use any of these procedures in your own applications; they’ve been thoroughly tested.

When setting out to learn the ropes in Omnis, the developer is faced with mounds of reading material. The technical content calls for a style of writing that puts a premium on correctness and precision; however, this often results in dry, vapid, tedious prose. The reading can be heavy going indeed! In view of this fact, I have tried my best to write in a style that is more direct and personal. On occasion I confess to having cut some linguistic capers, which you mustn’t take too seriously; the book was written with passion. Needless to say, the more colorful remarks should be taken with a grain of salt. At the risk of appearing dogmatic and too categorical, I have tried to cut my way to the bottom line through the dense and luxuriant forest of potential that Omnis represents and give you my personal recommendations based on practical experience.

Others’ experience may well be at odds with our own. No problem; this is as it should be. A good wholesome discussion never hurt anybody, and the one who stands to gain the most is the end-user. We welcome your viewpoints, suggestions for improvement, or requests for subjects you’d like to see dealt with.

Structure

“One Step Up” is divided into sections that correspond to main areas, which will be expanded from time to time with new chapters that address each problem area. These will be sent to subscribers on a continuing basis. Every time Omnis appears in a new version, those chapters of our book that need updating will be rewritten and sent out in the same manner. We have decided to use a 3-ring binder system in which chapters can be replaced individually, as needed, with new, updated versions. For this reason, each chapter must be numbered separately. Because new ones are always being added, the page numbering must take this into account. So the chapters themselves will not be numbered; rather, our hope is that a logical division of the contents and a lucid, well-arranged Table of Contents will serve as a satisfactory guide to the main topics you are looking for.

Layout and design

Paragraph prompts appear in the margin, in italics. The procedure examples follow directly after their respective subjects, framed and with the procedure title shaded gray. Where the procedure lines exceed one line, an ellipsis (...) indicates that the next line is a continuation of the previous one. A number of tips are scattered throughout the book; these appear in frames and reflect the subject matter that precedes them. Commands, format names, etc. that originate in Omnis are enclosed in single quotation marks, e.g.: ‘Load from list.’ A smattering of conclusions and personal comments of a more or less serious nature also appear here and there; these are set off by ornaments, like the following:



Could anything be as fun as programming with
Omnis?



How to Use the Table of Contents

Look for the main subject area in the Table of Contents. Grouped under each section is an overview of the chapters currently available under these main headings. There are no page numbers in the Table of Contents, because page numbering starts anew with each chapter. However, in the book proper, there is a footer at the bottom of every page that tells you which chapter you're in; and each chapter is separated by a colored divider. At the beginning of each chapter there is a complete Table of Contents (this time with page numbers); this is the place to look for specific page numbers.

A glossary at the back of Section 1 provides descriptions and definitions of a few difficult terms and abbreviations. In general, however, each concept and term will be explained along the way.

Before You Begin

It's time to roll up your sleeves and hunker down at the keyboard. If you haven't read "Introducing Omnis" and "Tutorial" (and browsed through "Reference 1" and "Reference 2") you should do so. Who said life was a bowl of cherries, anyway? You need to have some idea of what each command is for. Keep the manual close by, and don't hesitate to use it; there's an awful lot to remember. Try following up the tips on your own, using the resources of Omnis. That way you won't be as likely to forget, and you'll find out soon enough if you've understood the tip correctly.

Use test applications!

By all means, jump right in and test the waters – but preferably not with something you intend to sell or deliver. An application programmed during the learning phase is never very good; it can quickly become an impenetrable maze and a chore to work with.



It's often easier to start from scratch
than to work back from the other end.



How This Book Came About

“One Step Up” was initially conceived as an advanced course in Omnis way back in June of 1992. However, it proved to be impossible to limit the documentation I started out with. The fact is, there were just too many exciting things to write about! Bjørn Kjølsøth, Managing Director of AlphaBit, felt we should enlarge the project and make it into a book. With Bjørn’s enthusiastic support and professional counsel, work proceeded on a growing number of chapters. Needless to say, much of the work involved persistent testing and programming.

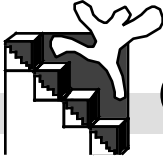
With the demanding learning phase still fresh in my mind, I wanted to write as simply and intelligibly as possible and supplement these efforts with appropriate figures – no easy task, as it happened! The work dragged on, but AlphaBit (Bjørn was indomitable) kept the faith. Meanwhile a welcome guest writer came on board in the person of Amund Haldorsen (who now has his own company in Porsgrunn); Amund wrote the important chapter on external routines.

The 1st edition of “One Step Up” was published in Norwegian in June, 1993. Since that time, most of the chapters have been expanded and updated; in addition, new chapters have been added. This was the first in-depth book to appear since Omnis 3 plus. After a major revision, including many new figures, a 2nd Norwegian edition is about to go to press. This 1st English edition is a translation of the 2nd Norwegian edition. Stephen Timmons has done an outstanding job of translating the book in the style and spirit in which it was written; he has also helped with the layout. I can’t count the number of hours he and I have sat and worked to make sure that the content was not only technically correct but was expressed in language that lives and breathes.

I also want to thank Geir Fjærli for designing the cover, and for all other help he has given to this project. Thanks are also in order for Birger Urbye, for his assistance in the printing of the book. And a hearty thanks to our expert on external routines, Amund Haldorsen, for his contribution. Finally, I want to thank all those terrific people at AlphaBit, and Bjørn in particular, for his faith in the project.


Oslo, May 4, 1994

Hallvard Lærum



Glossary

ACP	(Re v1.x) Abbreviation for Application Control Procedure. A user-defined procedure that runs in conjunction with all active fields, windows and menu lines within an application, provided the ACP is activated. See description of the 'Set application control procedure' command in "Programmer's Reference."
Alphanumeric	Refers to all the characters that can be written. The order in which alphanumeric characters are arranged and numbered varies according to the operating system. For example: ANSI, ASCII. As a rule, numbers (digits) come before letters. The order may be further investigated with the function 'chr().'
CRB	Abbreviation for Current Record Buffer. The CRB is defined as all the fields in all file formats in an application or a format library.
LCP	(Re v2.x and v3.x) Abbreviation for Library Control Procedure. LCP functions like ACP in v1.x, except that you may choose whether a procedure shall apply to the entire application or only to a library. LCP may be set so that it is runs even when all the windows are closed.

OPT/RB Our own abbreviation for Option/Right [mouse] Button. Where the Mac uses the option key (marked with the symbol ) , Windows uses the right mouse button. In Omnis, you may click with OPT/RB on just about anything.

Procedure address Every procedure may be evoked by some other procedure by means of what we term a “call.” However, the ‘Call procedure’ command requires an unambiguous address for the location of the procedure in question. This address will consist of the format name, “/,” followed by the procedure number within the format. The procedure address is written below in boldface type. The procedure name crops up automatically in braces.

Call procedure pProcedure/3 {Do something funny}

Call procedure mMenu/15 {Do something nice}

Call procedure wWindow/239 {Build a list}

RSN Abbreviation for Record Sequence Number. This field type is called “Sequence” in the field type list in the window where the file format fields are defined. In this book, the acronym “RSN” is used consistently in the name of all the sequence fields. The sequence number fields from the various files are kept separate by means of the first letters in the file name. (“SEQ” is another term used for the field name of the sequence field; but we have stuck to “RSN” because this is what the handbooks use.)

WCP Abbreviation for Window Control Procedure. This is a user-defined procedure

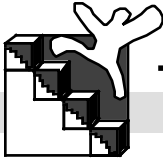
that is run in conjunction with all the active fields in a given window, as well as when moving between windows. WCP belongs to the window that is front (and active) when the 'Set window control procedure' command is given.

Window field

A type of field that occurs in windows. I have chosen the term "window fields" instead of "fields" to distinguish it from fields in file formats. Entry fields, Pushbuttons, List fields and Popup menus are all window fields.

Chapters:

1. The Planning Phase
2. Debugging
3. Layout & the User Interface



The Planning Phase

General Considerations	2
Establishing priorities	
Programming styles	
File structure	
Having the user's best interests at heart	
Evaluation of Data	10
Converting real-life information into file formats	
Fields and variables	
Comments and Labelling	20
Procedures	
Format names	
Field names	
Working with Procedures.....	31
Sorting and calling procedures	

General Considerations

Establishing priorities

After determining how the application is to be used, you should make a list of features, taking care to distinguish between those that *must* be included, those that should *ideally* be included, and those that would be *desirable* to have. Resist the temptation to include features that are technically impressive and offer great potential from a programmer's point of view but serve little or no functional purpose; even worse, they tend to consume an inordinate amount of time.

The list of features

Make your features list as brief as possible. Remember, you are aiming for an integrated whole, and every command must work – and work as intended, even in the unlikeliest of scenarios. Listen to those who will actually be using the product; this will make it clear which features should head your list. Those that wind up at the bottom of the list can – and should – be eliminated! They can always be added later, if the need should arise, after the application has undergone a debugging.

Programming styles

When starting out, try to anticipate future needs, especially those you feel sure are inevitable at some point. Although there is no denying that foreseeing and making provision for future developments is a mental challenge, you will simplify your task and build in safeguards by programming with few assumptions. Take nothing for granted; it's your job to ensure that the parameters and settings used in each and every procedure are just right. See the examples that follow:

- Set main file before every procedure containing commands that are affected by main file, and after every Enter data.

- Set Current list first in every procedure that handles lists and after every Enter data if the ensuing procedure lines contain commands that handle lists.
- Assign only one function to a given variable (no matter how insignificant).
- Keep the prerequisites for the naming of formats, the placement of various types of procedures, etc. to a minimum.
- Program procedures or windows that can be used in a variety of contexts – i.e., general procedures.

Few assumptions and heavy coding vss many assumptions and light coding

This kind of approach is diametrically opposed to the programming theory that calls for simplification wherever possible, for the use of programming procedures that are specifically geared to each task. “Applied” programming is much simpler, because the programmer is not concerned with whether a procedure is compatible with anything other than the specific information at hand. Most procedures, in fact, are of this type. Nevertheless, you would do well to employ general procedures for tasks that tend to recur but where the context varies somewhat. For example, you might, want to use the same window to display different lists at different times; but such a window will have to take into account the fact that the current list will change.

General procedures

General procedures, being difficult to program, are best left for the experienced developer. The whole point of using general procedures is to save time; if you use too much time trying to get them to work as planned, you will defeat your purpose. And if you have to establish a whole slew of restrictive rules in order to get a general procedure to work (for example, rules for naming and calling procedures), this will only complicate further development work. It is no easy task to fit a new feature to a set of rules formulated at an early stage in the development process.

Be especially careful not to make indiscriminate use of the Library Control Procedure or Window Control Procedure in this way. It will only result in “stop-gap” programming and at the very least prove hard to build upon! It is never easy to second-guess exactly where an application will ultimately wind up; but if you can strike a happy medium and avoid the potential pitfalls of both approaches, you will have succeeded admirably.

File structure

Decide on what file structure, fields and connection diagrams you want, and then sketch them out. A general overview of the database structure is indispensable for an effective database. Without it, you will never get the application to work properly. If the file structure is poorly designed, the result will be a data file that is larger than necessary, and searches will take longer than they should.

Understanding a file structure

In Omnis, as you know, the ‘Find’ command causes a connected record in the file (parent file) above to be read into the CRB. Files farther up in the hierarchy are read in manually, not automatically. The file overview provides quick and reliable information as to when it is necessary to locate records (manually) and the proper way of going about it. In fact it is the key to understanding how the application works.

Presenting a file structure

Any drawing program worth its salt can be used to present a file structure diagrammatically. Once you’ve drawn the symbols and the various types of connections, you can copy them and compile new file structures with ease. The following example (see next page) shows one way of sketching files and the connections between them.

Hierarchical file connections

Parent files appear above child files and are connected by a black vertical line, as shown in Figure 1.

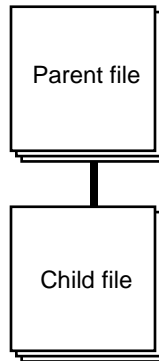


Fig. 1 Hierarchical connection shown by placement and a simple connecting line

Relational joins

The files are (preferably) on the same level and are connected by a gray, horizontal line. In addition, a small, black arrow points toward the file to which the connection is being made – that is, from “many” to “one” in a many-to-one connection.

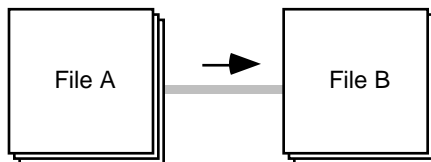


Fig. 2 Relational join shown horizontally by an arrow and a gray connecting line.

Alternatively, you can use a symbol for branched lines that merge into one to show how the files are related.

The branched portion comes from “many” and leads to “one,” as shown in Figure 3:

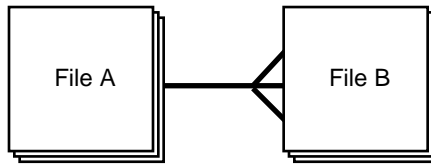


Fig. 3 Relational join shown by a branched line

Copying between files

Files that copy part of their content to another file are indicated by a gray, thick arrow that points in the direction of the copy. Refer to Figure 4.

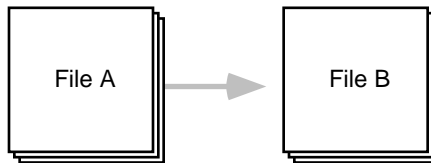


Fig. 4 Copying between files shown by an arrow

Designing a file structure

Figure 5 shows the file structure in a hypothetical application for a consultancy firm. The employees, fEmployees, have consultations with their customers, fCustomers, who belong to their respective companies, fFirms. This sketch makes use of symbols described above, and gives a fairly coherent picture of the interrelationships.

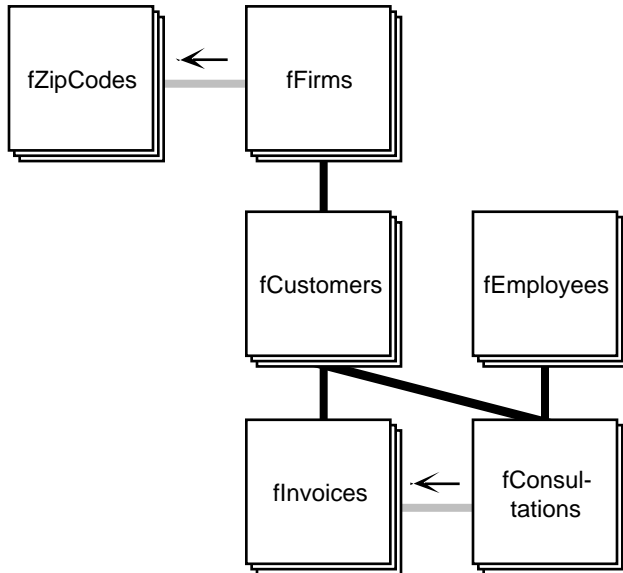


Fig. 5 A sample file structure

Analyzing the file structure in Figure 5

Here we have an example of hierarchical connections in three generations. The files fInvoices and fConsultations are both connected to fCustomers, which in turn are connected to fFirms. This means that a regular 'Find' with fConsultations or fInvoices as Main file will cause the connected record in fCustomers to be found as well. We subsequently get the connected record in fFirms with the aid of the 'Load connected records (fCustomers)' command. If we execute a 'Find' with fCustomers as Main file, then the connected record in fFirms will be found, just as for the generation below. The relational joins inside the diagram are between fConsultations and fInvoices and between fFirms and fZipCodes. Many consultations are linked to one invoice, and many firms can belong to one postal zone. If we combine these connections, we see that fConsultations are connected to fZipCodes in four stages. This means, for example, that we can

locate all the consultations that have been made with firms in any given postal zone. Moreover, all the information in the files between these outer points are accessible for analysis.

When programming, we work with one file at a time, and take into account connections to and from this file. The file structure itself tells us how the files are interrelated and which factors we must take into account as we go along.

Having the user's best interests at heart

The first rough drafts of how the windows will look, along with the kind of aids for entering data you have in mind, are all important considerations for the developer. Your sketches (or prototypes) will provide a basis for determining which tools will have the desired effect. A “defensive driving” mentality will stand you in good stead here. Remember, for the end-user, economy of movement is all-important. The value of minimizing the number of keystrokes and avoiding excessive use of the mouse cannot be overestimated. Developers, as a rule, are oblivious to the frustrations and irritation that users feel on account of some seemingly minor failing that could easily have been avoided but which, instead, has made registering the 5000th patient (or customer) an irksome chore. Often the problem is only a matter of “fine points.” The savings for the user, in terms of time and ease of use, may seem meager when viewed in isolation, but they add up in the long run. When all is said and done, it is the end-user who bears the brunt of the workload where databases are concerned.

Helping the end-user

Part of the philosophy behind providing the end-user with a good work environment is that he or she should not have to remain in deep concentration the whole time. Let there be no doubt about it: every extra reminder, every undo option, and every well-conceived shortcut is sure to be appreciated! Omnis gives ample scope for spoiling the end-user, but it's up to the developer to exploit the potential for doing so. In any case, the developer needs to work closely with the user. If you as a developer fail to put yourself into the

user's shoes, all your window reminders and shortcuts will not make good on their promise; in fact, the application itself will lose much, if not all, of its usefulness.

The beauty of foresight

Generally speaking, the planning phase is often the most overlooked and least emphasized of the entire development work. And yet it is just here that you can save yourself the most work and where the fate of the application is most likely to be decided. An hour's worth of concentrated brainstorming can save you many days worth of programming, because you will hit upon new ideas, discover new ways of doing things, and will be able to anticipate potential problems. When you haven't yet written a single line of code, changing course is a piece of cake! Your application will also be tidier, easier to debug, and the programming work itself will go a lot more quickly. The crux of all your creative and systematic mental work is careful planning. And this is done *before* procedures and variables enter the picture and complicate matters. From this point on, it's smooth sailing all the way – until something totally unexpected crops up – something that could also have been avoided with careful planning!



You program better when you keep
the big picture before your mind's eye.



Evaluation of Data

Converting real-life information into file formats

Part of your job as a developer is to evaluate how real-life information can be translated into file formats in such a way that it forms a systematic whole. This is often easier said than done. You need to determine which bits of information can be put together to form a whole, with as little redundancy as possible. If the same names or postal zones keep cropping up in separate records within a file, this indicates that some of the fields should have been taken out and placed in connected files of their own. Wherever several distinct bits of one type of information can be joined with one particular bit of another type of information (and where these recur in the data) these can all be put in their own separate files and linked. Repeat this to yourself 20 times, or read the example that follows.

Example of many connections

There are 20 counties in Norway, each of which is subdivided into municipalities. In each municipality there are a certain number of medical clinics, each with a certain number of doctors with their respective patients. The patients have all visited their doctor more than once and have had different kinds of examinations during each visit. Working upwards from the bottom, the files could be connected like this: Individual examinations – Consultation – Patient – Doctor – Medical clinic – Municipality – County, with one file for each generation. “Individual examinations” would be the child file, and “County” would be the great-great-great-great-grandparent file. Each file represents a unit in which the fields in each file are closely related in some meaningful way. If we had collected all the fields into a single file, we would then have had to store the name of the county (which belongs at the top) together with every single examination the patient had undergone from his doctor. The result would be reams of the same information copied over and over again. All the hierarchical levels above “individual

examinations” would expand exponentially, and the whole country would soon be awash in data!

File connections

Besides saving space on the hard disk, connections provide complete structural flexibility. An unlimited number of records in the child file can be connected to a specific record in the parent file; the same thing applies to each generation upwards. This makes it fairly easy to enter all kinds of information of varying types and amounts. Connections also give broad scope to the database engine in Omnis for manipulating data in searches, for statistics, and for the generating of reports. Information stored in the form of connections is more accessible for computers, generally speaking. Connections also facilitate the creation of indexes, which can be searched rapidly. Only human beings are going to understand free text anyway.

Should I put all the fields in one file?

On the other hand, there’s no denying the fact that searching for small bits of information spread out over many different kinds of connected files takes more time than retrieving one record in one giant file, in one quick go. If, for example, you know with certainty how many or which examinations a given doctor carries out in the course of a consultation, the smart thing to do would be to put this information into separate fields. This will speed things up. However, if the doctor were to do an additional examination, there would be no place to record it. It’s all a question of weighing losses and gains. Any information about the user’s situation can simplify the programming dramatically, but only if it is reliable.

Coping with file connections

You needn’t be afraid of programming with connections, provided you have a clear grasp of the file structure. Hierarchical connections require you to think in terms of generations in twos, whereas relational joins require the active intervention of the developer

when connecting or locating records. A sensible analysis of the data will tell you where connections are called for and where they are unnecessary. Some developers tend to connect every file to every other file, but that's rarely advisable. It usually results in madcap file structures that are in total disarray; it will also overload your computer (think of all those indexes that will have to be maintained!). Similarly, your procedures will in all likelihood be disordered and hard to interpret.

Indexes

Indexes are meant to help the database find its way in the data file. To all intents and purposes, an index is an internal tree structure; however, we may think of an index as a list that contains the RSN (Record Sequence Number) for all the records in a file when the content of the indexed field is sorted alphabetically (alphanumerically). This list is the database version of a table of contents. There is one for every field that is indexed. When a record is added, the RSN is inserted in various places throughout the index lists, depending on where the content of each index fits. Indexes are used in sorting, doing searches, and in the generating of reports. Index searches are extremely rapid, because binary searches are used (more on this subject later).

Too many indexes

The misuse of indexes is a fairly common sin among developers, one that often leads to the maintaining of index lists that are never actually used. A moment's reflection is usually all it takes to realize which fields are realistic candidates for a search. You can also do searches of fields that are not indexed, but for voluminous files this is time-consuming. In version 1.x of Omnis 7, there is a limit of 12 indexed fields per file. With a view to enlarging the application at a later date, it is important to use your quota sparingly so that you don't run out of indexes just when you need them most.

Boolean indexes

If you are running short on indexes, it is useful to know that Boolean fields are generally the least auspicious candidates for indexing. If you cannot avoid sacrificing one of the existing indexes in a file format, Boolean fields should be the first to go. This is especially true if the field has only two possible values (for example, man/woman), distributed among as many records. In such cases there is little to be gained by indexing.

In version 2.x you can have an unlimited number of indexes, so the problem is not as acute. Nevertheless, you should choose indexes with care, because if you have defined too many, it can take time to build them up.

Non-indexed fields

Analyzing non-indexed fields and doing searches on them can be carried out with the aid of reports, because the user must wait a while for the printout anyway. Users are usually more patient with reports, because many calculations are performed and often presented here at the same time. Thus the time spent in waiting is less tiresome than waiting for active feedback concerning some bit of information within a window.

TIP: The essential thing is to acquire as much reliable information as possible concerning the amount, type and distribution of the data. Assumptions are never more than just that – assumptions, a fact that must be taken into account during the programming by building into the application the potential for expansion and variation.

Amount per unit

There can be a set amount of information per sorting unit (file) in the material gathered during street polls.

The questions on the questionnaire do not change. For every question there will be a reply that must be entered. There is no need for file connections, because no schematic variations are anticipated. Each question is simply entered as a separate field in the file format. In an application of this type we can, theoretically, make do with just one file.

(It is customary, however, to facilitate the entry of data from such polls by recording the answers as selections from among a fixed set of possible replies. This enables us to add up identical answers and present the results statistically.)

Type of data

The type of data is a key component of the file structure. The field definitions are directly determined by the type of data. Time and date belong in one of the variants of the date field. Numbers and values with different decimal places and digits are placed in a number field that has the correct number of decimal places and a digital capacity with plenty of slack. Completely dissimilar free text is put into a regular text field with an ample surplus of the number of characters allowed. You should preferably use the maximum number of characters, since it will not affect the demands on memory. Knowing what the various field types represent and how they function is a prerequisite for utilizing them effectively. The fruit of choosing your field types wisely is simpler programming, rapid searches, and optimal use of your storage medium.

Distribution

How data is distributed determines the degree of standardization. The more dissimilar the information, the harder it is to sort it into fixed categories. This applies to text in particular. The most common method of translating text into number code consists of turning “woolly,” free-text descriptions into clear choices from a pre-existing list of alternatives. Computers can cope with numbers and code; but when it comes to prose, they are illiterate.

“Dynamic” versus “historical” data entry

When two files are connected to each other, they are usually dynamically linked. In the case of customers that are connected to a firm, all the customers will be updated if, for example, the firm changes its name. So the information on all the customers gets updated all at once, thus tempering the confusion that a change of name might bring. The relationship between the files is seen to be a dynamic one.

We do not invariably want the data to change in this way, however. A typical example is the registration of accounts and sales. If a product is sold at a set price, then the price does indeed remain “set,” regardless of whether the actual price changes later. We may call this kind of data entry “historical.” Its purpose is to keep information unchanged. So information on price, type of product, and time of sale must not be separated. Their integrity can best be ensured by putting them in the same file. The price and type of product are usually taken from a price list; but if all the relevant information is to be preserved, these two factors must be stored together with the time of sale. In other words, we need to copy information from one file to another.



To fully exploit the potential of the database,
we must closely analyze the data in its entirety.



Fields and variables

After a careful planning phase, defining fields in the file formats is a breeze. Your attractive list of fields, neatly arranged in logical order, may well leave an indelible impression of total control and benevolent perfectionism. However, the order of arrangement and

the presence of any empty fields in the file definition is wholly immaterial. Your computer doesn't give a hoot about which fields appear where, because they are referred to by their respective numbers. The end-user never sees the field list as it appears in the file format. It is not necessary to exchange one field for another, or to change the order of arrangement; that only makes for busywork. But you may fill in any empty fields.

Deleting a field in the file format

If you delete a field, procedures that refer to this field will show ‘#??’ in place of the old field name. The procedure “forgets” what the actual field was, because its related token has been removed. If at some later date you enter a new name on the same line in the file format (after doing other things), it will *not* show up later in the relevant places with ‘#??.’ You will have to insert the missing field name yourself. If, on the other hand, you are merely altering an existing field name, the new name will show up wherever reference is made to the field in the normal way. Indirect references to fields (for example with the aid of the ‘fld’ function and a text variable) will in any case have to be recovered by the use of ‘Find and Replace’ – or be edited manually.

Is memory a problem?

A roomy definition of fields has no bearing *per se* on how much space is taken up on the hard disk and in the internal memory. Only when they are filled with data do fields take up space, whatever amount the data requires. If there are many fields and variables in circulation, they will consume memory as they are used in the various procedures. Fields retain their content until the latter is actively deleted (for example, during certain calculations or unsuccessful searches). Lists, as you might expect, are the most crucial factor. A developer with a boundless zeal for definitions and an insatiable appetite for endless lists might well run into memory problems; otherwise there is little cause for worry. Just delete the longest lists after use.

The Sequence field (RSN)

A sequence field is actually just a concrete realization of Record Sequence Number for the user, as this number always exists in memory. The sequence number is very handy, since it is the “physical” link between parents and children in a hierarchical connection. Moreover, it is a sure means of identifying each individual record (or separating them from each other, even if they should happen to be very similar). It

pays to make a habit of defining a sequence number for each file format. It is sure to prove useful.

User-designed variables versus hash (#) variables

The 60 global number variables that come with Omnis should prove useful in many situations. For example, they are admirably suited as log variables in loops with a certain number of rounds, and for use in intermediate calculations. They tell us nothing, however, about what they are used for. As a result, they will gum up the procedure and mangle it beyond recognition. Not only that, the procedure will soon be swarming with errors. It is also risky to entrust them with vital information, e.g. flag, current status, etc., because it is so easy to lose track of which ones contain what and of which variables are already in use. The more hints the developer gives himself about what is actually taking place in the procedure, the less there will be to remember all at once, and the development work will be less taxing and more rewarding.

Where to use hash variables

There is one area in particular in which hash variables are unexcelled, and that is when values are being transferred from one application to another. When an application with its own file formats and its own data files is opened in Omnis, all the information from the old application is, not surprisingly, no longer available. For this reason, Memory Only files cannot be used for this kind of task. The hash variables are not deleted until you leave Omnis completely. Accordingly, they are well-suited as temporary storage areas during the transition from one application to another. The #-lists are especially useful in this regard, because they can contain information from all kinds of fields and behave much like an ordinary data file.

Variable file

For many tasks you need your own variables. These can be defined in a separate variable file, set to Memory Only. The fields in the file will be global

variables, and you may assign them descriptive names. For years now, Pascal and C developers have made a careful distinction between global and local variables, primarily with a view to conserving memory. Nowadays, with memory-hungry operating systems such as Windows and System 7, computers are usually configured with such an abundance of internal memory that we no longer need to ration memory. There is more than enough room for any global variables you might want to define. Moreover, with one variable for each task, we can be certain that all the variables contain what we expect them to, and the procedures that employ them will soon be error-free and reliable.

There are two tasks for which variable files are particularly well-suited; one is date variables. Hash variables don't contain date variables. That's a pity, because date variables are quite useful, especially for searches and reports. Search formats designed to be used with date fields will be more manageable if you make comparisons with a date variable instead of a number variable. In addition, the presence of a date variable in an entry field will cause Omnis to check the validity automatically – i.e. whether what is being typed in can be interpreted as a date. The other task involves display lists. You should declare any lists that are going to be on display in a window. Doing so will make it much easier to remember each list's function.

Library variables

In v2.x, you can define library variables, which can be used in place of Memory Only files as explained above. You would do well, however, to have the definitions gathered in one place (in one procedure); this will save you from having to look for them later.

Local variables and format variables

For complicated procedures it is a great help to be able to use local variables for all intermediate calculations and constants. With smart labelling, the procedure will be as well-arranged as it can possibly be, and the difference will certainly be noticeable! The various

procedure lines reveal their functions relatively clearly (which is not the case when hash variables are used). Even debugging is made much easier, because the user will not have to remember the function of, say, 15–20 hash variables. If variables are to be used elsewhere in the menu or the window, the developer will have to resort to format variables. Variables that are used in procedures and presented in reports must be global. (See also the chapter entitled “Data Structure in Memory & Hard Disk.”)



Use your own variables – each with its own unique function.



Comments and Labelling

Procedures

It is not always clear what a given procedure is all about. Only after following the logical drift of the commands and learning what the different variables represent (and what they contain) will you be able to understand the process. Care in labelling coupled with a good memory will spare you a lot of work at virtually every stage of the development.

Comments

In procedures it is good strategy to write comments between the commands (but don't write a whole book!). Not everything needs explaining, especially if you are using your own variables. The important thing is to make a note of what you intend to do with the procedure, and perhaps make use of comments between the different "paragraphs." Brief but important points (for example, assumptions about format names and the limitations of a procedure, etc.) can be made in the form of comments in fixed places, e.g. procedure line 0.

TIP: The names of variables, formats, etc. written in the comment lines can be used to bring up the function menu (OPT/RB). This is a quick and handy way to obtain "active" fields in a procedure.

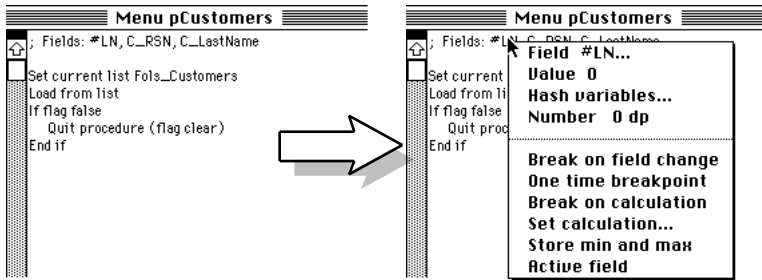


Fig. 6 Using comment lines to access information about a field

Naming a procedure

Finding good names for procedures is a real headache but well worth the effort, since good names can clarify an application markedly. Procedures often wind up doing more than they were originally intended to do, and incomplete and ambiguous names then become a major problem. Nevertheless, the question remains: How do you sum up an entire procedure in one brief sentence? Though personal preferences will always vary, every developer should strive to avoid superfluous phrases, at the very least. Short words are obviously preferable to whole phrases, and you can use incomplete sentences. But be careful with abbreviations. If someone else ever has to look at your application, all your so-called “self-explanatory” abbreviations will become a nightmare. Not only that, *developers have been known to forget their own abbreviations!*

Naming subprocedures before they are programmed

If you are in the habit of calling many small procedures from a main procedure, the logical thing would be to apply names to subprocedures before you start programming, which will break down your task into natural sections. This, in turn, will direct your programming in a natural way. This means, of course, that you will have to think through the entire procedure beforehand. A real chore, to be sure; but more often

than not it will result in better code. A certain amount of mental effort has got to be expended anyway; it might as well be sooner than later.

The numbering of procedures

Most applications wind up with a whole lot of main procedures and subprocedures. Programming is anything but a streamlined process! The procedure lists often exemplify this fact: the procedures appear helter skelter, in no apparent order. The titles don't always help us see which procedures and subprocedures belong together, because we only see the first 15–20 letters. Numbering the procedures goes a long way toward helping us find our way in a procedure list. For my own part, I usually use capital letters to signify main procedures, and begin with the letter A (within each format). I let the bullet symbol separate this capital letter from the rest of the main procedure title, as follows:

A•Monthly Accounts

The bullet symbol signifies that what follows is a main procedure. Each new main procedure is assigned a new letter (e.g. B, C, etc.). When you reach the end of the alphabet, begin again with double letters, like this: AA, AB, AC, etc.

The numbering of subprocedures

The following subprocedures are numbered consecutively: A1, A2, etc. Subprocedures don't need to be numbered correctly in terms of how they appear in the main procedure. The initial letters are separated from the procedure title by a period, as follows:

A1.Build FoLs_MonthAccount

Here we see the difference between main procedures and subprocedures, as well as where the various subprocedures belong. In other words, it is perfectly acceptable for procedures to lie strewn all about, as it were; that doesn't have to mean that we are going to

lose track of which procedures belong where. If there should be a need for sub-subprocedures (under subprocedures), number them with a lower case letter directly after the number of the mother procedure, as follows:

A1a.Get period

Still deeper levels can be indicated by alternating numbers and letters in each level, to keep them separate from each other.

Special characters and symbols in the procedure titles

Certain symbols, besides their use in numbering, can also suggest something of the content and nature of the procedure. Hidden away in the standard system fonts in both Macs and Windows machines are a variety of symbols that are suitable for this purpose. What is important here is that the developer choose his own symbols, so that he won't forget them. A little time and trouble here will reward you with a powerful and flexible system for the naming of procedures. It will also result in tidier applications and simpler debugging.

® “Registered trademark” – This symbol can be used to indicate a resource procedure that is called by many different procedures.

¶ “Paragraph” – This symbol can mean that the procedure has parameters and gives a return value. (Remember that this symbol customarily designates the RETURN Key in most word processors.)

⌘ It is useful to know whether a procedure contains Redraw windows or not, especially when it is being used as a resource procedure. The letter “r” can be used to indicate that the procedure does contain a ‘Redraw windows’ command.

☐ “System” symbol (in the Geneva font, size 10 pt.) can be used to designate distinct procedures, e.g.

collections of format variables, the Window Control Procedure, etc. For that matter, you may use any symbol you like, as long as it stands out in some way.

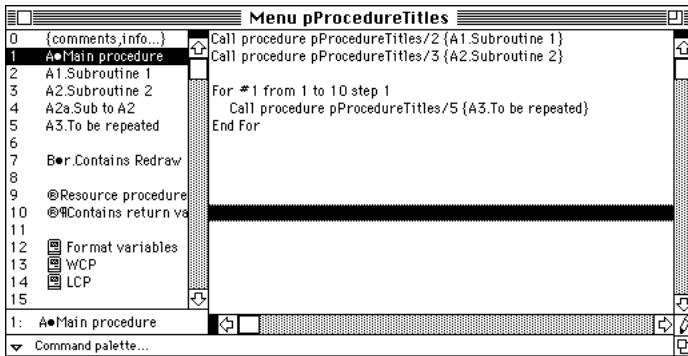


Fig. 7 Some of the naming conventions

“Cleaning up” procedures

Endless lists of procedure titles can be a chaotic sight. It is tempting to try to tidy them up by getting those that belong together to appear in logical sequence, and by separating different portions of procedures with blank lines. The first thing that happens if you do this is that the calls no longer work. You will wind up having to correct a number of ‘Call procedure’ commands before the application will work again. Not only that, it is often far from clear just where the different subprocedures belong when there is more than one procedure that calls them. All this makes for a lot of unproductive work. Collections of procedures will *never* be as tidy and well-laid-out as we’d like – nor do they need to be. Just try to keep procedures that belong together in the same general vicinity, and use a numbering system that you feel comfortable with.

Format names

All formats must have their own special name; but, unlike fields, they may always contain both upper and lower case letters. When windows are created automatically on the basis of file formats, the name “W_,” followed by a file name, will be suggested. It’s not a bad idea to let a letter designate the format type and put it first in the format name. Naming in fields is an analog process. Capital letters, however, are harder to read than lower case letters. In this case, the letter “W” does not represent vital information, so we can just as well use a lower case letter. Instead of using an underline character as a separator, we have chosen to put a capital letter first in the part of the name that follows, which gives us a net savings of one character, as follows:

wEmployees, fEmployees, rEmployees

Prefixes for format names

We suggest the following prefixes for the different format names:

m...	menus that will be installed
hm...	hierarchical submenus
pum...	pop-up menus
p...	menus used for procedure collections
w...	windows
iw...	import windows
f...	files
r...reports	
s...	search formats

The names of search formats

Finding good names for search formats is fiendishly difficult. If you insert all the search format conditions in the name, the result will be too long:

sField1EqField2_Field3greaterthan12_Field4lessthan99

A compromise solution would entail inserting only the field names:

sField1Field2Field3

You can also try assigning a name that describes the purpose of the search:

sFindCustomersInFirm

One of the biggest problems with this last method is that the name will often be an inept description of what the search actually performs; in addition, many names will tend to resemble each other too closely, and this could cause confusion. So the developer is left to brood and ultimately hit upon an appropriate name that is unique – and that’s hard!

Should we reuse the search formats?

The whole point of assigning descriptive names is to enable us to reuse them in other situations. However, you should ask yourself the following: Is this really necessary? Search formats don’t take up much room, and you can construct them in no time at all. As a rule, it takes less time to come up with new search formats than it does to agonize your way to a “good” name. The only requirement is that search formats must be identifiable by name. Therefore, you might just as well assign them a name according to the procedure address, and then number them consecutively within the procedure, as follows:

spProcedures5_1, spProcedures5_2, spProcedures5_3
swEmployees497_1, swEmployees497_2

The disadvantage of this method is that the names do not tell us anything about the content – which means that we’re not out of the woods *yet*.

Field names

Make a habit of assigning good field names. The more the name tells us about what the field is used for, the better it is. From time to time you will find yourself wishing you had an entire sentence at your disposal to do it justice. Until version 1.1 of Omnis 7, the limit for field names was 15 letters. In ‘Mixed Case/Long Fieldnames’ mode

we now have 255 letters at our disposal. That's all well and good, but the longer the field name, the longer the procedure lines. Screen space is not limitless, either. (Moreover, *developers hate scrolling with a purple passion!*) It pays to be as concise as possible and still convey the purpose of the field.

Non-unique field names

No two fields may have the same name. In addition, the developer needs to know which file format a given field belongs to. If the 'Unique Field Names' option in 'Preferences' has not been checked off, Omnis resolves the problem by putting the file format name together with the field name, as follows: `fFilename.FieldName`. This quickly leads to lengthy, ponderous calculations.

Unique field names

If you want to have short field names in your procedures, you should allow 'Unique Field Names' to be checked off. It will then be up to you to see to it that no two field names are the same. But this can be achieved by using the first letter (or first two letters) in the file format that are nearest the beginning of the field name; this will show us which file format the fields belong to. Unfortunately, such letter codes tend to make the whole field name cryptic or even render it unintelligible, unless they are separated from the rest of the name in some way or other. A space character cannot be used, but an underscore is permitted. Should a field name be typed in that contains a space character, this space character, as you probably know, will automatically be replaced by an underscore. In 'Mixed Case/Long Fieldnames' mode we can separate the abbreviation for the file format name from the rest of the field name by simply writing the code in lower case letters and the remainder of the name in upper case – or vice versa.

It's usually not all that difficult to remember which file formats had which prefixes in their respective fields. Don't forget, you can always view the file formats and

their fields in the List Field Names window, which can be brought up with CMND-9 or F9.

Abbreviations

After the separator, you must do your best to abbreviate as much as possible and still retain all the relevant information in the name. Eliminate superfluous or obvious information, e.g. “file,” “field,” etc. You will forget abbreviations that are too short or too numerous; and any fields that the developer doesn’t recognize are a major hazard! Consequently, you should try to settle on a set of standard abbreviations and stick to them; make a point of noting them down somewhere in the application. When dissimilar abbreviations are combined in a name, they should be separated visually, otherwise you won’t be able to make head nor tails of the names during a future revision of the application. This is all the more true if others will be looking at your application.

Example

Let’s try to abbreviate a really long field name. The unabbreviated field name reads as follows:

`'The_name_of_the_dog_who_belongs_to_the_boss's_wife_in_the_co
mpany'`

This field belongs to the file format fCompany. The abbreviated field name could be something like the following:

Upper Case:	CO_DOG_WIFE_BOSS	
	CO_DOGWIFEBOSS	(under
doubt)		
Mixed Case:	CoDogWifeBoss	
	CoDogWfBs	
	CO_DogWfBs	
	Co_DogWfBs	

The names of variables

Strictly speaking, you don't have to include the designation of the file name where the name of the variables in Memory Only files is concerned, because variables are usually collected in the same file. In this respect they are like local, format and library variables, which are all placed in their respective internal Omnis files. Nevertheless, it is important to distinguish between them, especially if the same names are used repeatedly (for example, in local and global variables).

We suggest the following prefixes:

me...	Fields in Memory Only files
gl...	Fields in Memory Only files (alternative prefix, abbreviation of “global”)
li...	Library variables
fo...	Format variables
lo...	Local variables
pa...	Parameters

The names of lists

Unfortunately, Omnis does not check to see whether you are giving a list in clear-cut list commands such as ‘Set current list.’ Lists are very specific types of fields, and you would be wise to mark them clearly. We suggest you mark them as follows:

(prefix)...Ls...	Mixed case
(prefix)...LS_	Non-mixed case

Examples, mixed case:

glLs_Companies
liLs_Companies
loLs_Customers
CoLs_Employees (File format fCompany)
DnLs_Notes (File format fDevelopersNotes)

Examples, non-mixed case:

GLLS_COMPANIES
LILS_COMPANIES
LOLS_CUSTOMERS
COLS_EMPLOYEES (File format fCompany)
DNLS_NOTES (File format fDevelopersNotes)

The names of Boolean variables

Boolean variables are often used as flags, switches, etc. Like lists, they stand out in some way or other. You may, if you wish, mark them in some special way, which means you won't have to use up characters in the field name for words such as "Flag," "Switch," etc., which are actually superfluous. See below:

(prefix)...Bo...	Mixed case
(prefix)...BO_	Non-mixed case

Conclusion

Some might think it pedantic of us to focus so intensely on the naming of fields and variables. But in fact, field names permeate the entire application, in every conceivable situation that might arise for the developer. Developers will do themselves a big favor by taking the creation of names seriously. It's all really just a matter of employing a few simple tools.

If you can remember the names of all the fields in all the files, this is a clear indication of excellent naming. This is a big advantage when it comes to putting fields in calculations, among other things, because typing in the field names manually is much quicker than using the List field names window (CMND-9 or F9) – *if* the developer is a skilled typist.

TIP: If you type in the first letters – just enough to fit one particular field name – Omnis will fill in the rest automatically. If the same letters fit more than one field name, the developer must select one of them from a list that appears. (This does not apply to calculations.) The entire naming system presented above is based on the assumption that it should be easy to remember the first couple of letters.

Working with Procedures

Sorting and calling procedures

The placement of procedures is often a reflection of individual taste, perceptions and personal bias. The choice is usually between placing them in a window's procedure list or in their own menus. We suggest that you create "procedure" menus. These menus will never be installed, but the procedures are called from the field procedures in windows. The result is a readily accessible library of procedures. It will be easy to use the same procedure more than once. You will avoid the calling problems that arise when field numbers (and hence the procedure "addresses") are altered, because this is unnecessary in procedure collections of this kind.

Calling procedures within window formats in v1.x

It is perfectly permissible to call procedures created in windows. However, such procedures, unless they belong to the window you are editing, will not appear in the procedure list when you type in the 'Call procedure' command in v1.x. You will have to remember the procedure address yourself. Editing procedures is also not as feasible here because the developer must locate the window format before the corresponding procedures. But there is a shortcut: use OPT/RB on the window format name. The command 'Modify procedures' will appear in the popup menu.

Calling procedures within window formats in v2.x

In v2.x, window procedures are dealt with on more or less equal terms with menu procedures, so we do not run into problems the same way here. Either way, if you place procedures to be called in window formats, it's a good idea to place them at the bottom of the procedure list starting from Procedure 500 and working your way "upward." This way the procedures are guaranteed to be in the safest possible location within a window format.

Large procedure collections

It's also possible to place all the procedures in a single procedure menu, but then one level of sorting disappears. This makes for a lot of scrolling. Moreover, 99 procedures (the maximum number allowable in v1.x) are often insufficient for an entire application. Even 500 may be too few, in which case we'll have to use more than one collection. We are then forced to search in a number of procedure collections, mostly by trial and error. The laborious job of reading through pProcedure1, pProcedure2, etc. is not exactly a barrel of laughs; nor is having to scroll through 500-odd procedures, for that matter!

Subdividing and calling

An important principle in the programming of heavy procedures is to subdivide them, or break them down into several small subprocedures which are called from the main procedure. This keeps the main procedure brief and it gives us a clear overall picture of what is happening in the procedure. The specific subprocedures can often be called from other main procedures. The most important advantage, however, is the general view this form of organization provides and the positive effect it has on the debugging. The subdivisions tell us a lot about where the error lies. We won't need to use as much mental effort during the initial programming phase, because we'll be able to concentrate on one thing at a time. In this way we can test small functions with few variables and parameters, and even the tests themselves will be easier.

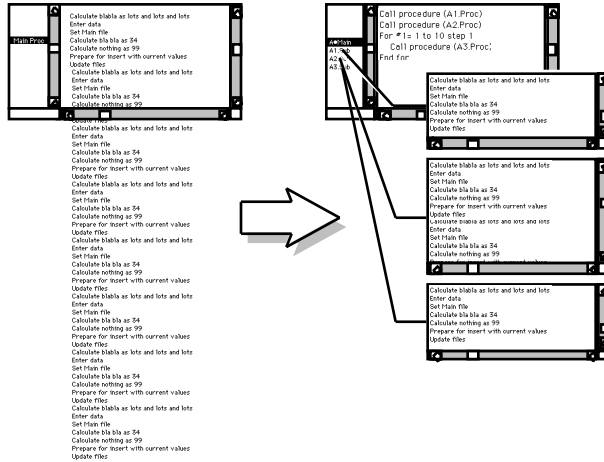


Fig. 8 An unduly long procedure on the left, and the same procedure subdivided and organized on the right

Cardinal rules

- Long procedures make for heavy reading.
- Long procedures tend to be error-ridden.
- Long procedures are not very nice-looking.
- Long procedures are definitely not good for your heart!
- No procedure is so special that it can't fit in somewhere or other and be reused (or so they say).

The subdivision of long procedures calls for a fairly good system for sorting and finding them later. It is up to the developer to determine how the procedures are to be kept separate; but the most important thing is to choose a system and stick to it.

Sequential calls

We have sequential calls when one procedure triggers another, which in turn triggers yet another, etc. If one of the procedures in such a row is run, the procedures that follow will also run, whether you want them to or not. Do your best to avoid calls within procedures which are themselves called, so that the Procedure stack can remain as small as possible. (See the chapter entitled “Sequence of Procedures.”)



Debugging cluttered applications is like trying to
catch butterflies
with a fly swatter – in rainy weather, no less!



Layout of the procedures

Procedures will be easier to read if we insert blank lines between dissimilar subtasks in the procedure. These have only a visual effect. The comment lines can be filled with just about anything when you want to highlight an important point; but remember: these lines can contribute to a cluttered and mushy appearance if they are overused. In v2.x, however, we can put the comments in the same line as the commands. Here they won’t disturb things much, and you may use them to your heart’s content. In Figure 9 we have a quasi statistical comparison that most people would agree on:

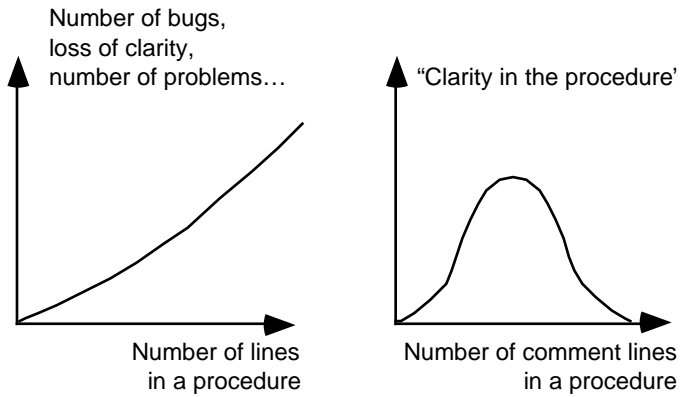
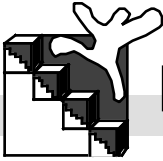


Fig. 9 Tidiness – the statistical kind , that is!



Debugging

General Considerations	2
The plague of bugs	
Avoiding errors	
Testing procedures	
Lengthy procedures	
Types of errors	
Localization	
Logical errors	
The datafile	
Window or Procedure?.....	11
How the Procedure Window Is Built Up	13
Finding the Right Procedure.....	14
Manipulating the Way Procedures Are Run.....	16
Manual control using “Run pointers”	
Manual control using “Run modes”	
Conditional stops	
Viewing and Altering the Values of Variables and Fields	25
Option/Right-hand Mouse Button (Opt/RB) menu	
The Field Value window	
The Field Value window for lists	
The Active Fields List window	
Examining the Sequence of Procedures.....	29
Trace all procedures	29
The Trace log	
OK messages	
Viewing the Procedure stack directly	
Finding Clutter in the Application.....	32
Buttons in the ‘Find and Replace’ dialog box	
Setting the criteria for the search	
Debugging in Multi-user Mode	35
In Conclusion.....	36

General Considerations

The plague of bugs

Debugging is both boring and demanding. Not many developers have the stomach for correcting problems that crop up unexpectedly when “everything was supposed to be in order.” Depending on the way we work, this process can make such great demands on our personal memory, sense of logic, and ability to juggle multiple elements that even the best of us often have to throw in the towel. In fact, this is why some applications are never finished. Failure to settle on a suitable approach at the outset is a sure-fire recipe for frustration. First we’ll take a look at a very general way of tackling the problem, then consider some examples of practical techniques; and then we’ll study the debugger itself in detail.

Avoiding errors

The simplest (and smartest) thing to do is avoid making errors in the first place. This requires a meticulous, somewhat sedate style of programming, which will not be to every developer’s taste. Be that as it may, “quick and dirty” programming is sure to create problems; and in any case, debugging is the most time-consuming process of all. So our guidelines for avoiding errors are simply a matter of good programming habits:

- Aim for good labelling and good field names.
- Let each field and each variable have only one task each.
- Take the time and trouble to make the structure of the application lucid and sensible.
- Don’t stop until you’re sure the procedures are well-organized.
- Avoid long procedures; break them up into subprocedures wherever possible.
- Do not continue until every subprocedure has been tested.
- Do your utmost to anticipate every conceivable situation that a given procedure could encounter.

Testing procedures

Continual testing at every stage is an indispensable part of a developer's work. This is particularly true of "new" procedures that the developer has not keyed in a hundred times before. Testing is the only way to mitigate the effects of Murphy's laws, from which no programmers are exempt. (Murphy's first law states: "If something *can* go wrong, it *will* – and when it's least expected and most inconvenient!")

Manipulating field values

In theory, the correct way to go about testing is to set the variables in use to known values and see whether you get the desired results when the procedure has finished running. We can use the OPT/RB menu to assign the values and to view them. If there are many variables to check, we can view them all at once in the Values list if the fields belong to the same file (via the 'File format' submenu in the OPT/RB menu.) You can also set the fields to be active and follow them in the Active Fields List as you step through the procedure. (More on this later.)

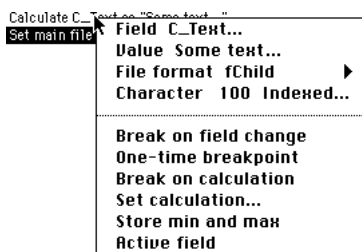


Fig. 1 The Option/Mouse Right-hand Button menu, abbreviated as the "OPT/RB" menu.

Lengthy procedures

Most procedures will contain a number of small, specific tasks that should be tested individually. The procedures must be run in segments, which you can do with the aid of the debugger. ‘One time breakpoint’ is useful here. It stops the procedure at the desired place and automatically disappears when the Go point passes. Regular Breakpoints placed in the margin are also acceptable. All of them can be deleted at the same time via ‘Clear Breakpoints’ in the ‘Breakpoints’ menu. But on the whole, your choice of method will be a matter of taste; in any case, there are options galore.

Thoroughly familiarize yourself with the debugger; you won’t regret it. The debugger allows for ample opportunity to test-as-you-go while you are inputting a procedure. Variables are dealt with at each stop, and it will be a relatively simple matter to correct any errors you uncover along the way. Errors are never as easy to catch as when you are inputting a procedure for the first time. All it takes is a little patience.

Types of errors

You will come across many types of errors as you go along. Some are due to a misguided use of commands. For the most part, Omnis will point these out, and they are fairly easy to correct. Other types, on the other hand, are the products of a fatal weakness in a procedure’s logical structure and are much harder to eliminate. But the worst that can happen is that a procedure does not perform the task for which it was intended, with the cause being that a certain command is not behaving quite as you expected. The procedure *appears* to be sound, and yet things still go wrong. So it is absolutely essential that you make sure that you know how every single procedure command actually works (and not just read a little and guess your way to the rest). In the paragraphs that follow, we will consider a general approach to eliminating most kinds of errors.

Localization

In all debugging, the first step, before you find out *why* something went wrong, is to find out *where* it went wrong. Usually, you already know *what* went wrong. (If not, then this is the first thing you need to find out.) Localization consists of zeroing in on an error. If your application is neatly sorted, finding the relevant procedure will not take long. The more you can narrow down the “list of culprits,” the fewer procedure lines there will be to comb, and the easier the debugging process will be. Everything hinges on how good the labeling is.

Logical errors

When the procedures run without protest from Omnis and the results are still not what you expected, logical errors are to blame. After a thoroughgoing localization, you may proceed as follows:

Has the test been carried out properly, and are your expectations rooted in reality?

It sometimes happens that a test appears to go smoothly and yet turns out to be the root of the problem. Not all developers appreciate what a chore it can be to run a proper test. Many factors and variables must be taken into account when attempting to produce a very specific situation in a somewhat cumbersome way. So just be patient! Learn to value the information a good test can provide and don’t begrudge yourself the time it takes.

Can the procedure be made clearer or broken down into subprocedures?

A little tidying up can go a long way; the work involved will not make great mental demands on you, and it can work wonders. You should not continue until you feel that you are in full control of what happens where. If you often find yourself tidying up procedures, you should probably be asking yourself if your programming style isn’t becoming “quick and dirty.”

Is the general structure right?

Who can't see the forest for the trees? Before you check out the procedure lines themselves, you had better check the bigger lines. In 'If'/'Else if' systems with many levels and a slew of 'End If's at the end, it is all too easy to assign certain parts of the procedure to the wrong level or put them in the wrong place. 'Repeat' or 'While' loops within loops will result in the same problem. Step back and see whether this is where the error lies. If the procedure is too lengthy to be viewed all at once, break it down into subprocedures and insert calls at appropriate places!

Procedure calls

Of course, it's possible to botch the calls as well. Be especially on your guard in situations where a "resource procedure" has been moved or deleted. There is always one more procedure that calls it. However, the procedure name of the subprocedure in 'Call procedure pProcedures/3 {Procedure name}' clearly shows whether the call has been wrongly placed. It's easy to spot errors of this type.

Missing or moved subprocedures

Subprocedures that are deleted can cause problems if there are other procedures that call them. We can catch these by inserting the procedure command 'Breakpoint' in the (otherwise) empty procedure. If this "empty" procedure is ever called, Omnis will stop here, and the culprit's name will appear in the 'Stack' menu! This method is a sluggard's delight. Another way of eliminating unwanted calls (which will appeal to the conscientious developer) is to use 'Find and replace.' For example: Search for *pProcedures/3*, with 'Allow wild cards,' 'Search from start,' 'All formats' and 'Literals' (or 'Complete text') checked off.

Which variables are used?

Don't be stingy: All the variables that are used in a procedure should be checked. Errors are often found where you least expect them.

Are the variables or fields the right type?

The OPT/RB menu shows the format in which the field or variable has been defined (date field, number field, etc.). If it isn't the right one, you can go directly to the file format from the same menu and correct it.

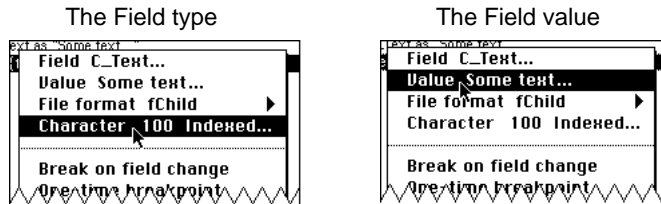


Fig. 2 The field type and value as shown in the OPT/RB menu.

What happens to them along the way?

That's a good question! The reason it's so hard to find out is that the computer runs too fast. You have to slow it down and get Omnis to show you what it is doing, or stop the procedures at strategic points.

Trace

The Trace function in the Debugger gives us a clear picture of where things are headed when a procedure is running. The Trace log gives a full accounting of what takes place when a given procedure is run; it can often tell us what we otherwise wouldn't know. (Remember to select 'Trace all procedures' if you're not already running a Trace.) Trace log is particularly useful for Library- or Window Control Procedures. Bear in mind, however, that Trace log will become quite large; it is also cumbersome to read.

Breakpoints

Breakpoints within a procedure are yet another way of controlling the procedure. The run stops by itself at preselected places and the variables can then be checked. A little experimenting with the Go point,

(Go) ‘To line’ and (Go) ‘From line’ will do the same job.

TIP: Remember that (Go) ‘To line’ moves up to (but not including) the line from which the command is given. In other words, the procedure in this line is not carried out.

So I’ve found the error; what now?

That’s hard to say. If the procedure line remains a mystery, then it’s high time to look up the command in the user’s manual. It’s just possible that you have a mistaken idea of what a certain command is used for, what it does, how it does it, and the kinds of demands it makes on Main file mode, etc. In data processing terminology there are a number of words that look the same but mean quite different things, so take nothing for granted. Don’t fool yourself into thinking that just because you have an elementary understanding of the term being used, you know enough about how the command actually works. Make a habit of consulting the user’s manual carefully and frequently, until you’re *certain* you know exactly what a given command is supposed to do. You can check yourself by running a little test in a test window; only then can you be sure.

Bugs in Omnis?

When you’ve done everything according to the book and the error persists, the problem might be a bug in Omnis. If so, it’s time to ring your nearest distributor, file a complete bug report (which will be forwarded to Blyth), and take your bows! But before Blyth can eliminate the bug, they need to know as much as possible about what was happening in the application when the error occurred. In fact, it’s not a bad idea to send them the entire application on a disk (or a screen dump at the very least).

Omnis as a whipping boy

Blaming Omnis for not being error-free is a pretty serious charge, you know; there must be grounds for declaring the defendant guilty as charged. Granted, there is a good chance you’ll stumble upon a bug or two in Omnis; but when you are struggling with a

difficult problem, Omnis bugs make a convenient whipping boy. Check carefully to make sure that the problem doesn't lie with the application instead; it could save you from filing a false bug report.

The datafile

The datafile itself has the potential for making your life miserable in certain situations. The problem might simply be due to the fact that the datafile doesn't contain the names, firms, postal zones, invoices, etc. that you thought it did. Since we cannot see the data in the same way that we can see the procedure commands, there is an ever-present danger that the datafile will not contain what we think it does. Most developers won't be able to remember exactly what data are in the file. The following procedure is a simple means of acquiring this information:

See the contents of fChild	1
Set current list #L1 Define list {fChild} Set main file {fChild} Build list from file	

See the contents of fChild (1)

Then click with OPT/RB on #L1, and with the aid of the popup menu bring up the Field Value window for this list. For connected files it is also useful to include the connections as well:

See the contents of fChild and fParent	2
Set Current List #L1 Define list {fChild, fParent} Set main file {fChild} Build list from file	

See the contents of fChild and fParent (2)

Here every parent record that is connected to the child records will appear together. For large files, the list will be correspondingly (and equally) long. So you can make do with selected fields from the two files; this will require a bit more typing and mouse clicks, however.

Datafile header not updated

There may also be inconsistencies in the format of the datafile and the file definition in memory. Failure to reorganize after making changes in the file format can lead to this very problem and produce the strangest results during a search. Once in a while it will be the datafile itself that is corrupted, which requires first aid from the 'Utilities' menu. If the file is a hopeless case, then save what you can by exporting.

Omnis error messages

If you do get an Omnis error message, correcting the error is fairly straightforward. In Appendix A of "Reference 2" there is a list of error messages and what each one means. Nevertheless, it will pay you to check everything that affects the relevant command and look up the description of it in "Reference 1." The error could be located further back from what actually set off the error message. The error message 'Find without an indexed field' could mean, for example, that the wrong main file has been set, not necessarily that the field has not been indexed.



Never succumb to the trial-and-error method. Errors can seldom be corrected by chance. The only thing that helps here is a thorough and systematic approach to the debugging process.



Window or Procedure?

Your first encounter with an error in an application will most likely follow this pattern: Omnis stops and shows the offending procedure. In other cases Omnis will not stop, and it is the developer that is left holding the bag. But in both cases the question remains: Where is the error? In windows, the natural thing to do is to check what's on the "front side" first, i.e. the window fields themselves and their settings, before you start on the procedures. Note the following in particular:

The 'local' option in window fields

Display fields where the 'Local' option has inadvertently been left unchecked will appear to be unresponsive. While the developer desperately searches for an error in the procedures, the error lies on the front side of the window.

Fields in wrong boxes

Display fields with 'Auto find' where the field that Omnis is supposed to do the 'Find' on, and the field that contains the value it fits, have been switched. It's not all that easy to keep track of which field goes where: The value that fits (or the variable containing it) should be in the calculation box (under 'Calculation') and the indexed field should be in the field box (after 'Name:')

The field order in the window

Is the field order correct? 'Auto find' fields are very sensitive in this regard, and whole groups of radio buttons will not work at all if the order of the window elements is not as it should be. If one or more field procedures are called by other procedures, you should exercise caution when you change the field order. This will cause the procedure addresses ('wWindow/3,' for example) which refer to that window to change meaning entirely, and the wrong field procedures will run. However, the calling of field procedures in windows is bad programming practice.

‘User defined’ not selected

Pushbuttons that are not changed from standard ‘Find,’ ‘Insert,’ ‘Edit,’ etc. to ‘User defined,’ when there are procedures behind them that are to run alone. This is an irritating and all-too-common error. In v1.x, the result was that the procedure was never run; from v2.x on, the standard button action is carried out in addition to (before) the user-defined field procedure.

How the Procedure Window Is Built Up

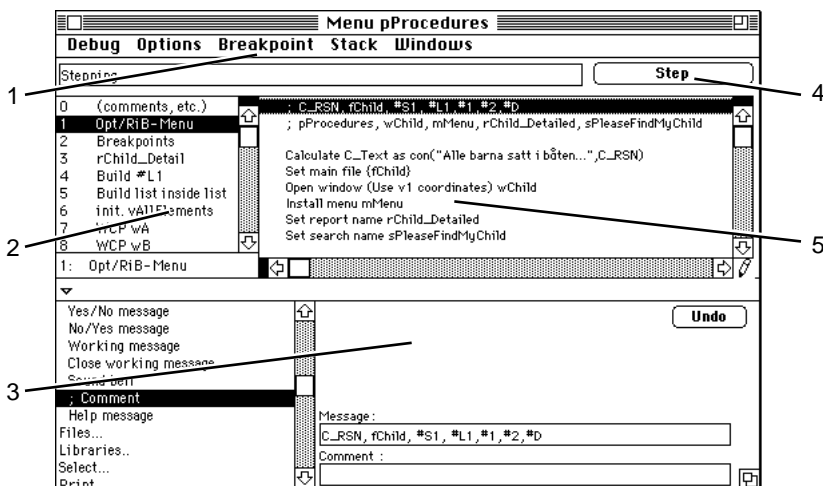


Fig 3. The Procedure Window showing the debugger menus

Since we will continually be referring to various parts of the procedure window, we should make it clear what they mean. In the text that follows, these numbers will be referred to in parentheses.

- 1 The debugger menus
- 2 Procedure title list
- 3 Procedure Tools window
- 4 The 'Go/Step/Step Over/Trace' pushbutton
- 5 The procedure itself

Finding the Right Procedure

If you feel reasonably certain that the error is in a specific procedure, you need a handy way of finding the right one in a jiffy. In v2.x, windows and menus contain up to 500 procedures each. Scrolling through all these procedures in order to find a particular one is inordinately time-consuming. Instead, try the following:

Looking under the field

Select a field in the window and press CMND-5 or F5 to view the procedure behind it.

Using specific procedure lines

If the procedure contains a reference to other formats, locating these is an easy matter. Simply select the procedure line and press CMND-8 or F8. Look for the following procedure commands:

Open window
Install menu
Set report name
Set main file
Call procedure
Set search name

By selecting one of these procedure commands, we also get a list of formats of the same type (in the Procedure Tools window). This, in turn, gives us access to the format names; we can click on one of them with OPT/RB and select 'Modify' from the menu that appears.

Using the comment line

Interesting formats (for example, menus and windows) can be jotted down in the comment lines, one format per line. These may then be brought up by selecting the comment line and pressing CMND-8 or F8. If the comment line contains more than one format name,

you must click on the desired format with OPT/RB and use the popup menu to locate it.

Using the Trace log

Relevant procedure addresses can be stored in the Trace Log. Run one procedure line using ‘Step’ in each procedure – preferably a comment line that contains a little explanation (or a copy of the procedure title); the procedure address will automatically be transferred to the log. Later all you’ll need to do is double-click on one of these lines in the Trace Log, and the corresponding procedure will appear.

Entering the procedure number

If you know the number of the procedure you want to see, you can click on the Procedure Title List (2) and enter the procedure number. The list will continue to scroll as the numbers are entered, just as when we select procedure commands during programming.

Using various hot keys

There are many keyboard shortcuts. (Macintosh developers who do not have an extended keyboard are advised to get one.) Click on Procedure Title List (2). The HOME key locates procedure number 0, i.e. the initiating procedure. The END key takes you to the last procedure, which is often the Window Control Procedure or list with format variables. In any case, since the last line is so accessible (you may also get here by dragging the vertical scrollbar all the way down), you would do well to place often-used procedures here.

The PAGEUP and PAGEDOWN keys work in this list just as you would expect them to, and they can be used to speed up scrolling. The chapter entitled “Keyboard Shortcuts” deals with a number of related functions.

Manipulating the Way Procedures Are Run

When the right procedure has been found, it's time to take over the steering wheel from Omnis; now it's *your* turn to drive. The procedures have to be run in small bits, one segment at a time, so that you can analyze every point in the run. You can control the process manually or establish criteria for each stop along the way. Whichever method you choose, Omnis is obliged to comply.

Manual control using “Run pointers”

By “run pointers” we mean pointers at specific lines which tell Omnis how to navigate between the different procedures. Manipulating these will give you effective control of those commands that are to be executed – and thus, control of what is taking place.

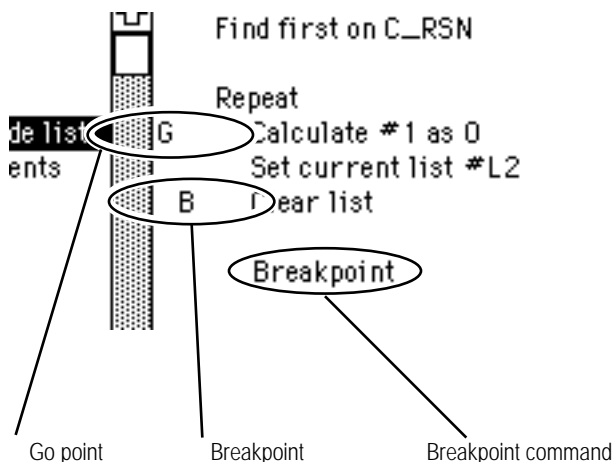


Fig. 4 Some of the run pointers

The Go point

When Omnis runs a procedure, it moves the Go point along. Go point is the “next line up” to be run. This is indicated by a “G” in the left-hand margin of the procedure line. A good way to set Go point is to double-click on the line you want. We know that Omnis will start here when we choose ‘Go,’ ‘Step,’ ‘Step Over’ or ‘Trace.’ Even if you are working on another procedure, Omnis will still run from the Go point. The procedure being run will appear immediately.

Running commands of your own choosing

By moving the Go point around in the procedure, you as a developer can skip over procedure commands that are unwelcome at that point, give individual commands, and in general create exactly the kind of situation you want.

Starting from the top

‘Execute procedure’ (‘Modify’ menu, CMND/CTRL-E when the debugger menus are hidden in v2.x, always in v3.x) runs the procedure from the top, regardless where the Go point is. This menu command is handy when an entire procedure is being run, i.e. when the testing of the individual subsections is complete. For that matter, we can put the Go Point on the top line and then select ‘Go’ (CMND/CTRL-E when the debugger menus are shown in v2.x, always in v3.x).

The Go point address

The name and address of the procedure containing the Go point will crop up in the Debug menu. If you are working on other procedures, you can return directly to the procedure where the Go point is located by selecting the address from this menu. In this way, the Go point acts as a book marker, for example when you are working with a main procedure that calls many other procedures.

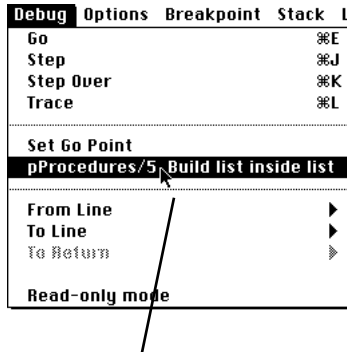


Fig. 5 The procedure address of the Go point

Breakpoints

Here is where Omnis stops. Breakpoints can be used to break up procedures “the hard way,” so that the run is halted at fixed spots. If you press ‘Go’ again, Omnis will continue to run from this point. The line containing the Breakpoint is executed only when the developer continues the run. Breakpoints can be set as a procedure command or marked off in the margin by means of the ‘Breakpoint menu’ (in the debugger menu group). The latter can also set a ‘One Time Breakpoint,’ which disappears after the Go point has passed it once during a continuous run. (Applies to ‘Go’ and ‘Trace,’ and not ‘Step’ or ‘Step Over.’) We can bring up the ‘Breakpoint’ menu as a popup menu by pressing OPT/RB in the left margin of the procedure (when the debugger menus are shown, that is).

Return point

When a procedure calls another procedure, it waits at the line that contains the command ‘Call procedure’ until the other procedure is finished. This line acquires a “label” called the “Return point,” marked by an “R.” (The Go point is somewhere in the other procedure.) When the other procedure is finished, the Go point reverts to the Return point and continues from there. When the Go point has returned from its little excursion to the other procedure, the Return point is no longer needed and is removed automatically.

The return point as a reminder

When the developer himself moves the Go point within a procedure, Return point appears as a reminder of where the developer last had the Go point. The little “R” has no further effect in this case.

Manual control using “Run modes”

There are several ways of running through the procedures. Each one has its advantages, and with a little imagination you can exploit these effectively and amusingly. Let’s take a closer look at them:

Go

‘Go’ means, pure and simply, that Omnis runs the procedure from the Go point without doing anything in particular for the developer along the way. The window with the procedure waits patiently until the procedure is finished and the developer can examine the result. If the Go point encounters a Breakpoint of any kind, it will of course stop there.

Trace

‘Trace’ runs the same way as ‘Go,’ but it also shows how the procedure lines are run. The line being executed is highlighted (inverse), and to the developer the whole thing will appear as a mostly black line that scrolls down the procedure lines. Procedures that are called are retrieved one by one as the Go point passes by. It can be terribly frustrating and confusing to try to keep yourself oriented in a window that is constantly blinking, scrolling, and changing appearance. On the other hand, we are able to trace Omnis’ every move, as well as keep tabs on which procedures are being run and the order in which it takes place. Keep your eyes fixed on the procedure titles in the list on the left and notice which procedures are being highlighted; this will ease your frustration. You’ll have a much more “restful” screen during Read Only mode; what is more, the run will zip right along. (See below.)

If you are quick with the ‘Stop’ pushbutton, you can use it to stop the run at the right places and check the content of the variables before continuing. During ‘Trace,’ all the procedure lines will be copied to the Trace Log, so you can return here later and peruse them. (But it gets filled up in hardly no time at all.)

Step

‘Step’ runs the procedure line with the Go point, then skips to the next one and stops. This is like giving single commands. Every ‘Step’ advances the Go point by one line. This is a thorough (and often time-consuming) method of going through the entire complex of procedures. ‘Step’ is easy to use, and together with a little moving around of the Go point, the developer will have the situation under full control. This method is perfectly satisfactory, and many prefer it to the fussiness and sense of powerlessness that ‘Trace’ entails. If the Go point comes to a ‘Call procedure’ command, the window will display the procedure that is called, and we can see it being added to the Procedure stack in the ‘Stack’ menu.

Step Over

This method of running the procedure is very similar to ‘Step,’ except that it doesn’t display the procedures that are called. Instead, the window remains inactive while the other procedure is being rapidly executed. This is useful when you are debugging procedures that occasionally turn calls into subprocedures and you *know* that these are working as they should. In such cases it isn’t necessary to comb the subprocedures once more; they will be executed as the Go point passes each line containing a call, and the debugging can continue undisturbed in the procedure at hand.

Change “Run mode”

Provided the debugger menus are displayed, you can switch back and forth between different run modes, which may be selected directly from the ‘Debug’ menu (farthest to the left); or, with OPT/RB, you can click on the ‘Go’ pushbutton in the right-hand corner (4). The former also starts the run, but the latter only changes the run mode. Click normally to start. However, you should learn how to use the four corresponding keyboard shortcuts that are shown in the ‘Debug’ menu; they are sure to come in handy.

Read Only mode

“Read Only mode” is a misnomer. It is a diffuse expression and can easily be confused with Read Only status for file formats. Simply put, Read Only mode is a means of speeding up ‘Step,’ ‘Step Over’ and ‘Trace.’ When the Go point moves downward in normal “change” mode, the highlighting keeps pace with the lines that are being run. This means that the procedure command in each line is displayed in the list in the Procedure Tools Window (3), and corresponding boxes and lists appear. This is the same thing that happens when the developer selects or clicks on a line. But it takes time. When you are in Read Only mode, the Procedure Tools window is hidden and your computer will be relieved of these tasks. This will make it easier for the computer to go from line to line. The speed premium is great, which makes stepping all the more useful.

When you go through a procedure to find the cause of an error, it’s primarily the variables and fields you’ll be dealing with, not the procedure commands. Only when you see a relationship between the two will it be kosher to edit the procedure. If you do, go into normal “change” mode by turning off Read Only mode.

Active fields and Read Only mode

Another potentially major hindrance to speed is the Active Fields List window. In Read Only mode this list will not be updated automatically. Select ‘Redraw’ from the “R” menu in the window to view the relevant values in the fields.

The advantages of manual control

The more familiar you are with the manual method of controlling a run, the easier it will be to control the testing when checking for errors in procedures. Since it is the test results that tell us where the problem lies, it follows that improving your testing skills will make you a better developer.

Conditional stops

The art of looking for bugs in applications is largely a matter of being able to stop the run in the nick of time. As a developer, what you are after is that magic moment when you catch a fleeting glimpse of the *Error* that you are trying so desperately to pin down. It is at this very point that the incriminating information in the variables and fields lies within your grasp. As a rule, you will get there sooner or later; all it takes is time, patience, and a bit of subtle maneuvering on your part. Those with keenly analytical minds might make more rapid progress, however, by getting Omnis to stop at the right set of conditions. The logic called for is strict and demanding, but getting a handle on it is not as frightening as you might think. In any case, an attempt is well worth the effort.

Break on field change

If a field or a variable is set to ‘Break on field change,’ Omnis will pause and display the procedure line every time the content of the field is changed from one value to another, notwithstanding #NULL. Although this method is straightforward enough, it can lead to the run stopping many times before the error is found. You should choose a field for ‘Break on field change’ with care. Don’t let too many fields be set that way; it will only cause Omnis to stop constantly, and the developer will lose track of what is happening.

Break on calculation

Where ‘Break on field change’ has its biggest weakness, ‘Break on calculation’ is on hand to help. To keep the run from continually being interrupted, you can insert a couple of conditions that must be met before Omnis can stop. The beauty of this is that you can insert the very thing you think is wrong!

Examples of conditional breaks

For example, if you see red because #1 becomes null in the course of a procedure, you can set the Break calculation to #1=0 (using ‘Set Break Calculation’) and see where this leads you. Omnis will stop as soon as something unforgivable happens, and all you have to do at that point is backtrack. We know that the error has just occurred, so it can’t be far away. If there is a

‘Trace on’ command at the beginning of the procedure, we will be able to see (in Trace Log) everything that has happened up to the break, including any procedures that are called. (To avoid superfluous procedure lines in Trace Log, place the ‘Trace off’ command at the end of the procedure.)

If the error lies in the fact that the content of the fields in the file format fChild vanish from memory, you can set the Break calculation to ‘C_RSN<1.’ This will cause Omnis to stop when the sequence number (C_RSN) is deleted from memory. The calculation may be ‘len(C_TEXT)=0,’ in which case Omnis will stop when the content of text field C_TEXT is deleted.

The Break calculation applies globally

The calculation is carried out every time the Go point passes a line that changes the value of a variable or field in the calculation. The condition(s) apply globally and, strictly speaking, are not dependent on the variable to which you set the Break calculation. There is only one Break calculation anyway, so it doesn’t matter which field you choose enroute to it. In other words, you are free to insert whatever calculations suit your fancy.

Limitations

Just as with ‘Break on field change,’ it is not advisable to set up too large a calculation, with many conditions and criteria. This will complicate the logic and make it harder for the developer to keep pace. And there is always the danger that the test situation itself is not right. Errors in the testing phase are sinister, and should be avoided at all costs.

Viewing and Altering the Values of Variables and Fields

You get to the root of an error by looking for the right information at the various stages of the procedure. In other words, the easier it is to find this information, the sooner you will find the cause of the error. Simple and effective methods for (primarily) viewing the values of variables and fields is all-important. In addition, you will need to examine other parameters, for example those concerning Main file, its file mode (Read Only, Read/ Write), etc.

Option/Right-hand Mouse Button (OPT/RB) menu

One of the most delightful things about Omnis 7 is how easily the developer can manipulate the content of variables and fields. The tool for this purpose is called the Field Popup menu, which we will refer to as the OPT/RB menu. For Macintosh users, this menu appears in various guises when you hold down the option key and click on, say, a field. In Windows, the right-hand mouse button (RB) is used instead of option-click. In the Field Popup menu you can directly observe the value of the field, as well as call up the Field Value window by selecting the topmost menu line. (The field name appears here.) Furthermore, the different variations of the menu all have shortcuts that you'll soon learn to appreciate.

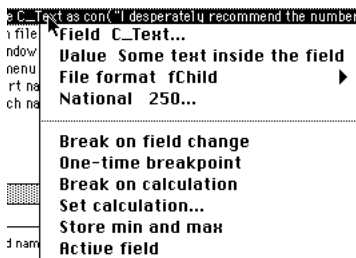


Fig. 6 The OPT/RB menu

Using the menu

By clicking around in window names, fields, lists, report names, #variables, you will gather all the information you need, and the situation will be clarified. Everywhere a full name is shown, you can click with OPT/RB and call up this menu. The various menu choices are familiar commands for the experienced user. If in doubt, turn to the chapter on the debugger in “Design and Development.”

The Field Value window

This window is the developer’s standard tool for editing values on the fly. Fields of all kinds can be altered directly, and Omnis will see to it that date fields are assigned dates, number fields assigned numbers, and so on. The window is indispensable during procedure testing. One of the developer’s most useful tools, it deserves to be used frequently.

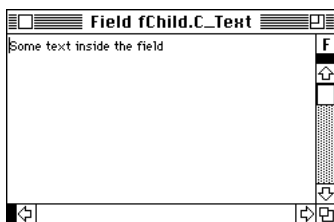


Fig. 7 The Field Value window

The Field Value window for lists

Lists have their own variation of the Field Value window. The content of each line can be altered directly by clicking where appropriate. You can also control #L and which lines are highlighted, as well as add new lines and delete others. Click with OPT/RB in the lefthand margin of the list, and the list window’s own popup menu will appear. The developer can simulate, within the procedure window, anything a user could think to do with the list, and can do so without having to open the user window.

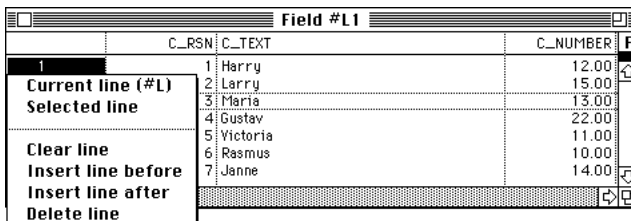


Fig. 8 The Field Value window for lists

The Values List window

This window shows the name of all the fields within a file format or a category (#variable, format variable, etc.) and their respective values. Since the window provides a great deal of information all at once, it is a very handy tool to have, especially at the beginning of an error search. It provides a fairly good overview. Later the window can be used to view or alter values of a field that is not being used in the procedure currently being displayed. The window does take up a lot of room on the screen, however, and is not something you will want to have open all the time. Example: Click with OPT/RB on the name of any file format, and select 'Show Values List.'

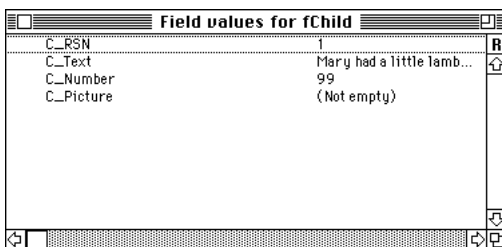


Fig. 9 The Values list window

The Active Fields List window

Fields that are active are sent to this window, where the values are continually updated. If the values of the variables change during the run, this will be reflected in the Active Fields List. You should position the window so that it isn't covered up by other windows. The easiest way to follow the changes is by means of stepping or slow tracing.

Set fields to Active by clicking on their names with OPT/RB, and then select 'Active' from the popup menu. When deleting the fields from the Active Fields List, reselect this menu item (via the OPT/RB menu) so that the "checked" character disappears.

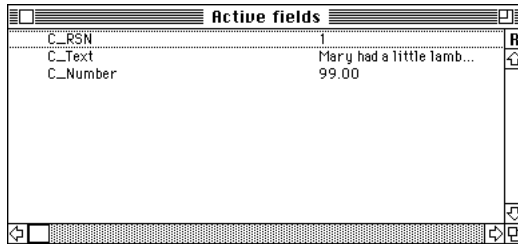


Fig. 10 The Active Fields List window

Examining the Sequence of Procedures

Control procedures, Timer procedures, procedure calls, and Push-buttons that are active under Enter data can all cause headaches for the developer. You need to know at all times which procedures are being run and in what order. The theory behind this is set forth in the chapter entitled “Sequence of Procedures.” Let’s take a look now at how we test the sequence.

Trace all procedures

The simplest method is also the dirtiest. If you select ‘Trace All Procedures’ from the (Debugger) ‘Options’ menu, you’ll get the whole story – but that might be a very long story! All procedure lines, bar none, are sent to the Trace Log with address and content. Every procedure takes up a lot of space in the Trace Log, and you have to read the addresses to see when the next procedure takes over. The procedure titles do not appear here, which makes the log hard to read. The developer will often have to do a lot of scrolling and searching.

The Trace log

The Trace log has been described as a kind of notebook, and rightly so. ‘Send to trace log’ is one of the commands that can be used to control it. If you equip every relevant procedure with such a command, it will serve as a kind of tag indicating that the procedure has been run. Thus, the Trace Log will be brief and concise. Take care, however, that you have included all the procedures that relate to the problem. The function ‘sys(85)’ will return the procedure’s address so that we can use it instead of writing the procedure title itself in the ‘Send to trace log’ command. It is also possible to send variables and fields with this command. The values will be noted as the various stages of the procedure sequence progress. Have the Trace Log open while the testing is in progress and the procedures are being executed.

If you wish to see the sequence in more than rough outline, you can trace the procedure lines themselves. It is a good idea to narrow the scope of the row of procedures by means of the ‘Trace on’ and ‘Trace Off’ commands, inserted at the appropriate places in the procedures. This will keep your Trace Log from getting filled up with more or less unwanted procedure lines.

OK messages

An ‘OK message’ in every relevant procedure allows the developer ample time to think, because Omnis waits for you to press OK. The disadvantage with this is that the onus is on you to remember the order – which is sure to tax your memory in the long run. If you want to view the values of variables and fields, put them in square brackets (“[]”) inside the OK message.

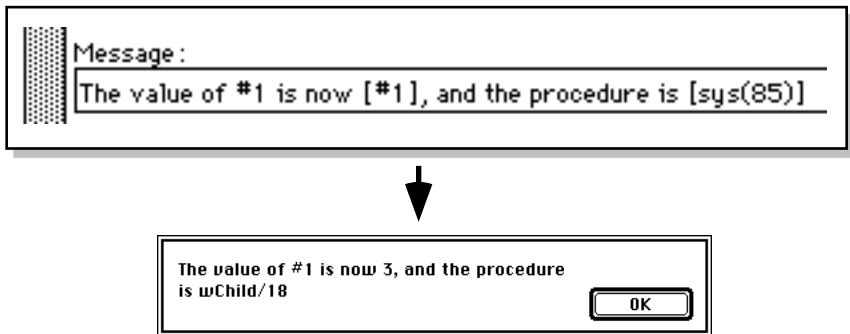


Fig. 11 How to use square brackets in OK messages

Viewing the Procedure stack directly

If the Go point meets a Breakpoint, the run will stop and the procedure containing the line with the Go point is displayed. We are, in effect, right in the thick of things at this point and can view the Procedure stack directly in the ‘Stack’ menu. A Breakpoint in the middle of the last procedure in a string of procedures that call other

procedures will thus bring up the entire sequence of procedures waiting to be executed at this very point. This string of procedures is named the “Procedure stack.” It’s impossible to explain the Procedure stack in two short lines, so please turn to the chapter entitled “Sequence of Procedures.”

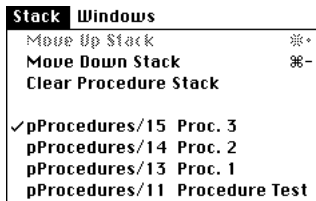


Fig. 12 The Stack menu

Finding Clutter in the Application

Changing your mind is a natural thing, and at times it can be the wisest thing as well. However, changes must be followed up consistently – and not just for a few procedures that we might think of offhand. The easy way to be thorough is to use ‘Find and Replace.’ This is a general tool that you will always discover new uses for. You can use it freely and fearlessly if you understand what the various choices in the dialogue box signify. Later, the ‘Find and Replace Log’ will be of practical use, especially in view of the fact that you are brought to the right format by double-clicking one of the lines. (Each line contains the right “address.”)

Buttons in the ‘Find and Replace’ dialog box

Knowing the buttons inside the ‘Find and Replace’ dialog box is a prerequisite for using this tool correctly, so let’s take a look at them.

Find

Executes a single search, and stops and displays the result as soon as it finds anything. The result is, in fact, the address telling us where the object belongs. The address is also noted in the ‘Find and Replace Log.’

Find All

Here Omnis runs through all the formats that have been selected for this particular search and makes a note of each address in the ‘Find and Replace Log’ whenever it finds something that fits. This takes a bit longer; on the other hand, it provides a better overview because all the results are shown at one time.

Replace all

The ‘Replace all’ button can be a bit risky because you can’t see exactly what you’re doing. You’re safest in executing one ‘Replace,’ examining the effect, and then running a ‘Replace all’ if the former succeeds.

Setting the criteria for the search

In addition to the actual text that the objects are supposed to match, you may further narrow down the search using one or more of the options represented by the check boxes.

Field names

Field names are searched for quite simply wherever they are used. If the name is only a part of the free text in a calculation, i.e. enclosed in quotation marks (" "), it will not be used.

Format names

Format names are searched for just like field names, i.e. when they are employed directly by the procedure commands.

Literals

This option refers to text, as set off in quotation marks (" ") in calculations, free text in windows and reports, as well as numerals in calculations. It also refers to the text in procedure addresses, such as in 'Call procedure,' 'Set window control procedure,' and other commands..

Procedure commands

Here the search is based solely on procedure commands. All parameters, settings, etc. are disregarded.

Complete text

When this option has been selected, the search text should fit the entire procedure command such as it appears (as text) in the procedure, with all parameters and settings inside their appropriate brackets and parentheses.

Ignore case

If you are careless in your use of upper and lower case, this choice will absolve you fully.

Allow wildcards

The two special characters “*” and “?” are useful when you aren’t sure how a given procedure command is written. The character “*” signifies any given number of letters or numbers (limited by each procedure line), whereas the character “?” is inserted for each and every letter you are unsure about.

C_*	finds everything that begins with “C_.”
*Child	finds everything that ends with “Child.”
C_??	finds elements that begin with “C_,” but they must consist of only 4 letters (here).
?Child	finds elements that consist of 6 letters and end with “Child.”

Search from start

‘Find’ normally continues the search from where it last stopped or found something, regardless of whether the conditions for the search have changed. By selecting this option, you will ensure that the search starts from the top of the first selected format.

Search backwards

The search starts from where it last stopped and works its way up the lines in the procedures (i.e. format “lines”). Later it works its way backwards in the formats alphabetically.

Debugging in Multi-user Mode

In v1.x you are not allowed to alter the application in multi-user mode. This means that the Design menu is not accessible, and you won't be able to get to the debugger menus. None of the virtues of the debugger can be exploited in multi-user mode. This leaves us with our trusty 'OK message' command. (Remember Omnis 5?) Variables and calculations are set off in square brackets ([]); if anything has changed, you will have to reopen the application in Single user mode. For this reason you should do as much testing as possible in Single user mode.

Version 2.x

Version 2.x doesn't have this limitation. As long as the developer has the right user level (#UL), and the format in question is not altered by others in the network, any format may freely be altered. During the development work you will not have to close and reopen the libraries every time the multi-user systems are being tested and errors are being discovered. Furthermore, several developers can work on the same project and the same application at the same time. For large projects this is welcome news indeed, and v2.x comes with a multi-developer system together with the version control (VCS).

A piece of advice

Bear in mind that you should always run a thorough battery of tests on site, i.e. where the application is going to be used, with all the different printers, computers, monitors, and other components connected. Remember Murphy's law!

In Conclusion

If you have ever sat and searched long and hard for an error in your application and sensed that you're getting increasingly irritated, then it's time to look at the clock. Very few people can engage in the kind of intense mental activity that debugging requires for more than an hour at a time. The galling fact that, for the moment, the mind cannot conquer matter can actually inspire within you an almost malevolent urge to press on. You will become obsessed with the thought of finding "just this one error" before you quit and won't notice that you are too exhausted to find it. It's not just that taking a break is a "good idea"; the fact is, that elusive error is simply *not* going to be found until you are alert and chipper again. And it's during the break that your batteries get recharged.

Many developers try to tough it out, fearing that if they stop, they will forget what the problem was and waste valuable time trying to get back on track after the break. For one thing, this is simply not true. For another, they will most likely have worked themselves into a blind alley and not have the energy to extricate themselves. And with his fuel tanks on reserve, the hapless developer will succumb to the temptation to use the dreaded "shot-in-the-dark" method, which is a total waste of time. He can even become so tired that he is unable to correctly interpret any key information that might arise. At this point, debugging is not only fruitless but a source of misleading conclusions, which in turn makes for a lot of extra work before the developer finally realizes what's going on.

Not only does time go by without bringing the developer any closer to a solution; he also becomes more aggressive, his adrenal glands start secreting copious amounts of adrenaline and corticosteroids (stress hormones), his muscles tighten, his concentration begins to slacken, and the vicious circle has begun. Then is when being a developer is no fun at all.

The most important aspect of the break is the change of scenery it provides. Go somewhere else and get your mind on something else altogether. The human brain is an expert in parallel processing, so you can be sure that the debugging is still going on – somewhere or other – in your head. Drink a cup of coffee (anyway, that's what I

do). Don't watch TV, or the battle is lost. Instead, give someone close to you a big hug (not a bad idea at all!).

After the break, you'll be able to approach the problem from a slightly different angle. That's when new ideas often surface, and points that previously escaped your notice will flash before your mind's eye. A solution is often teasingly close. With any luck, you'll soon be able to turn off your computer, switch off the lights, and head home in triumph.



If you see the Error of your ways, all the Bugs
will follow.





Layout & the User Interface

A Good Design.....	2
Effects	4
Use of colors	
Placement	
Letter effects	
Typefaces	
Choosing fonts in reports and windows	
Gray tones and shadows	
Logical Arrangement of Menus and Windows	13
Menus	
Windows	

A Good Design

Layout and the user interface are but two sides of the same coin. Both aim to facilitate communication between user and computer. Layout is concerned with the manner in which the various aspects of the application are presented to the user; the user interface encompasses practical solutions to the communications problem and to how the user interacts with the application. The primary goal of every user interface is to be as intuitive as possible for as many users as possible and obviate (or at least minimize) the need for an instruction manual. Or, to put it in plain English, the goal of any good user interface is to make an application easy to use.

The sorting of elements

Users need help in distinguishing important elements from less important ones. The layout directs the user's attention where the developer wants it and is thus the key to the sorting process. If every element in a window were to be given equal weight, the user would be left with the burden of finding out just what the application purports to do in a given window. Windows like that can be worse than useless. Your stock with the user will rise considerably if you can spare him or her much of the sorting work.

Prerequisites

There is a limit to how clearly a window can be made to look for the average user. We're all different, with different ways of looking at the world around us. And it won't be easy for you to decide on which method of presentation is absolutely the clearest so that everyone – without exception – will understand that this is how the application is to be used. Don't be pedantic about it. Applications are designed for typical use, where the user is on familiar ground. Anyone's first encounter with a particular window is going to be a special experience anyway. Trying to make the window explain everything and cover every conceivable base is like trying to print an entire user's manual on a postage stamp. In most cases all that's called for is a fairly

orderly presentation of the elements, perhaps a couple of small icons, a few drawings, and some good ideas. If your window is esthetically pleasing to boot, so much the better.



Guide the user's eye through the window.
Make the important parts easy to find.



Effects

An application's appearance is a fixed feature of the user's daily work situation. The same windows are going to be seen every day. Bear this in mind when you set about designing the overall look of your application. Better a neat, orderly and (dare we say) staid look than a flamboyant, overstimulating grabbag of effects, which will only divert attention from the task at hand and which – at worst – can be downright off-putting. As a rule of thumb we can say that only two (or at the most three) elements should stand out in relation to the others. Now let's consider a number of specific effects and the ways in which they can be effectively employed. And remember: where effects are concerned, sobriety is a virtue.

Use of colors

Subdued pastel colors are preferable to gaudy, glaring ones, especially on large surfaces. Color contrasts, where used to distinguish different elements, should be sharp. And don't mix similar colors (for example, don't use green text on a yellow background).

Colors can impart intuitive associations to elements. Red could signify commands that effect the data in some crucial (or potentially catastrophic) way; green could stand for "safe" commands, etc. Assigning specific meanings and associations to specific colors will always be a matter of personal preference. When you use colors to express something specific, you should be consistent. On the other hand, don't let color coding in itself feature so prominently that the colors are made to compete for the user's attention. The American essayist Ralph Waldo Emerson once wrote that "consistency is the hobgoblin of little minds." As we've seen, it has its place; but if you are dogmatically consistent and fixated on the use of color codes, you are sure to end up with glaring and unelegant windows. Think big: A pleasing surface appearance is more important than a "self-explanatory" window with color codes. Alternatively, you can set off elements with smaller color surfaces – perhaps even just a red line.

Placement

The placing of fields and text is an effect in itself and ought, as such, to flow logically from left to right. The crux of what you are communicating lies in the text, not in the pictorial expression. Arrange the fields in logical order, from top to bottom, and employ a line or column orientation. Line orientation is easier for the developer to work with than a column orientation, since the fields in the window will not need to be of equal width. Plenty of room (“air”) around a field or a title is also a pleasing effect. It is eye-catching and doesn’t slow the reading.



Let the order convey the message.



Letter effects

Where there are space limitations in a window, letter effects can be a pretty good alternative. You should bear in mind, however, that by their very nature they slow down the reading. The more pronounced the effects, the slower the text will have to be read – and the more attention it will call to itself. But if everything is equally heavy reading, we will have failed to achieve any contrast, and the user will have to spend more time trying to find out what the window is trying to say.

Capital (upper case) letters

Capital letters (“all caps”) are unsuitable for large blocks of text because they are harder to read than lower case letters set in italics or boldface type. The reason for this is that a word written in lower case letters is recognized largely with the aid of the outer contours of the word as a whole. Words written in all caps, on the other hand, have no distinctive shape

except that of a ragged rectangle; and because of this, the reader is forced to focus more or less on one letter at a time.

Boldface type and italics

Boldface type and italics are moderate, readable effects, and may readily be combined. These are among the most frequently used effects in general design and layout.

Underlining

Before the age of computers, underlining was one of the very few effects that a typewriter could produce. Even though most people are accustomed to seeing underlined text in typescript, the fact remains that in terms of readability, underlining is almost as poor as all caps. The underlining disturbs the reader's perception of not only the word's overall shape, but that of the individual letters that make up the word.

Size

It is customary in written material to make headings larger than the main body of the text. The sorting effect, however, only works when the differences are clearcut and well-defined. Much of the purpose is defeated when the developer tries to distinguish many levels of text by means of letter size. The result is often a cluttered look. Size is defined in terms of points, which is an old (and completely non-metric) typographical unit of measurement. When computers came on the scene, "points" made a simple and successful transition, and now means the same as "number of pixels in height."

Typefaces

Typefaces (often referred to as "fonts") are a major factor in the readability of any layout. As with all other text effects, it is essential to give careful thought to your choice and use of typefaces and to the specific functions they are meant to perform. There is an almost

endless variety of fonts. Some fonts are unexciting but highly readable; others, again, are eye-catching and stylish in themselves but useless for anything but headings or invitations. Don't be seduced by a font's artistic aspects; rather, consider its concrete characteristics. We will do just that in what follows.

Monospaced fonts

In monospaced typefaces characters are of equal width – both on screen and on paper. Even the space *between* words (produced by hitting the space bar) has the same width. Most typewriters employ monospaced characters. The advantage of monospaced typefaces is that we can place words and columns in vertical order with the use of the space bar. Where lists are displayed in a list field in windows, this is an important advantage. The disadvantage of monospaced typefaces, however, is that they do not help the eye to follow the flow of text in a line, because the space between the letters is too large. Such type styles suffer from poor readability. Most do make up for this, in some measure, by employing large serifs (more on serifs later). Monospaced fonts take up a lot of room, especially horizontally. Well-known fonts of this type are **Monaco** and **Courier**.

Proportional typefaces

The large majority of typefaces are proportional. The distance between the letters is determined by which character follows another. Each combination of paired letters has a predetermined space relationship, based on eye measurements of the designer of that particular font. Letters with proportional spacing appear closer together without overlapping each other, the shape of each word and each line of text can be clearly perceived, and the text itself is much more pleasant to read. The art of determining the distance between pairs of letters is called “kerning.”

With proportional fonts, you must use the tabulator to vertically place elements such as words, lines and columns under each other, or to place the various segments in their respective fields. The old method of

using the space bar will not work, because neither the character spacing nor the space between the characters is the same for any two kerning pairs. Examples of proportional fonts are *Helvetica* , *New York*, *Times* and *Geneva*.

Serifs

All typefaces can be divided into two basic categories: Serif and Sans serif. “Serifs” are the small lines or outcroppings that appear horizontally (or at straight angles) on the ends of curves and lines in the letters. A capital A, for example, will consist of 3 serifs: one at the point on top, and one on each of the “legs.”

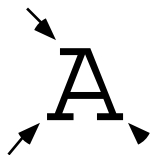


Fig. 1 Serifs

Sans serif fonts do not have these serifs, or small lines. The purpose of serifs is to guide the reader’s eye from left to right along the line. They help group the words in such a way that they can be perceived as distinct units. Large blocks of text can be more easily (and thus more speedily) read in a serif typeface. *New York*, *Courier* and *Times* are all fonts that have serifs, whereas *Helvetica* , *Geneva* and **Chicago** do not.

Choosing fonts in reports and windows

Sans serif fonts are usually not recommended for large blocks of text. Headings, on the other hand, are an altogether different matter; they are meant to attract attention. In any case, all the rules of good layout apply.

Fonts in window fields

For editable fields in windows, the situation is somewhat different. One of the most important

considerations is ease in editing; it must be easy to place the cursor between the letters. Proportional fonts in small sizes can sometimes cause big problems for the user in common editing tasks (typically when “i” and “l” appear in close juxtaposition, such as in the word “fill”). This factor must be weighed against the amount of room available for fields in a given window, since monospaced fonts generally take up more room. It is also possible to experiment with the size of the font to obtain an editable result.

Field labeling on screen

You will probably never be able to avoid writing beside each field in the window what the user is supposed to write in them. Without labeling, you risk having the wrong data in the different fields. Nonetheless, these labels are not central elements, because the user will soon learn the right order. The field names are meant more as “reminders” than as key elements. Therefore they can be set in small, inconspicuous letters. The same goes for reports. Even though labeling is very important, the labels themselves need not be read every time.

(The Omnis help system is an alternative, but it forces you to glance back and forth from the help bar at the bottom of the screen to the window field that you are pointing at.)

Gray tones and shadows

Dark text on a gray background is a mild, pleasing contrast, apt for less crucial elements. Entry fields filled with white will stand out clearly against a background that is light gray. In addition, the gray tone will make it easy to create shadows and three-dimensional effects. Many commercial applications make effective use of such effects. The illusion of inset and outset frames is quite popular, and with good reason: we are equipped by nature to cope with a three-dimensional world.

The color palette

The default colors in Omnis include few gray tones. But it is not all that difficult to make your own even gray tone scale with which to “fine-tune” your windows with subtle shadings, giving them a pleasing look. For example, let the left column in the tools palette contain black, white, and a couple of strong colors, and use the entire right column for gray tones all the way from dark gray to off-white. Let most of the tones be in the light end of the scale. (Windows users will for the most part have to resign themselves to using the existing set of colors and gray tones, or try their luck with “dither” patterns.)

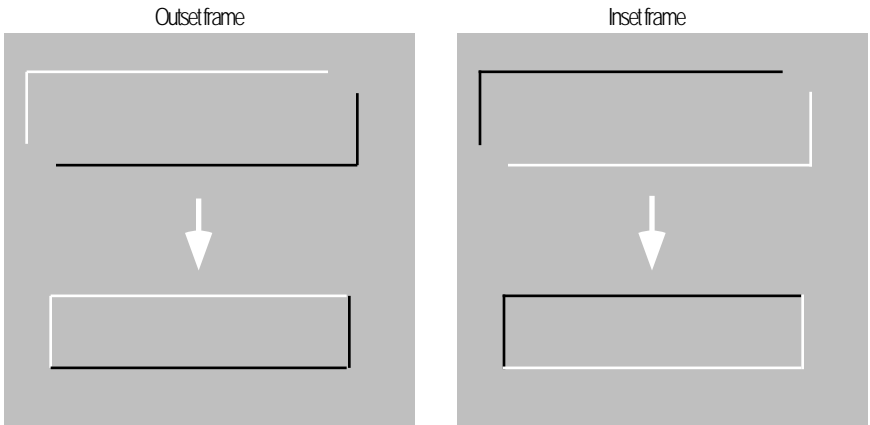


Fig. 2 How to draw outset and inset frames

Outset frames

Use a light gray background. Draw four separate lines in such a way that they form a rectangle just outside the field that you want to stand out three-dimensionally. (Hold down ALT/ CMND to draw the lines evenly.) Fill the field with the same gray tone as the background. Select the left and upper lines of the rectangle and set the foreground color to white (tools

palette, letters in different colors). Then select the right and lower lines and set the foreground color here to dark gray. You may also use black, but the contrast might be unnaturally sharp.

The white lines give the illusion of an edge. The impression is one of light shining in from the upper left. The dark lines give the illusion of a shadow cast by the outset frame. You can create the shadow illusion alone by just using an angle of dark gray lines. Alternatively, this angle can be laid just outside the edge of a monochrome, black rectangle, creating the impression that the figure itself is casting a shadow.

Inset frames

The procedure here is the same as for outset frames, except that the lines are colored (in pairs) the opposite way. The left and upper lines should have a dark gray tone, and the right and lower lines should be set in white – just the opposite of the outset frames. The light and dark lines give the illusion of edges and shadows, respectively, as mentioned above, and the “light” is perceived to be shining in from the same angle.

Greater depth

To give the impression of more pronounced inset and outset effects, the lines can be made thicker. Or you can draw an additional set of lines and move them one pixel down and one pixel to the right. You can copy each of your lines with the aid of ‘Copy’ from the ‘Edit’ menu or create a copy directly by holding down the OPT/CTRL keys. The length of the lines must often be shortened to avoid overlapping. Make fine adjustments to the placing of the lines with the aid of the arrow keys, which move the objects one pixel at a time. The effect of depth will be more convincing with lines that are set close together than with one thick line, because the ends of the lines ultimately determine the shape of the inside (or outside) of the three-dimensional box. The more lines that are under each other, the greater the sense of depth.

Saving the best for last

You will have use for all the imagination and creative energy you can muster when designing the appearance of your windows. Window design, however, is something you should save for last, because every alteration of a field's size and placement necessitates a corresponding adjustment of every depth and shadow effect.

3-D effects in v3.x

In v3.x, three-dimensional effects are already included. Now, all kinds of window fields may be given 3-D effects simply by selecting an option. Still, the techniques described previously may still have a purpose, as it is nice to divide various parts of your window by using 3-D boxes or bars.



Don't let more than 2 or 3 elements compete for attention.

And don't try to distinguish between too many levels of gradation.



Logical Arrangement of Menus and Windows

Apart from the file structure, the way in which windows and menus are logically arranged is the key factor in the effectiveness of an application. The user-friendliness of the application depends upon the windows' being arranged in a readily grasped pattern. This is necessary to make it easy to find your way in the maze of dissimilar (and increasing number of) functions in the application. A confusing application does nothing but give an impression of amateurism.

Menus

If the user can recognize certain elements, you're off to a good start. For Macintosh users this means that the Apple, 'File' and 'Edit' menus appear in their usual order. Windows users usually see corresponding menus in the same locations. Thus we already have two sorting groups. (The Apple menu can be edited only slightly.)

File

The 'File' menu, as a rule, contains commands that involve larger and more generalized tasks related to importing and exporting data, as well as entering and exiting the program. The following are typical commands in this regard:

- Open data file...
- Export data...
- Import data...
- Select Print Destination...
- Print Report...
- Page Setup...
- Choose Printer...
- Quit

Edit

This menu cannot be altered in version 1.x; but since it contains ‘Cut,’ ‘Copy,’ ‘Paste’ and ‘Clear,’ it is best to use it as it is. In version 2.x, however, even the ‘Edit’ menu is fair game.

Choosing menu titles

The application’s distinctive character is an important factor. If the user’s workday consists of three or four main tasks, each with its own window, then it is just as well to assign a window for each task. For most retail stores, the set of menus might look something like this:

**🍏 File Edit Sales Inventory Credits
Reports**

Fig. 3 A menu line in a concrete style

The layout shown in Figure 3 is easy to follow. If, however, you have many different windows, there might be *too* many, and the screen may be too small to accommodate them. In that event, you must distill the commands to a set of common features and sort them accordingly. One possible solution is the following:

**🍏 File Edit Windows Commands
Reports Tools**

Fig. 4 A menu line in a more generalized style

The most important thing is that the user should not have to search for the function that fits what he or she is trying to do.

Hierarchical (cascading) menus

According to Apple’s Human Interface Guidelines, hierarchical (or cascading) menus are to be avoided because they tend to hide commands, making them less accessible. Nevertheless, such menus are useful in

applications that contain a great many commands. The menu line, after all, is limited; but with hierarchical menus we can fit in a lot more. The important thing is to ensure that the title of each submenu is representative for all the commands it contains.

Windows

The same ground rules apply for menus and windows. Each window should have a main theme, so to speak, one that is broad enough to allow for more than one command. As a point of departure, it is a good idea to create a window for every file format that will be set to Read/Write. If the reports are not placed in their own menu, then we need a window with a list so that we can select reports for printouts. In addition, there are all the smaller subsidiary windows.

Subsidiary windows

Windows and dialogue boxes that crop up here and there are uniquely suited to attracting attention. They are perfect for receiving text, numbers, or user's choices when the application cannot proceed until these have been received. There is plenty of opportunity to steer the user away from dubious operations and, with the aid of labels and messages in the smaller windows, clearly indicate what is going to happen. The procedures, buttons, and fields will be separated from the mother window, which in turn makes the programming easier.

Hiding and showing fields

You can switch back and forth between hiding and showing fields within the same window in order to guide the user's input. This is an alternative method that works best for simpler tasks that don't require a great deal of feedback to the user. If the user is expected to do something that's not relatively self-explanatory, then it is best to reserve a special window for that purpose. The procedures for showing and hiding the remaining fields must be adjusted if the

order of fields in the window changes (which happens all the time!).



The application as a whole should be adapted to the user's daily work situation after the initial training phase.



Section 3: The Database Engine

Chapters:

1. Data Structure: Memory & Hard Disk
2. Field Types & their Function
3. File Connections



Data Structure: Memory & Hard Disk

Some Basic Terminology	2
Database terms	
Structure of Data on Hard Disk	4
Datafiles	5
File Formats	7
Main file	
Records	9
Fields	10
Indexes	12
What is an index?	
Building indexes	
Indexes spread over several fields (v2.x)	
Binary search	
The Function of the Internal Memory	18
How the Internal Memory is Organized	19
Current Record Buffer	
The rest of the memory	
Several datafiles in memory (v2.x)	
Variables	25
Local variables	
Format variables	
Global variables	
Manipulating variable declarations	
Hash (#) variables	
Parameters and parameter passing	

Some Basic Terminology

In computer terminology, the word “file” is a very broad concept. A file is really nothing other than a restricted entity within a storage medium that the computer can handle. Files are structured in a recurring pattern, which makes it possible to search them. There are files on both hard disks and diskettes. In these storage media, files are basically programs, applications, or documents, e.g. programming code or stored data – roughly speaking.

Database terms

For a database, the situation is a bit different. Here, a file or file format is a set of user-defined fields of different kinds; these fields determine the pattern into which the entered data will be organized. (In SQL terms the file formats are called “tables.”) The file is the blueprint, as it were, for the recurring structural entity. When the structure is filled with data, the concrete realization of this entity is called “records.” These records are placed sequentially according to a system that allows them to be searched.

The “shoe box” analogy

Figure 1 illustrates the realization of the concept of files, records and fields. Think of a little shoe box filled with cards, where the names and addresses of our friends are written down (one person per card); this describes the data in a file format, which in turn is the blueprint for how each card is supposed to look. The file format is also the basis for the way in which the data that is written in should be structured. When the end-user fills the fields with data, he or she is actually creating records in the datafile, one record at a time. So each card in the box in Figure 1 is a collection of records; these correspond to a set of values that fit the fields that were defined in the file format. In Omnis more than one file can exist at a time, and these are stored in a datafile, which can be likened to a large trunk filled with a number of smaller boxes with cards.

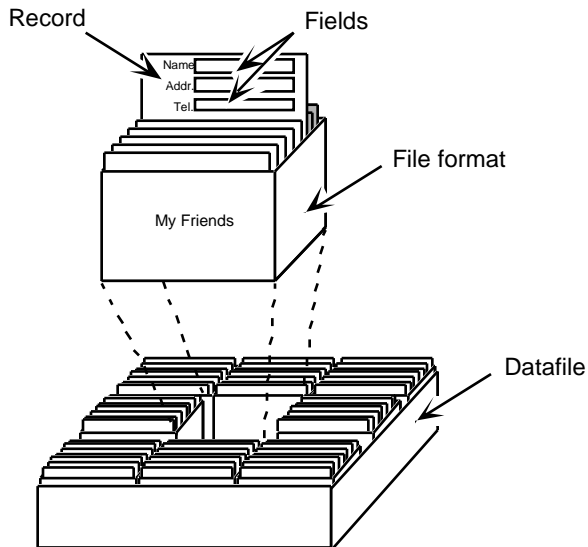


Fig. 1 The database terms in a shoe box

Structure of Data on Hard Disk

All information written in by the user is stored on the hard disk and is part of a dynamic system on four levels:

- 1 Datafiles
- 2 File formats
- 3 Records
- 4 Fields

This breakdown into datafiles, file formats, fields, and records gives the data a spread of four dimensions. In other words, it takes four indexing (pointing) “bits” of information to arrive at a specific value. You must know which datafile, which file format, which record, and which field leads to just the bit you are looking for. This means that all the information is spread out in a network of indexes, where you can get hold of it in a flash.

The first three dimensions are more or less stationary and are changed only as needed; the fourth dimension is specified in each instance (Entry fields or calculations). In what follows we shall look more closely at each of the four levels.

Datafiles

In Omnis 7 v2.x, several datafiles can be open at the same time. Unless the developer has stipulated that a file format belongs to a specific datafile, Omnis will store the data in the datafile currently in use. This is called Floating datafile mode. With the aid of the ‘Set current datafile’ command, the developer decides which datafile shall be used, just as with Main file.

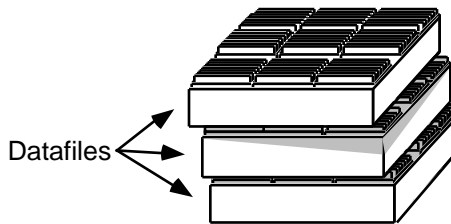


Fig. 2 How the datafiles might look in “real life”

Controlling the flow of data

A file format can be linked to a specific datafile by using the ‘Set default datafile’ command. After this command has indicated which file formats belong to which datafiles, Omnis will direct the data to the right datafile, regardless of what Current datafile is at that moment. The Current datafile setting is not affected. See the chapter entitled “Datafiles and Libraries.”

Datafiles and CRBs

Each datafile has its own independent CRB with its own Main file. This holds true even if a number of datafiles are using the same file formats. The field names are distinguished from each other with the help of the datafile’s name. If the ‘Unique Field names’ option isn’t crossed off in ‘Preferences,’ the name of the file format must also be given. These three names

are separated by a period. The full field name will read as follows:

`datafile.fileformat.fieldname`

If the datafile is not given in the exact form shown above, Omnis will favor the field that belongs to the CRB for the current datafile. The developer is free at any time to use fields from all the datafiles, provided he uses the full names.

File Formats

The datafiles consist of data entered in accordance with the pattern of the file formats. The latter determine the appearance of the records. File formats act as “templates” for how the datafile is organized and how the information is read in. A file format is the definition of a set of fields, their field types, and whether they should be indexed or not. The fields within a file format are identified by the order of their number in the file format window.

Main file

There are some commands that affect only one file at a time. They need to know which file this is; the ‘Set main file’ command comes to the rescue. This command tells Omnis which file you want to work with, and this is how commands that use Main file are directed. Apart from the foregoing, the command has no other effect, either on the fields or their content. The following commands need to know Main File:

- Clear main file
- Clear main & connected
- Prepare for insert
- Prepare for insert with current values
- Delete
- Delete with confirmation
- Build list from file
- Find
- Prompted find
- Next
- Previous

Main file and CRB

In Omnis we navigate through the CRB by indicating which file format we are working with at any given time. The ‘Set main file’ command is only a pointer, however, and has no other effect to begin with. It’s only when one of the Main file-oriented commands is executed that the Main file setting will have an important role to play. It then determines which file will be affected (the file contents, that is).

Most other commands are not concerned with what Main file is set to. The ‘Calculate’ command, for example, is completely independent of Main file and can combine fields from everywhere without any restrictions.

Records

Records can be thought of as the individual “cards” in the shoe box archive (datafile) shown in Figure 1 at the beginning of this chapter. When the fields in a file format are filled up with data by the user (or retrieved during import), the field values taken as a whole will constitute a record. Records are, in a word, a collection of the field contents within a file format as the file is being updated (saved to disk). If the user only fills in a few fields before the record is saved to disk, the record will only consist of data from these few fields. If, however, the user fills in *all* the fields, the record in the datafile will consist of a complete set of values from all the fields in the file format. The records are the finished product; the file format is the “recipe.”

Fields

File formats consist of fields whose contents are usually stored on disk in the form of records. Fields that are only present in memory are called “variables.” What both types of fields have in common is the fact that their contents (text, numbers, etc.) stay put until they are actively modified by a command. They may also be edited by the user or the developer, typically under Enter data, or by means of the Field Value window. The field type (text, number, etc.) determines the type of information the field will contain so that the former can be handled effectively. Omnis filters all data that is about to be entered in the fields, and does not allow the entry of any information that is incompatible with the given field type. The different types of fields are described individually in the chapter entitled “Field Types.”

The fields as drawers

We may regard fields as “drawers” that can be filled with values. The drawers themselves are of less interest than what they contain. Drawers can be emptied and filled and their contents copied from one field to another. When a field is empty, it has no length, and the expression ‘len(EMPTY_ FIELD)’ will be equal to 0.

The relationship between window fields and fields in memory

In Figure 3 I’ve tried to pictorially describe the relationship between the memory (containing the CRB) and the fields as they appear to the user. The windows only reflect the values of the fields in memory every time the window is redrawn. If the field values change, the window won’t show the changes until the next redraw.

Let us imagine that the CRB lies hidden behind the windows and that every time a window is redrawn, the CRB sends the needed field values to that window. Windows act as a kind of shell surrounding a core – the CRB; moreover, they behave much like pictures on

a wall. Both show a lot but have no real content. The “live” fields (and the actual content) reside in the memory, behind the windows.

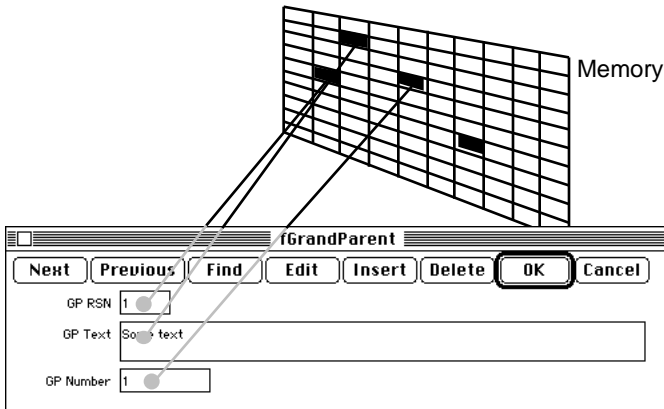


Fig. 3 A freshly redrawn window shows the contents of the CRB.

Redrawing windows

When fields are placed in windows, what we actually see are copies of the fields that are in memory. The field values in the CRB may very well change as the procedures are executed. It would be nice if the window fields were constantly updated; but as this would only slow things down unduly, Omnis has left this in the hands of the developer, who can update the window fields at just the right time using the various Redraw commands.

We are not restricted to fields from the same main file in a window, even though this is often advisable. As the values in the CRB change, we can update the window by executing a 'Redraw windows' command.

Indexes

What is an index?

We may regard an index as a list that shows the order of all the records in a file format in a given datafile. When the index is constructed, the records are sorted according to the field to be indexed; the order is alphanumerical. When a field is indexed, Omnis can use the index list to do a binary search (a very quick search method). Moreover, indexes can be used as pre-sorted sequences when printing out reports, during export procedures, etc.

(In reality, an index is a “tree structure” in which each branch represents an index element embedded in the relevant part of the index.)

Building indexes

Consider the following practical example. We start with the field definitions:

Field name:	Type:	
1 RSN	Sequence	Ind
2 NAME	Text	Ind

We write in 10 names in this file format. Every time a file is updated (by using the ‘Update files’ command), Omnis finds the last sequence number (RSN) in the file and increases this by one. Thus the RSN shows the chronological order in which the names were entered.

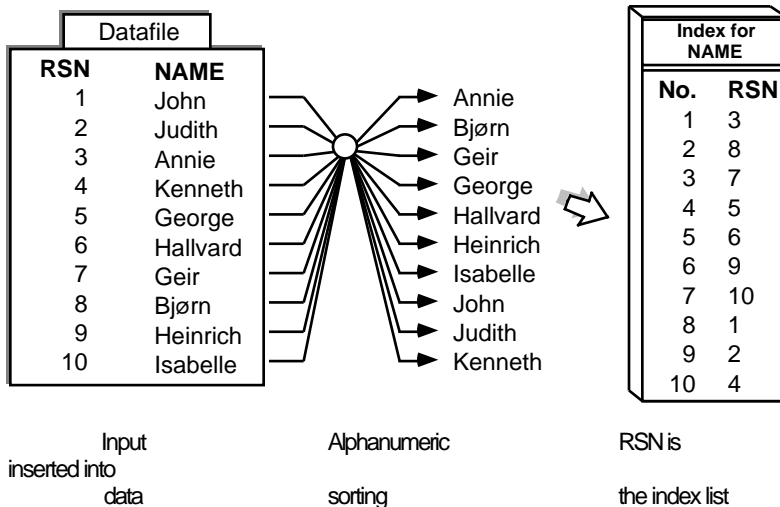


Fig. 4 Building an index

The index for NAME is constructed as the field is sorted alphanumerically, after which Omnis finds the related RSNs; thus the numbered list of RSNs constitutes the index. As new records are entered, Omnis places them in the various index lists where the field values belong, thus updating the indexes continuously.

Indexation during run-time

In reality, indexing takes place continuously as new data is entered. To illustrate the principle more clearly, we let indexing take place in the example above only after the first 10 entries. (This situation can arise if a field is set to be indexed after a certain number of records have been entered. During re-indexation, the process is approximately as illustrated above.)

Indexes spread over several fields (v2.x)

I Omnis 7 v2.x, indexes may be constructed from a number of fields. This is useful when many of the same values recur in the data for a given field and you wish to continue the sorting in another field. The developer defines indexes with the first, second, third, (etc.) field to be sorted within the same index. Within the index list, the order of entries having exactly the same values in the first field of the combined index will be arranged according to the alphabetical order of the next field in the “index field list.” That’s quite a mouthful, so let’s take a breather here:

Typical examples are fields such as FIRSTNAME and SURNAME. It is customary to sort according to surname; unfortunately for us, however, there are many people with the same surname. If we only follow the index for SURNAME, the order of persons with the same surname will be identical to the order of the RSN. This is rarely ever correct. In this case the best thing to do is to employ a combined index for SURNAME, which will consist of SURNAME followed by FIRSTNAME. Then everyone whose surname is “Smith” will be sorted further by their first name.

Binary search

The reason that fields which are indexed can be searched so rapidly is due to the use of what we call a “binary search.” When a character field is indexed, this means that the content in the fields is sorted in the same sequence as the current character set (for example, ANSI). Here each letter has a number, and the numbering is alphabetical. With the aid of the character set, Omnis can ascertain whether a word is “larger” or “smaller” than another word. A word that is “larger” begins with a letter that has a higher number than the word it is being compared with, i.e. the initial letter of the “larger” word is farther along in the alphabet. If both words begin with the same letter, the next letter in each are compared.

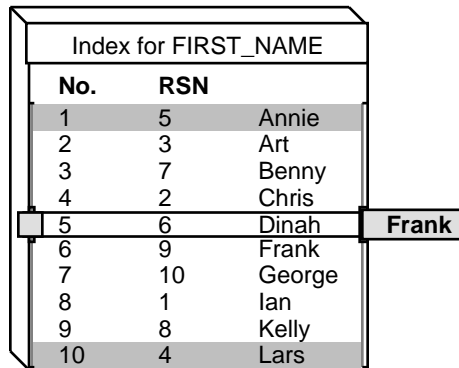
Logic principle

Binary searches exploit this by extracting a value embedded in the area of the datafile it is searching in. If the text to be searched (the word the search is trying to find) is larger than the value it extracts, this means that the desired record is in the bottom half, i.e. *after* the

value being extracted. If it is smaller, this means that the record is in the upper half of the area being checked, i.e. *before* the value being extracted from the sequence. This process continues until the field of search is narrowed down to one record per line, after which the search is terminated.

Practical example

The datafile contains 10 records with first names sorted alphabetically. The index for the field FIRST_NAME will be as shown below. We want to find “Frank.”




No.	RSN	
1	5	Annie
2	3	Art
3	7	Benny
4	2	Chris
5	6	Dinah
6	9	Frank
7	10	George
8	1	Ian
9	8	Kelly
10	4	Lars

Fig. 5 Searching for Frank, first try

Our first comparison is with record 5 in the index list. Here we see that “Frank” is larger than “Dinah,” so we can concentrate on the bottom half of the index list.

Index for FIRST_NAME		
No.	RSN	
1	5	Annie
2	3	Art
3	7	Benny
4	2	Chris
5	6	Dinah
6	9	Frank
7	10	George
8	1	Ian
9	8	Kelly
10	4	Lars




Frank

Fig. 6 Searching for Frank, second try

We had to choose between records 7 and 8 in the index list, since it would be pointless to try to extract records from between the index lines. Our comparison shows that “Frank,” being smaller than “George,” must lie in the top half of the search field. (The search field is between the two gray lines in Figure 6 .)

Index for FIRST_NAME		
No.	RSN	
1	5	Annie
2	3	Art
3	7	Benny
4	2	Chris
5	6	Dinah
6	9	Frank
7	10	George
8	1	Ian
9	8	Kelly
10	4	Lars



Frank

Fig. 7 Found Frank on third try!

Binary versus linear search

Binary searches require few comparisons to find the right record. Linear searches (which are used with the 'Search list' command) usually start from the top and work downward, one line at a time. If the amount of data should increase by one record or one line, a linear search will usually have to do an entire comparison again. Binary searches only do another comparison when the amount of data is doubled. This has a dramatic impact on the quantities of data that the various search methods can practicably cope with.

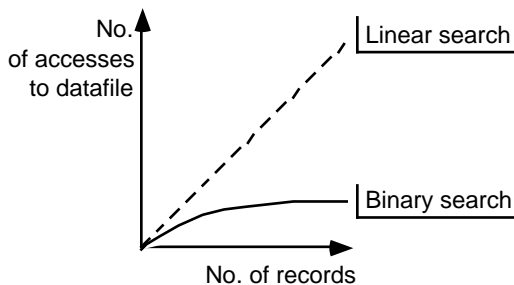


Fig. 8 Comparison of binary and linear searches

As we see in Figure 8, a binary search can handle, in practical terms, far larger amounts of data than a linear search can. The work Omnis does in maintaining your indexes is sure to pay off.

Binary searches on lists

When a list has already been sorted, a binary search should prove to be a useful alternative to the 'Search list' command. True, the programming involved is somewhat complex, but it is manageable, and the increase in speed is welcome. See the chapter entitled "Lists and Tables."

The Function of the Internal Memory

Every computer is amply endowed with a volatile medium of logically oriented transistors, which is often called Random Access Memory (RAM), otherwise known as the “internal memory.” This is the arena in which a computer can spread its wings, as it were. All data, i.e. everything that is routed through the CPU, is assigned to specific addresses in RAM. Later the data is retrieved, modified, compared, sifted, moved, measured and weighed, and restored. All this takes place at incredibly high speeds. Theoretically, it is possible to use stable storage media (e.g. hard disks); but not only do these run too slowly, they wear out too quickly.

The contents of RAM can disappear

When the power supply is cut off, everything in the internal memory vanishes. This is why we need a stable storage medium (diskette or hard disk) for storing a partially compressed copy of what we had in RAM. Diskettes and hard disks, as we know, do not need a power supply to retain information.

An arena for both “tools” and “products”

What the user does with the aid of the computer, i.e. writing letters, making drawings, composing music, typing columns of figures, etc. is assembled in the internal memory. This is done using a variety of programs, which also reside in the internal memory. Every program is allotted a portion of the total internal memory when the computer is booted. Omnis fills this space partly with its own program code, but leaves the rest for windows, graphics, field values and lists, etc.

Other parts of the allotted space are used for different kinds of buffers, i.e. temporary auxilliary storehouses whose task is to enhance the execution of a number of commands. Efficient use of the internal memory is a key element in any good program. Omnis is no exception.

How the Internal Memory is Organized

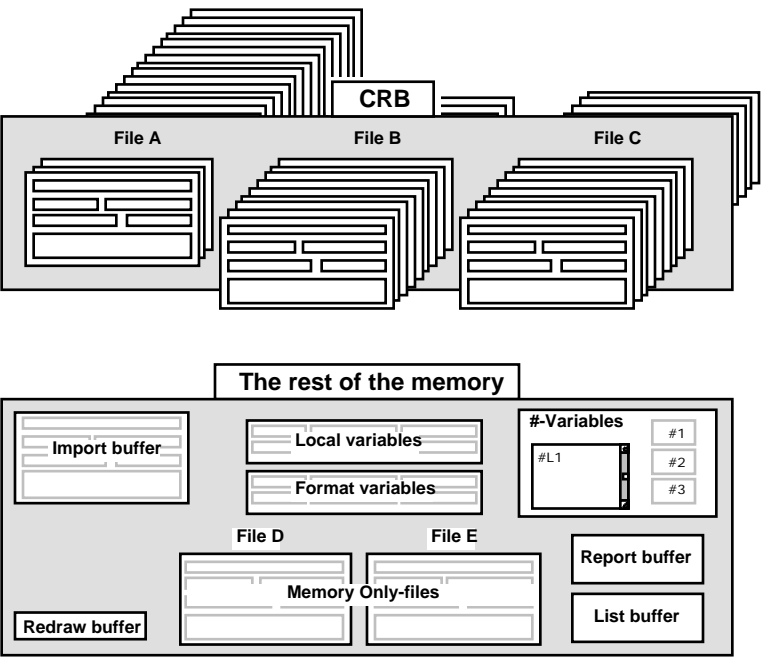


Fig. 9 Simplified diagram showing the organization of the non-code part of the memory allocated by Omnis

Current Record Buffer

Current Record Buffer (CRB) is a major concept in Omnis. CRB is defined as the content of all fields in all files (those to be stored on disk), i.e. all those fields from which the file formats are constructed. Data entered by the user generally winds up in the CRB. In the figure above, the Memory Only files are outside the CRB. We will come back to this later. CRB has two characteristics worth noting. The one

concerns the combination of records, the other concerns the updating of the datafile.

Combination of records

When a record in a given file is found with the 'Find,' 'Next' or 'Single file find' commands, its fields are filled with the relevant data. CRB thus becomes a new combination of field values within the file formats. As shown in the figure above, we can regard the CRB as an assortment of cards, one from each stack.

The CRB and 'Update files'

One of the guiding principles in Omnis is the way in which it updates files. All files are updated in one fell swoop when an 'Update files' is executed. When updating is performed by the 'Prepare for Insert' command, a new record is established in the main file slot, while the records in the rest of the CRB are saved to disk in their present form. For these files the old version is replaced by the new. With 'Prepare for Edit' the old version of the record (on disk) is replaced with the new version (in RAM) for all the files. This method of updating enables us to modify records in all kinds of files under the same Enter data and subsequently carry out a collective updating for all the files.

Memory Only files

In practice, fields in files which are eventually set to Read/Write will be given the most attention during CRB evaluations. Fields in Memory Only files are never saved to disk. In this respect they differ very little from hash variables. Nevertheless, technically speaking they are a part of the CRB. In Figure 9 I've chosen to place Memory Only files on the outside, because it will be helpful to regard the CRB as that which is being written to disk when an 'Update files' command is executed.

Read Only files

The contents of Read Only files are never saved to disk by 'Update files.' This is how these files differ from other files in the CRB. Read Only mode is a means of protecting files against modification. You can prevent all other files except one from being updated by setting these to Read Only. This is advantageous in multi-user environments, as it can prevent records from being locked. See the chapter on multi-user applications.

What does it mean that a record is in internal memory?

The 'Test for current record' command tests solely whether a record has been stricken in its entirety or not, i.e. whether it has been assigned a hatch in the internal memory. No matter how much of the field content is deleted or altered, this test will not be affected. However, we can only be absolutely certain that all the field contents of a record exist in memory just after a successful search has been carried out, or just after the user has entered the data for a new record. By "just after," we mean before other procedures are executed which contain commands that modify the CRB.

(To be sure that the information going into a datafile contains what it should, these tests in the field procedures should be run as the user inputs the data. The use of field procedures is the most orderly way to go about checking the contents of the fields.)

The rest of the memory

Along with the CRB, the internal memory consists of various buffers and variables, each with their own separate functions. The intention here is to avoid unnecessary field changes in the CRB.

Import buffer

The import buffer receives data from the port or import file before it is distributed to the various fields in the CRB. This takes place with the aid of the 'Import data' or 'Import field from file' commands.

Variables

The variables contain values just like the fields do, but their content is not saved to disk. The developer may create his or her own variables (local, format and library), or use those that already exist, i.e. #hash variables.

Redraw buffer

The redraw buffer contains the value that a field had the last time it was redrawn (either with a redraw command or by the user placing the cursor in the field). The redraw buffer is used as a basis for comparison for determining #MODIFIED.

Report buffer

The report buffer is used in building reports. It contains what is ultimately presented as the report itself, including text, graphics, field values, and what have you.

List buffer

The list buffer is used in the building of lists so that the CRB isn't altered when Omnis searches through the datafile, record by record.

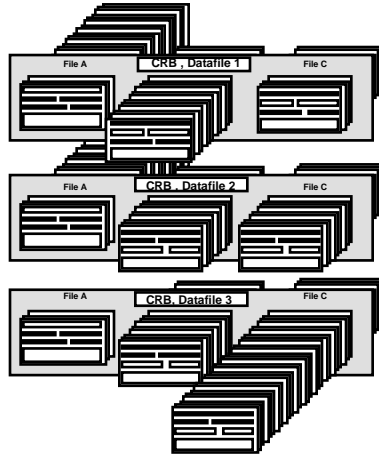


Fig. 10 The true complexity of CRBs

Several datafiles in memory (v2.x)

When several datafiles are open at the same time, each of them has its own CRB, even if they are coupled with file formats and fields with the same name. All the fields in all the datafiles can be accessed by including the name of the datafile in each field name, separated by a period. For that matter, each individual datafile is generally connected to a library with its own file formats and related fields. This opens the way for better modular programming, where each library can have its own function and datafile. Different modules can thus be tailor-made to fit the need.

Backup and security

Using several datafiles can be quite useful when making backup copies – in fact, when protecting your data in general.

Saving time and storage capacity

A complete backup of an entire datafile can take hours, even with speedy hard disks and tape streamers. Backups must be made frequently and represent a great deal of boring work, even though the backup routine can often be automated. One way of minimizing this work is to separate volatile data from stationary data

and put them in their respective datafiles. This will spare you from “dead weight” being repeatedly backed up without change.

If lots of new data is being produced continuously and the old data is only being kept for storage and for “just in case,” you can shunt data that is no longer used into special datafiles. For example, you can program your application in such a way that it creates a new datafile for each year. It’s a simple matter to choose the datafile that fits the year: just look at the registration date.

Security

Highly sensitive data can be sectioned off into separate partitions or hard disks that are well-protected from hackers. This is usually not necessary, however, because Omnis’ datafiles are coded in such a way that they can’t be easily read by disassemblers or other low-level editors.



Better method in your madness than madness in
your method.



Variables

Local variables

Local variables are variables that can only be used in the procedure in which they were created. They exist only as long as the procedure is running. When the procedure is finished, both the content and definition of the variable vanishes from memory. In v3.x you can set the initial value directly in the variable declaration.

Viewing the contents during the run

During debugging you will notice that all the local variables are empty when the procedure has finished running. To see the content of variables at the end of the procedure, you can insert a breakpoint just after the last procedure line; the procedure will halt before it is completed, and the variables will remain intact.

Areas of use

Local variables are invaluable for clarifying complicated procedures. You should try to replace hash variables with local variables in most situations, because the latter can be given meaningful names. This, in turn, will ease the programming process considerably and free you to concentrate on the logical problem in the procedure, instead of your having to remember a sequence of numerical variable names (those nasty #-variables).

User-defined constants

It's also wise to replace key numbers in the procedure with local variables so that they function like constants. In calculations this applies to any number that you're not sure about (for example, if the fixed value happens to change). If the number has to be changed at a later date, you won't need to insert the correct figure yourself; all you'll have to do is change the value of the constant.

Format variables

Format variables can be used freely within the same format. This means that the format variables in a window are visible to all the procedures in that window; the same goes for menus. The content of these variables is retained until they are actively deleted, or until the “owner” format is closed (if the ‘Clear format variables when closed’ option has been set). The declarations (the ‘Format variable’ commands) may be collected in a separate procedure, all in one place. Format variables work like local variables, but they hold the edge when other procedures are called.

Initializing values

Format variables must often be set to null or set to departure values at the beginning of the procedure. This is necessary in order to ensure that the value is what you want it to be. It can happen, for example, that another procedure from the same format has just been run and you have forgotten that both procedures make use of (some of) the same variables.

Limitations

The disadvantage of format variables is, not surprisingly, that they do not exist for other formats. This can throw a monkey wrench into everything if, for example, we want to print out a report from a procedure in a window or a menu. The format variables from the menu or the window will not be visible to the report format. Here you must resort to global variables.

Global variables

Global variables are variables that are visible to the entire application. One example is the fields in Memory Only files. These are visible for the “owner” library, but can be reached from other parts of the application as well. You can assign meaningful names to the fields, preferably within a common file format called, for example, “fVariables.”

Areas of use

Certain options, internal information, lists, settings, etc. that the developer wants to refer to in many different procedures and formats (and wants to keep from being used where they shouldn't) are prime candidates for global variables. Moreover, such file formats are handy to have if you need the kind of variables that are not covered by the group of hash variables – for example, date fields, time fields or picture fields.

Library variables

In Omnis 7 v2.x and 3.x there are also Library variables. These are visible to all the formats within a complete library and are defined in the same way as format- and local variables.

Manipulating variable declarations

Whether you are used to defining all the variables in a procedure before programming begins or prefer doing so as you go along, sooner or later a situation is bound to arise in which you'll want to change them a bit.

Changing name or type

All types of variables and fields are referred to in the procedures by a hidden number, not by their names. When a variable is created, it is added to an internal “file format,” which functions much like a regular file format. That's why it's easy to change the name of every variable: just edit the name directly in the declaration command ('Define local variable,' for example), and Omnis does the rest.

Deleting variables

If you've created a slew of variables that you later find you have no use for, you will quite naturally want to get rid of them. But it's not enough to delete the declaration commands. The variables are retained in memory until 'Remove unused variables' is selected

from the 'Modify' menu. But then the undesired declarations 'Define local variable,' 'Define format variable' or 'Define library variable' must first be deleted and the variables in question must not be used anywhere in the procedures. To track down all the occurrences, use 'Find and Replace' with 'Field names' checked off. When you have used 'Remove unused variables,' you can check the result by looking at the 'List field names' window. If the variables do not occur here, this means that they are no longer in memory, provided the correct format (and the correct procedure) are in view.

Perhaps this sounds complicated. It is sufficient to enter a definition to create a variable. (You never have to run the procedure line.) Why, then, go to so much bother to delete a variable? The answer is that this is how it works best in practice. Creating a variable is much easier and less risky than deleting one, because deleting a variable can render one or more procedures completely unrecognizable – and thus useless.

Moving the declarations

When we try to move a variable declaration with the aid of cut and paste, something happens that not everyone appreciates: Reinserting the declarations is interpreted by Omnis as an attempt to create new variables with the same name as the old ones. The existing variables are still in memory, even though the declaration commands themselves have been deleted by using 'cut' from the 'Edit' menu. The new variables are given the same names as those that already exist, plus a "1" (or the lowest number that makes the names unique).

In v2.x (and v1.3) you can delete the number at the end of the names. This is interpreted as a new declaration of the old variable (without the numeral appendage), and the connection is re-established. We can breathe easy. Finally, all we need to do is delete the unused variables hidden in the memory (for example, `Lo_Counter1`, `Lo_Length1`, `Lo_Height1`, etc.), which

we achieve by selecting ‘Remove unused variables’ from the ‘Modify’ menu.

Hash (#) variables

In Omnis we have a fixed set of variables, all containing names that start with the symbol #. These are called “hash variables.” They are pre-declared and may not be removed from memory. Some are also “read only,” which means that even the developer cannot modify their content. Hash variables can be subdivided as follows:

Editable variables (“Value” variables)

These are the familiar variables that everyone uses now and then when they need a variable or counter of some kind.

#1...#60: Number variables

These may be formatted with decimals by adding “D,” followed by the desired number of decimal places. Example: #1D2 (two decimals), #1D3 (three decimals), etc.

#L1...#L8: Lists

Quite useful when you need a list in a hurry, for example in procedure testing or as a buffer in a calculation. But lists presented in windows should be declared specially; then they will be left undisturbed by other procedures.

#S1...#S5: Text variables

The built-in text variables take up to 32,000 characters. Their most important use is in procedure testing.

Enter data messages (Read Only)

This is a large collection of variables dominated by Boolean and “read only” fields. They tell nearly everything about what the end-

user is doing under Enter data. Some are also activated outside of Enter data.

#BEFORE, #AFTER, #CLICK, #DCLICK, #TOTOP, etc...

Operating system variables (Read Only)

A handful of variables provide information gleaned from the operating system. Examples of these are #T – time, and #D – today’s date. You can’t change the contents of these variables.

Internal Omnis variables

This group consists of many variables that are useful to Omnis and developer alike. Key information about fields, printouts, etc. is collected for the developer, and some of the variables can also be manipulated.

#L, #LN, #LM, #LSEL

These are the list parameters for each list in memory. #L, #LSEL and #LM can be manipulated.

#FDP, #FD

Default presentation of number fields and date fields in windows and reports.

#UL, #MU

User level in the Omnis security system and in a multi-user network, respectively.

#ERRCODE, #ERRTEXT

Error code and explanatory text. Can be tracked for debugging.

#R, #P

Number of printed records, and page numbers in the report.

This is not an exhaustive list. The examples above are intended to illustrate what different groups of hash variables contain.

Transferring values between applications

Hash variables exist in memory as long as Omnis is running; this means that they can be used to transfer values from one application to another. Hash value variables are also useful because they are so practical. They are always available, and for really short procedures you will rarely use so many of them as to make them hard to keep track of. For testing purposes, hash variables are hard to beat.

Conclusions

Hash variables occupy a central place in Omnis' scheme of things. They are virtually in constant use, and I reckon that most developers will know them like the back of their hands. However, if it's been a long time since you made their acquaintance, there is no shame in consulting "Reference 1."

Parameters and parameter passing

When you set out to write general procedures, which is an art in itself, you naturally want them to be adaptable to some extent. And the product of a procedure is often a value that is placed in different fields each time the procedure is called. Global variables might help, but they tie up the programming and make heavy demands on the developer's memory. For these problems, parameters are the answer.

What are parameters?

A parameter is a local variable in a (general) procedure that receives its value from another procedure that calls it. Omnis sees to it that values are automatically passed between the correct procedures. The value for the parameter is sent over when the 'Call procedure with return value' command is executed. The field provided in the command is the drawer that receives the value that is sent back when the procedure that is called has finished. Commands that may pass values to parameters are the following:

```
Call procedure (F_Field1,F_Field2)
Call procedure with return value (F_Answer, F_Field1,F_
Field2)
Open window (F_Field1, F_Field2) ;; Sent to the O-
procedure
Install menu (F_Field1, F_Field2) ;; Sent to the O-
procedure
```

F_Field1 and F_Field2 contain the values we wish to send, and F_Answer will receive the result. (The "F" only means "File" here.) We may replace the fields inside the parentheses with actual values. Bits of text are enclosed in quotation marks (" "), as follows:

```
Call procedure pProcedures/15 (35, "Some text") {Text-handling
procedure}
```

Values to be used in the general procedure are received in the order in which the parameters are declared. When the general procedure has finished running, the parameters vanish from memory.

The return value

As previously stated, the procedure that does the calling will receive a value in return. Any field in such a procedure, provided the field is the right type, may receive a return value from the general procedure. A field is designated as the waiting receptor in the procedure that does the calling with 'Call procedure with return value.' In the general procedure a calculation is set as a return value with the aid of the 'Set return value' command. We may place fields, numbers, and calculations in the calculation line. The product of this calculation, when the procedure is finished, is passed to the waiting receptor field in the procedure that did the calling. The value ends up safely in the designated field, "sent" from the 'Set return value' command, and "received" in the 'Call procedure with return value' command line.

Calling procedure
Local variable First_Value (Short number 0 dp) Local variable Second_Value (Short number 0 dp) Local variable Here_is_the_Answer (Short number 2 dp) Calculate First_Value as 10 Calculate Second_Value as 20 Call procedure pParameters/2 {Resource procedure} with... ...return value Here_is_the_Answer OK message {The answer is [Here_is_the_Answer]}

1

The calling procedure

(1)

The field 'Here_is_the_Answer' serves as a drawer that waits politely in the procedure to which it belongs. Omnis handles the transport to and from the procedures.

pParameters/2 {Resource procedure}
Parameter Numerator (Short number 2 dp)
Parameter Denominator (Short number 2 dp)
Set return value {Numerator/Denominator}

2

The procedure that is called

(2)

When the procedure is finished, Omnis passes the product of the calculation in 'Set return value' to the procedure that did the calling. Note that we do not need to use global variables to carry out the parameter passing, nor do we need to remember which procedure did the calling. These commands solve an annoying problem in a rather elegant way.

To sum up

The main facts about parameter passing are as follows:

Transferring values to a resource procedure:

The values are placed within parentheses (in 'Call procedure,' for example) either directly or by means of variables. The receptor fields are determined by 'Parameter' commands in the resource procedure.

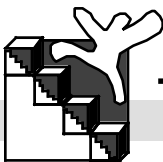
Transferring values back to the procedure that does the calling:

The calculated value to be returned (there is only one per procedure) is determined by the 'Set return value' command. The receptor field is already designated in the 'Call procedure with return value' command. The value is sent when the procedure is finished.



Show me your variables, and I'll tell you who
you are.





Field Types & Their Function

Introduction.....	2
Text Fields.....	4
Number Fields.....	6
8-bit numbers	
32-bit numbers	
64-bit numbers	
Conversion of text into numbers	
#NULL	
Boolean Fields.....	10
#NULL and Empty	
Boolean fields in windows	
Search formats and calculations	
Language differences	
Using Boolean fields with “record templates”	
Date Fields.....	15
Main date field types	
Units in Date fields	
Interpretation of text	
The year range in Short dates	
Calculations with dates	
Presentation in windows and reports	
Archeological dates	
Time Fields	21
Picture Fields	22
Memory requirements	
What is a suitable medium?	
Lists.....	25
Large lists	
Lists in file formats	
Binary Fields	27
Values	
Binary fields in notation	
Sequence Fields.....	28
How Omnis assigns RSNs	
Areas of use	
Resetting the RSN	
RSN and import	

Introduction

When defining a variable or a field in a file format, you must always designate the field type, which determines the type of data for which the field will be used. For example, to register the day a record was entered, we must use a Date field. On the other hand, if the field is intended to show the price of a certain model of car, it is advisable to let the field be of the “integer” type (meaning “whole number”); such numbers rarely ever have decimals.

We could have registered all the information as free text to start with, since this is a simple and straightforward way for your computer to receive data. But then we wouldn’t have been able to do any calculations on them. In some cases, free text would take up more space; and in any event, it wouldn’t be possible to store information in the form of pictures, lists, etc.

Filtering data to be entered

The field type serves as a kind of filter for data that is entered by the user (or imported). The purpose behind distinguishing between different types of data is to get the most out of the internal memory, make it possible to do calculations on field values in a reasonable way, and to act as a middleman between user and computer. The field type provides a backdrop of rules for the kinds of values that are allowed and those that are not. Date fields are typical examples of this. For example: if the user enters text that Omnis is unable to interpret as a date, Omnis will react with a beep, signifying that the input data has not been accepted. This minimizes entry errors markedly.

Filtering data to be read from disk

The field type also works the opposite way. When the value of a specialized field is loaded from disk, it is translated and displayed in a form that the user can easily interpret and understand. On the diskette, all data is present as number codes in the form of ASCII or ANSI characters. Thus a field type is a recipe for how letter characters are to be translated from (and to)

ASCII/ANSI before they are presented to the developer or user. Text (and integers) need no special translating, whereas dates, numbers containing decimals, and images must be interpreted and processed according to specific rules before the content can become useful. Let's now take a closer look at the way specific field types are defined, the amount of memory they require, and how they function.

Text Fields

Text fields accept all kinds of characters that can be entered from any kind of keyboard, and stores them without modification. (Exceptions to this are, of course, TAB, ENTER, RETURN, and similar functions keys.) The length that is specified in the field definition is the limit for how many letters the user will be permitted to type in, *not* how much actual space the field occupies in the memory. One of the advantages in setting an upper limit on the number of characters in a field is to encourage the entry of brief codes with a predetermined number of characters – or, quite simply, to force the user not to write so much. Memory requirements will depend on how much text is present in the field at any one time (1 byte per character). So all the text fields in the file definition can be set to their upper limit. Fields don't take up so much room, either on the hard disk or in the internal memory. Nevertheless, you can save room by actively deleting variables that contain lots of text when they are no longer needed. In v1.x, the upper limit is 32,000 characters; in v2.x (and 3.x) you can have up to *10 million* characters. (The latter represents approximately 15 times the number of characters there are in this book!) This means, among other things, that we can insert relatively large text files directly into text fields and process them with Omnis's procedure commands afterwards.

ASCII

On the hard disk, all information is stored as characters in a “character coding” system. ASCII is an internationally recognized alphabet consisting of 255 “letters” (i.e. characters). The letters of the alphabet are stored in 8-bit format, i.e. eight 1's or 0's are required to identify a letter. For every letter there is a corresponding number. ASCII contains every letter in the English alphabet as well as all the numbers (digits), most national characters for foreign languages, many special characters, and a number of control characters (new line, tabulator, new page, etc.). The precise definition is fixed, but different programs and machines nevertheless deviate slightly from the standard, especially with regard to little-used characters. For example, Macintosh uses a variation of

ASCII where the special “foreign language” characters in particular have a non-standard placement.

ANSI

Windows utilizes ANSI, which is also an 8-bit character code system. The sequence of special characters is different from that in ASCII. This inconsistency is dealt with by Omnis during all conversion between platforms, however. The relevant character sets reside in the file called ‘Omnis 7.ini.’ The difference between ASCII and ANSI is significant for SQL, and we will return to this in a later chapter.

The National and Character field subtypes

The National and Character field subtypes determine how the field is sorted when indexes are built. The National fields are sorted with upper and lower case versions of the letters in lettered pairs, with upper case first {A, a, B, b, C, c...}, whereas Character fields are sorted in the order of the relevant character set {A, B, C, D...a, b, c, d...}.

As a rule, information is written into a text field with a capital letter at the beginning of the first word; the information is often in the form of sentences or names. If the user forgets to start with a capital letter, the record in question will “vanish” to the end of the index if the field type has been set to Character. In later searches, it will appear as though the record has been deleted. To avoid this situation, we set such fields to National. (It is rarely appropriate to sort fields according to ASCII, so we can just as well set all the text fields to National.)

Number Fields

Number fields refuse everything but numbers, commas, and minus signs. The field type may be specified further, depending on how large the numbers are and how many decimals they contain. The basic definition is related to how much room the field types take up in the internal memory, expressed in terms of the number of bits.

8-bit numbers

Eight-bit integers lie between 0 and 255. In other words, the range is limited; on the other hand, the field only takes up 1 byte. This kind of field cannot store numbers with decimals. Fields that are displayed with Radio buttons may be of this type.

32-bit numbers

This format occupies a bit more space – 4 bytes – but also has a much larger range. The first 30 bits are used for the number itself; the remaining two bits indicate whether the number is negative or not.

Long integer	+/- 2 billion
Short number 0dp	+/- 1 billion minus one (9.99×10^8), whole numbers.
Short number 2 dp	+/- 10 million minus one (9.99×10^6), two decimals.

64-bit numbers

The 64-bit format is the largest; it takes up 8 bytes. The exact range depends on the number of decimals and the number of relevant digits. The numbers shown below should really be subtracted by one, but for the sake of simplicity they have been rounded off to the nearest exponent:

Number Floating	+/- 1.0x10 ¹⁰⁰	No. of decimals varies	15 valid digits
Number 0dp	+/- 1.0x10 ¹⁵	whole number	all valid digits
Number 1dp	+/- 1.0x10 ¹⁴	1 decimal	all valid digits
Number 2dp	+/- 1.0x10 ¹³	2 decimals	all valid digits
Number 3dp	+/- 1.0x10 ¹²	3 decimals	all valid digits
Number 4dp	+/- 1.0x10 ¹¹	4 decimals	all valid digits
Number 5dp	+/- 1.0x10 ¹⁰	5 decimals	all valid digits
Number 6dp	+/- 1.0x10 ⁹	6 decimals	all valid digits
Number 8dp	+/- 1.0x10 ⁷	8 decimals	all valid digits
Number 10dp	+/- 100,000	10 decimals	all valid digits
Number 12dp	+/- 1000	12 decimals	all valid digits
Number 14dp	+/- 10	14 decimals	all valid digits

Larger numbers

Numbers with greater accuracy or greater range must be put in text fields. As we are talking about *extremely* large numbers, however, for most people this limitation is academic.

Conversion of text into numbers

If you use Text fields inside calculations in which the answer is expected to be a number, Omnis automatically converts the contents of the Text field into a number. The conversion process follows these rules:

- Depending on whether your computer has been set to use a period or a comma as the separator between whole numbers and decimals (i.e. in decimal numbers), the content is interpreted accordingly. The text “12,231” is interpreted (by Omnis) as the number 12231.00 if the decimal separator is a period.
- If the Text field contains a minus sign (regardless of where it is in the field), the number is interpreted as a negative number.
- The plus sign is ignored.
- If the Text field contains characters other than those mentioned above, then the value of the field is interpreted as 0.

#NULL

If the ‘Can be Null’ option has been selected, the Number field may also have the value “undefined” (termed #NULL in Omnis). This means that the field has not been touched, either by the end-user or in calculations. It is sufficient to place the cursor in the editing field; the value of the number field will be replaced by “0” or by the whatever number is written in. There is no value ‘Empty’ for Number fields; it is synonymous with the number 0. (“Empty” means that the field is saying that *there should be no value here.*) In open windows, the #NULL value and the number 0 will both be represented by the character “0.” This likeness, however, is only skin-deep. In procedures we can search for records in which the field value has not been written in.

Example: Temperatures

As an example, take the recording of temperatures. Here the number 0 has meaning, because it is an acknowledged temperature. Records with a field value “temperature of 0” and records with untouched fields will both appear the same in the windows. If the end-user, having placed the cursor in the editing field, subsequently discovers that the temperature was not recorded for that day, we should then set the field’s value to #NULL. This would mean that the record has not been considered at all, as far as this field is concerned. We can set the value to #NULL using a ‘Clear range of fields’ command. If we do *not* do this, the temperature will be recorded as “0.” On a hot summer day it’s highly unlikely that this represents the reality of the situation! You can read more about #NULL in the paragraph on Boolean fields.

(If we had used a Text field, we could have set the field to value ‘Empty’ by using ‘Calculate FIELD as "".’ The value ‘Empty’ would mean that the record for this date *has been considered*, and it was found that the temperature measurement had in fact not been recorded at the weather station, – in other words, the real-life information does not exist.)

#NULL in procedures

For some reason or other, we cannot locate those records in which the number field has the value #NULL by setting the search criteria 'FIELD=#NULL.' Instead we must exploit the fact that #NULL is sorted as an absolute zero, i.e. before the lowest value possible for the relevant number field. We can look up the range a particular field type has, or use a negative value chosen at random. This value must be totally different from whatever might be entered. If we want to look up unregistered temperatures, we may set the search criteria as follows: 'TEMPERATURE < -100.' This will give us all unregistered temperatures (#NULL), but it will also yield erroneous entries (unless, of course, we have been gathering data from the far side of the moon!).

Insert as 0

If we have a field that is allowed to have the value #NULL (i.e. "undefined"), it will get this value at the outset. It will keep this value through 'Update files,' provided the editing field in the window has not been touched. This applies, however, only if the 'Insert as 0' option has not been selected in the field definition. If it has, Omnis will insert the number "0" when a 'Prepare for insert' command is executed. Thus the field receives the number "0," even though the end-user has not touched the window field. If the user wants the field to be undefined, this must be done in some procedure, using the 'Clear range of fields' command.

Boolean Fields

Boolean fields may be compared with electrical switches in the sense that both have two possible values: Yes or No (or 1 or 0, on or off, man or woman, etc., as the circumstances dictate). Boolean fields occupy 8 bits in the memory. They are often used in Check boxes.

Areas of use

In every situation where there are only two possible choices, Boolean fields are always a good one. For example: Man/Woman, Yes/No, Vacant/Occupied, etc. Typical areas of use are “course” type divisions, but then you must be sure that the division is not broadened to include more than two choices. Nevertheless, if such a thing were to happen, it wouldn’t be hard to get out of this bind by composing a little conversion routine. Running such a routine on larger amounts of data is time-consuming, however.

#NULL and Empty

In practice, these fields have several possible values: Yes, No, Empty and #NULL. An empty Boolean field signifies that the value is unknown; thus it should be neither Yes nor No. On the other hand, if the value is #NULL, this means that the field has never been touched by the end-user or used in any calculation since the record was established (or since the variable was declared). Thus #NULL means that the field contains no information whatsoever, and so it is often called “undefined.” To get #NULL values in Boolean fields, you must have selected ‘Can be Null’ in the File format. (The ‘Insert as empty’ option should also be turned off.)

Example

Let’s look at an example involving a poll. Imagine that your assignment is to ask 100 specific people in a company whether they had ever had a backache. Furthermore, let’s say that of these 100, you were

actually able to poll 80. Five answered Yes, 65 answered No, and the remainder (5) couldn't remember. As you know that 20 were not asked, they get the value #NULL. Five people had had a backache and 65 hadn't. Naturally, these two groups get the values Yes and No, respectively. The last 5 people couldn't remember whether they had had a backache or not. The value for these is Empty. We would be justified in assuming that these people probably had a careless attitude toward their own health – which, however, is not the same as not having been asked! (#NULL)

#NULL is usually not discernible

Having explained the difference between #NULL and 'Empty,' we should also mention that Omnis makes no distinction between the two in its calculations. A Boolean field that is calculated as #NULL apparently behaves as if it were Empty in most contexts. There is one difference, however; #NULL is lower than Empty in the sorting sequence. We can exploit this fact in Search formats (see below).

The easiest way to differentiate in a clear manner is to create our own codes. In the foregoing example it would have been simpler to use a Short integer field in which, for example, 0 meant "Not registered," 1 meant "Don't know," 2 meant "No," and 3 meant "Yes." With the aid of Radio Buttons, the system would be easy to use.

Boolean fields in windows

Within windows you usually encounter Boolean fields in Check boxes or in Radio Buttons. You click on Check boxes so that the "X" appears inside the box. Then the value is Yes. If the value is to be No, you must first click and bring up the "X," and click it again to make it disappear. If the field is left untouched, the value will be #NULL (provided the 'Insert as 0' option is not checked off.) For Radio Buttons you need two (in Window field sequence), representing the values 0 and 1. The values will be No and Yes, respectively. The black bullet pops up in the Radio button

representing “No,” whether the value is No, Empty, or #NULL. To get the value to be Empty, it must be deleted by a procedure containing ‘Calculate FIELD as “”.’ To get the value to be “No,” you must click on the Radio button for “Yes,” and then on “No.” Alternatively, you can use the command ‘Calculate FIELD as 0.’

Search formats and calculations

To pick the values Yes, No, Don’t know (Empty), and Not asked (#NULL), we have to know the means we have at our disposal. The order of values is as follows (from lowest to highest):

NULL, Empty, No, Yes

Suggestions for search criteria and their interpretation are summed up in this table:

Test:	Finds:	
If BOOLEAN<“”	NULL	Not registered
If BOOLEAN=“”	EMPTY	Answer non-existent
If BOOLEAN=0	NO	No
If BOOLEAN=1	YES	Yes
If BOOLEAN<0	EMPTY and NULL	Ambiguous
If len(BOOLEAN)=0	EMPTY and NULL	Ambiguous
If BOOLEAN>=0	YES and NO	Clear answer
If len(BOOLEAN)>0	YES and NO	Clear answer
If BOOLEAN<>1	NO, EMPTY and NULL	Not positively
If BOOLEAN<>0	YES, EMPTY and NULL	Yes or “Not denied”
If BOOLEAN<>“”	YES, NO and NULL	Not ‘Empty’
If BOOLEAN>=“”	YES, NO and EMPTY	Only registered records

Language differences

Within calculations the content of the calculation field is interpreted in terms of the nationality of the particular version of Omnis you have. Different languages have different words for “Yes” and “No,” which is only natural, and Omnis must make allowances for this. To avoid any discrepancies between different versions, the numbers 1 and 0 should be used. These are interpreted correctly by all the versions and can be used both in procedures and reports.

Using Boolean fields with “record templates”

In many situations you can save both time and work if you use a pre-constructed template when entering a record. This template contains text that fits a typical record in the file in question; this content may be inserted when a new record is created. The user makes needed changes, and then saves it as an ordinary record. Such templates are particularly convenient when there are large amounts of monotonous text in the data.

The “Template flag”

We can insert a system of templates by creating an indexed Boolean field in the file format. Example: “L_Letter_is_Template.” (The “L” is an acronym for the filename fLetters.) By using this field, normal letters are distinguished from “letters templates.” To locate the letters that will serve as a general point of departure for the actual letters, you merely conduct a search with ‘L_Letter_is_Template=1.’ Since the templates and the records are in the same file, any changes in the File format will affect both. The templates and the actual letters will not be confused because we will have used L_Letter_is_Template as a flag in order to distinguish between them. There is no extra burden on the internal memory, and the programming goes surprisingly smoothly. (All you need to do is turn the flag L_Letter_is_Template on and off at the right places.)

The template title

If the File format does not have a text field that is suitable as a template name (for example, the letter’s

title), such a field should be established. (Example: 'L_TemplateName'). Without a descriptive title or a separate name, the user won't know what the template actually contains.

Using the templates

By setting 'L_Letter_is_Template=1' as a search criterion and building a list that is sorted according to L_TemplateName, we will be able to view all the templates in the file. Template names may be shown to the user in the form of a list. The template is selected, is located in the datafile, and the content of the "template record" is used by a subsequent 'Prepare for insert with CV' command (CV = Current values).

Date Fields

Date fields are number fields in which the content is converted to day of week and date before being presented to the user. Date = 0 signifies AD (actually, December 31, the year 0). Date = 1 signifies January 1, the year 1; subsequently the days are counted until the year 9999.

Main date field types

Date fields can be divided into two main types: Short date and Long date. They differ a bit, so we'll take a brief look at each of them.

Short date

Short date fields always have “days” as their minimum unit, and they require 4 bytes.

Long date

Long date fields contain a point in time with a precision of up to 1/100th of a second, in addition to the date. The field requires 8 bytes. The minimum unit depends on the field definition.

Units in Date fields

The subtypes of the Long date fields determine the minimum unit in terms of which the date field will be treated. For example, if this unit is 1/100th of a second, allowances must be made for this fact in every calculation. Look at the results in the following table:

Format	Calculation	Result
Date D m Y	DATETIMEFIELD+3	3 days later
Date D m Y H:N	DATETIMEFIELD+3	3 minutes later
Date D m Y H:N:S	DATETIMEFIELD+3	3 seconds

later
Date D m Y H:N:S.s DATETIMEFIELD+3 3/100 sec.
later

It is clear here that calculations with the date field are wholly determined by the way in which they are formatted. When only calculating with Short date fields, Omnis always uses days as the minimum unit, so you shouldn't run into this problem here.

Interpretation of text

When translating a text segment into a date (or date with time), the following rules must be observed:

- In calculations, text that is enclosed in quotation marks (" ") will be treated just like text enclosed in parentheses in the 'dat()' function. Sheer numerals (i.e. numbers without quotation marks) are grabbed without further translation and *must* therefore numerically represent the correct date, counted from year 1. It is rarely ever advisable to assign integers to date fields, because this will mask a field's true value and make the procedures or reports harder to read and debug.
- Any character that can be written (not including control characters) may be used to separate the day, the month, and the year. (You don't usually need to separate the day, the date, and the year.)
- The first two connected numbers (characters before the first number are ignored) are interpreted as the day. The next two connected numbers are interpreted as the month (In American versions, the reverse is the case.)
- If the year is omitted, Omnis will choose the current year. If both the month and the year are omitted, Omnis will choose the current month. Both "01" and "0107" become June 1 1994, if this is today's date.
- An attempt will be made to reinterpret values that exceed the natural limits for day and date. Example: "440312" will be interpreted as December 3, 1944.

The year range in Short dates

Subtypes of the Short date fields tell us how the date is to be understood, *if* the text by which the date is interpreted does not contain information about any century. Example: “12/3-67” will be translated as March 12, 1967 if the date field is of the type ‘Short date (1900-1999).’ If the field had been of the type ‘Short date (1980-2079),’ the date would then be translated as March 12, 2067. If we type in “12/3-1967,” all doubt will be eliminated.

Calculations with dates

In general, Date fields can be used the same way that Number fields are used in calculations. In practice, however, you will only need subtraction (and possibly addition). The inverse integral of dates doesn’t contribute very much that is useful. We’ve already seen that the formatting of the specific Date fields determines the unit in terms of which it will be reckoned. Calculations operate in terms of the minimum unit of all the fields that appear in the calculation. The field with the minimum unit determines, in other words, how any isolated numerals should be interpreted. When the calculation is completed, the answer is inserted in the receptor field (date or string) in ‘Calculate DATE as,’ with the time, day, and year in the right place. If the receptor field is a number field, the date will be converted to a number that is expressed in terms of the unit used in the calculation (for example, seconds or days).

#1 = Days between dates

Calculate ShortDate1 as "Jan 01 1990"
Calculate ShortDate2 as "Jan 10 1990"
Calculate #1 as ShortDate2-ShortDate1
, **** Answer: #1 is 9 ****

1

No. of days between two dates – Short date format (1)

If we want to find out how many days there are between two dates, this is no problem at all, provided both dates have days as their minimum unit. We can use a simple number field to receive the date, as shown in Procedure 1. Here, the difference in days is placed in #1.

Using a Short date field as “converter”

2

```
Calculate LongDate1 as "07.Feb 90 20:15:15"  
Calculate LongDate2 as "07.Apr 90 21:15:15"  
Calculate ShortDate as LongDate2-LongDate1  
Calculate #1 as ShortDate  
; **** Answer: #1 is 59 ***
```

No. of days between two dates – Long date format

(2)

However, it sometimes happens that we use Long date fields that include time, and the minimum unit might be seconds. Using the method above, we get the answer expressed in terms of seconds. Not many of us, however, would be interested in having 9 days expressed in terms of seconds; we would have trouble relating to the result! We must convert the date values from the minimum unit to the desired unit. We can do this ourselves (by dividing 86400 ($60 * 60 * 24$) if it is seconds). But it's easier to let Omnis do it for us. We need a Date field with days as the minimum unit in order to complete the translation of the two date formats. See Procedure 2.

Days and hours

The field LongDate2 is 2 months and 1 hour after LongDate1. The number of days is calculated correctly on the basis of the specific months we're talking about. The exact time is disregarded; only whole days are counted. If, however, we wish to show how many hours there are beyond the number of days, we may use the 'jst():' function. This is all shown in Procedure 3 below:

The # of days and hours between two Long dates

3

```
Local variable DifferenceDate (Date D m Y H:N:S)  
Local variable NoOfDays (Long integer)  
Local variable NoOfHours (Long integer)  
  
Calculate LongDate1 as "02.Feb 90 20:15:15"  
Calculate LongDate2 as "02.Apr 90 21:18:15"  
Calculate DifferenceDate as LongDate2-LongDate1  
Calculate ShortDate as DifferenceDate
```

```
Calculate NoOfDays as ShortDate
Calculate NoOfHours as jst(DifferenceDate,"T:H")
, *** Answer: NoOfDays is 59, and NoOfHours is 1 **
```

Taming the Date field beast

By using the function 'jst(..."T:H"),' we get the time portion of the date DifferenceDate. This will be satisfactory for all situations, because the time value we get in Procedure 3 is always less than 24 hours. It would not have worked to use 'jst(..."D:D")' to get the number of days from DifferenceDate, because the number of days will exceed the total number of days in a month. The "date" DifferenceDate has the date value "February 28, 0001," and so the value of 'jst(DifferenceDate, "D:D")' will be 28! If you count, you will find that this date is 59 days away from December 31, 0000, which is day 0 for Omnis. Therefore we use the date field 'ShortDate,' and get the time period expressed in the proper unit.

Calculating # of hours between two long dates

Local variable Hours (Date DMY H)

```
Calculate LongDate2 as "05 Mar 90 20:15:15"
Calculate LongDate1 as "06 Mar 90 21:18:19"
Calculate Hours as LongDate1-LongDate2
Calculate #1 as Hours
, *** Answer: #1 is 25 ***
```

4

Number of hours

(4)

We can use the same technique with hours. But then we need a Long date field (not Short time; it has an upper limit of 24 hours), in which the minimum unit in the date format is hours. Before running Procedure 4, a new date format, 'DMY H,' must be added, with which we will format the local variable 'Hours.'

Presentation in windows and reports

The subtypes of Long date fields tell us how the various date fields should be displayed in windows and reports. The subtypes determine, for example, whether the month shall be written out in full, whether the number for the century shall be included, etc. This can also be controlled with the aid of 'jst(L_DATE;"D:...")' in each instance.

Archeological dates

Dates before Christ (B.C.) must be stored in Text fields or Number fields and be interpreted by the developer. Another option is to use regular Date fields, but then the developer must make allowances for the reverse calculation (the higher the number, the older the date); and somewhere in the application it must be made unequivocally clear that the dates preceed year 0.

Time Fields

Time fields (Short Time) resemble Date fields; they are 2-byte (number) fields representing the number of minutes that have elapsed since midnight. Translation takes place in the same way as for Date fields, and follows these rules:

- In calculations, the text that is enclosed in quotation marks (" ") is treated just like text enclosed in parentheses in the 'tim()' function. Sheer numerals (i.e. numbers without quotation marks) are grabbed without further translation and must therefore numerically represent the correct time, expressed in terms of the number of minutes past midnight. It is rarely advisable to assign integers to time fields, because this masks the time field's actual value and makes the procedures or reports harder to read and debug.
- The first two connected numbers are interpreted as time. Remaining numbers are interpreted as minutes.
- Every character that can be written (not including control characters) may be used to separate hours from minutes (or 1/100ths of a second, for that matter). In fact, it is not really necessary to separate hours from minutes, minutes from seconds, and so on.
- "PM" or "AM" placed anywhere in the text segment prescribes a 12-hour interpretation. Default is a 24-hour interpretation.

If you wish to register seconds or hundredths of a second, you must use the Long date field. See the paragraph on Date fields.

Picture Fields

Picture fields are designed especially for pictures (images); naturally they may not in themselves be used for calculations or searches. The purpose of these fields, when it comes right down to it, is solely to display drawings, pictures and figures for the user, as well as to receive (via 'Paste' or 'Paste from file') and store them. When the these fields are used in windows or reports, you may choose between displaying a reduced version of the entire picture, or just a section thereof, in the event that the picture's borders exceed the size of the field. Generally speaking, information residing in the picture field can't be presented in any other type of window field. In addition, all image modification must be done using external routines.

Memory requirements

Pictures, by their very nature, hog memory and they put a distressing load on our poor old computer. The most demanding tasks for any machine involve various forms of image handling. The crux of the problem is the enormous amount of graphics information, especially when it's in "bitmap" format. (Bitmap images consist of a mosaic of miniscule squares in a variety of colors.)

Resolution

The memory requirements depend on the picture's overall area and the number of colors that are employed. As you may know, every image, when scanned, is translated from its natural, analog form (with "infinitely" large resolution) to an artificial, digital form in which a set resolution is employed. The process involves placing a hypothetical grid (or "netting") over the screen; each square in the netting is assigned the color that fits best. The higher the resolution, the finer the mesh, and the more the digital image will resemble the analog one. But you pay a high price in the form of dramatically increased demands in terms of hard disk space.

Calculating space requirements

Each “square” in our hypothetical netting is called a “pixel.” Every pixel corresponds to a single point on the screen. Monochrome images contain only all-black and all-white points, each point taking up 1 bit in the memory (which is what we would expect). Such an image makes demands on memory according to the following formula:

$$\text{Bytes} = \frac{[\text{Horizontal pixels}] \times [\text{Vertical pixels}]}{8}$$

The figure 8 in the denominator of the fraction is there because there are 8 bits in a byte. We see that the demands on memory increase proportionately with the size of the image (both horizontal and vertical).

Colors

On the other hand, if we want color, each color must be numbered. Each point is assigned a number that states which color it has. This number becomes a kind of color code. If each point receives a 2-bit number, the points may choose from a list of 4 colors (for example: White=00, Black=01, Blue=10, and Red=11). This means that a 2-bit picture may display a maximum of 4 colors. The number of bits in the code used by the points to designate their color is called “pixel depth.” For color images we derive the following formula for memory requirements:

$$\text{Bytes} = \frac{[\text{Horizontal pixels}] \times [\text{Vertical pixels}] \times [\text{Pixel depth}]}{8}$$

An image with a measely 4 colors requires twice as much memory as its monochrome counterpart. For reasonably realistic color images, 256 colors are not all that much. Remember that every nuance of color and gray-tone is defined as a separate color; and in shadows, soft edges, and overlapping transitions, there are a great many such nuances! 256 colors requires 8 bits per pixel, which increases the demands on memory by a factor of 8. To the human eye, a pixel depth of 8

is ridiculously small. It's only when the concentration increases to 24 bits that picture quality even *begins* to approach photographic quality; moreover, the demands on memory are correspondingly astronomical. A 24-bit picture with 512 x 512 pixels (about 18 x 18 cm at 72 dots per inch) requires *6 megabytes*. True photographic quality requires a pixel depth of 32 bits.

What is a suitable medium?

These are all factors you should bear in mind when planning how to include pictures in a database. Though Omnis manages images quite efficiently, high-capacity media are nearly always needed. For larger libraries of pictures, you must resort to hard disks in the gigabyte class, or even to CD-video (Video disk).

Lists

Generally speaking, a list can be compared with an empty sack. We know what it's name is and what it's for, but not what it contains. The list is only ready to be used when the exact fields it is to contain are defined in a procedure. Each line in the list contains a value corresponding to the fields that once gave the columns their titles (in 'Define list'). Thus the list becomes a kind of "mini"-database, in which the lines correspond to records.

Large lists

The upper limit for how many lines a list can contain is circumscribed by the amount of memory available at any given time. (In Omnis 5, the limit was 30,000 lines.) For lists to be shown to the user, this is not the practical limit. Large lists take too long to build, are cumbersome (too much scrolling), and tend to saddle the user with the work of searching for a record in a file. This negates the list's usefulness. It was never the intention that lists should be used as mirror images of big files. If you get lists with lines that can be numbered in three digits, it's time to pause and reflect on the usefulness of your list and try to recall what you originally had in mind. As a rule, limiting the number of lines in the list from which the user shall choose should pose no problem. (Use a search!) The application will "look its best" when as much of the file as possible is kept on the hard disk.

Lists in file formats

Lists defined in file formats are stored together with the other data in the file format (that is, provided the File format has been set to 'Read/Write'). The list definition, #L, #LM, #LN, selections, and the 'Save selections' buffer are all stored, along with the values in the list lines themselves.

A fourth dimension

From a programmer's point of view, this opens up a whole new world of advanced data processing, because

we get a 4th dimension in which to index data; and if we reckon with differing datafiles as a dimension, we're up to 5 dimensions (and now you've lost me!).

Other aims

Lists on which we do not intend to perform rapid searches, and which do not change very much, may be stored in file formats. This is an alternative to file connections in which the linked records are merely added chronologically and where there is no need to analyze the “connections” within a given time frame. This is particularly true of large datafiles with a huge number of connections, in which case it will be a lot quicker to retrieve lists directly from disk than to build them separately. You can read more about this in the chapter entitled “Lists and Tables.”



Lists are a developer's best friend.
And the best part is: unlike Fido, you don't even
have to walk them!



Binary Fields

Binary fields contain untreated, “raw” data stored with no fuss. This field type is designed to be used by external routines, and its content cannot be displayed by Omnis. Omnis assists the external routines by administrating large amounts of data – which, after all, is its forté. The point of all this is to pave the way for other types of data, e.g. sound and video for presentation and processing, and to secure an uncomplicated integration of external routines with the Omnis database engine.

Values

Binary field values themselves are not displayed. However, Omnis does inform you whether the fields are empty or contain values. This is ascertained by using the OPT/RB menu (click on the name of the Binary field), as well as the Field Value window.

Binary fields in notation

Certain attributes in the notational system are in binary format. This applies, for example, to \$formatdata and \$datadict. For processing these it's useful to have suitable fields for this kind of data.

Sequence Fields

Sequence fields contain the Record Sequence Number (RSN). This number is assigned to each and every record that is inserted, and is unique for each one. In any case, as the RSN exists in the memory, it costs nothing to define a Sequence field. You may alter the contents of such fields, but the changes are never stored in the datafile. Where it concerns RSNs, Omnis is in the driver's seat.

How Omnis assigns RSNs

The first record to be created in the file is assigned RSN = 1. The next record is assigned RSN = 2, etc. If a record in the file is deleted, the number for that record is not reused, and the numbering of the other records is not changed. Thus the highest RSN does not necessarily indicate how many records the file contains. On the other hand, it does tell us how many times a new record has been *added* to the file. (You get the number of records in the file using the 'sys(83)' function.)

Areas of use

Fields defined as sequence fields provide the programmer with a convenient opportunity to do something useful with the RSN. Here are some of the possibilities:

- Making a sure identification of the various records. There is no danger of confusion in Single user systems, and minimal risk in multi-user systems. We get a rapid and unambiguous search for very specific records.
- Creating serial or invoice numbers in applications in which it is impossible to delete records. If the user is allowed to delete records in the file, the developer may still accept RSN as the serial number, provided the numbers don't have to follow each other in unbroken succession.

- Showing the chronological order in which records were entered and the number of new records that have been inserted all together.
- Finding the record most recently inserted.

TIP: Always define a sequence field, even if you suspect you'll never use it. Sooner or later you will need the RSN, for example, in order to recover a specific record.

Resetting the RSN

When a file format slot in a datafile is deleted, the RSN for this file format is reset to 0. (Use the dialog box 'Examine datafile' in the 'Utilities' menu.) If there are copies of the deleted datafile, and these copies are reused, we run the risk of getting different records that share the same RSN.

RSN and import

During an import, the RSN that accompanies the main file will be disregarded, and new RSNs will be assigned (as with a regular 'Insert'). This, in turn, affects the import and export of hierarchically connected files. See the chapter entitled "Import and Export."



Care in your choice of field types
will assure you of an efficient file structure.





File Connections

In General	2
Types of Connections	6
Hierarchical connections	
Relational joins	
Different Ways of Linking Files	7
One-to-One	
Many-to-one	
One-to-many	
Many-to-many	
Commands for Editing Data in a Datafile	12
Prepare for insert	
Prepare for edit	
Delete	
Update files	
Update files – Do not cancel pfu	
Modifying Contents in a File of a Higher Level	
while Modifying Another File	15
Theoretical solution	
Practical solution	
Expanding the system	

In General

File connections are used a great deal to simplify data storage when working with databases. Much has been explained about this technique, and many find it difficult to understand. Basically, we are dealing with an attempt to mirror the way information is organized in the real world. We are used to information being linked in some way and we are used to having to know the context in order to understand it. If a friend asks you over for dinner at six o'clock, you already know where it is, who else might be coming, and what to expect. The information "dinner at six" is connected to other data which you already possess about your acquaintances. If you hadn't recognized your friend, this information would be unintelligible.

Avoiding repetition

The beauty of file connections is the fact that certain types of data inherently belong together. When making the connections, we sort the data by groups and thus avoid saving all kinds of information on top of itself and continuously repeating ourselves. If we were to register all the inhabitants in a certain geographical area, they would all share the same information concerning weather conditions, public facilities, etc. It makes no sense to key in this information together with each name. It's more efficient to separate data that is often repeated, store it in a file, and then link each inhabitant's individual data with the shared data.

Child files and parent files

When we discuss file connections, we are talking about a possible link between two independent file formats. In principle, one file will contain data that is unique to the file, while the other one will contain more general information that can apply to a number of files. We often call the first file the "child file," which is where most of the data is entered. The other file is called the "parent file," where we place the shared data; the various records in the child file are then linked to the appropriate records in the parent file. Each connection

consists of one record in the child file, which is linked to one record in the parent file. A file will often be linked to several other files. This enables a record in the child file to be linked to one record from each of the parent files.

The nature of linking

Such links are uni-directional. A record knows which record it is linked to, whereas the latter is not aware that other records have “attached themselves.” The child file records have appropriated a name or a number, which they use to point out who their parents are in the superior files. Using the parent–child analogy, we can say that the child record knows its father’s or mother’s name. Its father is in the parent file “Fathers,” and its mother is in the parent file “Mothers.” When the child looks up its mother’s name in the “Mother’s” parent file, it will soon find out who she is, where she lives, etc. Likewise, the child will be able to do the same with its father’s name. However, neither the child’s mother nor father know each other, nor do they know their own child. Of course, we’re being somewhat cynical in using a model in which it is the child who holds the family together.

The connection itself

In reality, a connection (or link) is based on an item of information that identifies a record in another file that is stored in a separate field in the first file. The data must serve as an identifier. For example, if a person is given an identification number, this number must be unique to that person. This number can be used when linking, because it can be used to locate a particular person. The child–parent example used the parents’ name for identification purposes. We can call the name fields in each parent file `F_FATHERS_NAME` and `M_MOTHERS_NAME`. They function as identifying fields, also known as “key fields.” Each name was copied and saved in its own field in the child record. Let us call them `C_MY_MOTHER` and `C_MY_FATHER`. These fields are often called foreign

keys. The entire link is based on the content of these fields.

Visualizing the file connection

Figure 1 shows us how the linking procedure can be visualized. The sequence number *C_RSN* belonging to parent file *fCompany* is saved as a label on all the connected records in the child file *fCustomers*.

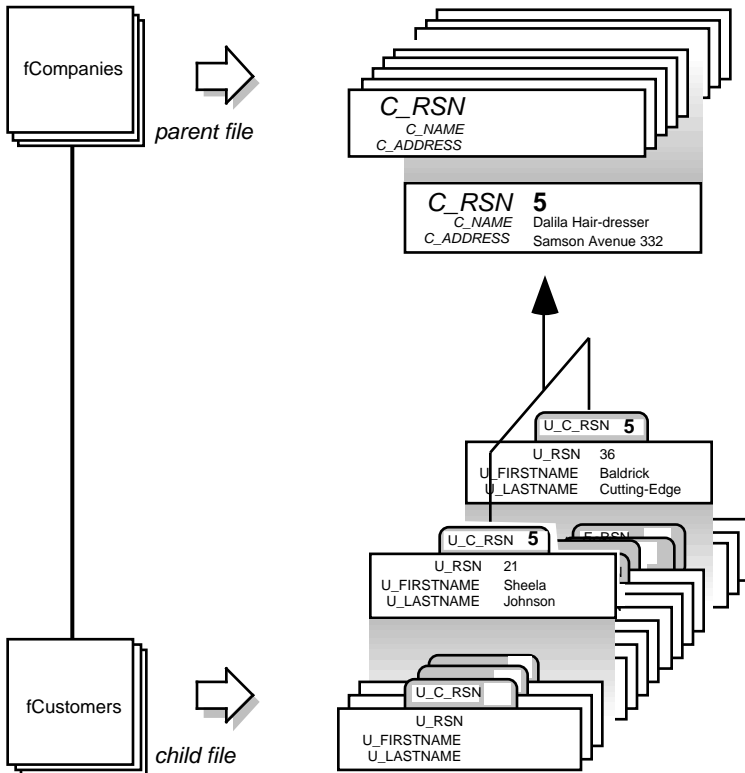


Fig. 1 Visualizing a file connection

Figure 1 shows that the records in fCustomers have a small, extra label where the U_C_RSN field has been placed. This is the foreign key from the fCompanies file. Wherever we wanted to create a link, we have copied the value of the fCompanies' identifying field, C_RSN, to the foreign key U_C_RSN. The record found in fCompanies has a C_RSN value of 5. We also find this number in the foreign keys of the two records linked in fCustomers. The value of U_C_RSN is also 5, thus establishing the link.

Types of Connections

Hierarchical connections

Omnis uses hierarchical connections to eliminate most of the programming work involved in file connections without adversely affecting flexibility. When creating hierarchical connections, the Record Sequence Number serves as the identifying data for the foreign keys. Omnis generates this number itself, so the user (or developer) will be less likely to mess things up. The developer doesn't actually see the foreign keys; Omnis takes care of organizing foreign keys and copying values. When you need to refer to the foreign key, you may use the name of the field containing the sequence number. As long as the main file is set to the child file, Omnis will understand the difference.

Connected fields in windows

Moreover, the linked record in the parent file will be retrieved automatically when 'Find' is used to locate a record in the child file. Fields from the parent file can be placed in the same window as the child file, with the linked data being shown together with the data in the child file.

Relational joins

The developer uses relational joins in Omnis to control the connection itself. Connected files are not retrieved automatically, making it necessary to type in a bit more code. On the other hand, the link is more stable, in the sense that it is more amenable to imports and exports, since it is not dependent on RSN (see the chapter entitled "Import and Export"). On the whole, relational joins give us a greater degree of flexibility, as they leave more in the hands of the developer. However, all this takes place at a relatively advanced level. In actual use, relational joins are just as quick as hierarchical connections.

Different Ways of Linking Files

The fact that there are several ways of linking files allows you to choose the option that makes them optimally suited to the data they are to contain. What we'll now be looking at is not specific types of connections as such, but rather the result of how the developer controls the way the user makes these links.

One-to-One

A one-to-one connection means that only one record may be connected to a record in another file. Your procedures must ensure that there are no cases of “double booking.” This is achieved by only allowing procedures to create records in the child file and ensure the link, with no interference from the user. A typical example is that of an invoice (child file) linked to a task (parent file). Instead of using one-to-one connections, we could have placed all the fields in the same file format. This would result in our calling up much unnecessary information, except in certain situations, e.g. printing the invoice. Furthermore, making it easy to change which bits of information are connected (as we can when we use file connections) helps the developer provide the user with a means of undoing erroneous connections.

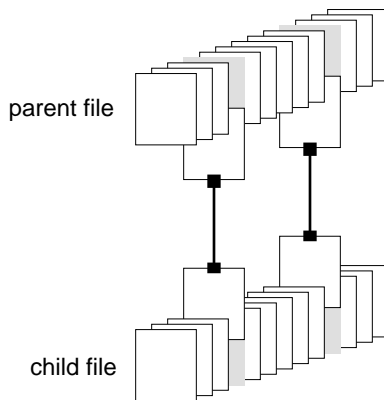


Fig. 2 One-to-One file connection

Many-to-one

This is the most common way of linking records. Just be sure that the parent record (to which one of the child records is going to attach itself) is in the CRB when the file is updated. With Omnis' hierarchical connections, key values are copied automatically, whereas with relational joins the developer must do this himself. ('Calculate ForeignKey as KeyField'). Example: A club with scads of members.

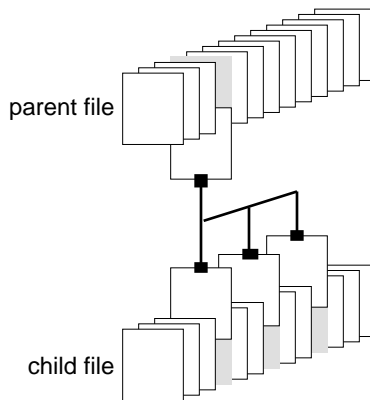


Fig. 3 Many--to-one file connection

One-to-many

A one-to-many file connection is only possible when using relational joins. The field we use in the child file to carry out the link contains a code that coincides with several records in the parent file. In this case, the parent “key” field would not be unique, but would fit several parent records. It is the procedures which must generate a list of the linked records, which is then shown to the user. This is a unusual type of connection, which can usually be replaced by many-to-one connections.

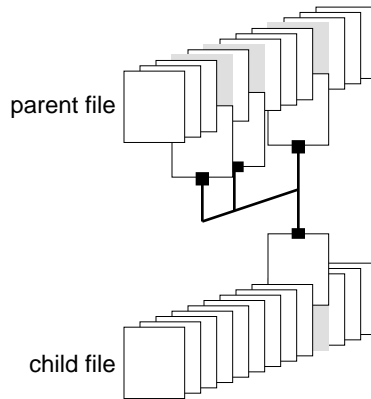


Fig. 4 One-to-many file connection

Many-to-many

This type of connection requires an indirect approach. To keep our bearings in a linking structure of this complexity, we need to make contact with every single connection. We need to create an extra file that links the two files and acts as a go-between. Each record in the connector file is linked in a many-to-one connection to each parent file. Actually, there is nothing special going on here. Each record in the connector file is linked to a record in parent file 1 and to a record in parent file 2.

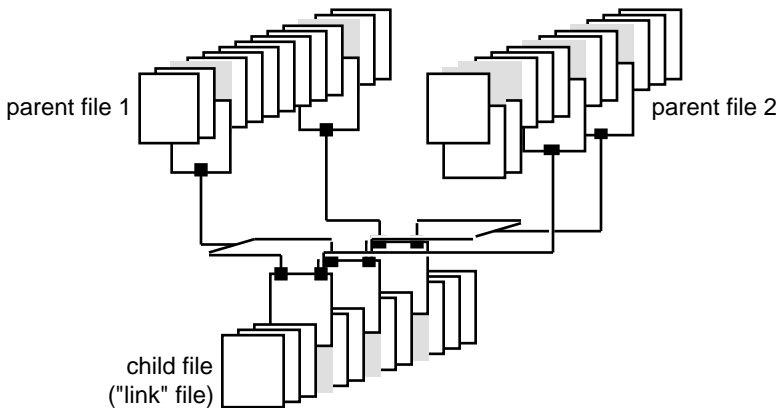


Fig. 5 Many-to-many file connection

Searching

We see the results of a many-to-many connection when we count the links, keeping one of the parent files constant. We can find out which records in parent file 2 are linked to a record in parent file 1 by searching for all the records in the connector file that have a foreign key that corresponds to the record in parent file 1. In other words, we look for all the records in the connector file carrying the label that fits the record in parent file 1. When we see which other labels appear in the fields that show the links, we'll know

which records in parent file 2 are indirectly linked to parent file 1.

An example

Imagine a video rental shop. Its archives contain many customers' names and many video tapes. Every time a customer rents a tape, the transaction is registered. This is a classical example of a many-to-many connection. The customers are stored in one file and the video tapes in another. A third file contains information on each rental, e.g. date and price. This rental file links the customer file to the video file in a many-to-many relationship.

Any given customer may have rented many videos, and any given videotape may have been rented by many different customers. The rental file serves here as a connecting file. The data being stored is basically the combination of a customer ID number and a video tape number, together with the date. Every time the video is rented, the name of the customer and of the tape should be read into memory and a new record created in the rental file.

Most searches are made in the rental file, whether it's printing out a bill, creating a monthly budget, finding out which customers rent what, or determining which films are popular. For a further explanation of how to find records using the many-to-many file connection, see the chapter entitled "Search and Find."

Commands for Editing Data in a Datafile

When working on an application, the developer has to consider so many aspects of the user interface – lists, tables, and appearance – that he may forget the main point of the application which, basically, is to administer and save data to disk. The commands involved in this process are not especially complicated, but it is an advantage to be well-acquainted with them.

Prepare for insert

This command prepares for the insertion of a new record in the main file. This means that the fields in the main file will be cleared. Only one file at a time can have a new record added to it. Before you can carry out a new ‘Prepare for insert,’ you must finish the old one by using ‘Update files’ if you wish to keep the contents of the fields. Any new ‘Prepare for edit’ or ‘Prepare for insert’ will abort any previous ‘Prepare for insert’ command. That’s why it is wise to take one file at a time and to finish using ‘Update files’ before beginning any new tasks. The ‘Test for a current record’ command will not be true until the file has been updated.

There is nothing wrong with carrying out ‘Update files’ while still in Enter data mode. The user will hardly ever notice anything and won’t be able to interrupt the procedure anyway. This method can be used, for example, when toggling between windows.

Prepare for edit

In principle, this command prepares the whole CRB for editing. In Single user mode, the contents of CRB fields remain unchanged, whereas in multi-user mode, the main file record is retrieved again from disk. This results in the latest version of the record appearing in memory. This may not match the contents of the fields just before the ‘Prepare for edit’ command was executed. For this reason, in multi-user mode you must assume that the contents of the main file will “disappear” each time you carry out this command.

The entire CRB

If the windows and their control procedures allow it, the user can move from window to window and edit all the data in the CRB files. When 'Update files' is carried out, all the modifications, minus the files set to Read Only or Memory Only, are saved – with one exception: Modifications are not stored for files where 'Test for a current record' results in false. The entire record is deleted from memory when the commands 'Clear main file,' 'Clear main and connected,' and 'Delete' (record), are given, or when failed searches take place. This also leads to 'Test for a current record' command resulting in the value 'false.'

Fields that are cleared

It's all right to delete single fields, e.g. with 'Clear range of fields,' or when the user, while in a window, uses an entry field to delete it. The record is still registered as being in memory, and the modifications are saved using 'Update files.' In addition, you can retrieve a record (with 'Find,' 'Next,' etc.) and edit it between the 'Prepare for edit' and 'Update files' commands. It is only when the update takes place that the CRB is analyzed to determine which records are in memory; and these records are saved to disk.

Test for a current record

Even if a record has been deleted from memory, you can still enter values into the fields. The values won't be saved to disk during 'Update files,' though. The reason there is confusion about 'Test for a current record,' and whether a record has been saved or not, has to do with the way Omnis tests for this. Keeping a constant record of whether every single field has been changed or not requires too much work. Therefore, only successful searches and an updated 'Prepare for insert' or 'Prepare for import' will be able to establish whether the record actually exists in memory. The record retains this status, even though most of its fields have been modified or deleted.

The record won't lose its status until a 'Clear main file,' 'Clear main and connected,' or a failed search takes place. This leaves us in the reverse situation. 'Test for current record' will still result in 'false,' even though the data in all the fields is kosher. In its defense, we can say that the test is useful despite its limitations. In other words, in practice it isn't necessary to completely follow the logic in the command's title, because its main function is to check whether the user or the procedure has really entered or found a record that can be linked using a file connection.

Delete

This command simply deletes the main file record stored on disk. This function is irreversible. 'Test for a current record' will result in false. Therefore you should always make sure that the user is certain about carrying out this command, e.g. with a 'Yes/No message.'

Update files

Any files set to Read/Write are updated, i.e. saved to disk. This ends the 'Prepare for insert' or 'Prepare for edit' mode, unless something else has been decided. The file that was prepared for a new record (i.e. the main file) receives this new record, while the other files are updated in the same way that they would be when a record is edited. This happens even though the other files haven't been specifically prepared for editing. You can protect files against this type of modification by keeping them in Read Only mode, or by preventing the user from changing windows during Enter data.

Update files – Do not cancel pfu

When you choose the 'Do not cancel pfu' option, Omnis repeats the previous 'Prepare for insert'/'Prepare for edit' command. What this means for 'Prepare for insert' is that a new record is added to the *same* main file that had been set during the previous 'Prepare for

insert.’ For ‘Prepare for edit,’ this means re-entering ‘Prepare for edit’ mode, without specifying Main file.

Modifying Contents in a File of a Higher Level while Modifying Another File

When you input data for a record, this involves either creating a new record or modifying an existing one. If the file you're working with is linked to another one, the (parent file) record must already exist. But this is often not the case; the record isn't always there. Normally you would have to cancel (or save, with the wrong connection or no connection at all), enter the window containing the parent file, and add the necessary information as a new record here. Later you can go back and correct the record in the child file. This involves a lot of awkward maneuvering between different windows and records, as well as having to enter some data more than once. This is a typical file connection problem, because you are dealing with two files at the same time. Thus it is important to make it easy to move from editing records in one file to editing records in another.

Theoretical solution

A good solution would be to leave the child file window directly without saving to disk, insert the missing record in the parent file, and return to the child file window in order to continue editing, without having altered the contents in the child file. These fields should contain the data entered before the user left the window. Most users would like such an option for each level upwards in the file hierarchy, applicable to every file in the system.

Problems

The trouble with this procedure is that it leaves some field values in danger of disappearing as the user moves from the child file to the parent file and back again. The commands we'll be using will change the CRB. 'Prepare for edit' retrieves the saved version of the record when Omnis is in multi-user mode. The 'Prepare for insert' command always deletes the file's CRB. Furthermore, you must take into account the fact that, if given the chance, the user might try to carry out a search on one of the files. In that event, it wouldn't be long before all the data the user entered into the

child file's record had disappeared. There is a way around this, though.

Practical solution

We use a list to save all of the fields in the main file as field names with their respective values. Underneath each Insert button we place an ordinary “Prepare for Insert” procedure. Just after the ‘Prepare for insert’ command, we call up the list’s old values, exactly as the user keyed them in. With the help of a Window Control Procedure in every window, we can discover when the user changes windows during Enter data mode. This is the signal for the Main file fields to be added to the list. When the user returns to the window later and presses the ‘Insert’ pushbutton, the old values are read from the list and reinserted in the fields.

Setting up the procedures

The following steps show how to set up a working system that should be able to cope with the problem mentioned above. First, we need to prepare the window(s) for the procedures to come.

- 1) Sown in procedure 1 are the global variables that are used in these procedures:

500 Library variables

Library variable LiLs_AllFiles (List)
Library variable Li_MainFileName (Character)
Library variable LiLs_OneFile (List)
Library variable Li_FieldName (Character)
Library variable Li_Value (Character)

1

- 2) When the application starts, the list LiLs_AllFiles should be defined. This contains the name of the main file, as well as a “sublist” of the field names and field values.

499 A•Init procedure

2

Set current list LiLs_AllFiles

Define list (Store long data) {Li_MainFileName,LiLs_OneFile}

- 3) Every window that is part of the system must be set to 'Allow clicks behind,' and preferably not to modeless Enter data. Each Window Control Procedure should look something like this:

wChild/240 {Window Control Procedure}

3

If #WCLICK&#EDATA

 Call procedure STARTUP/494 {C•Save values into list}

 Clear procedure stack ;; Cancel old Enter data

Else If #TOTOP

 Set main file {fChild} ;; The main file for this window

End If

(To avoid having several Enter data modes within each other, we make sure that the old one has been aborted. Thus we also avoid carrying out the extra update that follows 'Enter data' in the 'Insert' procedure.)

- 4) The insert pushbutton should be set to 'User defined,' and its procedure should be as shown in Procedure 4:

wChild/8 {Insert}

4

If #CLICK

 Prepare for insert

 Call procedure STARTUP/497 {B•Insert Values into CRB?}

 Redraw windows

 Enter data

 If flag true

 Update files

```

Else
    Cancel prepare for update
End If
Redraw windows
End If

```

How the procedures work

Now, let's take a look at what procedures need to be run and how they react to the user's input.

- 5) When the user clicks on one of the other windows, Window Control Procedure receives a #WCLICK, and the following procedure is run:

494 C• Save values into list

```

Set current list LiLs_OneFile
Define list {Li_FieldName,Li_Value}

Calculate Li_MainFileName as sys(82)
Build field names list {[Li_MainFileName]}
Redefine list {Li_FieldName,Li_Value}
For each line in list from 1 to #LN step 1
    Load from list
    Calculate Li_Value as fld(Li_FieldName) ;; Get the field value
    Replace line in list
End For

Set current list LiLs_AllFiles
Add line to list

```

5

The Main file ('sys(82)') fields are placed in a short list: LiLs_OneFile. In order to have values from several files on hand, we place this list inside another list: LiLs_AllFiles, together with its file name: Li_MainFileName. After this, every line in the comprehensive LiLs_AllFiles will consist of a file name and a subsidiary list of field names and values.

- 6) When the user returns and presses 'Insert,' Procedure 6 is run just after the 'Prepare for insert' command in Procedure 4.

497 B•Insert Values into CRB?

```
Set current list LiLs_AllFiles
Set search as calculation {Li_MainFileName=sys(82)}
Search list (From start)
If flag true
    Call procedure STARTUP/495 {B1.Insert values and remove from
list}
End If
```

6

First we search in the long LiLs_AllFiles list to see whether the values from the main file have already been stored here. If a line contains the name of the main file, it means that the user has left this particular window during Enter data, and the data the user previously entered is stored here. Then they should be transferred from Lils_OneFile to the fields in the main file, using Procedure 7:

495 B1.Insert values and remove from list

```
Set current list LiLs_AllFiles
Load from list

Set current list LiLs_OneFile
For each line in list from 1 to #LN step 1
    Load from list
    Calculate Li_FieldName as Li_Value (Use fld() of name)
End For

Set current list LiLs_AllFiles
Delete line in list
```

7

Here we call up values from our little LiLs_OneFile list. We are given the name of the field and the corresponding value. Once the values have been transferred to the CRB's fields, we will have no more use for the list values, which may then be deleted.

Expanding the system

This system can also be used with the 'Edit' pushbutton. All you need to do is insert the appropriate call into the procedure, as shown in Procedure 8. You can use as many windows as you like, as long as all the above procedures have been included. In addition, you must repeat the changes described in points 3 and 4 for every window.

Modified Edit pushbutton

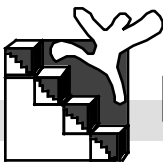
```
If #CLICK
  Prepare for edit
  Call procedure STARTUP/497 {B•Insert Values into
CRB?}
  Redraw
  Enter data
  If flag true
    Update files
  End if
End if
```

8

Section 4: Elements of an Application

Chapters:

1. Datafiles & Libraries
2. Sequence of Procedures
3. Lists & Tables



Datafiles & Libraries

Application Structure.....	2
A word about Omnis' structure	
Libraries ("Format libraries")	
Formats	
STARTUP menu	
Opening and Closing Libraries.....	5
From the 'File' menu	
From the Format browser	
From procedures	
Internal names	
Extension libraries	
Controlling Datafiles	8
Two main routes	
Floating file mode	
Set default datafile	
Summary	
CRB and Data Files.....	13
Adding data to different datafiles	
Reading and modifying values in different datafiles	
Opening and Closing Datafiles.....	16
From the Format Browser	
From procedures	
Example of a Datafile Handling Procedure	17
Main procedure	
Subprocedure: Get Path	

Application Structure

A word about Omnis' structure

Omnis has always contained an impressive amount of repeated code. Options are presented with the aid of the same lists, buttons, etc. that we use in our applications. There are many unique advantages to this, one of which is the fact that application files take up little space in memory and on the hard disk. Considering its vast potential, Omnis occupies surprisingly little space. Another result of this introspectively logical connection is that the application file is treated in about the same manner as the datafile. In this sense the application file *is* a datafile, with indexes and “file formats” that contain the formats we use. This helps increase Omnis' efficiency, no matter what the size of the project involved.

Omnis v1.x

In Omnis 7 v1.x we program applications that consist of a number of formats we create ourselves. If we want to use the same formats in several applications, we have to copy them over first. The same limitations apply to the datafiles. We can only have one datafile open at a time and have to make do as best we can with external lookup datafiles, which are read-only. The application is constructed as shown in Figure 1. Each application consists of all the formats we have created:

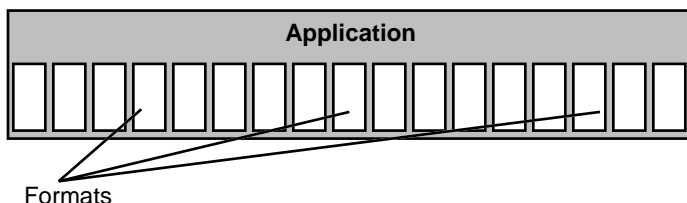


Fig. 1 Crude application structure in v1.x

Omnis 7 v2.x

However, this was “back in the old days.” In v2.0, both the application and the datafile(s) were given a new structure. The application is now broken down in the following manner:

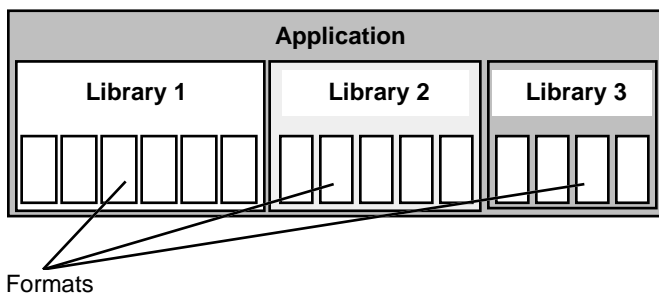


Fig. 2 Crude application structure in v2.x

Libraries (“Format libraries”)

The application isn’t a fixed entity anymore. It now consists of whatever combination of open library files that exists at any given time. A format library corresponds to what was previously known as the application; the difference is that we can now have several format libraries open at the same time. They can be run on their own as an application or be combined as modules.

Combination of formats

As before, the library file itself is a collection of formulas for file formats, menus, windows, reports, and search formats. Libraries that are opened have their menus installed beside the others. The user can employ certain windows from one library and others from another, and won’t notice any difference. Loading libraries means adding new menus and windows – and thus whole sets of new commands and possibilities for the user. This means that the software

we create can be constructed with building blocks. We can offer smaller solutions for smaller needs, and easily expand if necessary.

Formats

Most developers are already familiar with formats. There are file formats, window formats, and menu formats with their accompanying procedures, as well as reports and search formats. All this is loaded into memory when needed. Each format library consists of the sum of these resources, in addition to a number of internal preference tables.

STARTUP menu

When a format library is opened, the STARTUP menu (any menu format so dubbed by the developer) runs procedure 0. All initiating procedures take place or are evoked here. It is also here that we find the key to controlling every action the user is allowed to carry out. For example, this is where passwords are keyed in and where certain datafiles are linked to specific file formats. Taken as a whole, the STARTUP/0 procedure functions as a transitional zone, which enables the developer to ensure that the merging of new libraries goes smoothly. We'll take a closer look at this later.

Opening and Closing Libraries

There are several ways of calling up libraries, and every method has its virtues.

From the ‘File’ menu

When you select ‘Open library’ from the ‘File’ menu, you call up the “main” library. By this I mean the library that is in charge from the outset and that usually controls the loading of other “sub”-libraries. Other libraries are closed and removed from memory before the new one is installed and booted.

From the Format browser

Two lines appear in the lowermost part of the list in the Format browser tools window: ‘Libraries’ and ‘Datafiles.’ By clicking on ‘Libraries,’ we see which library is in memory at the moment. We retrieve new libraries from the hard disk by clicking on the ‘Open’ pushbutton.

Design library

The format library we are working with is marked with an asterisk (*), and is called “design library” (\$dlib). In the Format browser we only see formats within this library. From this perspective, libraries are usually isolated from each other. To change design libraries and work with formats in other modules, you double-click on the desired library name (in the Format browser).

When you load a format library from here (using the ‘Open library’ pushbutton), the STARTUP menu is not installed, and the 0-procedure is not run. The developer has to install any menus himself. The advantage of this is that it provides full control over the events taking place when libraries are loaded.

\$Ignore external

The notational attribute \$Ignoreexternal is set for every library. It is normally turned off; but when switched on, it allows any format library to read and modify the variables and formats of other libraries. You then have to use the complete format address to reach the right formats. This means using the library's name whenever you refer to it. Usually only the neighboring library's variables and fields are relevant, so we needn't do anything with this attribute.

From procedures

It's a good idea to create your own procedures in order to ensure complete control of the way the different modules are loaded. Here we must use our wits to locate the various libraries, even when they are in redundant directories or in folders on the disk. We can also enter our own password routines and send parameters to the STARTUP/0 procedure in the library being opened.

The 'Open library' or 'Prompt for library' commands are every bit as versatile in loading libraries as the above. We can choose the 'Do not close other libraries' and the 'Do not call startup procedure' options. (Their meaning should be self-explanatory.) If the path isn't specified, Omnis will search in the EXTENSION/Omnis Extension folder. The path must include the library's file name, as well as any internal name and password. Giving the password here makes it unnecessary for the user to key it in himself when a new module is loaded. Finally, you send a list of any parameters in parentheses. This works just like you're used to from your previous experience with parameters (see the chapter entitled "Field Types & Their Function.")

Internal names

To distinguish different libraries and datafiles from each other, they are given names of their own. Each name is usually the file name. Windows users in particular should remember that there is a difference between the file name and the internal name. Unless you specify a separate internal name, Omnis will remove the path and extension from (its copy of) the file name before using it internally.

The ‘path’ is a long sentence telling you which folder the current file is located in and which disk it is on. The extension is a combination of three letters (separated from the rest of the information by a period), which defines an MS-DOS file type. The “.LBR” extension is removed. The conversion works as follows:

C:\OMNIS\APPS\INVOICE\MONTHSUM.LBR
Internal name: MONTHSUM

Internal names for Macintosh users

For Macintosh users, the internal name will usually turn out to be the same as the file name. However, if there are three letters separated from the rest of the file name by a period, they will not be included in the internal name. In this respect, there is no difference from Windows.

Extension libraries

You can use Omnis to improve the Omnis environment itself. Any format library placed in the extension folder (Omnis Extensions for Macintosh, EXTENSION for Windows), is loaded when you start up Omnis. There are a few differences, however. The familiar STARTUP/0 procedure will be run, but the STARTUP menu will *not* be installed. Any menus installed by the STARTUP/0 procedure will not appear until a library that *hasn’t* been placed in the extension folder is opened. Omnis is virtually in a state of suspended animation (closed state) until “real” libraries are opened. When you open an ordinary library, Omnis will go into its usual (open) state.

In the extension folder you can place a library that should always be present. You might as well place any control systems for loading and unloading modules here, which will give you a large measure of control over other libraries.

Controlling Datafiles

Omnis 7 v2.x can keep several datafiles open at the same time. This affords a previously unheard of degree of flexibility, and is one of the main reasons that modular applications are possible. On the other hand, this is a decidedly complicating factor, and any such complication of datafile manipulation will be off-putting to many developers. If the developer is not absolutely sure of himself here, he risks riddling the application with errors so serious as to render it unusable. Full control is absolutely essential. How, then, shall we manipulate these datafiles, and when do we know what data goes where?

Two main routes

There are two principal routes for this control to follow: A common one that can be varied, and one or more individual ones that are locked.

The common route

The common route starts from all file formats that are in Floating datafile mode. All file formats are in this mode at startup time. The data fed into them is sent down to the datafile indicated in Current datafile. Figure 3 illustrates this point with a faucet that can be turned. Right now it is pointing to datafile DATA 1. Text, pictures, dates, numbers, etc. from file formats FILE A, FILE B and FILE C all run down into datafile DATA 1. The current datafile is the one the faucet is pointing to. If we want to “turn” the faucet, we use the ‘Set current datafile’ command and ask for one of the other datafiles.

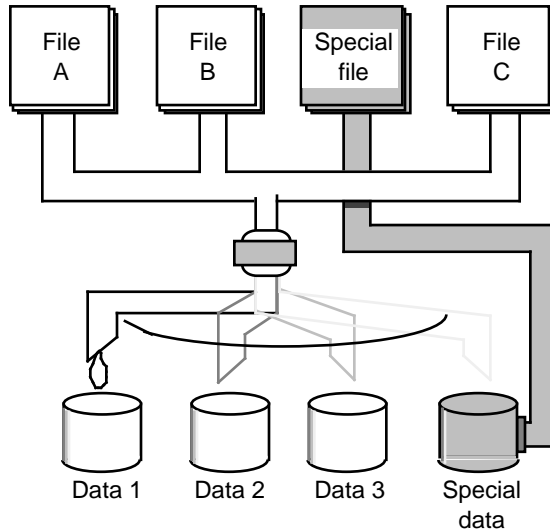


Fig. 3 Floating and default datafile mode

Locked, individual routes

The other type of route is individual, and establishes a fixed link between specific file formats and a specific datafile. It doesn't care what the current datafile is, nor does it act on it in any way. This means that all of the data entered in the SPECIAL FILE file format in the figure will automatically end up in the SPECIAL DATA datafile.

Floating file mode

When a file format has been set to Floating file mode, this means that the file format follows the common route. This one empties its contents into the current datafile. It might be the last datafile opened or the last one mentioned with the 'Set current datafile' command. If all file formats are in Floating file mode, you get something like Figure 4. Here, the data ends up wherever the faucet is pointing.

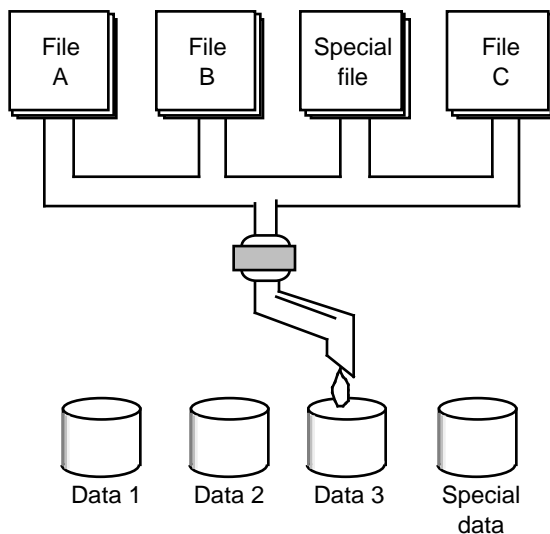


Fig. 4 All file formats in floating datafile mode

Set default datafile

The individual method is thus the handiest. This is because each file format has its own task, so it is only natural to divide datafiles up in a similar way at a higher level. This can be quite useful when making backup copies. It will be possible to separate fixed data into its own datafiles, minimizing the number of files that must be backed up. Furthermore, particularly sensitive data can be separated from the rest and placed in well-protected partitions on the hard disk.

The procedure in practice

Using the 'Set current datafile' command, we specify the datafile we wish to connect to. This file must be open, or we will get a 'General purpose error,' which causes Omnis to abort. After choosing the correct datafile, we set a list of file formats to be funneled into this file using the 'Set default datafile' command. For the FILE A file format, the procedure will be as shown

in Procedure 1. Once the link has been created, we can return 'Current datafile' to its previous setting.

Associate datafile and file format

```
;; Alternatively, a datafile name-checking routine first  
Set current datafile {Data 1}  
Set default datafile {File A}
```

1

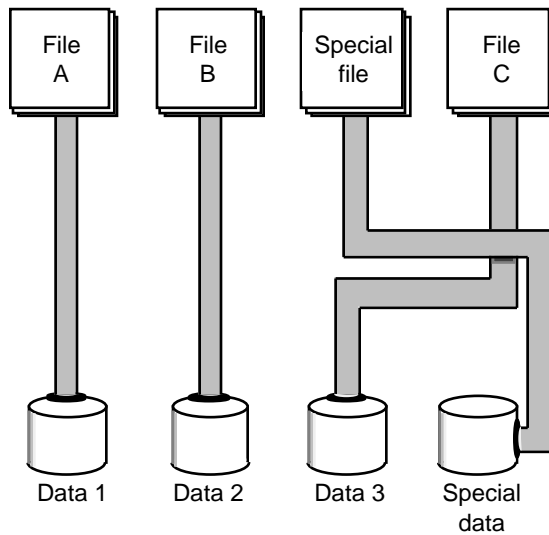


Fig. 5 All file formats set to default datafiles

Linking all file formats to their own datafiles

If we link every file format to specific datafiles, the result will be as illustrated in Figure 5. Here, 'Current datafile' will have no influence on where information is stored.

Resetting from 'Default file' to 'Floating file' mode

If you want to reset the file format from a locked, individual route to the common route, use the ‘Set floating file mode {file format}’ command.

Summary

To summarize Figure 6, we may regard the faucet as representing ‘Current datafile.’ The gray fixed connections show how the ‘Set default datafile’ command works.

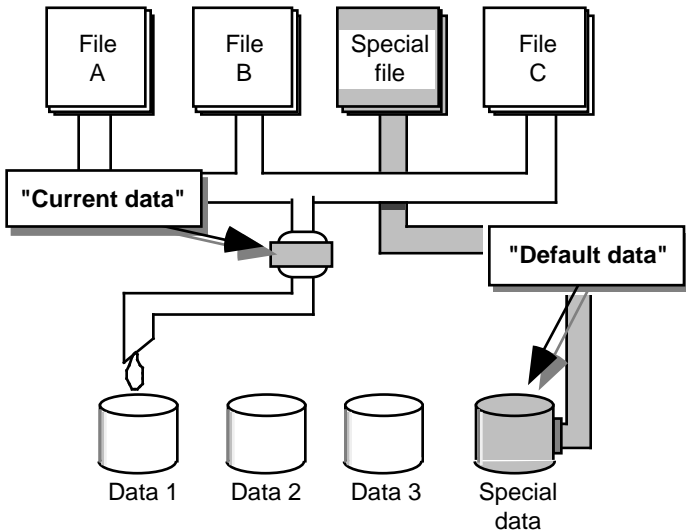


Fig. 6 Summarizing datafile handling

CRB and Data Files

Adding data to different datafiles

Current Record Buffer is a fundamental concept in Omnis. It can be defined as every field in every file format. Another way of putting it is to say that it is the combination of one record from each file in memory at any given time. Together, all these field values are written to disk when an 'Update files' command is given. Naturally, files set to Memory Only, Read Only or Closed are exempt from this rule. As we locate records, the CRB becomes an ever-changing combination of field values. So far this has been a relatively simple term. (See the chapter entitled "Data Structure: Memory & Hard Disk.") But now it's time we made things a bit more complicated!

A complete CRB for each datafile

In principle, when you have several datafiles open, each one has its own complete CRB. Imagine sheets of paper placed side by side in piles. The "page layout" is similar for all the sheets, but the content varies. Each sheet of paper represents a record and each pile a datafile. The top sheet would then represent the CRB. Keep in mind, though, that this example uses the same file format in each datafile. If we were to look at each pile, we would see that fields in the file format contain different values in each pile.

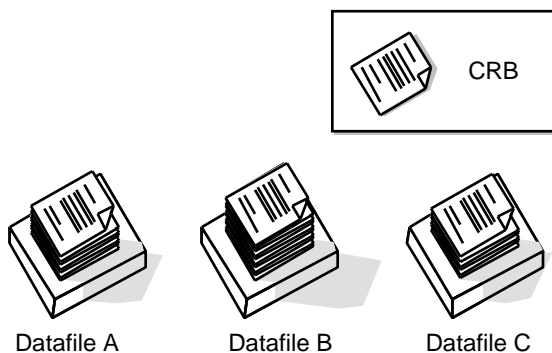


Fig. 7 One CRB in each datafile

We usually only notice this fact when the same file formats store data in different datafiles. When we change ‘Current datafile,’ the field values in these formats also change.

RSN

When we add data to different datafiles, we see that the Record Sequence Number (RSN) is numbered separately in each datafile. Thus RSN will start with number 1 every time we start adding data to a file format in a new datafile, even though we have previously used this format in other datafiles. From this perspective, datafiles are completely unaware of each other’s existence.

Reading and modifying values in different datafiles

If we use several datafiles simultaneously, we ought to know how to read and modify values in CRBs that belong to datafiles other than the current one. The principle here is not new. We add extra information to the field names to reach the values we want. Once the correct addresses have been set, we can deal with the fields the way we are accustomed to.

File formats within the same format library

The general way of addressing field values in CRBs belonging to other datafiles is as follows:

`datafile.fileformat.fieldname`

If our datafiles are called MYFRIENDS and YOURFRIENDS, the file format is fPersons, and the field is F_Name, the procedure will probably look something like this:

Calculate #S4 as MyFriends.fFriends.F_Name

Calculate #S5 as YourFriends.fFriends.F_Name

(Prl. 1)

After this, #S4 will contain “Roger,” which is the name of one of my friends. #S5 will contain “Harry,” who isn’t really one of my friends.

File formats in other libraries

If we want to call up values belonging to file formats in libraries other than \$clib, we have to turn on \$ignoreExternal:

Calculate #F as \$clib.\$ignoreExternal.\$assign(1) (Prl. 2)

The address notation is as follows:

library.fileformat.fieldname

If we want to call up the name of a pet (P_Name) in the fPets file format in the ANIMALMODULE library, the procedure will look like this (Prl. 3):

Calculate #S4 as AnimalModule.fPets.P_Name (Prl. 3)

If no specific datafile has been specified, the current datafile will be used.

Several datafiles

If the library we are looking for uses more than one datafile, the correct address will be:

datafile.library.fileformat.fieldname

Let's look at the first example we used in this sub-heading; but imagine that we are now programming a procedure in ANIMALMODULE (or any other) library. The library we want to get to is MAINLIB, and we want to know the name of the persons in the CRB that are related to the datafiles MYFRIENDS and YOURFRIENDS (i.e. the CRBs that go with their datafiles).

Calculate #S4 as MyFriends.MainLib.fFriends.F_Name
Calculate #S5 as YourFriends.MainLib.fFriends.F_Name

#S4 will be "Roger," one of my friends; and #S5 will be "Harry," one of your friends, even though the Go point is in the 'AnimalModule' library.

Opening and Closing Datafiles

From the Format Browser

At the bottom of the main list in the Format browser tool window we find ‘Datafiles.’ By clicking here, we see all of the open datafiles. Current datafile is marked with an asterisk (*). The pushbuttons allow us to open and close datafiles at will, and double-clicking sets a datafile to Current.

From procedures

We find ‘Open datafile,’ ‘Prompt for datafile’ and ‘Close datafile’ among the procedure commands. There isn’t much more to say about them, except for the fact that ‘Open datafile’ also gives you the option of mingling the datafile to be loaded with the those that already exist in memory, or of replacing them altogether. Moreover, any datafile opened will automatically become the current datafile. Keep in mind, however, that this part of the ‘Open datafile’ command is irreversible, so it’s up to you to return Current datafile to its original setting unless you want to change it.

Example of a Datafile Handling Procedure

Some people might find the expanded structure of many datafiles intimidating, especially the idea of the “floating” datafile. (What? Do you mean to tell me that my data just “float”?!). Unless the procedures handle things properly, data could well end up hiding in the most unlikely datafiles. To put you at ease, here is a practical example of how to solve the problem. We’ll show you how to carry out the most conventional procedure – namely, how to link specific file formats to specific datafiles.

Main procedure

There are a few tasks in this procedure that need to be dealt with in a responsible manner. Let’s take a look at the elements we ought to include:

Remembering the name of the previous current datafile

We must remember the setting of the current datafile, because the ‘Open datafile’ command will change it. To do this, we use the `Lo_Last_Datafile` variable. When we have finished connecting the Default datafile, we set the Current datafile to the name saved in `Lo_Last_Datafile`. If `Lo_Last_Datafile` is empty, this means that no datafile was open to begin with and we won’t have any file to return the current datafile to. (So don’t even try!)

Where is the datafile?

The next problem is guessing where the datafile is. It’s worth checking the same folder as `$clib`. This means that we’ll need a path to `$clib`, but won’t need `$clib`’s name. No problem! Procedure 3 will take care of this nicely.

Find it yourself!

If the datafile can't be found, you'll have to ask the user to find it himself (the 'Prompt for data file' command). Before you do this, though, you should alert the user, or you might end up with your support phones ringing at all hours of the day (and night!), or some similar nightmare scenario.

Link the file format and datafile

The magic 'Set default datafile' command takes care of this. It's up to the developer to decide on a name for the file format and enter it.

Set Current data to Last_Data

Finally – if necessary – we clean up and reset the current data to its previous setting.

A•Open datafile

```
Local variable Lo_Last_Datafile (Character)
Local variable Lo_Path (Character)

Calculate Lo_Last_Datafile as $cdata().$name
Call procedure pTest/3 {A1.Get $clib path} with return value Lo_Path

Open datafile (Do not close other data) {[Lo_Path]MYDATA.DF1}
If flag false
    OK message {Please locate the datafile MYDATA.DF1}
    Prompt for datafile (Do not close other data)
End If

Set default datafile {fPersons}

If len(Lo_Last_Datafile)>0
    Set current datafile {[Lo_Last_Datafile]}
End if
```

2

Subprocedure: Get Path

This subprocedure (Procedure 3) returns the path name, complete with delimiters according to the platform in use. It finds the position of the library name within its path name, and extracts the string up to

this position. This corresponds to '\$clib()\$.pathname,' up to and including the final delimiter.

Platform-independent code

This procedure should work in both Macintosh and Windows environments. This means we must bear in mind that each operating system uses different folder/directory delimiters. The Macintosh uses a colon (:) between its folders; Windows and OS/2 use a backslash (\) between their directories. We circumvent the delimiter problem by finding the position of the library name in the complete \$pathname, sending back, untouched, the remaining information as a whole path.

A1.Get \$clib path

```
Local variable Lo_Path (Character)
Local variable Lo_Path_and_Libname (Character)
Local variable Lo_Libname
Local variable Lo_Pos_of_Libname

Calculate Lo_Path_and_Libname as $clib()$.pathname
Calculate Lo_Libname as $clib()$.name
Calculate Lo_Pos_of_Libname as pos(Lo_Libname,
Lo_Path_and_Libname)
Calculate Lo_Path as
mid(Lo_Path_and_Libname,1,Lo_Pos_of_Libname-1)

Set return value {Lo_Path}
```

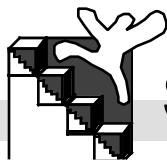
3

(If you don't understand the notations in this chapter, please turn to the chapter entitled "Introduction to Notation.")



Libraries are more than books!





Sequence of Procedure

Introduction.....	2
Field Procedures	4
Window Control Procedures (WCPs).....	5
Library Control Procedure (v2.x and v3.x)	7
The Timer Procedure.....	9
The Jig-Saw Model.....	11
The individual parts	
Changing houses	
Returning to house	
Inactive fields	
Windows in the Jig-saw Model.....	19
Opening a window via menu or procedure	
Outside Enter data	
Table Fields in the System (v2.0)	23
Set Next Action (SNA)	25
Queue Action.....	27
Procedure Stack.....	28
Move up stack’/’Move down stack	
The ‘Clear procedure stack’ command	
Quit to Enter data	
Quit all procedures	
Tables of the Jig-Saw Model.....	36
Tables – Macintosh.....	37
Tables categorized according	
to the jig-saw model (Macintosh)	
Alphabetical overview (Macintosh)	
Tables – Windows.....	41
Tables categorized according	
to the Jig-saw model (Windows)	
Alphabetical overview (Windows)	
Events as Evoking Factors (Macintosh and Windows).....	45
“Real” events – evoking FWL or WL	
Status messages – accompanying various events	
Events that must be activated before use	

Introduction

Ever since computers came on the scene, the way they are programmed has naturally been the most crucial factor in their usefulness. The first machines had to be programmed in a code that was compatible with their internal structure. On the earliest machines this meant connecting wires between vacuum tube circuits. The arrival of microprocessors made it possible to program properly in machine code: long sequences of digits in which each sequence represents a severely restricted command. The programmer himself had to move digits from one address in the memory to another in order to be able to make calculations. He was usually a mathematician (or plumber, if you will) and had direct contact with the machine's central processor, the CPU. This was first-generation programming language.

Later, Assembly and other letter-based languages were developed. In these languages each command (which consisted of three letters) could carry a somewhat heavier load, and the language acted as an intermediary between programmer and microprocessor. It translated what the programmer wanted into machine code. Assembly was a second-generation language. When Pascal, Simula, Basic, Forth and C arrived with their structured handling of variables, increasingly advanced commands, and potential for well-ordered programs, these were called third-generation programming tools.

Omnis 7 is a fourth-generation development tool. Individual commands are “big” – i.e. they each perform many tasks for the user. Each command is little more than the title of a long list of minor commands that the user doesn't need to think about. In addition, the developer is placed in a logical system of empty building blocks that follow a particular sequence, depending on their function and the context. The developer fills these blocks with procedures. Care has been taken to combine the highest degree of flexibility with cohesiveness and reliability, so that the applications never “derail.” Knowing how this system is put together provides valuable insight into how Omnis functions as a specialized programming language for databases.

Omnis was designed on the assumption that the developer wants to communicate with the user by means of a number of standardized elements. We have access to all the windows we want, which we can fill with fields of various kinds, and we can create menus that will display our key application commands. In addition to this, we can let the pushbuttons activate a more narrowly focused subset of commands within each window. For instance, we can group related commands in popup menus, allowing the user to choose alternatives from lists or popup menus. On the whole, these are easy to use. Situations can arise, however, where the developer needs to find a

special solution, where field or menu procedures alone do not suffice. This is where the control procedures come in. We'll be looking at the following elements:

- Field procedures
- Window Control Procedures
- The Library Control Procedure
- The Timer procedure

In what follows we will discuss each of these individually before we start combining them and putting everything into a coherent system.

Field Procedures

Field procedures should be familiar to most developers. Under each field in a window there is a procedure that generally runs when the user employs this field. This means going in or out of Entry fields, clicking on check-boxes or radio buttons, selecting from lists, etc. Because they are run only when they are needed, precise and highly focused procedures can be placed behind the fields.

This is but one of Omnis' strengths. Since we don't have to make sure that the procedures are run at the right time, we can concentrate on the short, simple procedures that are to be added. For instance, we can assist the user maximally in entering the correct values. On entering Entry Fields, we can insert today's date or any other appropriate value. On exiting, the supplied values can be checked to see that they are within the limits of the data. If there are any incorrect values, you can return to the field and give an 'OK message' or the like. Example: scale of marks 1 through 6, date within the current year only, etc.

Window Control Procedures (WCPs)

Each window may have its own control procedure, called Window Control Procedure (abbreviated WCP). This is a procedure that is run in relation to each active or inactive field inside a given window. It is also run before you enter a window by clicking on it or by opening the window from a menu. The same thing happens when a window goes from Design mode to ‘open’ state (CMND/CTRL-W). In addition the WCP runs when the window itself or one of the background objects are clicked on. Thus the WCP is related in some way to “everything” having to do with the window in question. If you change windows, a different WCP is in charge.

Activating and deactivating the WCP

The WCP for the current window is set with the ‘Set window control procedure’ command. It applies to the window where we find the procedure that contains this command. Although the control procedure itself may be located elsewhere, the ‘Set window control procedure’ command must be in the window to which the WCP applies. The WCP is deactivated when the window is closed, or when it is activated by another procedure with the ‘Clear window control procedure’ command.

Running the WCP

Generally speaking, the WCP is triggered by all events (Enter data messages) that occur because the user has done something in the window to which the WCP belongs. #CLICK, #BEFORE, #AFTER, and #WCLICK are typical events.

Areas of use

The WCP is, in fact, a jack-of-all-trades. We hope the following hints will come in handy:

- Setting Main file and Current list in main windows for the various files.
- Preventing other windows from opening inadvertently.
- Preventing a window from closing during Enter data mode.
- Performing special manipulations of fields or responding to user input in a tailor-made way.

- Detecting and reacting to clicks on inactive fields or the window itself, giving rise to different variants of the user interface.

Remember, though: the WCP is evoked both during and not during Enter data mode, e.g. after ‘Prepare for insert.’ So take care not to do anything without first making sure that you are not interfering with the poor user trying to input a few humble bits of data.



Watch out, windows, Big Brother is watching you.



Library Control Procedure (v2.x and v3.x) or Application Control Procedure (v1.x)

In Omnis version 1.x, the global control procedure is called the Application Control Procedure (or ACP). This applies to the application as a whole. In v2.x, each application consists of one or more libraries. Here we have the Library Control Procedure (or LCP), which rules the roost within the bounds of its library. LCP and ACP are generally run where the WCP runs, except that LCP and ACP apply to all the windows in a library.

Activating ACP or LCP

ACP can be activated by windows and menus anywhere in the application, whereas LCP primarily applies to the library that contains the ‘Set library control procedure’ command. In this way the LCP is exactly like the WCP, but on a higher level. The LCP of one library will also apply to windows of other libraries, provided they are opened or installed by the library to which the LCP belongs.

Global LCP

If you check off the ‘All libraries’ option in the ‘Set Library Control Procedure’ command, the LCP will apply to the entire application, not just the particular library within which the LCP was set. We call this a global LCP. Each library can have only one LCP, whether it is global and applies to all libraries or is local and applies only to this library. If more than one library has a global LCP, they will all be run in sequence throughout the application in the order in which they were activated.

The ‘No windows’ option

If the ‘No windows’ option has been checked off, the LCP will remain activated even when all windows are closed. Normally, the LCP and the ACP are not run when all windows are closed, even when selecting from a user-defined menu. On the other hand, if the ‘No windows’ option is checked off when the LCP is activated, the latter will run also when only menus are in view.

Running LCP or ACP

Basically, LCP or ACP is run every time the user does something that triggers an event, e.g. #CLICK, #WCLICK, #ENTER, #TAB, #KEYEVENT, etc. All events that evoke the field procedures also evoke an ACP or LCP if either of these are activated.

Areas of use

LCP and ACP, the developer's "fine tuners" in special situations, are excellent tools for special purposes. By the same token, they can easily be misused as a "last resort" and as an easy way out. Since they reach everywhere except reports, special care must be taken in programming them. The developer must think in terms of the application as a whole and remember everything at the same time – not the easiest task in the world!

Enter data

LCP and ACP, like the Timer procedure and the WCP, can intrude during Enter data. In any case it is important that the procedure not interfere with what the developer wants to do in the procedure that is waiting. When the user presses OK, a number of things may have changed, and the updating could be done with the wrong data. We can check in the control procedure to see whether we are in Enter data by looking at the value of #EDATA. See the chapter entitled "The Ins & Outs of Enter Data."



With its pervasive reach, the LCP can make your day
– or totally wreck it –



The Timer Procedure

The characteristic feature of the Timer procedure is that it can be run at regular intervals set by the user. Omnis has a sort of alarm clock built into it which ensures that the correct procedure is run at the right time. The time interval may be designated in increments of one second or more.

The sequence of procedures

This is straightforward enough, but the nervous developer might perhaps wonder whether the Timer, by some statistical fluke, might worm its way in among the other control procedures. Fortunately, that doesn't happen. The Timer stays right at the back of the line in every situation and waits until the cursor has performed all its duties and has returned to an appropriate house. Even with the shortest time interval, the Timer procedure places itself at the bottom of each evoked procedure stack. So we don't need to explain how the Timer works systematically; the Timer will take care of itself.

Exceptions

In both v1.x. and v2.x, the Timer pauses when no windows are open. It restarts when a window is opened. The Timer also stops when the developer is changing a format.

Uses

Any operations to be carried out regularly may profitably be placed under the Timer procedure. Suitable tasks include the following:

- Automatic updating of the time of day while entering a record.
- Updating of "shared" lists in a multi-user system.
- Testing whether there are recent changes in datafiles in multi-user systems.
- Time limit for interrupting Enter data (and 'Prepare for update' mode) when the user takes a coffee break (i.e. the technique mitigates the locking of records in a multi-user system).
- Checking for incoming electronic mail in a multi-user system.

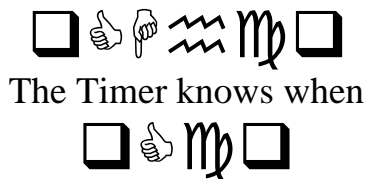
- Checking the appointment calendar for meetings that start in five minutes!

Pitfalls

The Timer procedure can become annoying to the user if it demands constant feedback and is run too often. Try to avoid putting OK messages or Yes/No boxes in this procedure.

It's also important to notice that there is nothing to keep the Timer from operating when the user is entering data and Omnis is in 'Prepare for update' mode. If Read/Write- or Read Only settings are being modified, parts of a record are being deleted, or new records are being accessed, data on the disk can be modified beyond recognition. The links between records may also be modified in this way. To avoid the problem, remember to check for #EDATA in the Timer procedure.

If some strange error appears to have crept in unawares, you should have a look at the Timer procedure. This procedure is very useful; but, like the LCP, it should be programmed with reference to the entire application.



The Jig-Saw Model

Probably the most problematical aspect of Omnis programming is knowing when, how and why application, window and field procedures are run. Thus far we have had recourse only to the simplified and rather obscure “roundabout” model in “Design and Development” (Ch. 8, pg. 9). Those who have ventured to test and see when the control procedures are run have usually ended up with a bunch of their own test messages, and they are none the wiser.

Why read about something so boring?

Some might think that having to know the exact sequence of the different control procedures is going a bit too far. But this knowledge is vital to you as a programmer. Searching for errors in procedure chains is practically impossible unless you’re completely sure when the different procedures are run. But if you are, you’ll also be able to come up with practical solutions to seemingly intractable problems, because the control procedures take over where the field procedures leave off. It is crucial to the effectiveness of the applications that the developer has a good understanding of the sequence.

You can compare this to the importance for professional musicians of practicing scales. Scales and sequences may both seem uninspiring and “un-useful,” but they are (indirectly) vital to music-making and programming respectively. So read on, and be comforted. It is more important to understand the model and the mentality behind it than to remember everything. You can use this understanding in testing the sequence of procedures on your own.

Windows and Macintosh

There is a difference between the way the sequence of procedures is set up and dealt with in the Windows and Macintosh operating systems. The various window fields behave somewhat differently. This is because Windows can be used without a mouse, whereas Macintosh presupposes the use of a mouse. It must therefore be possible to operate the Windows version of Omnis by using the keyboard only. This means that it must be possible to place (or focus) the cursor on all kinds of active fields when the user is selecting them,

i.e. moving the cursor here using the TAB key. In Windows, the user can simulate mouse clicks via the keyboard. For this reason, the two operating systems are kept separate in the model we will be discussing.

The individual parts

It is surprising how extensive the list of waiting procedures can become. The sequence depends on many factors, one of which is the position of the cursor at any given moment. In order to make sense of all this, a set of jigsaw puzzle pieces and a homeloving little fellow have been developed into an explanatory model.



Fig. 1 The cursor, shown as he appears “in real life”

The cursor

Let us imagine the cursor to be an anonymous person, fleet of foot, who runs between windows and fields but is more comfortable indoors, i.e. inside one of the fields that can house the cursor.

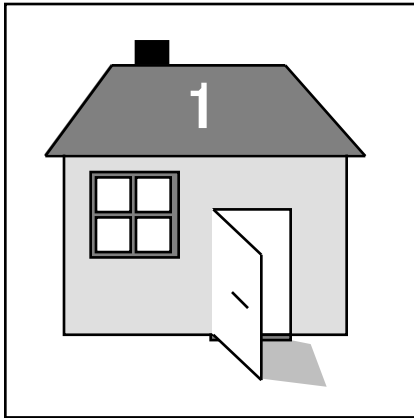


Fig. 2 The ultimate shelter for the cursor

Where is my house?

The cursor can run from house to house within the same window; but if he has to enter a field that won't give him shelter, he returns to his own house on the double. The houses are located inside the windows. When the user is changing windows, the cursor must first exit the house, and then the window, before entering the next window. Here the cursor will find its way to the house the end-user last clicked on, provided the 'Keep bringtofront clicks' option is on. (Window parameters). If this option is not set, or the end-user did not click on a particular house, the cursor will enter the house with the lowest field number.

Enter data

When the user enters the Enter data mode, many fields that previously did not shelter the cursor now begin to do so. Consequently, the cursor may often seem to enter a house spontaneously. This applies to Entry fields, among other fields that accept direct data entry from the user.

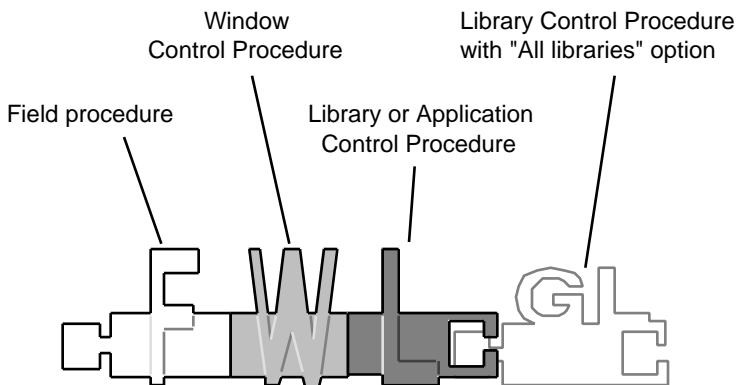


Fig. 3. The triad in the jig-saw model

The triad

The basic unit in the jig-saw puzzle is the piece shown in the figure above. We will call it “the triad.” It consists of the field procedure for the field in question, followed by the WCP of the window in which the field is located, and finally by the LCP (or ACP). The piece marked “L” stands for both LCP and ACP in the figures that follow. After these, any global LCPs from other libraries, as shown by the piece marked “GL,” are appended to the end of the triad. However, for the sake of clarity, we will not include these kinds of control procedures in the model. They generally follow the same pattern as ordinary LCPs. (See the chapter entitled “Datafiles and Libraries.”)

“Field” stands for any active window element in which the user can place procedures. The triad is the canonical sequence that is generally followed when the cursor enters and leaves houses, runs through different window fields, and changes windows. In isolation it corresponds to Pushbuttons (Macintosh), or Check boxes during Enter data mode (Macintosh and Windows).

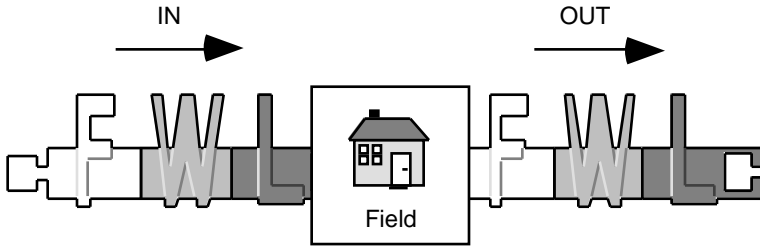


Fig. 4 The sequence of procedures on entering and leaving a field

What is a house?

We may define a house as a field where the cursor can reside. This is indicated by a flashing horizontal bar. If the cursor has to leave to enter a field that is not a house, it will return to its own house. When the cursor leaves the house, the “out” sequence of procedures is run; and when it enters a house, the “in” sequence is run. In both cases this is usually the triad. The field that typically houses the cursor is Entry field under Enter data.

Changing houses

When it’s time for the cursor to change houses, we get the sequence of procedures by piecing together the out-procedures and the in-procedures for the next house. (These are usually one and the same.) If the cursor is in Field number 1 in a window under Enter data, and the user clicks on Field number 2 (which, like Field 1, is an Entry field), the sequence of procedures will be as shown in figure 5.

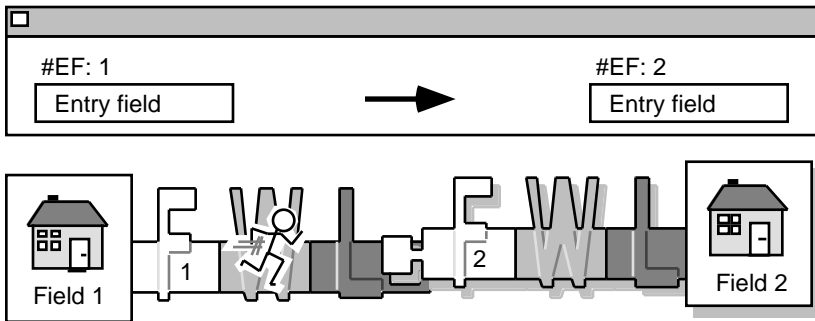


Fig. 5 From one Entry field to another

In the ordinary triad the field procedures are run before the control procedures. Nevertheless, the WCP and LCP steal a march on the field procedures of Field 2, since they are part of the triad leaving Field 1. The full sequence is as follows:

1. Field procedure for Entry field, Field number 1
2. Window Control Procedure (WCP)
3. Library Control Procedure (LCP)
4. Field procedure for Entry field, Field number 2
5. Window Control Procedure
6. Library Control Procedure

Returning to house

When the cursor is in a house and the user clicks on an active field that is *not* a house, we get the entire sequence by simply adding the triad (of Field 2) to the out-procedures for the house. Afterwards the cursor returns to Field 1 and things are back the way they were before. No extra procedures are evoked when the cursor returns to a house. The next time the user clicks on another field, the out-procedures are run again as the house is vacated.

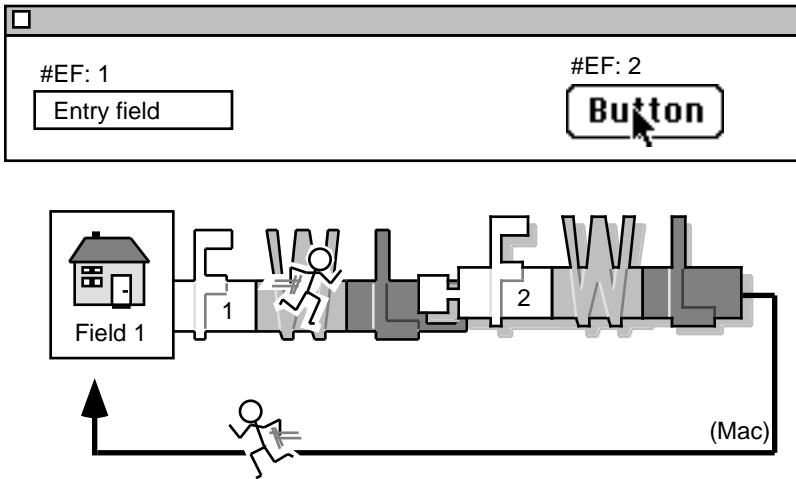


Fig. 6 From Entry field to Pushbutton and back

This example holds true for Macintosh, but it is not altogether representative for Windows, where all fields that run the triad will also house the cursor. The sequence of procedures is the same, but the cursor does not return to the Entry field shown in Figure 6. The full sequence is as follows:

1. Field procedure for Entry field, Field number 1
2. WCP
3. LCP
4. Field procedure for Pushbutton, Field number 2
5. WCP
6. LCP

Inactive fields

Inactive fields are rather strange. They will neither house the cursor nor run field procedures. They are unable to lure the cursor from his house; our friend the cursor just stands there and waits for the inactive field to run its control procedures.

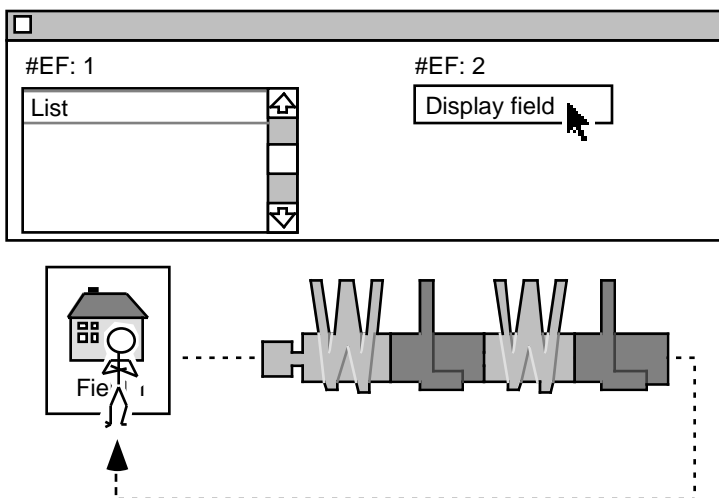


Fig. 7 Clicking on a disabled field

The control procedures are run twice. The first pair is, in effect, an out-procedure without any field procedure, but it receives no Enter data messages. Only in the second pair is it possible to test #CLICK, #ER and #EN and trigger an appropriate reaction. These are:

1. Window Control Procedure. #ER reflects the previous house (Field 1), but no #AFTER is evoked.
2. LCP
3. Window control procedure. Here #CLICK crops up. #ER is the inactive field (Field 2).
4. LCP

(If the house is a Combo box, the first pair is deleted and the sequence is just "WL.")

Windows in the Jig-saw Model

Much of what applies to houses also applies to windows. There are in-procedures, consisting of the WCP for the window in question, and the LCP. The WCP may be compared with the field procedure for the window elements, since it is unique for each window. (The WCP must be activated, however. For that matter, the LCP must also be activated in order to function.) The out-procedures are the WCP followed by the LCP.

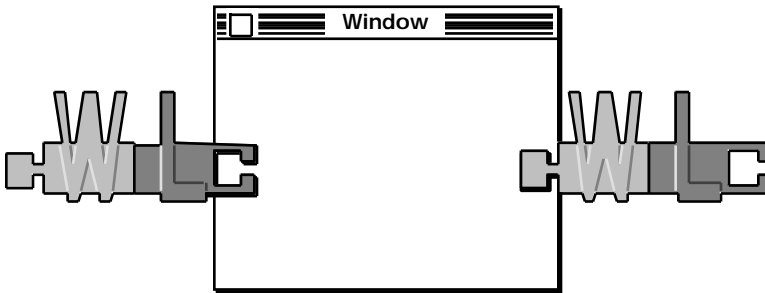


Fig. 8 Control procedures in a window

The initiating procedure (or 0-procedure) is run when a window is *opened*, i.e. not just brought to front. The initiation procedure actually belongs to the in-procedures; but since it is run just after the 'Open window' command, it is best to separate it from the triad. We will come back to this later.

Changing windows

To cause the cursor to move from one window to another, you start by piecing together the out-procedure of window A and the in-procedure of window B. Like this (next page) :

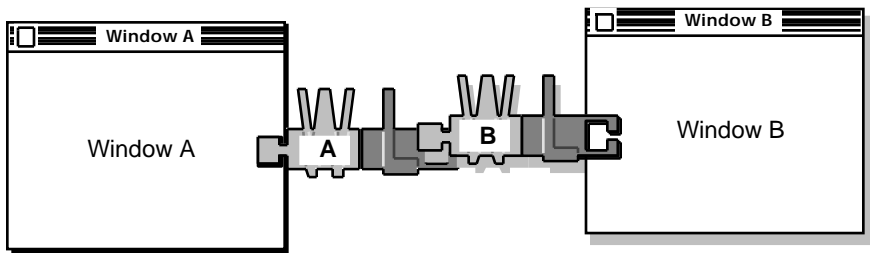


Fig. 9 Moving between windows

We can extend this by adding one house in each window. Now the cursor has to begin by leaving the house before he can run from one window to the other. When the cursor reaches window B, he will head for the first house in this window. (If the 'Keep Bringtofront clicks' option is on, it will move to the house being clicked on.) The resulting chain of procedures looks like this (Figure 10):

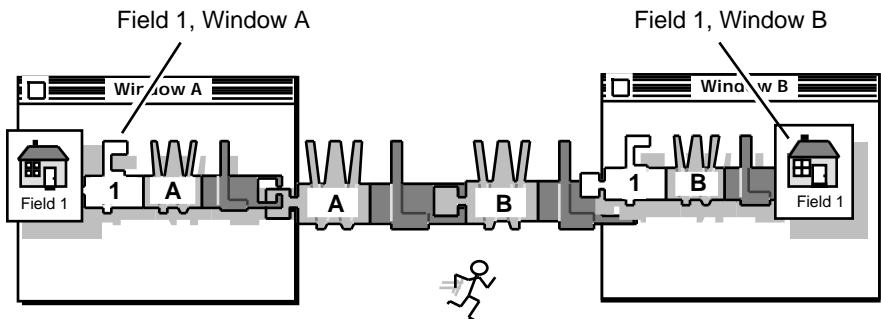


Fig. 10 Moving from a house in window A to a house in window B

Here the houses are Entry fields under Enter data. The complete sequence is as follows:

Exit Field 1 in Window A:

1. Field procedure for Field 1 in Window A
2. WCP for Window A
3. LCP

Exit Window A:

4. WCP for Window A

5. LCP

Enter Window B:

6. WCP for Window B
7. LCP

Enter Field 1 in Window B

8. Field procedure for Field 1 in Window B
9. WCP for Window B
10. LCP

In this way we can piece together the complete list of procedures for a given situation. Windows behave the same both inside and outside Enter data.

Frustrated?

If all this seems somewhat mindboggling, you may take comfort in the fact that it could have been a lot worse! Any programming situation involving general modules that are combinable will inevitably result in a lot of code and a host of procedure lines – the modules in this case being, of course, windows and fields with control procedures. Brief, concise code must be sacrificed to achieve full flexibility and predictability when the combination of modules is run.

Opening a window via menu or procedure

When a window is opened, procedure number 0 for the window is run. This happens immediately after the ‘Open window’ command is executed. When the procedure has been run, the WCP and the LCP take over. The same sequence is run when the window is opened from a field procedure (for example, under a Pushbutton.)

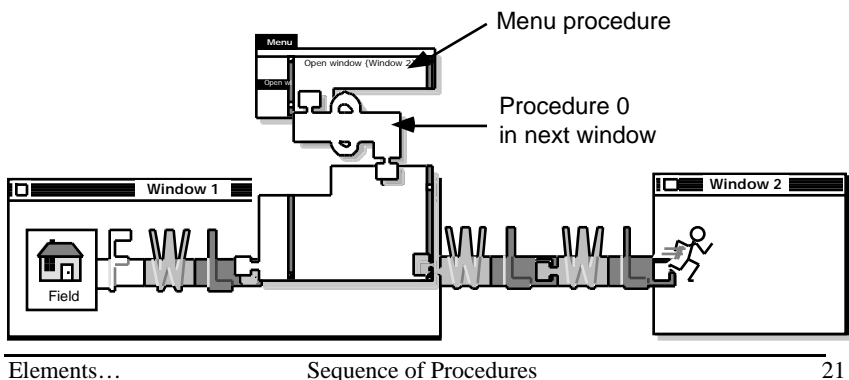


Fig. 11 Opening a window when the cursor resides in a field

If the menu is used when the cursor resides in a house, the out-procedures for the house are run before the menu procedure. If the window is opened, the 0 procedure follows immediately after the 'Open window' command. Next comes the rest of the menu procedure, followed by the out-procedures of the window being vacated. Finally, the in-procedures for the new window (and any in-procedures for the nearest house) are run.

Outside Enter data

The same basic rules apply both inside and outside Enter data, although some of the fields behave differently. The most important differences are that Entry Fields and Picture fields no longer house the cursor, nor do they run their field procedures. Display fields, Check boxes and Radio buttons also stop running their field procedures. This holds true for both Windows and Macintosh. The other fields generally behave as they normally do (see tables). There is no significant difference in the sequence of events when changing windows or using menus.

Table Fields in the System (v2.0)

Each table has its own field procedure that acts like a control procedure. Fields within the tables are treated in a special way. The tables may be regarded as “windows within windows.” The field procedure of the table is placed next to the front of the triad of each field, as if it were a Window Control Procedure for the “mini-window.” The sequence is as follows:

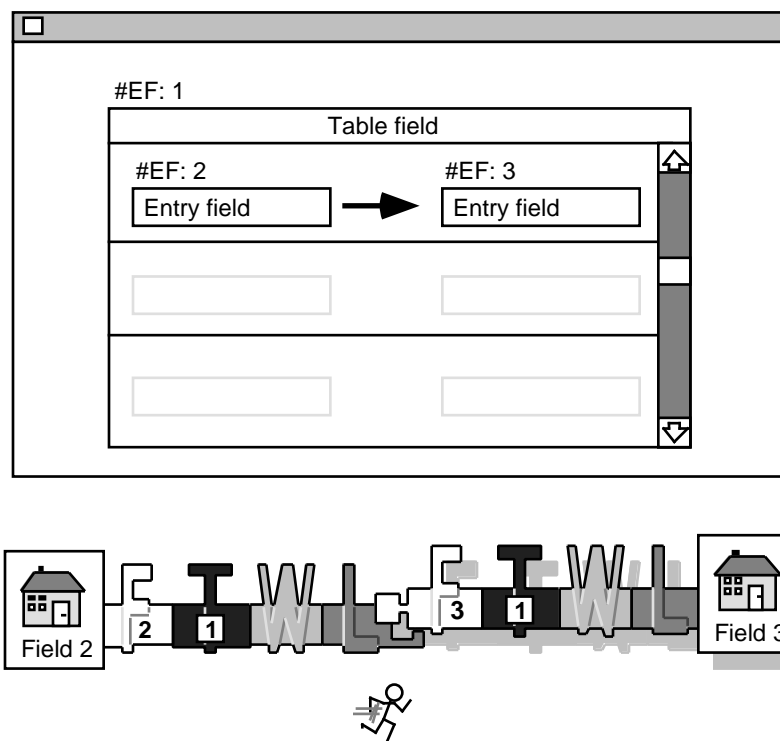


Fig. 12 Inside a table field, moving from one entry field to another

This sequence also applies when changing lines within the table, i.e. going from a house in one line to a house in another line. If the table is not set to ‘Enterable,’ it will function like an ordinary list, even where control procedures are concerned.

Complete sequence:

1. Field procedure for Field 2
2. Field procedure for Table field (Field 1)
3. WCP
4. LCP

5. Field procedure for Field 3
6. Field procedure for Table field (Field 1)
7. WCP
8. LCP

(As the cursor enter the table, the triad is run as usual, and from then on the table field procedure is included in the triad, as long as the cursor moves between houses inside the table.)

Set Next Action (SNA)

This command gives the developer an opportunity to force the user to do what the developer wants him to. ‘Set Next Action’ replaces the user’s last action with a “simulated user action,” which is carried out after all the procedures waiting to be run are completed. Structurally, SNA is on the level above the individual control procedures, since each user action works in triads. The last action the user performed (called “default action”) triggers the out-procedures, and all of these are run. However, the triad that normally follows (i.e. the in-procedures), belongs to the “default action.” Using the SNA command, we redirect the cursor, thereby canceling the triad or swapping it for in-procedures belonging to other fields.

Example

If the user clicks on another window, a #WCLICK is triggered. This can be intercepted with the aid of a WCP. If we give the ‘SNA remain on current field’ command, this means that the cursor is to remain in the field where he was. The window that was clicked on does not appear; the #WCLICK event has been avoided. The simulated user action is to return to the field. Other possible simulated user actions are the following:

TAB	Move to next house
SHIFT-TAB	Move to previous house
OK	Quit Enter data with Flag True
CANCEL	Quit Enter data with Flag False

Sequence

It is important to note that the procedure where the SNA command is found must have been completed in full before the SNA is carried out. This means that an Enter data in the same procedure as the ‘Set Next Action’ command will not be influenced by SNA, because the procedure terminates after the user has quit the Enter data of the procedure. If the developer wants to place the cursor in a field that is not the first house in the window field sequence, the field procedure of the first house in the window should contain an SNA command that directs the cursor to the correct field.

SNA can be used to prevent the user from doing something he or she shouldn't. For example, if the LCP discovers that the user has chosen 'Quit' from the standard 'File' menu (#ER=11009, see the 'Call procedure' list in Procedure Tools Window), the developer can prevent this from being carried out by using the 'SNA remain on current field' command. However, a number of the SNA commands can lead the developer into a never-ending loop, where his only recourse is to turn off the computer and lose some programming work. So look alive!

Standard Omnis window

When creating a window from a file (Make»Window format...), Omnis automatically inserts a standard WCP, which checks to see if the user is trying to change windows under Enter data, and prevents this by means of 'SNA remain on current field.' This effectively cancels the user's click on the background window.

Limitations

Omnis permits only one SNA at a time. If the developer inserts more than one SNA in the same procedure, only the last one will be carried out. 'SNA perform default action' cancels the effect of an SNA given earlier in the procedure (or earlier in the procedure sequence), so that what the end-user was trying to do is eventually respected. In v2.x, we can place simulated user actions one after the other in a sequence with the 'Queue' commands. See next page.

Queue Action

In v2.x we are allowed to place many “user actions” outside each other. The commands in this group behave in the same way as Set Next Action, with two exceptions. First, the most recent user action is allowed to pass, i.e. to be carried out, evoking all the procedures applying to that command. Second, these commands may be arranged in a sequence of user actions. Each new ‘Queue action’ command will be added to the sequence as it appears in the procedures. When the entire procedure sequence has been run, Omnis starts on the list of user actions, each of which may evoke long sequences of procedures. The actions in the list are removed from the queue as they are performed.

“Let me tell you a thing or two...”

Here is your chance to run circles around the poor end-user. With a repertoire of “bells and whistles,” the paternalistic developer may show and tell the user just what he or she should have done.

Queue keyboard event

One of the ‘Queue’ commands deserves special mention, the ‘Queue keyboard event.’ This can “record” any series of keystrokes from the keyboard and repeat the sequence later. This is a very comprehensive and interesting tool with a wide variety of uses. Within a fair-sized text field it will prove useful to move the cursor by means of the functions of the four arrow keys, the HOME key, and the END key.

Procedure Stack

Procedure stack represents the list of procedures which, at any given time, are waiting to be carried out. It may be visualized relatively easily by using 'Call procedure.' If procedure A calls procedure B, the procedure stack will consist of procedure B followed by procedure A, as long as procedure B is being run, because procedure B must be completed before A can resume running. If procedure B calls procedure C, B must wait till C has been completed before it can resume running. As long as C is running, the Procedure stack is as follows:

3. Procedure C
2. The rest of procedure B (after 'Call procedure C')
1. The rest of procedure A (after 'Call procedure B')

Procedure A is waiting for procedure B, which in turn is waiting for procedure C. The waiting procedure lines are marked with gray in Figure 13, below.

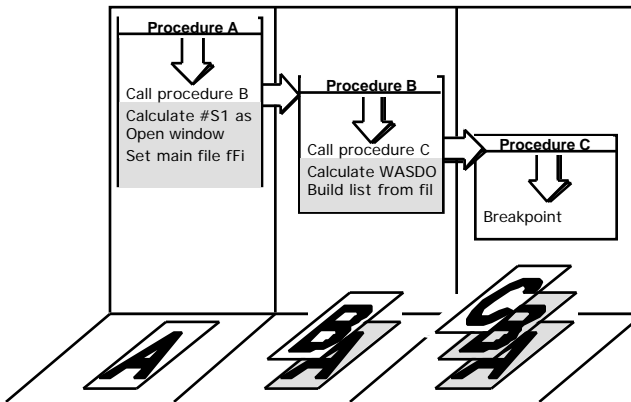


Fig. 13 Visualizing the Procedure stack

The Procedure stack is a product of sequential calls. Control and field procedures are also added to the Procedure stack, arranged nicely and neatly after the calling procedures that are waiting to be run.

‘Move up stack’/‘Move down stack’

The ‘Move up stack’ and ‘Move down stack’ commands represent one of the less intuitive aspects of the debugger. The menu they belong to provides us with a good way of searching for the procedures in question when several calls follow each other. To move “down” the Procedure stack is to move to the procedure that is next in line when the current procedure is completed. In practice, this means the rest of the procedure that called the current one.

Downward and upward in time

If A calls B, which in turn calls C, and we are focusing on C, Move Down will bring us to procedure B. When we have arrived there, Move down will take us to A, and Move up will take us back to C. Thus “Down” is ahead in time, whereas “Up” is back to the present.

About Figure 13

Let us imagine that when a new procedure is called (B), it is placed like a piece of paper on top of the procedure that called it (A). In the figure above, we see that C is placed on top of B, which in turn is placed on top of A; this lends a figurative meaning to the directions in ‘Move up’ and ‘Move down.’ ‘Move down’ is to work our way down to the bottom of this stack; ‘Move up’ is to work our way back to the top again.

The ‘Clear procedure stack’ command

This procedure command will burn all our bridges or remove every sheet in the stack except the top one. Everything that was to come after the current procedure will be canceled. The procedure that is currently running will, however, be completed. If A calls B, which in turn calls C, and C contains a ‘Clear procedure stack’ command, the remaining lines of B and A (after the lines containing ‘Call procedure’ in each respective procedure) will not be run, no matter how patiently they may have waited in line for procedure C. (That’s gratitude for you!)

Enter data

If one of the field procedures contains a ‘Clear procedure stack’ command and Omnis is in Enter data, all control procedures that are waiting in line will be abruptly forgotten and Enter data will promptly come to a halt. (See Figure 14.) This

is an efficient way of closing Enter data and starting afresh, since the procedure commands that are waiting to be carried out after Enter data will not be carried out at all. Updating of the file, particular calculations etc., is typically avoided. The cursor leaves the scene without a trace – and without remorse.

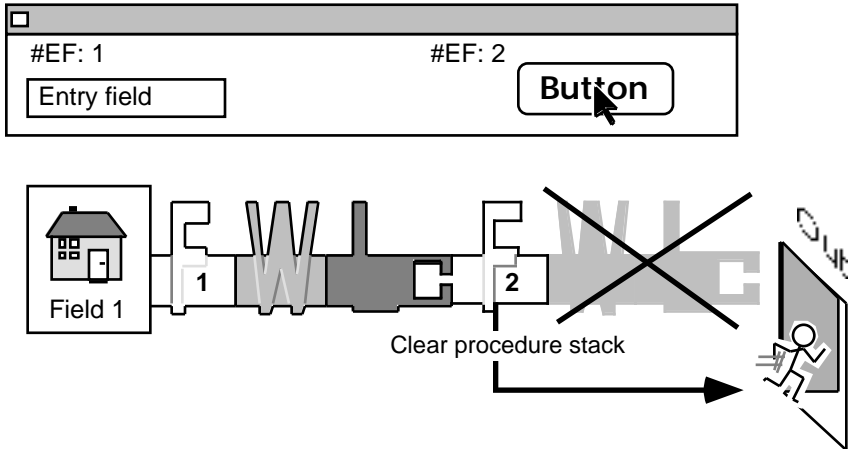


Fig. 14 Clearing the procedure stack

Leaving Enter data mode

Let us imagine that our cursor has encountered a pushbutton containing the 'Clear procedure stack' command. He immediately leaves Enter data, and all remaining procedure lines are canceled. The cursor himself leaves the site, but will turn up again in the nearest house outside Enter data.

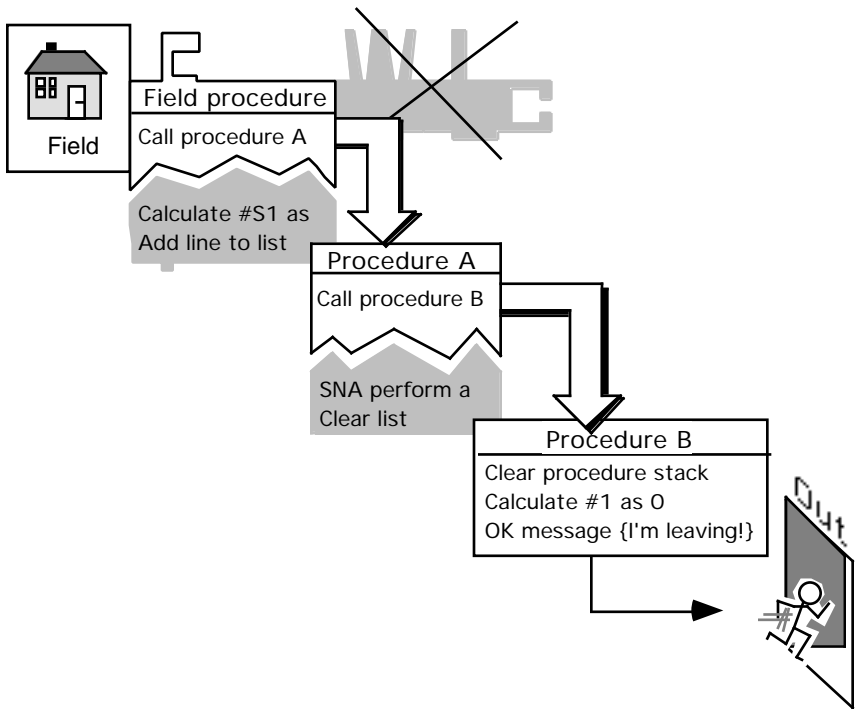


Fig. 15 Running away from a humongous procedure stack

Clearing a big procedure stack

In Figure 15 we have included a couple of calls in the sequence of procedures. That which is marked gray will not be carried out, on account of the 'Clear procedure stack' command in procedure B. Procedure B will, however, be completed in its entirety before the cursor leaves Enter data. The result of the sequence will be that the commands in procedure B will run, i.e. #1 will be set to 0, and an 'OK message box' will appear.

The procedure containing the 'Enter data' command

If the house was an Entry field under Enter data, the procedure containing Enter data will be canceled too, like the gray procedure lines in the figure. This procedure is not shown in the figure, because that would have made it too messy. If, on the other hand, the house had been a list field (housing the cursor outside Enter data), the situation in the figure might

have arisen outside Enter data; in that case the cursor could rest easily in the list field. Only if the field stops housing the cursor as Enter data mode ends will it be necessary for the cursor to move to the nearest house.

Quit to Enter data

This command is not quite as drastic as 'Clear procedure stack.' The cursor immediately halts in mid-procedure and also cancels the triad of procedures set with a particular event. It is soon ready for more, however. Enter data runs as before and will be closed in the normal way when the user presses OK or CANCEL.

Avoiding control procedures

If you don't want to go so far as to stop Enter data and not update the file etc., and would rather just skip the WCP or LCP, you can use 'Quit to enter data' at the end of the last procedure to be executed.

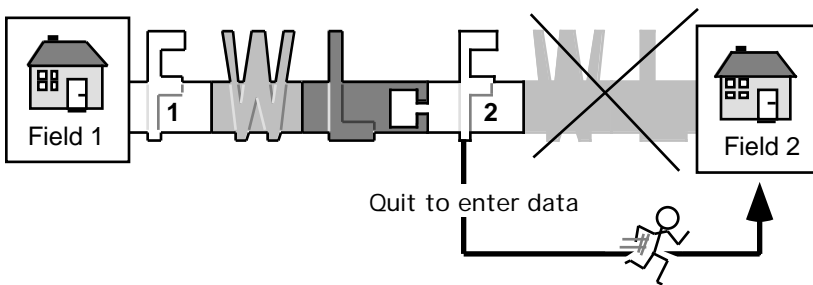


Fig. 16 Skipping the last control procedures on the way into the house

'Quit to enter data' entering a field

In Figure 16, the cursor encountered a 'Quit to enter data' in the field procedure for Field 2. He simply skipped the remaining 'WL' and went straight to the house in Field 2.

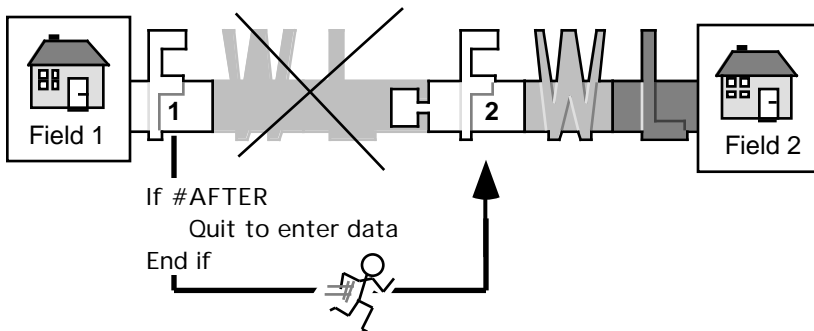


Fig. 17 Avoiding control procedures on the way out of the house

'Quit to Enter data' leaving a field

In Figure 17, 'Quit to enter data' lies in the field procedure for Field number 1. The cursor skips 'WL,' which is waiting in line, but bounces back to face the new challenges that Field 2 has in store. This means that in order to skip the following 'WL,' the field procedure for Field 2 must also contain a 'Quit to enter data' before the cursor resides safely in its house. (A 'Clear procedure stack' could have done the trick here, but then Enter data would have lost out, i.e. would have been canceled.)

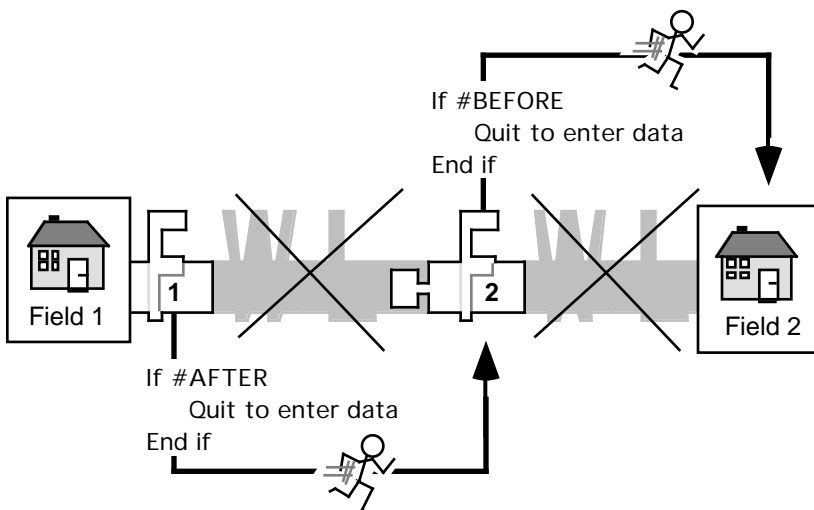


Fig. 18 Combining two 'Quit to enter data' commands, thus avoiding all control procedures while still in Enter data mode

Avoiding all control procedures

Combining two 'Quit to enter data,' as shown in Figure 18, makes it possible to avoid all control procedures when exiting one house and entering another. The prerequisite is that both field procedures are active so that they will run.

Quit all procedures

'Quit all procedures' works in the same way as 'Clear procedure stack,' but in this case the procedure that is being run is canceled immediately without being allowed to finish. 'Quit all procedures' is thus (if possible) even more drastic. Nothing marked gray in the figure will be carried out. If there is an Enter data, it too will be interrupted. Nor will the remainder of the procedure containing the 'Enter data' command (not shown in the figure) be completed. All control procedures are canceled. The cursor is in the nearest house waiting for the next user action, so a 'Quit all procedures' is not the end of the world. Compare Figure 19 with Figure 15 in the paragraph on 'Clear procedure stack.'

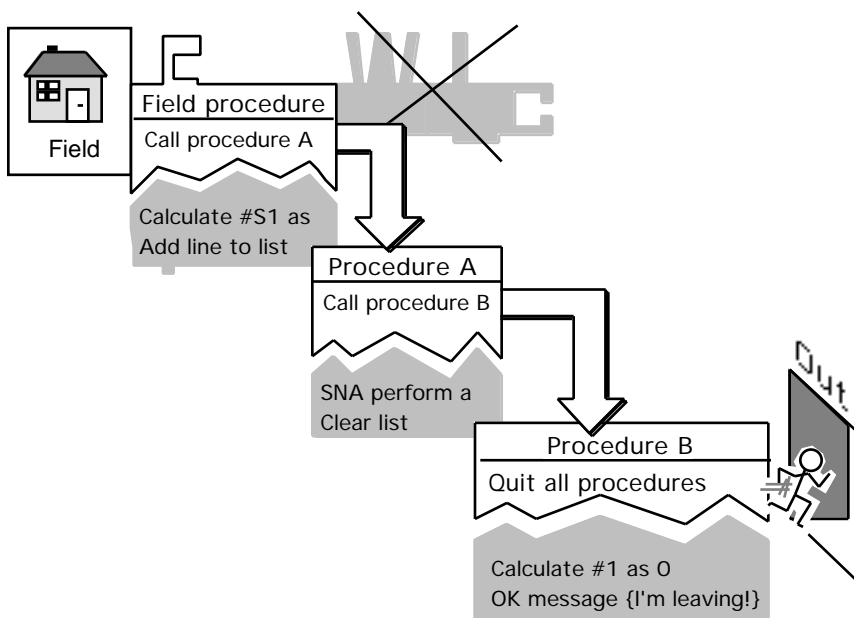


Fig. 19 The instant escape provided by 'Quit all procedures'

Tables of the Jig-Saw Model

The following tables provide a complete overview of the way the different fields function. Because Macintosh and Windows differ in a number of respects, each of these systems is given its own table. First the fields are classified according to the jig-saw model; then they are listed alphabetically for easy reference.

By “active” fields we mean fields that run field procedures. The tables apply to Omnis 7 (v2.0 and later), but on the whole can be used for version 1.x as well.

Tables – Macintosh

Tables categorized according to the jig-saw model (Macintosh)

Abbreviations

- F Field procedure
W Window Control Procedure
L Library Control Procedure (corresponds to Application Control Procedure)
o Cursor exits previous house and runs its out-sequence.
r Cursor returns to previous house.
I Internal action, i.e. a command executed by Omnis by means of standard pushbuttons.

Cursor houses (Macintosh)

Cursor houses when not in Enter data mode	#BEFORE	#AFTER
Combo box	FWL	FWL
Dropdown list (“Combo box” in v1.x)	FWL	FWL
Display Fields with scroll bar	WL	WL
Entry fields with scroll bar	WL	WL
Extended area	FWL	FWL
List field	FWL	FWL
Table field	FWL	FWL
Cursor houses during Enter data mode	#BEFORE	#AFTER
Combo box	FWL	FWL
Display fields with scroll bar	WL	WL
Dropdown list (“Combo box” in v1.2)	FWL	FWL
Entry field (all types)	FWL	FWL
Extended area	FWL	FWL
Picture field	FWL	FWL
List field	FWL	FWL
Table field	FWL	FWL

Non-Cursor houses (Macintosh)

Active during Enter data and when not in Enter data mode.	Action	(Same reaction during normal and Enter data mode.)
Pushbutton Button area Standard pushbuttons	#CLICK #CLICK #CLICK	o FWL r o FWL r o FWLWLI r
Popup list	Open list (click on the field) Release list, no selection #CLICK (new list line chosen) or #DCLICK (same list line)	o r FWL r
Menu	Open menu (click menu title) Menu selection	Nothing happens o F r
Popup Menu	Open menu (click menu title) Release menu, no selection Menu selection	o r WLF r
Active during Enter data	Comment	(#CLICK)
Radio button Check box OK, Cancel pushbutton Display field (no scroll bar)	After this, Enter data is terminated No out-procedures are evoked leaving the previous cursor house.	o FWL r o FWL r o WLI (r) WLFWL r

Inactive fields (Macintosh)

No cursor houses, no field procedure and no out-procedures.	#CLICK
Entry field, when not in Enter data	WLWL r
Display field (no scroll bar), when not in Enter data	WLWL r
Picture field, when not in Enter data	WLWL r
Check box, when not in Enter data	WLWL r
Radio button, when not in Enter data	WLWL r
Disabled pushbuttons and disabled standard Pushbuttons	WLWL r

In a WLWL, the first part (WL) applies to the house in which the cursor resided; the last part applies to the inactive field. Any Enter data messages concerning the inactive field will thus show up in the last pair with the WCP and ACP.

Alphabetical overview (Macintosh)

Abbreviations

Hs	Cursor house at all times.
Hs ED	Cursor house during Enter data only.
Not Hs	Not a Cursor house.
FP	Able to run field procedure at all times.
FP ED	Runs field procedure under Enter data only.
#BEFORE	Sequence of procedures inside a house, i.e. in-procedures. (Entered for houses only.)
#CLICK	Sequence of procedures when clicking in a field or selecting from a menu, list or combo box. (Selection when the cursor resides in the house of the last two. In other cases only the in-procedure is evoked by clicking and simultaneously selecting a line.)
#AFTER	The sequence of procedures exiting a house, i.e. the out-procedures. Entered for houses only.

(The abbreviations F, W, L etc. are explained in the previous tables)

Window field type	Hs	Hs ED	Not Hs	F	F ED	#BEFORE/ #CLICK	#AFTER
-------------------	----	-------	--------	---	------	--------------------	--------

Elements...

Sequence of Procedures

Button area		x	x	oFWL r	
Check box		x	x	oFWL r	
Combo box	x		x	FWL	FWL
Dropdown list	x		x	FWL	FWL
Display field, no scroll bar		x	x	oWLFWL r	
Display field w/scroll bar	x		x	WL	WL
Entry field, no scroll bar		x	x	FWL	FWL
Entry field w/scroll bar	x		x	WL	WL
List	x		x	FWL	FWL
Menu		x	x	oF r	
OK, Cancel button		x	x	oWLI (r)	
Picture field		x	x	FWL	FWL
Popup list		x	x	oFWL r	
Popup menu		x	x	oWLF r	
Pushbutton		x	x	oFWL r	
Radio button		x	x	oFWL r	
Standard button		x	x	oFWLWLI r	
Table	x		x	FWL	FWL

Tables – Windows

Tables categorized according to the Jig-saw model (Windows)

Abbreviations

- F Field procedure
W Window Control Procedure
L Library Control Procedure (corresponds to Application Control Procedure)
o Cursor exits previous house and runs its out-sequence.
r Cursor returns to previous house.
I Internal action, i.e. a command executed by Omnis by means of standard pushbuttons.

Cursor houses (Windows)

Cursor house when not in Enter data	#BEFORE	#AFTER
Button area	WL	WL
Combo box	FWL	FWL
Display Fields with scroll bar	WL	WL
Dropdown list (“Combo box” in v1.2)	FWL	FWL
Entry fields with scroll bar	WL	WL
Extended area	FWL	FWL
List	FWL	FWL
Pushbutton	WL	WL
Standard pushbutton	FWL	FWL
Table	FWL	FWL
Cursor house during Enter data	#BEFORE	#AFTER
Button area	FWL	FWL
Checkbox	FWL	FWL
Combo box	FWL	FWL
Display fields w/scroll bar	WL	WL
Dropdown list (“Combo box” in v1.2)	FWL	FWL
Entry fields (all types)	FWL	FWL
External area	FWL	FWL
List	FWL	FWL
Picture field	FWL	FWL
Pushbutton	FWL	FWL
Radio button (treated as group)	FWL	FWL
Table	FWL	FWL

Non Cursor houses (Windows)

Active during Enter data and when not in Enter data mode.	Action	(Same reaction during normal and Enter data mode.)
Popup list	Open list (click on the field) Close list, no selection #CLICK (new list line chosen) or #DCLICK (same list line)	o r FWL r
Menu	Open menu (click menu title) Menu selection	Nothing happens o F r
Popup Menu	Open menu (click menu title) Close menu, no selection Menu selection	o r WLF r
Active during Enter data	Comment	#CLICK
OK, Cancel pushbutton	After this, Enter data is terminated	u WLI (r)
Display field (no scroll bar)	No out-procedures are evoked leaving the previous cursor house.	WLFWL r

Inactive fields (Windows)

No cursor houses, no field procedure and no out-procedures.	#CLICK
Check box, not Enter data	WLWL r
Disabled pushbuttons and disabled standard Pushbuttons	WLWL r
Display field (no scroll bar), not Enter data	WLWL r
Entry field, not Enter data	WLWL r
Picture field, not Enter data	WLWL r
Radio button, not Enter data	WLWL r

In a ‘WLWL,’ the first part (WL) applies to the house in which the cursor resided, the last part to the inactive field. Any Enter data messages that apply to the inactive field will show up in the last pair of WCPs and ACPs.

Alphabetical overview (Windows)

Abbreviations

Hs	Cursor house at all times.
Hs ED	Cursor house during Enter data only.
Not Hs	Not a Cursor house.
FP	Able to run field procedure at all times.
FP ED	Runs field procedure under Enter data only.
#BEFORE	Sequence of procedures inside a house, i.e. in-procedures. Entered for houses only.
#CLICK	Sequence of procedures when clicking in a field or selecting from a menu, list or combo box. (Selection when the cursor resides in the house of the last two. In other cases only the in-procedure is evoked by clicking and simultaneously selecting a line.)
#AFTER	The sequence of procedures exiting a house, i.e. the out-procedures. Entered for houses only.

(The abbreviations F, W, L etc. are explained in the previous tables.)

Window field type	Hs Not			F		#BEFORE	#AFTER
	Hs	ED	Hs	F	ED		
Button area	x			x		(F)WL	(F)WL
Check box		x			x	FWL	FWL

Combo box	x		x	FWL	FWL
Dropdown list	x		x	FWL	FWL
Display field, no scroll bar		x	x	WLFWL r	
Display field w/scroll bar	x		x	WL	WL
Entry field, no scroll bar		x	x	FWL	FWL
Entry field w/scroll bar	x		x	(F)WL	(F)WL
List	x		x	FWL	FWL
Menu		x	x	uF r	
OK, Cancel button		x	x	oWLI (r)	
Picture field		x	x	FWL	FWL
Popup list		x	x	oFWL r	
Popup menu		x	x	oWLF r	
Pushbutton	x		x	(F)WL	(F)WL
Radio button		x	x	FWL	FWL
Standard button	x		x	FWL	FWL
Table	x		x	FWL	FWL

“(F)” indicates that the field procedure is run during Enter data only.

Events as Evoking Factors (Macintosh and Windows)

Field, window and application control procedures are triggered by events. These are events evoked by the user's actions, and each of them is represented by a hash variable. If the field in question is active, the field procedure is run, followed by the activated control procedures.

Other message variables (hash variables) elaborate on the status and situation of these events but do not themselves evoke an FWL "triad" (or WL). These we have dubbed "status variables." Here is a list of events and status variables:

"Real" events – evoking FWL or WL

#AFTER	Leaving all kinds of fields housing the cursor and all windows opened through procedures.
#BEFORE	Entering all fields housing the cursor and all windows opened through procedures.
#BEFORE1	Same as above.
#BEFORE2	Same as above.
#CLICK	On all kinds of fields except Entry fields during Enter data. Otherwise on the window's background.
#DCLICK	On all kinds of fields except Entry fields during Enter data, and not on pushbuttons.
#DISABLED	When a field is set to disabled. Requires \$statusevents set to 'kTrue.'
#DROP	When a field has something dropped on it. Requires \$drop for the field set to 'kTrue.'
#ENABLED	When a field is set to enabled with 'Enable fields.' Requires \$statusevents set to 'kTrue.'
#HIDDEN	When a field is set to hidden with 'Hide fields.' Requires \$statusevents set to 'kTrue.'
#HOLD	When the user points to a field while holding down the mouse button. Requires \$drag for the field set to 'kTrue.'
#HSCROLLED	When the user uses the horizontal scroll bar.

#KEYPRESS	When the user presses a key on the keyboard and the cursor resides in a field that is a cursor house. Requires \$keyevents set to 'kTrue.'
#MOUSEDOWN	When the user points to any field and presses the mouse button. Is also sent for window background. Requires \$mouseevents set to 'kTrue.'
#MOUSEENTER	When the mouse moves into a field area. Is also sent for window background. Requires \$mouseevents set to 'kTrue.'
#MOUSELEAVE	When the mouse moves out of the area of a field. Is also sent for window background. Requires \$mouseevents set to 'kTrue.'
#MOUSEUP	When the user releases the mouse button, and the arrow is pointing to any field or the background of the window. Requires \$mouseevents set to 'kTrue.'
#MOVE	Requires \$drag for the field set to 'kTrue.'
#OK	(May be counted as an event. Is sent simultaneously with #AFTER.)
#RESIZE	Not associated with any particular field. Evokes WA only.
#RHOLD	Requires \$drag for the field set to 'kTrue.'
#RMOUSEDOWN	Requires \$mouseevents set to 'kTrue.'
#RMOUSEUP	Requires \$mouseevents set to 'kTrue.'
#SENT	
#SHOWN	Requires \$statusevents set to 'kTrue.'
#TOTOP	
#TOTOP1	
#TOTOP2	
#VSCROLLED	
#WCLICK	

Status messages – accompanying various events

#ALT	
#CANCEL	Sent together with #AFTER
#CLOSE	Sent together with #AFTER
#COMMAND	
#CTRL	
#DELETE	
#EDATA	
#EDIT	
#EF	
#EFLD	
#EM	
#ENTER	(Enter key evokes an #OK, which may be counted as an event.)
#ER	
#FIND	
#INSERT	
#INSERTCV	
#INSIDE	In tables only.
#KEY	
#LCHANGE	In tables only.
#NEXT	
#OPTION	
#PREVIOUS	
#RETURN	
#SHIFT	
#SKEY	
#STAB	
#TAB	

Events that must be activated before use

Some events make heavy demands on the processor and are therefore normally turned off. Below is a list of events that must be activated before they are used. The notation object (to be set to 'kTrue') heads each group. The accompanying status variables are placed under each event.

`$clib.$prefs.$mouseevents`

`#MOUSEUP`

`#MOUSEDOWN`

`#MOUSEENTER`

`#MOUSELEAVE`

`$clib.$prefs.$rmouseevents`

`#RMOUSEUP`

`#RMOUSEDOWN`

`$clib.$prefs.$keyevents`

`#KEYPRESS`

`#KEY`

`#SKEY`

`$clib.$prefs.$statusevents`

`#DISABLED`

`#ENABLED`

`#SHOWN`

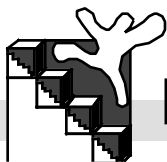
`#HIDDEN`



Learning the sequence of procedures is a lot like
learning scales

– not exactly music to your ears, but useful
nonetheless.





Lists & Tables

What Is a List?.....	2
List Settings.....	4
Manipulating Lists	6
Creating and pointing to a list	
Defining a list	
Building a list	
Editing lines in a list	
Editing single values in a list	
Automatic editing control (Local)	
Multiple line selections in lists	
Displaying Lists in Windows.....	24
“Simple” List fields	
Dropdown lists (Combo boxes in v1.x)	
Popup lists	
Combo boxes	
Field value window for lists	
Tables	
About the calculation field	
Suggested ways of using lists	
Displaying Single List Values.....	33
Using ‘Load from list’	
The ‘lst’ function	
Lists Stored in Datafiles.....	35
Lists within Lists	37
Redrawing Lists	38
Redraw named fields	
Redraw numbered fields	
Binary Search in Lists	40
Tables.....	43
What is a table, anyway?	
Window elements in tables	

What Is a List?

A list is an isolated area in memory that contains values in an ordered system. Think of a list as a table containing rows and columns, just like those in textbooks and newspapers. We use fields from the file formats or hash variables to give the columns their titles ('Define list'); but in themselves, the contents of the lists are totally independent of the fields. When we give the 'Add line to list' command, the contents of the fields in the list definition (which gave the columns their names) are copied to the new line in the list. Thus the numbers and values have their allotted space in memory, independent of the original fields. The values can be transferred to the respective fields again with 'Load from list.' Lists are usually manipulated whole lines at a time so that all the values on that line are affected. New lines can be added and old ones removed.

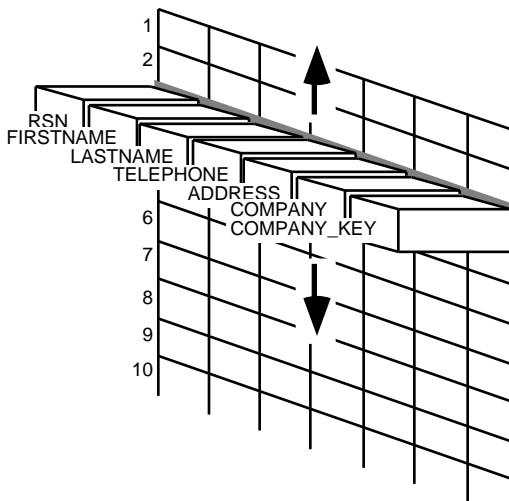


Fig. 1 One way of visualizing a list. The boxes with labels represent fields receiving values using 'Load from list.'

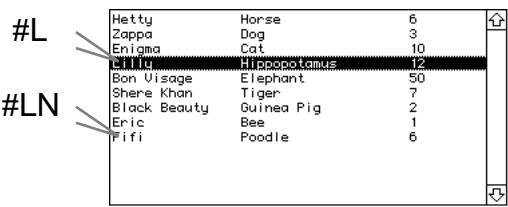
Lists and the CRB

It's important not to mix the lists' values with the actual fields (in the CRB) which have given the list columns their titles. The titles can be

regarded as “beacon” connections between the lists’ contents and the Current Record Buffer. If we select a list line and give the ‘Load from list’ command, the values in that line will be transferred to every field that was set as a column title. This means that we will have copied values from the selected line to its related fields in the CRB.

List Settings

For every list there is a group of settings that help us in our programming. Let's begin by looking at the most important ones.



The diagram shows a list window with a table of animals and their counts. A horizontal bar highlights the row for 'Horse' (6). An arrow labeled '#L' points to this highlighted row. Another arrow labeled '#LN' points to the first row of the table, 'Hetty'.

Hetty	Horse	6
Zappa	Dog	3
Erigma	Cat	10
Horse	6	
Bon Visage	Elephant	50
Shere Khan	Tiger	7
Black Beauty	Guinea Pig	2
Eric	Bee	1
Fifi	Poodle	6

Fig. 2 #L and #LN in a list

#L

The #L variable contains the ‘current line’ number. When the user clicks on a list field in a window, the value of #L for the field’s list will be set to this line. (We can safely assume that #L is the line the user has clicked on. The field procedure is run directly after #L has been set, anyway.) Many commands, such as ‘Load from list,’ ‘Insert line in list,’ etc. use #L unless otherwise specified. Remember that ‘Search list’ changes #L to the line that satisfies the search criteria. Each list has its own #L, so you must check what Current list is when #L is being modified or its value is being used.

#LN

The #LN variable tells us how many lines the current list contains. This number also corresponds to the line number on the last line of this list. If, for example, you wish to find the last line in a list, #LN can be of service. To modify #LN, you have to add or delete lines from the list.

#LM

The #LM variable limits how big a specific list is allowed to be. It is usually set to around 10 billion, so it shouldn’t give you any headaches. However, we can use it to keep lists from becoming too

large. When the user is allowed to have a say in how to set up a list, we risk having the whole datafile fed into it; the lists gain weight dramatically. This may be avoided by using the #LM variable. When #LN (number of lines) becomes greater than #LM (maximum number of lines permitted), every command that automatically builds lists will abort. This applies to the ‘Build list from file’ command, and a number of others. There are other settings, and we’ll take a closer look at them later on in this chapter.

Manipulating Lists

Creating and pointing to a list

We can use one of the eight built-in lists, #L1–#L8, or generate (i.e. declare) one ourselves. We create global lists as a field in a ‘Memory Only’ file, or as a Library variable (in v2.x). Lists that are only needed within the same format (menu or window) should be declared as format variables. In a simple procedure we just need to create the list as a local variable. Most list commands only work with one list at a time, which is why we have to point out which list we wish to work with. Here we use ‘Set current list (List name),’ which specifies the name of the list we’ll be working with. The #CLIST variable always shows us the name of the Current list.

Defining a list

It’s about time we decided on the contents of the list. It can use any sort of field from any file format, whether connected or not. We use the ‘Define list’ command to include whatever we want on the list. Naturally, the list is deleted in the process. In the interests of simplicity, we can key in the name of a file format in the list definition, thus setting all the fields (in the file format) as titles for the columns.

Building a list

In principle, building a list is simplicity itself. The process consists of finding records in a certain order and copying selected field values into the list as the records appear. We normally use one or more search criteria to decide which records to read in – for example, using ‘Set search name.’ We have already decided which values to insert in the list by using the ‘Define list’ command. When a record has been found, this means that we have a set of field values in the CRB. These may be freely inserted in a new line in the list, e.g. with ‘Add line to list.’ Omnis will see to it that the right values wind up in the right column. Figure 3 illustrates what this process might look like.

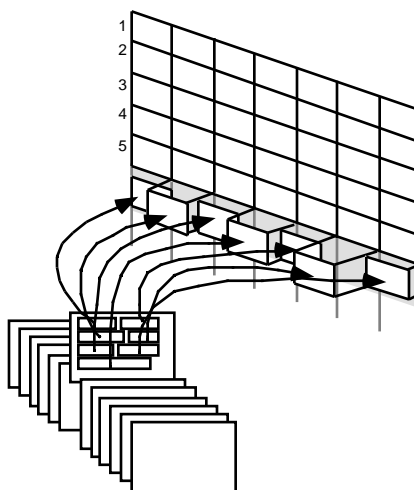


Fig. 3 When a list is built automatically, the values from the fields corresponding to the list definition are copied into the list as each record is located.

Whether the list building takes place by using ‘Add list’ within a loop or by choosing the ‘Build list’ commands, the principle remains the same. We can call the former, “procedure-controlled” and the latter, “automatic.” The difference is that automatic list building takes care of all the procedure commands that you would have placed in the loop yourself. Let’s take a closer look at this.

Procedure-controlled building

Procedure-controlled list building puts a tool of great potential in the developer’s hands, and it isn’t difficult to use either. We use a procedure with an ‘Add line to list’ command inside a loop. The only thing you need to make sure of is that the fields that were set up in the list definition have the correct values when ‘Add line to list’ is carried out. This means using familiar commands like ‘Find,’ ‘Next,’ ‘Calculate,’ etc. Calculations or searches are often placed inside the loop.

Show Character set	1
Set current list #L1	
Define list {#S1,#1}	
For #1 from 33 to 255 step 1	

```

Calculate #S1 as chr(#1)
Add line to list
End For

```

List with character set

(1)

This procedure builds a list that displays the computer's character set, i.e. all the letters with their corresponding number. The loop ranges from 33 to 255. For every repetition we place the character corresponding to this number in the #S1 variable, which is then placed in the #L1 list.

Find names using contents of #S1

2

```

Set current list #L1
Define list {C_LastName }
Set main file {fCustomers}
Calculate #1 as len(#S1)

Find on C_LastName {#S1}
While mid(C_LastName,1,#1)=#S1
    Add line to list
Next
End While

Redraw windows

```

Search list

(2)

The example in procedure 2 builds a list of records in fCustomers where C_LastName begins with the letters that #S1 contains. The #S1 variable thus functions as a search code for C_LastName. If the user types "Jo" in #S1, the list will receive names such as Johnson, Jones, Jordan, etc. This procedure works very quickly, beating out v1.2's 'Build list from file (use search).'

Automatic list building

Omnis can virtually build us a list automatically – by using the ‘Build list from file’ command, for example. It works like this: Omnis goes through Main file in the order of the designated indexed field and adds a new line to the list whenever a record is found. Any records in connected files are called up during the process, so we can easily retrieve values from fields in different connected files and place them in the same list. This requires that Main file be set to the file at the bottom of the connection hierarchy. The logic here is the same as for the ‘Find’ command. This also means that when building lists automatically, we should only use fields from connected files that are directly above each other in the hierarchy. The procedure looks like this:

Build list with connected records	3
Set current list #L1 Define list {fChildren, fParents} Set main file {fChildren} Build list from file on C_Name Redraw windows	

After Procedure 3, list #L1 will contain values from the connected records in both files.

Expansions

There is no reason to get stuck at this point. We can go on to build the list using search criteria with fields from the parent file. Similarly, we can sort the list according to fields in the parent file. It isn’t necessary to add fields in the list definition that are part of the search criteria. The search itself decides only which records to read in and in what order, whereas the list definition decides which values to insert in the list every time a record is retrieved.

Find connected records	4
Set current list #L1 Define list {C_Name,C_RSN} Set main file {fChildren} Calculate #1 as P_RSN Set search as calculation {P_RSN=#1} Build list from file on C_Name (Use search) Redraw windows	

Find connected records (4)

If, for example, we are in the window for the fParents file, Procedure 4 can find the connected records in fChildren. The field P_RSN contains the Record Sequence Number of fParents.

Find record represented in selected list line	5
If #DCLICK Set current list Fols_Connected_Children Load from list Single file find on C_RSN (Exact match) Redraw windows End if	

Find record represented by the selected line in the list (5)

If the user double-clicks on a line in the list, a field procedure like Procedure 5 will be able to find the appropriate record in the child file. (The Fols_Connected_Children list is a format variable, and its declaration is not shown here.)

Main file and Current list

I had better mention the fact that both Main file and Current list are stationary objects which are normally not changed without issuing the 'Set main file' or 'Set current list' commands. That's why it isn't necessary to specify Main file and Current list in each of these procedures; once is enough. However, both commands have been included in the procedure examples to make the code unambiguous.

In practice, this means not having to worry about whether they've been set properly for the procedure in question – if you've set them

yourself. For example, you can place a ‘Set main file’ command at the beginning of every procedure that has one or more Main file-dependent commands. When the procedures make sure that everything is in place, the code will be nice and stable.

Editing lines in a list

Usually the developer manipulates whole lines at a time. ‘Add line to list’ adds a set of values to the *end of the list*, and the number of lines (#LN) increases by one. #L remains unchanged. ‘Insert line in list’ inserts a set of values *in the designated line*. The rest of the lines are displaced downward by one line with no loss of content. ‘Replace line in list’ replaces the designated line with the current values of the fields specified in the line definition.

Adding a line

Figure 4 shows us how a whole set of values (from the fields in the CRB) are placed at the end of a list when the ‘Add lines to list’ command is used.

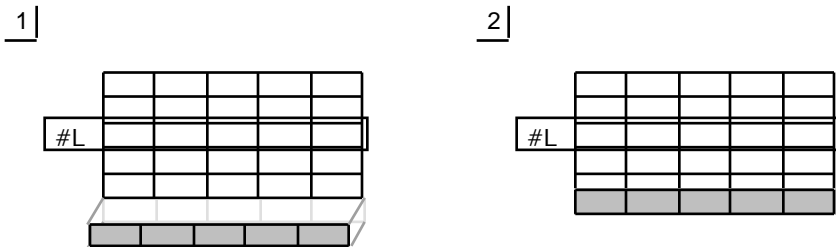


Fig. 4 Adding a line to a list

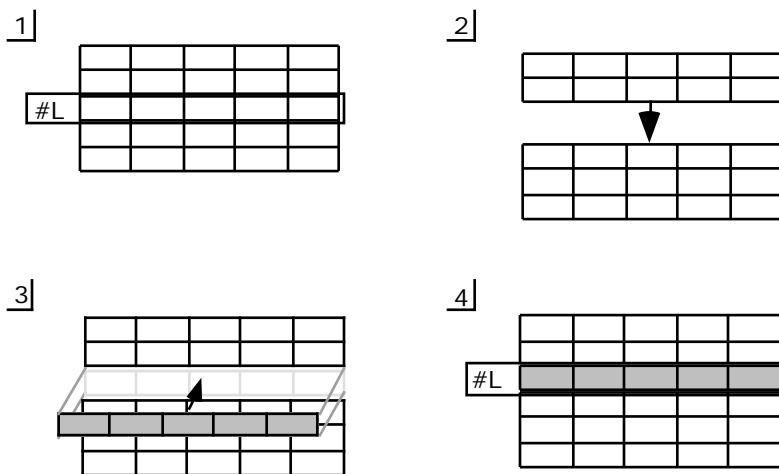


Fig. 5 Inserting a line in a list

Inserting a line

Figure 5 shows what happens when the ‘Insert line in list’ command is carried out. The current line is moved down a notch, and the new set of values is inserted in its place. #L remains unchanged during the operation.

Moving lines

Whole lines can be moved from one list to another by using ‘Load from list’ and ‘Add line to list.’ The fields in the CRB become a buffer. Only the common fields in the list definitions are transferred, so in this case both lists should be the same.

Merging lines

The ‘Merge list’ command “bakes” two lists into one. This is a useful command when moving many lines between lists. For one thing, it is quick; for another, the CRB remains unchanged. Current list is the list that receives values. If we want, we can tell which lines should be included by resorting to a search format or a search calculation.

Deleting a line

The ‘Delete line in list’ command removes a given line and moves up the lines below it, so that #LN decreases by one. The #L variable doesn’t change value unless it marks the last line in the list. If it does, #L will be equal to the new #LN, which is one (line) less. The ‘Clear line in list’ command only deletes the values in the designated line so that the latter remains empty. In other words, ‘Clear line in list’ doesn’t alter #LN.

Editing single values in a list

There are times when you will only want to extract or alter a few of the values in a list. There are several ways of going about this.

Replace line in list	6
Define list {B_RSN, B_Text, B_Number, B_Boolean} Build list... ;; Build up contents of list	
Replace line in list {3 (, #S1, "1")}	

Replace line in list (6)

The ‘Replace line in list’ command can change selected field values in a specific line in a list while leaving the other list values untouched. The fields (or values) to be inserted are enclosed in parentheses, and their placement is determined by the use of commas.

In procedure 6, value of the #S1 variable is inserted as a list value in column ‘B_Text,’ line 3, and the list value in column ‘B_Boolean’ is set to 1. The commas separate the columns in the list definition, and this indicates where the values should be inserted.

Calculate #L1(column,row) as...

It is possible to change values in a list directly, one at a time. To point to the right cell, we must specify the two coordinates, which are the column number and the row number. This method uses numerals only, and doesn’t take into consideration any field (column) names.

Using the ‘Calculate’ command, we change the value in the second column, third row of the #L1 list in the following way:

```
Calculate #L1(2,3) as "New Value"  
; Syntax: Calculate listname(column,row) as Value
```

Load from list

Likewise, ‘Load from list’ can read in selected list values by designating the line number and column(s) by position. This is done by correctly placing the fields to receive these values in relation to the order of the fields in the list definition. The columns are separated by commas. Try this:

```
Load from list {3 (, #S1, #1, )}
```

In this example, we copy the values from line 3, columns 2 and 3, into #S1 and #1, respectively. (In the embedded parenthetical expression, the first comma separates the second column from the first; the last comma separates the fourth column from the third.) Within calculations you can also use the ‘lst’ function, which has the following syntax:

```
lst (listname, line number, column title)
```

This represents a single value in calculations. Since Omnis will locate the right list value for the designated field (i.e. the column title), the developer doesn’t need to specify the column position. This is how to read the value of the ‘#S1’ column, line 3, in list #L1:

```
lst(#L1,3, #S1)
```

Automatic editing control (Local)

There is a very simple way of saving some time when you have a list and want the user to be able to edit its contents. Place the fields to be edited as Entry fields in the window. Assign the List field the lowest window field number (the window field sequence within the window), and afterwards assign window field numbers an uninterrupted sequence. Example: the List field is field number 17, and the Entry fields are given field numbers 18, 19, etc. The ‘Local’ option in each Entry field must be checked off. This means that the list values will automatically be

transferred to the corresponding fields when the user clicks on a line, and during Enter data, any changes will be updated in the list. The developer should allow for the user to add new lines by creating a pushbutton, which would run an ‘Add line to list’ command.

Saving the list contents to disk

When the user is satisfied and has clicked on OK, the developer can allow a subsequent procedure to save the changes in the list to disk. If the list is part of a file format being saved to disk, an ‘Update files’ command is all that’s needed. However, if the lines in the list correspond to records in a Read/Write file format, you go through the list line by line, read in the corresponding records, and compare them to see if there are any differences. If there are, the values from the list should be transferred to the CRB and the new edition of the record should be saved to disk.

Update the lines of #L1 to records of fChild	7
Enter data	
If flag true	
For #L from 1 to #LN step 1; Steps through all lines in list	
Single file find on C_RSN {lst(#L1,#L,C_RSN)}	
If flag false ; The line has no corresponding record	
Add a new record	
Else	
Difference between CRB and line in list? (Procedure 8)	
**Update record on disk	
End If	
End For	

Updating lines in a list to records in a file (7)

Our example uses list #L1, which contains fields from fChild. Initially, the list is built by using values from the datafile (the ‘Build list from file’ command), after which the user is allowed to edit it (under Enter data).

If the line in the list is entirely new, we insert it as a new record. We have to read in the values after ‘Prepare for insert,’ because this command clears the field values in the main file. If we had transferred the values from the list before the ‘Prepare for insert’ command, it would have caused the field values in Main file to vanish.

Adding a new record based on a newly added line

If a line is totally new, Omnis will not be able to find a corresponding record in the data file. Thus a ‘Find’ or ‘Single file find’ command resulting in ‘flag false’ would suggest that the line has been added by the user. However, we can also use the RSN, represented here by B_RSN. Generating sequence numbers is not the user’s responsibility, which is why B_RSN will be zero in lines added by the user. It’s up to the developer to choose an appropriate method. Main file’s part of the CRB will be cleared if a failed ‘Find’ or ‘Single file find’ occurs, but this has little practical bearing on the procedure as we have written it.

If the procedure discovers that the record already exists, we check to see whether there is a difference between the line in the list and the field values for the corresponding record, which is now in memory:

Difference between list line and CRB?	8
Local variable DIFFERENCE (Boolean)	
If C_Text<>Ist(C_Text)	
Calculate DIFFERENCE as 1	
Else If C_Number<>Ist(C_Number)	
Calculate DIFFERENCE as 1	
Else If C_Boolean<>Ist(C_Boolean)	
Calculate DIFFERENCE as 1	
End If	
If DIFFERENCE	
Quit procedure (flag set)	
Else	
Quit procedure (flag clear)	
End If	

Is there a difference between the contents of the list line
and the record in memory? (8)

The procedure has its own separate subprocedure to ensure that the main procedure doesn’t become too long and complex. By making use of the ‘Quit procedure’ commands that affect the flag, we get a call that behaves pretty much like a command. If there is a difference, i.e. the flag is ‘true,’ we update the record on disk. This is all done in Procedure 9:

```

Enter data
If flag true
  Set main file {fChild}
  Set current list #L1
  For #L from 1 to #LN step 1
    Single file find on C_RSN {lst(C_RSN)}
    If flag false
      Prepare for insert
      Load from list
      Update files
    Else
      Call procedure vChild/20 {Difference, #L1 and CRB?}
      If flag true
        Prepare for edit
        Load from list
        Update files
      End If
    End If
  End For
End If

```

Multiple line selections in lists

When the 'Show selected lines' option has been checked off for a List field in a window, the user can select several lines at a time. Omnis has a number of commands that make it possible to fully exploit this feature. The 'Save selection for line(s)' command saves the selection of lines to their own buffer in memory, which follows the list. If the list is saved to disk, this extra buffer will be stored together with the current line selections and the contents of the list. Among other things, the buffer is easy to use if the user wants to undo and retrieve the original selections ('Swap selected and saved.')

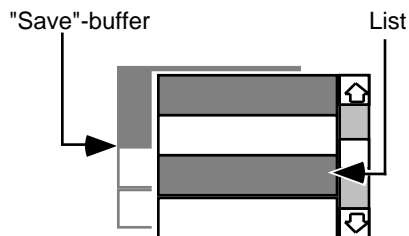


Fig. 6 A list and its “save” buffer for line selections

Logical operations with the ‘Save’ buffer

Omnis gives us a choice of commands which are meant to make logical comparisons between the “save” buffer and the selected lines in a list. However, the logic of AND, OR and XOR often escapes those who are not used to programming arithmetically with these operands. So we will take a closer look at the nature of each of these comparisons.

Figure 6 shows the list and the buffer behind it. The list itself shows at all times which lines the end-user has selected in the window in question, while the buffer behind it shows which lines were selected the last time the ‘Save selection for list line(s)’ command was executed. (The “save” buffer itself will not be affected by the ‘AND/OR/XOR selected and saved’ commands.)

AND selected and saved

The operator ‘AND’ requires that the line must have been selected both in the “save” buffer and in the list itself for the line to be selected as the command is executed. If a line is selected in the list itself only or in the “save” buffer only, the line will be left unselected.

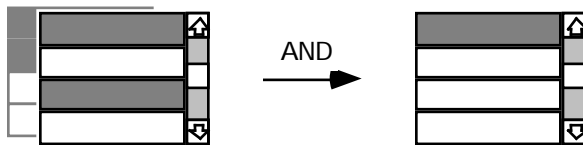


Fig. 7 AND selected and saved

OR selected and saved

The operand ‘OR’ only requires the line to be saved in the list itself or in the buffer. If the line has been selected in both places, it will remain so after the command has been carried out. In other words, this command is rather inclusive.

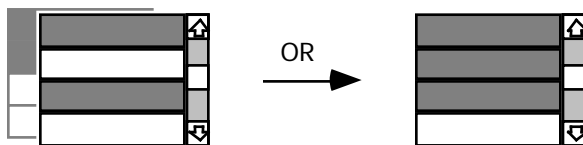


Fig. 8 OR selected and saved

XOR selected and saved

XOR stands for “Exclusive OR.” It will not select the line if it has been selected both in the list itself and in the buffer; in every other respect it behaves just like OR. This means that the line must either have been selected in the buffer or in the list itself, but not in both places. The lines in question will be have been selected when the command has been executed.

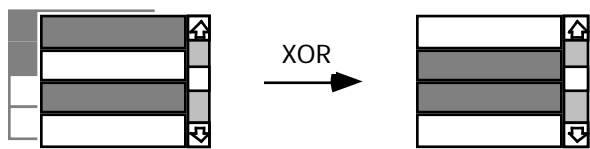


Fig. 9 XOR selected and saved

The woes of line selecting

Let’s take a closer look at a problem having to do with the selection of several lines in an ordinary List field. When a user holds down the CMND or CTRL keys, he can select lines one at a time without clearing the previous selections. However, if he clicks without holding down one of these keys, all the other selections will disappear. This can be very irritating, because making selections is fairly demanding work, and this kind of accident is not all that uncommon. We should make our own “Undo” pushbutton to use in recovering the lost group of selections.

And here to save the day...

Offhand, it might appear that all we needed to do every time we clicked on a line was to use the ‘Save selection for list lines’ command. Alas, it’s not that simple. This only works the first time the user clicks on a line. What we really want is an “Undo” pushbutton that still works after the user has clicked more than once.

The procedure for the List field	10
<pre>If #CLICK Calculate Window_list as Undo_list Set current list Window_list Save selection for line(s) (All lines)</pre>	


```

Set current list Undo_list
Restore selection for line(s) (All lines)
End If

```

The procedure for the List field

(10)

Let's imagine that we have placed a list called 'Window_List' in a window. We want to safeguard the previous set of selected lines (the last time the user clicked on a line), so that the user can undo if he inadvertently deselects all the lines. We have to save the selections every time the user clicks on a list line, to accommodate the undo function each step of the way. The main problem is that the user can't see that the damage has been done until Procedure 10 is run. Thus we have no way of keeping the new set of selections from being saved. The only solution is to use an extra list, to assist the "save" buffer.

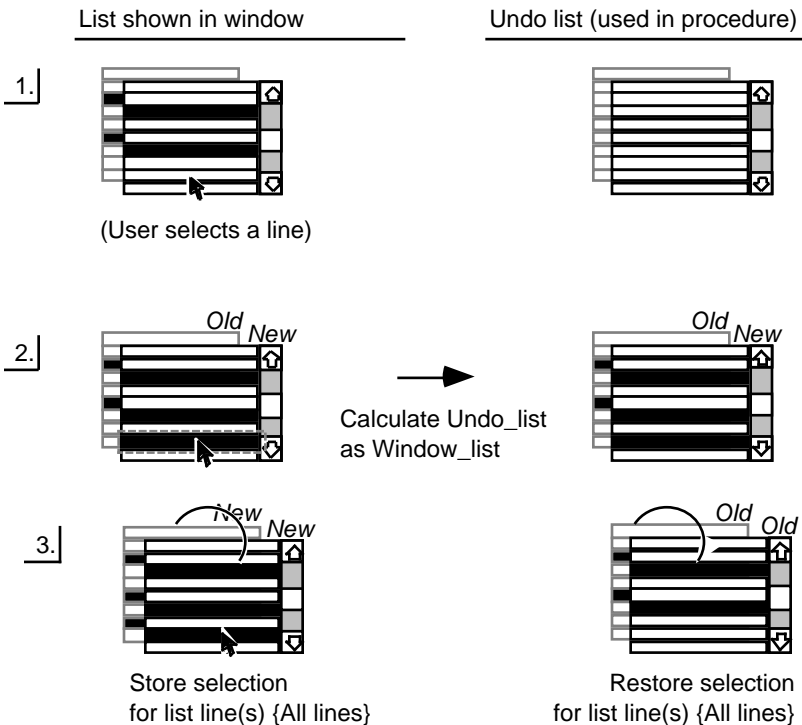


Fig.10 A diagrammatic description of what happens when the list selections are stored, as shown in Procedure 10. (The figure starts on previous page.)

Storing the previous selections

Figure 10 illustrates what happens when the user selects a line using CMND/CTRL-click. First, Window_list is copied over to another list, called Undo_list here. This is done right at the beginning of the procedure, before the current set of selections is saved to Window_list’s “save” buffer. In Undo_list we read in the contents of the “save” buffer with the ‘Restore selection for list line(s) (all lines)’ command so that it is ready to be transferred. Finally, we save the current set of selections in Window_list in the companion “save” buffer.

So we actually have three sets of selections: 1: The active one, which is always visible in Window_list and which we cannot control with procedures. 2: The “save” buffer in Window_list, which acts as an intermediate storage place. 3: The Undo_list itself, which contains the previous set of selections.

Restore selections	11
If #CLICK	
Calculate Window_list as Undo_list (Redraw field)	
End If	

Restore selections (11)

I have given my “Undo” pushbutton the name “Restore Selections.” If the user wishes to undo an inadvertent action, he can use this pushbutton. It simply copies Undo_list to Window_list, thereby restoring the previous selections. Figure 11 illustrates how this takes place.

List shown in window

Undo list (used in procedure)

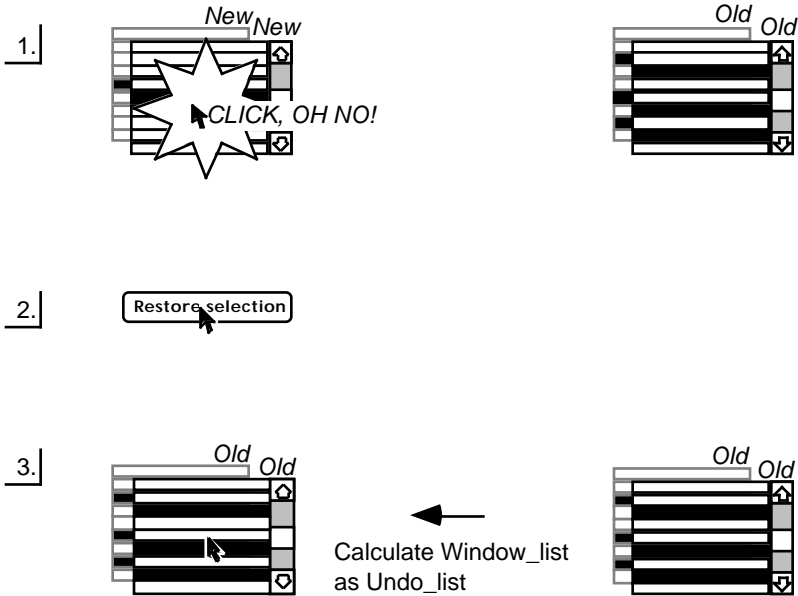


Fig. 11 Our end-user, having fallen prey to bad luck, is saved by the “Restore selections” pushbutton.

Displaying Lists in Windows

When a list has been defined and filled with data, it can be displayed in a number of interesting ways, each one meeting a specific need. It is important here to differentiate between fields and file formats on the one hand and fields in windows on the other. We call the latter “window fields”; they are used in windows. Entry fields, List fields and Display fields are all window fields. The various window fields allow the end-user to view and edit fields from the file formats.

The main purposes of List fields

Except for Combo boxes (v2.x), window fields for lists do very little with the lists they contain. Their purpose is merely to serve as a link between user and procedure. The procedures behind the List fields determine what will happen when the various fields report that the end-user has now clicked on a field, double-clicked, or selected a line, etc. Values from the list have to be fetched using commands in the procedures (for example, using ‘Load from list’). It is all in the hands of the field procedure. An important exception to this is #L, which is always set directly when the user clicks on a list line. Moreover, the end-user has control over which lines are selected, provided the ‘Show selected lines’ option has been switched on.

“Simple” List fields

The list field was the first field to show up in Omnis, and is probably still the most popular one. Both easy to learn and easy to use, it provides a good overview of the list contents. If you can’t see all the contents at once, just use the vertical or the horizontal scroll bar.

Searching manually in the List field

When the cursor is placed in a list field, we can key in the first couple of letters that appear (on screen) in the line we want. A search of the list will then begin. The first suitable line will be selected, and Omnis will scroll to it. Such searches are just the thing for lists under approximately 100 lines, especially if they haven’t been sorted. We can also move up and down the list using the arrow keys, or by using the “+” and “-” keys to find other list lines that match our search letters.

Selecting several lines

When ‘Show selected lines’ has been checked off, the user can select one or more lines. If the cursor resides in the list field (which it will if the user has clicked on the list field, for example) a continuous block of lines may be selected by holding down Shift when selecting. Holding down CMND or CTRL allows us to select one line at a time by clicking on them. If (using CMND/CTRL) you click on a line that has already been selected, this will de-select it. However, if you click on a line *without* holding down any of these keys, you will annul all the other selections, leaving the only line selected the one you just clicked on. This can be a galling experience if you do this inadvertently after painstakingly selecting, say, 10–15 lines from a list of 50!



Fig. 12 “Simple” List field

Advantages

- Can show the whole list at once with selections without forcing the user to “open” anything.
- Selections made by a single click.
- Easy for the user to understand.
- Generally faster than tables.
- Can use letters, the +/- keys, or arrow keys for searching.

Disadvantages

- Takes up a lot of room in the window.
- Redraw can take time.
- Not as advanced as tables.

Dropdown lists (Combo boxes in v1.x)

This kind of List field is used a great deal in Windows programs. It provides a number of options for choosing from the list, which makes it very flexible but somewhat harder to grasp. The fact is, there are two basic ways of selecting a line. When you move with the aid of TAB, it might appear as though you were manipulating a field in a file format. However, this box is actually a place where we can key in the first couple of letters and in this way single out a list line. The selected list line is displayed in the box. If we click on the Dropdown list field, a little window will appear that acts just like an ordinary List field. (Not multiple choice.) It will close when we select a line (double-click, ENTER or RETURN), or when we move away from the field by clicking somewhere else or using TAB.

As with the list field, we can move up and down the list using the arrow keys, or by using the “+” and “-” keys to find other list lines that match the search letters.



Fig.13 Dropdown list (“Combo box” in v1.x)

Advantages

- Takes up very little space in the window.
- Can use letters, the +/- keys, or arrow keys for searching.
- If #L=0, the field will be empty and send a clear signal to the user that the list has not yet been used.
- From v2.1 on, the number of lines in the “popup list” part can be increased by as many lines as you wish.

Disadvantages

- The list that appears may overshadow other fields the user wants to click on, making it difficult to exit the Dropdown list field.

- Selections made with the mouse are fussier.
- In versions before v2.1, the number of lines that can be displayed at the same time in the list field section is limited to 5, regardless of font style or font size.
- Cannot be used to select several list lines.

Popup lists

This window field works about the same way as a Popup menu. It has all the advantages of a menu, in that it allows quick selections and takes up less space, but isn't always immediately noticeable. If you can't proceed without selecting from a list, you should choose a window field type that takes up more space and attracts more attention. The Popup list can display more than one selected line, which it does with check marks. The selections aren't as "fleeting" as in ordinary list fields, because each line must be selected or deselected one at a time.

Popup lists and popup menus

Confusingly enough, the popup list resembles a popup menu, which reacts immediately once a line has been chosen and runs the selected menu line procedure. The Popup list only sends a #CLICK message to the procedure behind it (and all of the control procedures); it is then up to this procedure to react to the message.



Fig. 14 Popup list

Advantages

- Takes up very little space in the window.
- Mouse selections are efficient, as with menus.
- If #L=0, the field will be empty and return a clear signal to the user.

Disadvantages

- Only the system font (12 point) can be used. This type of character takes up a relatively large amount of space. The number of lines that can be shown all at once is restricted by screen size.
- It doesn't cut a very striking figure in a window.

Combo boxes

We finally got a real Combo box in Omnis v2.0, i.e. a true combination of a field and a list. It is more active than the other list fields, because it automatically copies the list value over to the designated field (in the CRB) as the user selects a line. The value thus transferred is the one that appears on screen at the beginning of the list line.

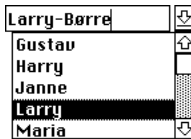


Fig.15 Combo box

Advantages

- Takes up very little space in the window.
- Can use letters, the +/- keys, or arrow keys for searching.
- If #L=0, the field will be empty and return a clear signal to the user, indicating that the list has not yet been used.
- Provides a direct link between the list values and the field receiving the value.
- The field (in the CRB) can be edited directly by the user. This means that the field can receive values that are standard selections (from the list) in addition to those that are completely new (i.e. those typed in by the user).
- The 'Entry field' part reacts like an ordinary Entry field and reacts in the normal way to such standard commands as 'Find,' 'Next,' 'Insert,' etc.

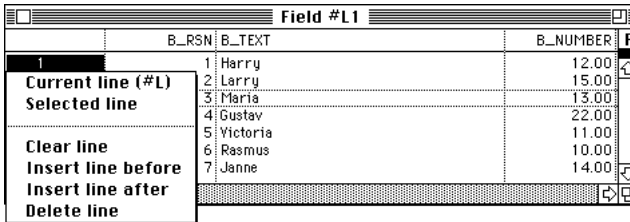
Disadvantages

- The list that appears may overshadow other fields the user may want to click on; this could make it cumbersome to exit the “list field” part of the Combo box .
- Mouse selections are fussier.
- Prior to v2.1, the number of lines that could be shown at the same time in the ‘List field’ part was limited to 5, irrespective of font or font size.

Field value window for lists

Even though this window is basically reserved for developers, it remains a very simple way of displaying a list. It is called up with the OPT/RB menu by clicking on the list name, or by using the following command:

Field menu command: Open Value Window {ListName}



	B_RSN	B_TEXT	B_NUMBER	F
1		Harry	12.00	
2		Larry	15.00	
3		Maria	13.00	
4		Gustav	22.00	
5		Victoria	11.00	
6		Rasmus	10.00	
7		Janne	14.00	

Fig. 16 The Field value window for lists

The Popup menu

In this window, the OPT/RB menu has a lot going for it. This particular Popup menu can add and remove lines, change selections and set #L (‘Current line’), all of which come in mighty handy when testing procedures.

Limitations

In v1.x, the window is only available in Single user mode.(The Design menu doesn’t have to be on screen.) In v2.x it is always accessible.

Tables

Tables provide another way of displaying lists in windows. They are extremely flexible and useful, but differ significantly from other list fields. That's why we have decided to discuss them separately at the end of this chapter.

About the calculation field

For all window fields custom-made for lists, it's important to remember to fill in the calculation field. This is where you decide which values to display, and in what order. The formatting instructions are set, as is the line orientation (left/right, distance to next value, etc.). You are perfectly free to adjust the appearance of the list lines to get them to look just the way you want. A typical example of this is the 'jst' function, a good description of which you'll find in "Reference 1," pp. 2–24 to 2–29. Most people like columns to be neatly aligned underneath each other, which makes it important to use a monospaced font in the list (otherwise the letters will not be the same width). See the chapter entitled "Layout & the User Interface."

The mechanics of the calculation

Calculations in the list field are not all that different from other forms of calculation. However, it is repeated for every list line and calls up its field values from each list line instead of from the CRB. The fields in the CRB remain untouched, so we needn't worry about them. Actually, we can insert anything here, e.g. a Boolean condition, which can be used to show (or not to show) one or more letters, depending on the list's field value. Confusing? Consider this:

```
...mid("Occupied",1,(A_AppartmentRented=1)*8)...
```

If the value in A_AppartmentRented equals 1, the value of the condition (A_AppartmentRented=1) will be 1 (YES), and 1*8 will obviously (er, uh...) be 8. Thus all eight letters in 'Occupied' will appear in this list line.

Pitfalls

These calculations can take time, especially for long lists. The same calculation is repeated for each list line every time the window is redrawn with 'Redraw windows.' Long, complicated calculations in lengthy lists can be cumbersome in the extreme, and then there may be no point in carrying them out. That's why as many of these

calculations as possible should be done in advance and the results inserted in the list as values. There is always enough room in memory to indulge little luxuries such as this, to ensure that our redraws are fleet of foot.

Suggested ways of using lists

You should be familiar with most aspects of the different kinds of list fields by now, and eager to make the most of them in your applications. Nevertheless, to speed you along, the following list should suggest some classical ways of using lists.

To survey all the records in a file

This should preferably only be done with smaller files. The List field of choice is the “simple” list field, which can show many lines at a time.

To display connected records from the file below

Clients linked to a firm is a typical example. You get a nice overview at a glance.

To present simple options to the user

If a search results in more than one record, you can use a list to enable the user to decide for himself which record to use. For example, from a list of clients with the surname “Jones,” you could choose whichever Jones you wanted.

Free combination of On/Off options

Multiple selections in a list with the aid of CMND/CTRL-click when the ‘Show selected lines’ option has been set. When many such Boolean choices are to be made, a List field (preferably a table) should prove useful.

General auxiliary buffer for complicated procedures

Lists can clean up your procedures by replacing some complicated “pointer juggling” with their own built-in functionality. For example, to safeguard the integrity of the parts of a text variable when the text

is split up, modified and reassembled, you can temporarily store them in a list.

Displaying Single List Values

Using ‘Load from list’

Values can be loaded from lists to other variables that are shown in the window. This can be done in separate procedures. The method is handy in cases where you wish to change the value and, possibly, return it to the list. Omnis can also control this automatically (see the section entitled “Automatic editing control”).

The ‘lst’ function

This function can be used to read in a single value from a list; and being a function, it can be placed anywhere in the procedures, in the calculations for most window fields, and in report fields.

```
lst(listname, line, fieldname)
```

The syntax

Here you can state the name of the field (i.e. column) you want to get a hold of, in contrast to the ‘Load from list’ command, which must be adjusted by the use of commas. If no list is specified, the current list will be used. If the line number is left out, #L will be used (the field name must always be designated).

The ‘lst’ function in procedures

This is a useful function for reading in a single value from a list, but remember: #L, #LN and #LM still reflect the current list and not the list designated in the ‘lst’ function.

The ‘lst’ function in windows

Not only display fields can benefit from the ‘lst()’ function. Text in Pushbuttons, Radio buttons, and Check boxes can all have a ‘lst()’ in square brackets ([]), providing us with a good way of “flipping” the text when we have the sort of field procedures that behave differently in different situations. The various messages can be placed in a fixed list and shown in the pushbutton text, depending on the ever-changing nature of the situation (Main file, Enter data, stages in an incremental entry of data, etc.).

The 'lst' function in reports

The advantage of using 'lst()' in report fields becomes apparent when lists stored in field formats are presented in reports. We need to have one 'lst' field for each list line that is displayed. In addition, we should test for the length of the list, so as not to end up with too many blank spaces between the records in the report. (See the chapter on reports.)

Lists Stored in Datafiles

Lists stored in data files together with their file format have their advantages and disadvantages. First and foremost, they can be read in extremely quickly, and will help save time when fixed lists are loaded. Their flaws become apparent when you try to change them, because then the whole list must be updated. The greatest limitation involves searching the contents of the list. To accomplish such a search, the developer must write a procedure that locates records in the file one at a time and searches every record in the file. This usually takes quite long. For small lists (up to an 200 lines), there is still a great advantage in being able to use lists built from indexed files; it's quick enough, and a whole lot more flexible.

What is saved with the list?

When a list is saved to disk, its value is naturally saved in the lines, but we also save all the information and settings that apply to each list, which are the following:

- The selected lines in the list itself and in the “save” buffer
- #L, #LN and #LM
- The list definition, i.e. the titles of the columns

The ‘Prepare for insert’ commands clears all these settings and deletes the contents of the list. This means that every new list in the file format is completely “clean.”

Areas of use

Most people perceive the main function of lists stored in data files to be the registration of information chronologically. This is all the more true when the content doesn't need to be analyzed or manipulated further, or when the number of registrations (or any field type) varies from record to record.

In addition, it is helpful to be able to place fixed lists (price lists, etc.) in a file format and save them on disk. Reading them will go like greased lightning. Depending on which record we locate, we get completely different lists. The field List_Calculation could contain a ‘jst(...)’ expression that would be used to format the list lines when the list is displayed in a window. If we put the expression ‘eval

(List_Calculation)’ in the List field calculation, the “formatting calculation” stored with the list (in the file format) will apply. If the list’s calculation and titles are placed in their respective text fields, we’ll have a handy file format for every kind of list.

Lists within Lists

At first sight, lists within lists can appear inordinately complicated. Actually, they aren't. The setup provides for a three-dimensional (instead of a two-dimensional) table. Imagine a pile of ordinary tables placed on top of each other. The definition, selected lines, contents, and all the other information concerning the list is stored, together with the "sublists," on each line. In other words, we can have totally different lists in each line. This resembles saving a list to disk. Defining such a "big" list is easy; just check the 'Store long data' option. It is immaterial which list is defined first – the "big" one or the "little" one.

Retrieving single values from lists within lists

The easiest way to retrieve a single value from a big list is to use the '1st' function in two stages. First we copy the sublist from the big list's line into an ordinary list, and then we fetch values from this list in the usual way. See the following:

Calculate #L2 as 1st(#L1,1,#L2)

Calculate #S5 as 1st(#L2,1,#S5)

'Load from list' works the same way as '1st' in this respect. It takes two steps to retrieve a single value here as well. Tables are the only window field type that can show lists within lists directly (see the section on tables).

Areas of use

You can use lists within lists to handle a pile of invoices (each sublist contains the invoice items) or a list of employees and (for example) the courses each of them have attended.

Redrawing Lists

You shouldn't redraw lists and tables unless it's absolutely necessary; it takes a long time and can irritate the user. You may, however, carry out any other redraws whenever you want, because they take place so quickly that the user doesn't notice anything. A rule of thumb is that you should only redraw lists (and tables) when a line has been edited, deleted, or added. This means that selections from lists shouldn't result in the redrawing of the whole window, only the fields directly affected by the selection. Any window with both fields and lists should have a "redraw" procedure that doesn't include the list, a procedure that can be called at any time and that will replace 'Redraw windows' in many cases.

Redraw named fields

If we specify two field names in this command, Omnis will carry out a selective redraw of all the fields between these two (including the specified fields themselves) in the order of the fields within the file format. This is the same order in which the fields appear within the 'Field Names' window (CMND-9), which in turn means that this window can be used to ensure that the right fields are included. The example below shows the list being defined as field number 11 within a file format. We have to use two commands to keep from including the list in the redraw.

Redraw with field names (file format field numbers)	12
Redraw named fields Field_01 to Field_10 ; The list is Field number 11 in the file format Redraw named fields Field_12 to Field_30	

Redraw with field names (12)

The advantage of the 'Redraw named field' command is that it doesn't depend on the order of the fields in the window, which can easily be changed. This kind of redraw will not be affected by a change in the window fields sequence. By the same token, it will not include particular window elements, i.e. text in Pushbuttons, Check boxes, Radio buttons and "free" background text. These elements must be redrawn using 'Redraw numbered fields' or 'Redraw windows.'

Redraw numbered fields

As you know, we can also allow for redraws with a selection of window fields. This means that the window fields in the window will be redrawn. If the same field (in the CRB) turns up in window fields not included in the range stipulated by this command, nothing will happen to the other window fields.

Redraw with window field numbers	13
Redraw numbered fields 1 to 15 ; The list is field number 16 in the window Redraw numbered fields 17 to 30	

Redraw with field numbers in the window (13)

This way of doing a selective redraw is extremely sensitive to changes in the window. If new elements are added, the developer will often have to change the order of the fields, abruptly making the redraw procedure unreliable. The advantage of ‘Redraw numbered fields’ is that it also includes calculations in square brackets throughout the window, in addition to all sorts of settings that have a bearing on the window fields (e.g. #L).

Binary Search in Lists

The ordinary ‘Search list’ command is a thorough but sluggish command. It starts from #L or from the top of the list and works downwards, line by line. Although it works, it isn’t the best way to get results. The list is usually sorted according to a specific field, which can be used to do a binary search. Binary searches are much quicker than line-by-line (linear) searches. Omnis uses this very same principle when it peruses an indexed field in the datafile.

The principle

We compare the search text with the text in the indexed (i.e. sorted) field in the list, and determine whether the search text is “larger” or “smaller.” The former means that the line with the desired value is found after the line it is now being compared to. The latter means that the desired lines are found before the line we are now searching. The comparison is in terms of ASCII or ANSI – the numerical value of the letters in the fields. We start with the line in the middle of the list. If the search text is “larger,” this tells us that the line we want is somewhere in the bottom half. If it is “smaller,” it will be in the upper half. We proceed by testing the line that is in the middle of the relevant half. The value in this line is once again compared to the search text, and we find out which half it lies in, within the area we have narrowed our search down to. We continue halving the list until we know exactly where the desired value lies.

Practical solution

The following is a general procedure that can replace ‘Search list.’ It does an automatic search of Current list. To call it, we must send values to the following parameters:

Pa_FIELD (Field name)
Pa_SEARCH_VALUE (National)

Pa_FIELD is the field we will use to search the list, which must be sorted according to this field. And the field has to be part of the list definition. Pa_SEARCH_VALUE is the search text, i.e. the value the sorting field should have. When we call the procedure, the parameters are entered in the following manner:

Call procedure pBinarySearch/1 (NAME, "George")

Binary search on current list	14
<p>Parameter Pa_FIELD (Field name) Parameter Pa_SEARCH_VALUE (National)</p> <p>Local variable LINES_START (Short number 0 dp) Local variable LINES_END (Short number 0 dp) Local variable LINES_MIDDLE (Short number 0 dp) Local variable LENGTH (Short integer (0 to 255)) Local variable VALUE (National) Local variable GOT_IT (Boolean)</p> <p>Calculate LINES_START as 0 Calculate LINES_END as #LN Calculate GOT_IT as 0 Calculate LENGTH as len(Pa_SEARCH_VALUE)</p> <p>Load from list {#LN} Calculate VALUE as mid(Pa_FIELD,1,LENGTH) If Pa_SEARCH_VALUE>VALUE Sound bell Quit procedure (flag clear) Else If Pa_SEARCH_VALUE=VALUE Calculate GOT_IT as 1 Calculate LINES_MIDDLE as #LN Else Repeat Calculate LINES_MIDDLE as LINES_START+int((LINES_END-... ...LINES_START)/2) Load from list {LINES_MIDDLE} Calculate VALUE as mid(Pa_FIELD,1,LENGTH) If VALUE>Pa_SEARCH_VALUE Calculate LINES_END as LINES_MIDDLE Else If VALUE<Pa_SEARCH_VALUE Calculate LINES_START as LINES_MIDDLE Else If VALUE=Pa_SEARCH_VALUE Calculate GOT_IT as 1 End If Until GOT_IT=1 (LINES_END-LINES_START)<=1 End If</p> <p>If GOT_IT=1 Deselect list line(s) (All lines) Select list line(s) {LINES_MIDDLE} Calculate #L as LINES_MIDDLE</p>	

```
If mid(lst(#L-1,FIELD),1,LENGTH)=Pa_SEARCH_VALUE|...  
...mid(lst(#L+1,FIELD),1,LENGTH)=Pa_SEARCH_VALUE  
    OK message {Several lines were found.}  
End If  
Quit procedure (flag set)  
Else  
    Sound bell  
    Quit procedure (flag clear)  
End If
```

But there's a catch to it –

If the search criterion fits several lines of the list, we won't be able to know for sure whether the correct line has been found. The search should then continue within the same area, but with stricter search criteria. For example, we can search linearly on each side of the initial line until we find one that is a perfect match. But this should be up to the one who will be doing the programming to decide. Searches more advanced than simple, localizational ones should be carried out on the datafile, and not in lists.

Tables

What is a table, anyway?

Tables are the most flexible way of displaying lists in windows. They are a type of window field meant for lists, just like “simple” list fields, Popup lists, Dropdown lists, etc. Tables, however, are more flexible and more powerful in use, because they can contain all sorts of window elements in their lines. This opens up a wealth of options for creating good user interfaces. Tables are one of the most exciting features of Omnis 7.

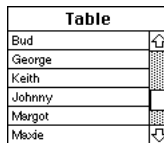


Table	
Bud	
George	
Keith	
Johnny	
Margot	
Maude	

Fig. 17 A downright boring way of using a table

How they work

The list to be shown in the table must, of course, be defined and generated, just like any other list. Unlike other kinds of list fields, the table allows you to use a sort of “window area” in each line where you can insert Display fields, Entry fields, lists, text, pushbuttons, etc. at will. This is the table line; whatever you can do elsewhere in the window you can also do here. The area is repeated below, in synch with the number of lines in the list.

Values in fields

Everything you put in a table line (during Design mode) is repeated each line down the list. Fields that are part of the list definition for the table’s list get the correct value from their respective lines. List values can also be altered directly under Enter data, provided the ‘Enterable’ option has been checked off. However, expressions set off in square brackets ([]) in bits of free text, pushbuttons, etc. retrieve their values from the CRB.

Lines

In tables, the #CLICK message variable is sent whenever the user changes lines or clicks on another field within the same line.

Happily, #LCHANGE comes to our rescue, making it clear what the user has actually done. Clicking on a different window field within the same table line will trigger a #LCHANGE, which you can take as an indication that the user has not moved to another line, just to another field. In most other respects, the table fields work just like any other list field.

Enterable

When ‘Enterable’ has not been checked off, the user can highlight only one or more line areas. When ‘Enterable’ has been checked off, the user can use the line areas as though they were a normal part of the window. This means that the values in Entry fields can be edited, pushbuttons pressed, menus used, etc.

Auto extendable

For this function to work, the line area has to contain at least one Entry field (or a field that houses the cursor. See the chapter entitled “Sequence of Procedures”). When the user presses TAB to exit the last Entry field in the bottom row, a new line is automatically inserted. That’s it. You can’t delete lines the same way; to do that, you must create your own pushbutton.

When a list is ‘Enterable’ and ‘Auto extendable,’ it is advisable to start with one line in the list instead of an empty list. It will be easier to “tab” one’s way out of the last field (and write new lines) when one already exists.

Window elements in tables

Tables will vary greatly in terms of appearance and function, depending on which elements we decide to put in the lines. We should bear this in mind, since it can lead to some pretty exotic results. Let's take a look at how the most common window elements shape the table.

Entry fields

Fields from the list definition can be inserted, making it possible to edit the list values directly. 'Entry field (Border)' gives a nice square pattern. All other aspects of Entry fields apply here as well, provided Enterable has been checked off. By holding down shift and dragging from the left edge, you can extend vertical lines with which to split the table into suitable sections or, alternatively, make it look like a spreadsheet.

Tables can be used, among other things, to edit records to be entered or updated in a child file. When the user is finished, we transfer the values in the list to the data file as previously shown in this chapter.

Display fields

When we use Display fields in tables, we can show their contents in proportional fonts (e.g. Helvetica, Garamond, etc.), and still have them in perfect vertical alignment. And we don't have to use the 'jst' function to do so either (which we must in ordinary list fields).

Check boxes

A set square pattern of Check boxes can be used as a powerful selection console. These may be combined with a list defined with Boolean fields and a fixed number of lines.

Picture fields

We are finally able to display pictures stored in lists. Just put a Picture field inside a table line.

Pushbuttons and Button areas

Complexes of pushbuttons in fixed square patterns are useful when you want to make something reminiscent of agendas and calendars. All of the pushbuttons will be copied into the lines below and are

really the same pushbuttons all the way, but you can use #L to adjust the procedures' different reactions to the different lines.

Popup menus

The menus behave normally, but in reality it is the same menu that recurs in every line.

Text and graphics

Text is copied into every line below. However, you shouldn't use calculations in square brackets to alter the appearance of different lines, because the latter derive their values from the CRB. Graphics are treated the same way, which means that different symbols and markers can be used to improve the layout of the tables. There's nothing wrong with inserting gray-tone graphics and three-dimensional effects in the tables either. And remember: the dividing lines and the border itself can have different patterns.

List fields and tables

For the first time, we can display and edit lists within lists in our windows. (The list definition of the main list must contain the sublist, and the option 'Store long data' must be checked off in the 'Define list' command.) We can insert all sorts of list fields in the table – including other tables. Unfortunately, if we want to edit the sublists' contents, we have to use (even in v3.x) ordinary list fields and a custom-made editing window. The technique isn't all that demanding; and it's useful, to boot.



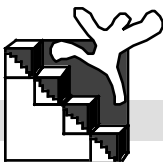
Tables eat List fields for dinner!
Program them with care and they might even
fetch your slippers for you.



Data Input

Chapters:

1. The Ins & Outs of Enter Data
2. Import & Export



Import & Export

Introduction.....	2
File Types.....	3
Standard Export Tool.....	6
Export via Reports.....	8
Exporting to Word Processors	9
Assembling the file manually	
Controlling layout in other programs by using codes	
Exporting with the space character	
Sequence Numbers	11
Importing Connected Files.....	12
Principle for independent import	
Importing hierarchically connected files	
with the help of a separate file	
Importing hierarchically connected files	
with the help of a list	
How to avoid the problem altogether	
The 'Import Field From File' and 'Import Data' commands	20
Import data	
Import field from file	
Update or Insert New Record?	22
See if the record exists	
A successful search	
Update or add record?	
To save or not to save?	

Introduction

Import and export is one of the ways in which large amounts of data may be transferred from one application to another, from one program to another, or from one type of machine to another. This kind of data exchange helps integrate and coordinate a variety of platforms. The ability to import is essential in making the transition from one database to another. Export is inherently no less important, as it provides an “exit” option.

Making the transition to other databases

It might seem paradoxical that a database should ease the transition to a “competitor,” but this provides a marked element of safety. It makes it possible to switch platforms without the necessity of writing in all the data one more time. Moreover, a database with powerful import tools will be able to handle data from almost any kind of program.

External tasks

Sometimes we will need to carry out special tasks that lie beyond the province of a database. This might involve advanced statistics, special graph plotting, or demanding desktop publishing. We can turn to other special programs and export the data to be used, or we can resort to an automatic exchange of data. Both Windows and Macintosh platforms have advanced routines in their operating systems for coping with this (DDE, OLE, Apple Events, and Publish-and-subscribe) – all of which support Omnis; we’ll come back to this in another chapter.

File Types

“Standards are so nice, we made lots of them.” — In our quest for greater efficiency (and less extra work), file standards have been an elusive but much-sought-after goal. As it happens, there are few true standards, partly because of the wealth of programs and types of documents, and partly because software manufacturers can’t agree on the rules of the game. Nevertheless, a number of simple, relatively widespread file types have turned up, and many programs are able to make use of them. The whole rationale behind common file types when importing and exporting data is based upon the fact that programs at both ends must be able to use them. The sender must be able to save the data to disk in accordance with the rules for the file type, and the receiver must be able to read them in accordance with those same rules. The structure of each file type is important, so we will discuss the various types briefly. Omnis can export and import files that are constructed according to all these file types.

Tab-delimited

For regular text and numbers, this is perhaps the most common file type of all. Fields are separated by a tab, i.e. character no. 9 in the ASCII system. The last field in the file format concludes with a carriage return character, i.e. character no. 13 in ASCII. It is usually not possible to place tabs within text fields (because this will cause the user to skip to the next field). However, it is theoretically possible to add tabs by means of calculations. Omnis purges these characters from the fields during export. If this were not the case, these characters would be mistaken for delimiters; the result would be incorrect separation of fields and total confusion.

The content of the text fields is enclosed in quotation marks (" ") if the ‘Enclose exported text in quotes’ option in the ‘Preferences’ window is checked off. Export to word processing programs usually takes place via a tab-delimited file type.

Comma-delimited

The comma-delimited file type is similar to the tab-delimited one, except that its fields are separated by a comma instead of a tab.

SYLK

SYLK is a special format for spreadsheet programs. It includes not only data but coordinates, i.e. information concerning field placement in the spreadsheet's coordinate system of cells. This is how we avoid the proliferation of delimiters. Omnis arranges the contents of the fields in rather staid columns, with records aligned vertically. All carriage returns and tabs are deleted in the text fields when they are exported.

Omnis Data Transfer

The 'ODT' file type is designed specifically for the exchange of data between different applications or platforms that run Omnis. Here is where all information concerning field type, field name, etc. is kept. With this file type there is virtually no data or formatting loss, hence it is the safest. Omnis Data Exchange format is the only file type that can export images.

Data Interchange Format (DIF)

The 'DIF' file type can be read by a large number of applications. Information contained in the field content includes: names of the programs from which the data originates, the number of fields in the file format, the total number of records, and the field names themselves. The control character is replaced by a space character.

dBase

Using this file type, dBase-compatible applications can read the data directly. Even though dBase is a somewhat dated programming language, there are still many databases that use this file format.

Lotus

The file type of one of the world's bestselling spreadsheets for PCs has also been included; it can be read directly by all Lotus-compatible programs.

One Field per line

All the fields are exported with a carriage return between them, resulting in a long "ribbon" of data. No character is inserted that would tell us when the next record begins in the export file. The user himself must keep track of how many fields there are in the file format, to avoid phase displacement during import. This is the only type of export that we have gotten the 'Import field from file' command to work with very well.

Standard Export Tool

This tool becomes available when we choose ‘Export data’ from the ‘Utilities’ menu. For uncomplicated data exchanges it is more than equal to the task. Fields can easily be retrieved by using the ‘Load fields from file format’ menu option. The fields appear in the order of definition, and the RSN fields for any hierarchically connected files appear automatically.

Index

The ‘Index field’ is the field that determines the order in which the records are sent out. Before export, the records are sorted alphabetically by field content and then reconstituted as the export file.

Export file

The name of the export file should be as informative as you can make it, because export files have a way of proliferating. Use the file format’s name (abbreviate, if necessary), together with its file type. You may also provide common information about small files by placing these in the same folder and imparting the information to the folder’s name. Windows users are unfortunately restricted to an 8-character name + file type labeling (extension), separated by a period.

Abbreviations

Below are some suggestions for how to abbreviate different file types when exporting data from Omnis.

File type	Abbreviation
Tab-delimited	(<i>TAB</i>)
Comma-delimited	(<i>CMA</i>)
SYLK.....	(<i>SYL</i>)
Omnis Data Transfer	ODT
Data Interchange Format	DIF
dBase	DBF

Lotus..... WKS
One Field Per line..... (*IPL, OPL*)

The parenthetical abbreviations are my own suggestions. The rest are the actual file type labels that occur in Windows and MS-DOS. (One benefit for windows users is that Omnis suggests the extension before export.) Although it is conventional to let tab- and comma-delimited file types, as well as ‘One Field Per line,’ have the extension TXT, I don’t feel that this serves our purpose very well. Be that as it may, we’ll have to know the difference between them in order to complete a successful import.

Organization

It’s not a bad idea to place export files from the same datafile and same export time in the same folder or directory. This will simplify the treatment of files, and just might save an already befuddled developer from a veritable copying nightmare.

Selection

If you wish to do a selective export, you may ask Omnis to utilize an appropriate search. Check off ‘Select using search’ before the export. Of course, this means you must take care to choose the right search in the first place, before the export commences. The records are located according to the search criteria, sorted, and finally poured into the export file.

Export via Reports

The powerful resources we have at our disposal for presenting data in reports can also be employed to transfer large amounts of information to other programs. Reports provide a good opportunity for making selections from as many files as you want, and for exporting the fields in the exact order you want. (The fields are exported in the same sequence as they appear in the record section: from left to right, line by line.) Not only that, but debugging is a breeze; all you need to do is swap destinations between file and screen.

When you check off the 'Print as export format' option in the 'Report parameters' window, the dialog box where you select the file type for export will appear. After file type has been selected, the destination for the report must be sent to file ('Report destination' >> 'Send to file'). Then the dialog box for determining the name of the export (actually print) file appears. This file is retained as destination until another destination is selected. The file itself is not released for use in other programs until another destination or print file is selected.

Exporting to Word Processors

Since most users prefer proportional fonts when working with word processing, the chief aim of such an export is to get exported text separated with tabs and each line to end with ‘New line.’ You can do this handily with standard export tools. (If the file type uses tab-delimited text, then ‘Enclose exported text in Quotes’ should not be checked off.) The resulting text file can be read by every word processor.

Assembling the file manually

You obtain maximum control by assembling the ‘Print file’ yourself. The Report destination is set to ‘File.’ This enables us to include the column titles in a convenient way, and we can clearly see everything that is happening. By deciding directly what will be placed in an export file, we are in control of everything that has to do with the file. The entire weight of the procedure language in Omnis is at our disposal, with no information being hidden away in report formats.

Export to text file (tab-delimited)

```
Prompt for print file      ;; This creates an export/print file

; Write column titles to file:
Transmit text to print file (Add newline) {#S1[chr(9)]#S2[chr(9)]#S3}

; Write values to field, separated by TAB (chr 9):
Transmit text to print file (Add newline)...
...[{con(#S1,chr(9),#S2,chr(9), #S3,chr(9))}]

; Close export file:
Close print file
```

1

Exporting to text file

(1)

First we generate a print file and place the column titles there. In this case this will be #S1, #S2 and #S3. All text that is placed in the command ‘Transmit text to print file’ will be taken literally and “dumped,”

unceremoniously, into the file. All field values and calculations must therefore be set off in square brackets ([]) so that the calculations will be performed and the result inserted before the text is dumped to disk. This is repeated for each record to be exported (for example, within a 'For...Next' loop). Finally, we close the file.

Controlling layout in other programs by using codes

If we wish to retain a layout from a report, this requires that we know which special character codes the word processor uses for indicating boldface style, italics, ruler settings, etc. Full-featured DTP programs and word processors usually provide a list of these control characters, and they vary greatly. If we know the codes for styles, fonts, page breaks, etc., these can be inserted together with the field values. It would be natural to use the principle as shown in Procedure 1. This opens the door to highly advanced database publishing.

Why this is rarely advisable

As a rule, this is a job for hackers, not mere mortals such as ourselves. Inserting codes in all the right places can be a daunting task, and can really only be justified when exporting frequently to word processors. The report generator in Omnis is powerful enough to meet most needs. In a pinch, you can export raw text with personalized letter codes at the head of each paragraph. Later you can do an automatic search within your desktop publishing program (or word processor) to locate the codes, add appropriate styles, and then delete the codes themselves.

Exporting with the space character

When a report is sent to the clipboard, the data being exported contains a number of space characters between the fields. The same thing happens when the report is sent to a Print file. As mentioned earlier, the resulting text is unsuitable for use with proportional fonts in word processing, and this kind of field placement is therefore a thing of the past.

Sequence Numbers

Sequence numbers can pose problems when data is being exchanged between different databases. During export, the RSN is stored unchanged; but during import, a conflict arises. Records to be added are always assigned new sequence numbers by Omnis, otherwise several records might wind up with the same number. This means that the RSN fields are assigned new numbers, and this spoils things for those who use RSNs as invoice numbers or other 'key field' tasks. If the user should ever have to import a hierarchically connected file (child file), the connections will also be completely different.

Importing Connected Files

When connected files are imported, the parent file should always go first, so that the connection can take effect as the child file is being imported. However, a number of requirements must be met before the right connections can be made:

- The parent file being exported must have consecutive sequence numbers and must be exported in its entirety. (If a record is deleted, the RSN sequence will be broken.)
- The datafile receiving the records in the parent file must be empty, so that RSN begins with 1.

The requirements are not met

In practice, some of these requirements will often *not* be met, which is only to be expected. After all, we rarely have datafiles in which no records whatsoever are deleted. Moreover, the receptor datafile is rarely ever empty. So we have to find a way to tackle this.

Principle for independent import

The developer must take care to store the old sequence numbers (of the parent file to be imported) in a safe place together with the new sequence numbers that will be assigned during import. The sequence numbers can be stored in a list or in a file format connected to a separate datafile.

Importing the child file

When the soon-to-be child records are read in one by one from the import file, they take with them the old sequence numbers of the parent file (i.e. old foreign keys). This will probably not match the parent records they are connected to after the import, because the sequence numbers in the parent records (i.e. the key field) were displaced when *they* were imported.

Locating the “true” parent record

Fortunately, we have stored the old sequence numbers from the parent file together with the new displaced ones, either in a separate list or in a separate file. We need these to “clean house.” With the aid of the *old* sequence number, we can locate the parent record’s *new* sequence number. Then we execute a ‘Find’ (on the parent file) with the *new* sequence number in order to locate the *original* parent record. Then all that remains is to update – and finally establish – a successful connection between the correct imported parent record and the imported child record.

Importing hierarchically connected files with the aid of a separate file

It’s not all that easy to “catch” the point of the description in the previous paragraph – at least, not right off the bat. So let’s go through the entire process, as if we were following a cookbook. In the example below, a file is used as a buffer for old and new sequence numbers, mostly because it is so simple to search in a file. Further, we connect this buffer to a separate, detached datafile. The requisite procedures will be displayed as they are needed.

Fields used in the example:

fParent	fChild	fImportbuffer	Comment
P_RSN numbers (...etc)	C_RSN (...etc)	I_RSN I_PA_RSN_OLD I_PA_RSN_NEW	Sequence Old fParent RSN New fParent RSN

I. Create a file format with its own datafile as an import buffer

Define a new file format that will retain old and new RSNs from the parent file to be imported. In our example we will call the file ‘fImportbuffer.’ We define two Long integer fields: I_PA_RSN_OLD and I_PA_RSN_NEW. The former should be indexed. Subsequently we run Procedure 2. Here we create a datafile called ‘IMPORTBUFFER.DATA,’ which is all tied down to fImportbuffer by the ‘Set default data file’ command.

Create IMPORTBUFFER datafile

Local variable LO_CDATA (Character)

Calculate LO_CDATA as \$cdata().\$name

Create data file (Do not close other data) {IMPORTBUFFER.DATA}

Set default data file {flimportbuffer}

If len(LO_CDATA)>0

 Set current data file {[LO_CDATA]}

End if

Why a separate datafile?

One of the reasons for using a separate datafile is that it's extremely easy to delete an entire datafile when the import is finished. (Just chuck it in the wastebasket (Macintosh), or select 'Delete file' from within the File Manager (Windows).) We don't have to wait for any reorganization. However, we do have to see to it that Current datafile has not been altered when the procedure is finished running. We use a local variable, LO_CDATA, to remember its name, and employ the 'Set current data file' command with the local variable in square brackets at the end of the procedure.

II. Create import windows

Create an import window for each and every file to be imported. It's easiest to use 'Make >> Window...' in the 'Design' menu. An import window tells Omnis what the field order is in the import file. The window must be visible during import. The field order of visible 'Entry fields' in the window determines how the data will be loaded during import. Be sure to include all the fields in the file format, in the order in which the data was exported. In this example the windows are called 'vImportParent' and 'vImportChild.'

Note: When a window is automatically built up from a file format, neither list field nor binary field is included! The user must add these himself, adjusting the field order in the window afterward.

III. Import the parent file

Import the parent file with the aid of Procedure 3, shown below. Here we copy the old sequence number (PA_RSN) over to I_PA_RSN_OLD before the record is added to the parent file. When the parent file has been updated, we'll then know the new sequence number and can copy the content in P_RSN over to I_PA_RSN_NEW. Finally, the content of fImportbuffer (I_PA_RSN_OLD and I_PA_RSN_NEW) is saved with the aid of the 'Insert with Current values' command. This is repeated until 'Import data' yields 'flag false,' and the loop is completed. In this example, we utilize 'Omnis data transfer' as a file type, but there is nothing wrong with using another file type.

Import parent file	3
<pre>Begin reversible block Open window vImportParent End reversible block ;; As the procedure ends, the window closes. Prompt for import file If flag false Quit procedure End If Prepare for import from file {Omnis data transfer} If flag false OK message {Wrong format in Import file. Please use an ODT file.} Quit procedure End If Repeat Set main file {fParent} Prepare for insert Import data ;; A record is loaded from Import file to CRB. If flag false Cancel prepare for update Else Calculate I_PA_RSN_OLD as P_RSN ;; Remember old P_RSN Update files ;; Update fParent, new P_RSN is generated Calculate I_PA_RSN_NEW as P_RSN ;; Remember new P_RSN Set main file {fImportbuffer} Set read/write files {fImportbuffer} Prepare for Insert with current values</pre>	

```

        Update files      ;; Save RSN's in flmportbuffer
        Set read only files {flmportbuffer}
    End If
Until flag false

End import
Close import file

```

Technical comments

The updating of flmportbuffer is a bit special. To keep 'Update files' from updating both files, flmportbuffer is kept in Read Only until it's time for a new record to be added. The simultaneous updating becomes a problem from the second time the loop is run, because by then I_PA_RSN_OLD will have received a new value when the loop encounters the first 'Update files' (shown in boldface type inside the loop in Procedure 3).

Since fParent is kept in Read/Write mode the whole time, both files are updated at the next 'Update files' command. For fParent, however, this doesn't matter, because none of the fields in fParent have changed content from the first to the second 'Update files' command. (Naturally, we could have treated fParent just as we do flmportbuffer with respect to Read Only or Read/Write status, but this isn't strictly necessary.)

IV. Import the child file

When the records in the parent file have been imported and the relationship (flmportbuffer) has been established between the old and the new P_RSN, the child file may be imported. As the records are loaded, the child file's copy of P_RSN (i.e. the foreign key) follows. But now that the parent records have new sequence numbers, we have to turn to flmportbuffer to find out what has become of them. With the aid of I_PA_RSN_NEW, we find the correct parent record in fParent once again, and the right connection between child record and parent record can be established under 'Update files.' This is illustrated in Procedure 4.

Import child file

```

Begin reversible block
  Open window vImportChild
End reversible block

Prompt for import file
If flag false
  Quit procedure
End If

Prepare for import from file {Omnis data transfer}
Set main file {fChild}

Repeat
  Prepare for insert
  Import data
  Redraw windows (All windows) ;; Displays the imported data.
  If flag false
    Cancel prepare for update
  Else
    Single file find on I_PA_RSN_OLD {P_RSN} ;; Locates P_RSN in
    Single file find on P_RSN {I_PA_RSN_NEW} ;; Retrieves the
parent                                     record itself, using
I_PA_RSN_NEW.
    Update files
  End If
Until flag false

End import
Close import file

```

V. “Clean up”

When the child file has been imported and the correct parent records have been successfully linked, the datafile IMPORTBUFFER.DATA may be deleted. However, this is not a “must.” When the datafile was generated in Procedure 3, it wound up in the same directory or folder as the Omnis 7 program. There was nothing to keep us from placing it elsewhere, but this would have meant designating a correct path in the ‘Create datafile’ command. In fact, the difference is only cosmetic. It doesn’t really matter whether the end-user deletes the file or not. If a datafile with the same name and same directory or folder already exists when a new one is created, the old one will simply be deleted. If you’re a bit of a perfectionist, however, you

may run Procedure 5, which makes use of Omnis extension 'DeleteFile.'

Delete IMPORTBUFFER.DATA datafile
DeleteFile ("IMPORTBUFFER.DATA") with return value #1 If #1<>0 OK message {The file was not deleted} End If

5

Deleting the datafile IMPORTBUFFER.DATA

(5)

When using the external routine 'DeleteFile,' it is absolutely essential that the correct path and the correct file name be set, otherwise the wrong file could be deleted. In Procedure 4 we used the same directory as the Omnis program, i.e. we didn't specify any particular path, sparing ourselves the effort. There is no overriding reason why we should tuck it away in a folder, because it is deleted in Procedure 5 anyway. During import, the end-user is spared from having to see an extra file here – if he's the type that can't be swayed from doing other things with the computer while it is doing an import. He still won't be able to delete the file, because it will be "busy."

Importing hierarchically connected files with the help of a list

If we want to avoid having to reorganize a datafile or avoid manipulating datafiles, a list in the internal memory is the way to go. Since the list is not stored on disk, we don't have to work with the hard disk. However, the disadvantages of this approach are painfully apparent when the power fails or your system crashes. Instead of searching in a file, we use 'Search list' (or possibly a binary search). See the chapter entitled "Lists and Tables."

How to avoid the problem altogether

The *preventive* solution to the problem is to utilize a relational join instead of a hierarchical connection. All the connections will remain intact after the import, since they are not dependent on the RSN. All files that are connected using relational joins can be imported freely, and it doesn't matter which one is imported first.

Finding a valid key field

If the parent file contains another field that is unique, this field can be used to restore the link. The value of the parent key field (or the concatenated contents of several) can be stored in one of the fields of the child file, which gives us a connection as good as any.

The ‘Import Field From File’ and ‘Import Data’ commands

Both ‘Import data’ and ‘Import field from file’ can lead to confusion. Let us take a closer look at them.

Import data

The ‘Import data’ command utilizes the fields placed in visible Entry fields in the forward-most window as a blueprint for how to interpret the stream of data flowing in during import. The order of the fields in the window determines how the data is distributed among the different fields in the CRB. Extraneous fields will be disregarded or set to Empty, depending on whether you have specified too few or too many fields in the window. (Omnis sees to it that the “framing” in the file is correct.)

The command only transfers the values from the import file to the CRB. (This can be compared with the process by which the user enters data.) Consequently, the developer himself must keep tabs on ‘Prepare for insert’ and ‘Update files.’ (But that’s not all that hard, is it?)

Concerning window formats generated automatically

Windows that are created automatically with ‘Make» Window...’ are particularly apt as import windows. But before they can be used, a couple of modifications need to be made:

- Only Entry fields and Picture fields are taken into consideration by Omnis.
- Boolean fields appear as Check boxes, which will have to be changed to Entry fields.
- The field representing the RSN appears in a Display field. To avoid “field displacement,” the field that shows the sequence number must be changed from Display field to Entry field.

- The order is usually correct at the outset, and Omnis only looks for Entry fields and Picture Fields. It doesn't matter if there are other kinds of fields (e.g. Pushbuttons) between Entry fields, because they will be disregarded anyway. In fact, you can just as well forget about Pushbuttons when you create the import window.
- It's a good idea to delete the Window Control Procedure for this window. Even though the control procedure is relatively benign in such windows, this will be one less thing to think about when debugging. In any case, you won't be needing it here.

Import field from file

This command takes one “line” – raw and untreated – and places it in the designated field. A line for this command is *not* restricted by character 13 (carriage return), as is the case with tab-delimited file types. This means that virtually every kind of character in the file can be loaded, including formatting characters and index characters. While this command may not be all that useful for most developers, it could prove invaluable for the most advanced ones. For example, it can be used to locate a specific point in the file from which to start the import. A related command, ‘Import field from port,’ can be used to monitor a port so that a specific value triggers an import or another procedure.

If we have an import file of the type ‘One Field per line,’ we can use ‘Import field from file.’ But then we must know how many fields there are in the file format, otherwise we get a phase shift, and the imported data is corrupted. The advantage of the ‘Import field from file’ command is that it contains the ‘Leave in buffer’ option. If this option is activated, the import values will be retained in the import buffer. When we’ve made sure that the imported record doesn’t already exist in the datafile and are ready to insert it into the CRB, we can retrieve it directly from the internal import buffer once again. We do this by running the ‘Import field from file’ command once more, this time without activating ‘Leave in buffer.’ This technique keeps us from having to create our own buffer.

Update or Insert New Record?

Import is appropriate when we wish to gather information from several different datafiles and combine them into one big datafile. There is often a need for this – for example, in major investigations involving data entered in various offices at separate geographical locations. With repeated import (especially of connected files), we often find that the same records keep turning up. We don't want to place these in the datafile as new records. Unfortunately, the commands we will be using to mitigate this (e.g. 'Find'), can all too easily delete what we import from memory. How do we get around this?

See if the record exists

After obtaining the data with the help of 'Import data,' we run a search on the file with an indexed field for which the content should be unique. With certain reservations, this could be the name of a customer, for example. If the search is unsuccessful, this tells us that the record does not exist; consequently it can be inserted as new. The CRB for 'Main file,' however, will have been deleted, because the search was unsuccessful. (The content of the import buffer is also emptied when we use the 'Import data' command)

A successful search

If the search is successful, we are suddenly left with the record from our own datafile, and the imported values have vanished from the CRB. If we try to retrieve the import values anew, the next "record frame" in the import file appears. In multi-user mode we run into further trouble: When we use the 'Prepare for edit' command, Omnis reloads the record from the datafile, which means that what we are trying to import disappears. To avoid all this, we use a list as a buffer before the search is run.

Update or add record?

If the datafile already contains a record like the one to be imported, the user may want to consider whether to update the record or insert it as a new one. Records that otherwise seem identical might actually represent different objects in real life (e.g. lots of people sharing the same name). The user should be provided with all available information so that he can make an intelligent decision as to the true identity of the imported record.

Is the identification good enough?

If the sample material contains lots of seemingly identical records, the fault might lie with the identification scheme. In all good investigations, the identification of objects is reliable, which means we avoid the above-mentioned problem. If the indexation is truly to be trusted, we don't need to burden the user with the task of determining the true identity of the record to be imported.

Import: Insert or edit?

6

```
Set current list #L1
Define list {fPersons}      ;; All the fields in the file format
Add line to list            ;; Replace line in list requires at least one line
in the list                  to work properly.

Prompt for import file
Set main file {fPersons}
Open window iwPersons      ;; Import window
Prepare for import from file {Omnis data transfer}

Repeat
  Import data
  If flag false ;; No more records to be imported.
    Cancel prepare for update
    Break to end of loop
  End If
  Redraw windows

  Replace line in list {1}  ;; Field values of fPersons are stored in the
list.

  Find on P_Name_DateBirth (Exact match)  ;; Must be indexed and
```

```

unique.
  If flag false ;; Record does not exist in datafile.
    Prepare for insert
    Load from list {1} ;; Restore imported data from list.
    Update files
  Else ;; Record probably exists in datafile.
    Yes/No message {Update record [P_Name_DateBirth], address
    If flag true
      Prepare for edit
    Else
      Prepare for insert ;; Insert as new record.
    End If
    Load from list {1}
    Update files
  End If
Until break

End import
Close import file
Close top window ;; Close import window.

```

The procedure “Import: Insert or Edit?”

(6)

In this procedure, the file fPersons is used; the key field (P_Name_DateBirth) is composed of the person’s name and birthdate. Thus we can be reasonably sure that the field is truly identifying. (Remember that the records must in some way be identified on the basis of the values in the import file.)

To find out whether the record already exists or not, we do a search with the imported value of the key field. If the search is unsuccessful, a ‘Prepare for insert’ command is executed, after which the import values are retrieved from the list. If the search is successful, this means that the record is to be updated. We execute a ‘Prepare for edit’ command and read in values from the list accordingly.

To save or not to save?

In doubtful cases, the decision is the end-user’s, on the basis of the values in the remaining fields that are imported. In Procedure 6 we use a ‘Yes/No message.’ No matter what the user chooses, the

imported record will be saved some way or another, either as a new record or as an updated version of an old one. Those who preffer the option of not saving the imported record at all must have a separate window showing both the imported record and the record to be updated, and providing Pushbuttons such as 'Insert new,' 'Update existing,' and 'Do not store.' In any case, such a window would offer a lot more than our feeble, puny Yes/No message box.



Import and Export are the freeway into and out
of your application.



Section 6:

Data Output



Search & Find

Introduction.....	2
Searching Within a Single File.....	4
Locating a specific record	
The relationship between indexes	
Find tables	
Find	
Next	
Unsuccessful searches	
Searching in Connected Files	19
Key fields and foreign keys	
Hierarchical connections and relational joins	
Find parent record	
Find connected records	
One-to-One	
Many-to-One	
Many-to-Many	
Searches Spanning Several Generations.....	37
Find Grandparent	
Find grandchild	
Search Formats.....	44
Prompt for search format	
Flexible search formats	
Search formats versus 'Set search as calculation'	
Creating search formats with notation	
Speed Tests	53
The test method	
Simple searches on an identified record	
Field larger than a certain value	
Two fields, each with their respective values	
Conclusion	

Introduction

Searching, a sensitive subject, is the most important thing that a database can do. Being able to locate specific input is *the* main reason for taking the bother involved in keying in large quantities of text and numbers. Had that not been the case, it would have been simpler to store all the data on reams of index cards and file them away in a card catalogue. Good search tools separate good applications from poor ones. Not only do they provide the basis for analysis and presentation, they are the single most effective means of “breathing life” into a mass of data – that is, making it flexible, available, and dynamic.

Don’t panic!

The annoyingly prominent status of searches is at odds with the sad fact that they aren’t always that easy to execute. You are quickly drawn into general database theory, where you are required to think abstractly on many levels. It’s easy to be intimidated by a complicated file structure, especially if you’re trying to understand how all the components of the system work together at the same time. Fortunately, you don’t have to. We have only to look at a couple of file formats at a time and ignore the others. We’re also going to try to learn something about indexes and how to use them. Getting to know the separate parts well should put you fully in control. You’ll gain a better understanding of the whole process as you gradually become used to the different forms of file connections.



If there’s one thing I’m sure about, it’s that I’m sure that
I’m unsure
whether I’m in doubt about whether I’m sure.
— And that’s for sure!



The mysteries of speed

Quick searches are something we all strive for. Some even try to become experts in this area. When databases are compared, it's search speeds that are measured, just as horsepower and optimal velocity are standards of measurement where sports cars are concerned. Persistent developers sometimes use unorthodox methods in their quest for rapid searches. This often results, however, in procedures with mysterious contents that are difficult to decipher. These procedures are usually spun around the paradoxical effects of various bugs and eccentricities in Omnis' native code. I've decided not to attempt to follow these nitpickers. The goal of this chapter is to ensure steady progress based on simple principles. The conventional procedures are more than sufficient in everyday life.

SQL and Omnis 7 v.3.0

In line with current trends, Blyth is moving towards client/server solutions for Omnis 7. It is becoming increasingly common to use SQL as the main language and to have a smaller machine connected to a large one, which can run the most powerful database engines. This is what is known as "frontend" and "backend." To guide developers in this direction, Blyth has used SQL expressions for virtually every type of data access in the v3.0 manuals. The description of Omnis' own database engine has been relegated to a tiny appendix. I hope that this chapter will prove useful to those who still wish to use 'Find tables' and 'Next/Previous.'

Combination SQL and Find tables

Even in larger configurations, there are always a number of small files that might as well be saved locally on the frontend machine. Here Omnis' own database engine could be used to access the data. You can compare the two engines with a train (SQL) and a car (Omnis' core code), respectively. The car is large enough and handy enough to use for short trips to the store, but you need a train to transport large quantities

of goods (data). A good developer should how to choose the mode of transportation that is most appropriate.

Searching Within a Single File

Locating a specific record

You will often need to find a single, specific record in a file. To do this, you need to know a field value that is unique for this record. If the file is an address list, you can try the person's name. But because of the commonness of certain names, this is not a sure-fire method. If you decide to use the name, include the birthdate if you want to be fairly sure of finding the right record. The search will then compare the name field (as well as the birthdate) with a certain value. These two constitute a search with a complex comparison, which shouldn't be necessary when all you want is to find the record you have in mind. We'll take a closer look at complex searches later on. Right now, you want to carry out a simple search, which means you need a unique, identifying piece of information.

ID (Social Security) numbers

If we had all had completely different names, we wouldn't need identification (or Social Security) numbers. However, the world isn't that simple, which means (in most European countries, at least) that everyone is assigned a number at birth. Most people are - justifiably - unwilling to give out their number, knowing full well the potential for abuse. This is why it is mostly banking, welfare and health institutions that can count on receiving this information. Even they have problems obtaining ID numbers, either because people forget it, or because they are unwilling to give it out.

Personal ID numbers

This difficulty with ID numbers means that all kinds of clubs, organizations and businesses often create their own ID number codes. As long as the numbers are mainly for internal use, this is no problem. However, making the client, member, etc. remember his or her membership number in order to be recognized by the database is an example of bad programming on the developer's part. In principle, such numbers should

only be used to make simple searches as quick as possible in the different procedures of the application. Ordinary customers should be identified with information that the customer remembers at the moment in question and which they are not afraid to give out. Examples of this type of non-sensitive information are the customer's name and perhaps their address or telephone number (instead of their birthdate).

RSN unique codes

The developer can choose to set up his or her own unique numbers or codes, or use Omnis' built-in system: Record Sequence Number. The latter doesn't require any extra programming and is easy to use. Furthermore, you can create your own code based on the information available. For example, you can add an 'M' if the person is male and 'F' if the person is female. Codes with numbers and letters can thus contain more information than mere numbers. In addition, codes are easier to remember than numbers (if remembering numbers should ever be necessary). Naturally, the meaning of each letter in the codes must be jotted down somewhere in case you forget them.

Locating a record

When you actually know which item of information you wish to find, it is a simple matter to search the file in question. For every successful search, one or more conditions are invoked which must be satisfied. We call this set of conditions our "search criteria." Simple searches have only one search criterion, i.e. they only look at one field. This criterion may be 'C_RSN=12', for example. Provided that this field is uniquely indexed, we don't need any other information to locate this specific record. If we only want to look at the values from one file, we use the following command:

Single file find on C_RSN {12} (Procedure line 1)

Here, the Record Sequence Number value in the record we're looking for is 12. The field representing the RSN

is called C_RSN. ‘Single file find’ runs a search in the file format (to which the field belongs) and only reads in data from this file. This makes the command independent of Main file, which can then be used without any further ado.

Single file find on C_RSN

(Proc.I. 2)

When no comparison value has been entered in the command, Omnis use the current value of the field being searched. This can be seen in Proc.I. 2 (“Proc.I.” is the abbreviation for “Procedure line”). Unless we know that the C_RSN field contains the right number, this is a handy way to begin a search.

Single file find on C_RSN (Exact match)

(Proc.I. 3)

‘Exact match’ signifies that the search should fail unless a record is found that proves to be an exact match. Proc.I. 3 shows an ‘Exact match’ search. Provided this option is *not selected*, the search will accept similar records if the one we wanted doesn’t exist. This is why we should select ‘Exact match’ whenever we wish to find a specific record, or we may risk continuing the procedure with a record that is not the one we want.

Find on C_RSN (Exact match)

(Proc.I. 4)

‘Find’ works the same way as ‘Single file find.’ It automatically calls up every connected parent record (one in every connected file) as soon as it has found the appropriate child record. This applies to every file linked by means of hierarchical file connections, i.e. standard Omnis file connections.

The relationship between indexes

In a file format, we usually set up an index for every field we might want to use in a search or a sort. An index shows the order of the records in the file when they have been sorted alphabetically (alphanumerically) according to the field in question. (Read the chapter entitled “Data Structure in Memory and on Disk” for more

information on indexes.) Among other things, indexes are used by such commands as 'Find,' 'Next' and 'Previous.' Let's take a look at a file format and see how these commands work here.

Find on RSN_FELT (Exact match) (Proc.I. 5)

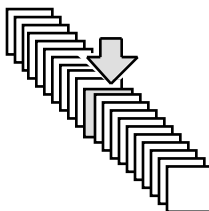


Fig. 1 "The pointer" has found a record.

Figure 1 shows us what the 'Find' command in Proc.I. 5 does. We have called the field that contains the Record Sequence Number RSN_FIELD. The value of RSN_FIELD is used as a basis of comparison, and the corresponding record is read into the CRB.

The pointer

'Find' always starts the search from the beginning of the file. Let's imagine that when a record is found, a pointer moves to this record and forms the starting-point for the direction of subsequent 'Next' or 'Previous' commands. The pointer knows which index it looks through on its way down, but this direction can be easily changed later. This is a simple search through a specific index. We can see that the records are arranged neatly and orderly. Once a record has been found, we can continue up or down the same index (Fig. 2) with the aid of 'Next' or 'Previous.' This moves us one record at a time.

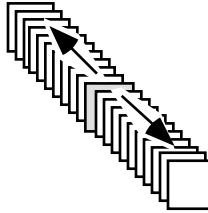


Fig. 2 We can move up and down the index.

Several indexes

On closer inspection, we see that this file contains several indexes, each crossing the other. The record we found has indexes in many directions, as shown in Figure 3.

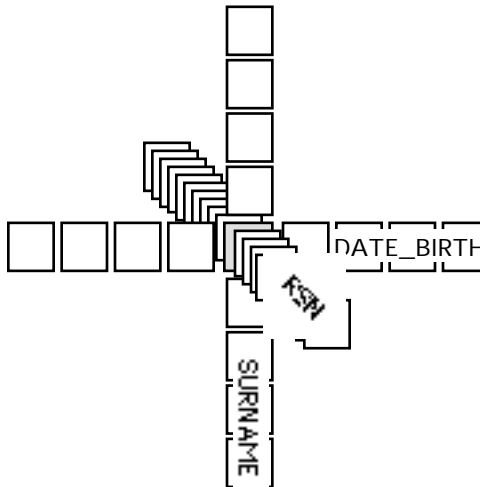


Fig. 3 Indexes that stretch in every direction

Changing indexes

The record is placed in each index, depending on where the field value of the corresponding fields appears in the alphabet. Thus a record can be in front of one index and in back of another, depending on the contents of the the records' fields. If we want to “change tracks,” as it were, we can use ‘Next’ or ‘Previous’ along with the name of another index. (The name of the index is usually the name of the related field.) Thus we end up with the next record appearing in the alphabetical order of this other field.

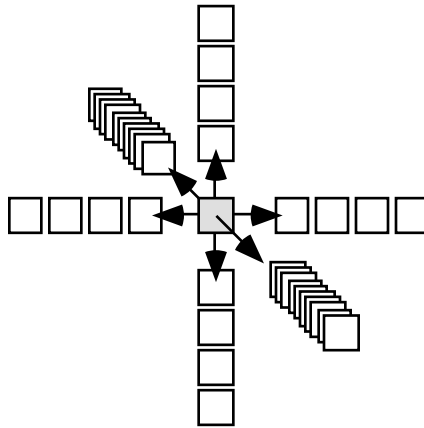


Fig. 4 Take your pick!

Indexes show the way

As Figure 4 shows, there are indexes in every direction starting from the record we found. This leaves us free to choose the direction we wish to move in.

Previous on SURNAME

(Proc.I. 6)

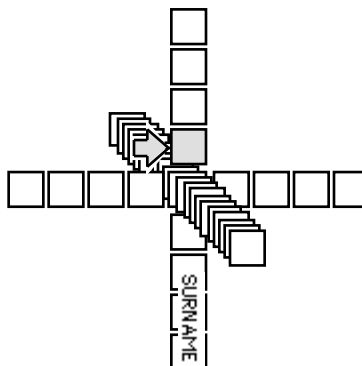


Fig. 5 Taking one step back in the SURNAME index

When the command in Proc.1. 6 is carried out, we are taking one step back along the index SURNAME, as shown in Fig. 5. If the field value of the first record is 'Smith,' then this (previous) record's value must be nearer the front of the alphabet and its SURNAME field may contain the name 'Sadowy.' Starting from this new record, indexes extend in every direction, according to the contents of the fields in the other records.

Find tables

A 'Find table' is a set of records that matches the search criteria of a specific search. Typically, it is set up after a 'Find'; and with the aid of 'Next,' we go through the table one record at a time.

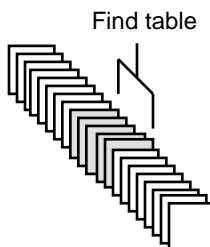


Fig. 6 A 'Find table'

A “constrained” Find table

When we use a search format or a search calculation, we usually get a limited selection of records somewhere in the middle of the file, as shown in Fig. 6. The example in Procedure 1 uses ‘Set search as calculation.’ Here, only animal names that start with “G” will be included in the Find table (which is generated with ‘Find.’) The subsequent ‘Next’ command only follows the Find table that had previously been generated. In this procedure, whenever there is a turn in the procedural loop, the record that has been read in is added to the #L1 list. When ‘Next’ passes the end of the Find table, the result is ‘flag false.’

Find table to list
Set current list #L1 Define list {fAnimal} Set search as calculation {mid(AnimalName1,1)="G"} Find on AnimalName (Use search) Repeat Add line to list Next on AnimaName (Use search) Until flag false

1

Line no.	Animal name
1	Goat
2	Gorilla
3	Pig

The contents of #L1 (after Procedure 1 has been run) are shown in the table below the procedure (previous page). This isn’t an especially speedy example, but it does show how Find tables works. Using Find tables often results in quicker procedures than using ‘Build list from file.’

A Find table in every Main file

Every Main file can have its own Find table. You change the Find table by switching Main file and use the right indexes in the ‘Next’ commands. Procedure 2 shows you how to do this.

Changing files and Find tables		
Set main file {fPlants}		
Find on PlantName {"Horseshoe"}		:: PlantName="Horseshoe"
Set main file {fAnimal}		
Find on AnimalName {"Goat"}		:: AnimalName="Goat"
Next on AnimalName		:: AnimalName="Gorilla"
Set main file {fPlants}		
Next on PlantName		:: PlantName="Hibiscus"

2

Changing files and Find tables

The fPlants and fAnimals files are independent of each other. Having found a record in fAnimals, we can go back to fPlants and continue moving on from our previous position. We see that ‘Hibiscus’ comes after ‘Hazel’ in the alphabet, so everything should be just fine.

1

Find

‘Find’ starts at the beginning of the specified index and quickly finds the first record in the index order which meets the search criteria. This record is read from disk into the Current Record Buffer. If there are several similar records containing the same value, ‘Find’ will usually call up the first record in the lineup, i.e. the one with the lowest RSN.

Non-discriminatory searches

When we set a single search criterion, we choose an indexed field and a field value that matches the record we wish to find. ‘Find’ looks for the record that best fills these criteria. For text fields, this means that as many letters as possible (starting from the left) in the text of the search criteria should match the text in the record we’re

looking for. The first letter takes precedence over the second one, which in turn takes precedence over the third one, etc. The same rules apply when we do a sort.

The number “0” and the letter “O”

To make it easier to work with data of varying quality, Omnis has purposely been programmed to distinguish between the number “zero” and the letter O. If the original data has been entered according to some out-of-date system, or by an inexperienced user, the letter “O” may have been used to represent the number “zero.” In most text-based systems, the number “0” appears with a diagonal line and thus strongly resembles the Norwegian letter “Ø”. The intention is to distinguish between the number “0” and the letter “O”. Some inexperienced users do, in fact, mistake the zero with a slash through it for the letter “Ø” and are therefore reluctant to use it.

This problem dates back to the time when typewriters reigned supreme. Since the number “0” and the letter “O” look so much alike, many old typewriters didn’t even have a separate character for “zero.” Those who grew accustomed to the lack of this character (as well as those who think that the “Ø”-zero character is the letter “Ø”) might very well use the letter “O” instead. This could prove to be more than the minor annoyance it would appear to be at first glance. You basically get the same result you would get if you were using a low quality database: undependable results when searching.

What it all boils down to is that uppercase “O” and lowercase “o” in the data file can be interpreted both as numbers and letters when files are being searched. If the text you are searching for contains the number “0,” the Find table will include records in which the *letter* “O” is supposed to mean “zero.” However, if the record in the data file contains the *number* “0,” it is interpreted as a zero, and nothing else.

Number values

As for number fields, the first record chosen is the one that has a value equal to *or greater* than that of the search criterion. When there is no exact match, Omnis will pick the record that's next in line (in the index, that is). If, for example, the search criterion uses the value 97, 101 will be chosen instead of 96, since 101 has a higher numerical value. The fact that 96 is closer to 97 has no bearing on the matter here.

Simple searches

If the only search criterion is a simple comparison, and 'Exact match' has not been selected, this is more like pointing at a record in the file than making a limited selection.

Find on AnimalName{"Goat"}

(Proc.I. 7)

No "constrained" Find table is generated here. 'Next' will start from 'Goat' and continue to scroll through the file until it reaches the end of the index.

Discriminatory searches ('Exact match')

If we select 'Exact match,' this means that the field's value should match the contents of the search criteria and nothing else. The number of letters should match, as well as the use of upper and lowercase letters. If a letter "O" has crept in where there should be a zero, the record will not be accepted.

Find on AnimalNames {"Goat"} (Exact match)

(Proc.I.. 8)

If, as shown in Proc.I. 8, 'Find' is used, any subsequent 'Next' will scroll only through those records where AnimalNames equals 'Goat.' This gives us a short Find table.

Advantages

Since 'Exact match' searches don't accept many exceptions or extra conditions, they can search large files somewhat more rapidly.

Single letters

If we only give the first letters, 'Find' will locate the first record (in the index sequence) containing these letters. In this case we will only use one letter:

Find on AnimalNames {"M"}

(Proc.l. 9)

The cursor starts at the first record beginning with 'M,' e.g. "Monkey." Since there are no other search conditions, the search will only indicate where the cursor should begin. Each ensuing 'Next' will then gradually work its way through the entire file, starting at the first record beginning with 'M.'

If we had selected 'Exact match,' the search criterion would have dictated that AnimalNames should be 'M,' neither more nor less. But since there is no animal by that name, the search would fail. What we usually need in such cases is a search format that gives AnimalNames beginning with 'M' as a condition.

Find on AnimalNames

(Proc.l. 10)

If no specific point of comparison is given, the search will use the field value in, say, AnimalNames to start the search, as shown in Proc.l. 10.

TIP: Make sure that you know the difference between text sorting and numerical sorting, especially where numbers are concerned. For the numbers 1 to 10, the sort will look like this:

Numerical sorting:	1,2,3,4,5,6,7,8,9,10
Text sorting:	1,10,2,3,4,5,6,7,8,9

During text sorting, the number 10 will come between 1 and 2, because this type of sort looks at one digit (i.e. one character) at a time. Values with the lowest character come first, followed by all the other numbers that begin with the same digit. Both 100 and 1000 would be placed between 10 and 2.

The solution is to place enough zeros in front of the numbers that don't have the maximum number of digits within the data file, like this:

Text sorting:	01,02,03,04,05,06,07,08,09,10
	001,002,050,070,071,100

Next

This command works in much the same way as 'Find.' The difference is that it always starts where 'Find' left off. It follows the Find table that was generated last, using the search criteria that were given then. 'Next' can narrow the search by using 'Exact match' or 'Use search' where the original 'Find' didn't. However, we cannot broaden the search this way. A 'Next' command which is less complex (i.e. which has fewer options selected than the original 'Find' command) will follow the same pattern that has been established. In other words, it will use the Find Table that has been generated.

'Next' can also change direction, i.e. decide to continue along the index of another field. This will lead to a new Find Table being generated in this direction, the size of which will be determined by the search criteria that were given with this particular 'Next' command ('Exact match' or 'Use search').

Unsuccessful searches

When the conditions for a search using "Find" are too rigid for the data contained in the file, the search will fail. This means that the fields in the Main file and in all hierarchically connected files will be deleted from memory; in addition, the flag will be "false." The Main file's Find Table will also be gone. If the value we wanted to find only existed within the field we searched, we'll have to retrieve it from somewhere else, such as a variable or a list, because the field is now empty.

```
Find on AnimalNames(Exact match) ;;  
AnimalNames="Bull" (Proc.l.11)
```

Immediate effects

If the search in Proc.l. 11 fails, the value of the field AnimalNames will be deleted. Let's imagine that in Proc.l. 11 the file doesn't contain a record with an AnimalName that matches "Bull." If we try to repeat the command without adding any other information, this will amount to carrying out a search without any search criteria. The value we want Omnis to find is #NULL, which is always at the very front of the index. In this case, the first record in the AnimalNames index is called up, giving the same result as with 'Find first on AnimalNames.'

Find tables

The same thing happens when 'Next' or 'Previous' have strayed outside the bounds of Find table. The field values in the Main file and connected files are deleted from memory. It will not be possible to "sneak back into" Find table again with 'Next' or 'Previous,' because essentially what we have here is a failed search. Both Find table and the field value that are being used in the search are lost, the latter being a prerequisite for rebuilding Find table with the aid of 'Find' or 'Next.' This is one of the reasons why lists are often handier than Find tables for carrying out complex search operations.

Find tables and search formats

Failed searches won't lead to this problem if we use search formats or 'Set search as calculation.' This will have the effect of putting the value we plan to use for comparison in a safe place, whether the fields in Main file are deleted or not. If we use 'Next,' with 'Use search' selected, the first 'Next' that fails will result in blank fields in the Main file. The following 'Next' will use comparison values in the search format (or calculation) to generate a new Find table, after which it

will find the first record here. 'Next' will appear to have abruptly halted (blank fields), and then restart at the beginning of Find table.

Connected files

Failed searches delete the fields in both the Main file and all hierarchically connected files. This can all too easily be overlooked in complex procedures and may thus lead to numerous inexplicable failures. As long as you bear this point in mind, it should be relatively easy to eliminate or avoid such errors.

Searching in Connected Files

When we work with files that are linked, everything appears to work fine until we begin searching for data. Searching in connected files may seem problematic, but the only thing we really need is to know how to search for single files. Omnis itself takes care of some of the hierarchical searches automatically, whereas with relational joins we have to do everything ourselves. The searches always take place one at a time, one file at a time. You only need to know the order they're in and the information you need to carry out each search.

Key fields and foreign keys

When considering file connections, a couple of concepts will help us describe how the fields are linked. We have already explained how a file always has a field that can identify the different records. In other words, this means that none of these records ever have identical values in this field. This identifying field is essential, which is why it is called the “key field.” (It isn't very helpful to think in terms of keys and locks and the like; in fact, it will only confuse the issue.)

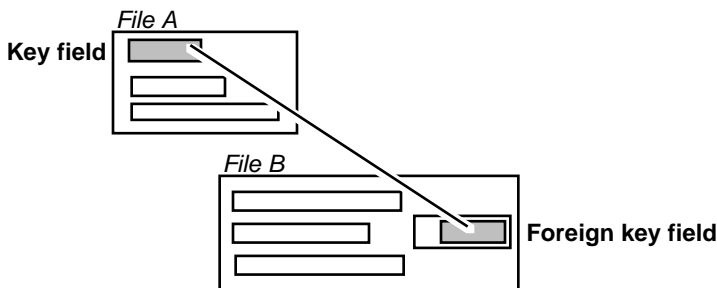


Fig. 7 Key field and foreign key field

Among other things, Fig. 7 shows that file B is connected to file A. The field (in file B) which enables every record in file B to remember which record they are connected to in file A is called a “foreign key field.” This field (also known as the “foreign key”) is thus a foreign file's characteristic “fingerprint,” which can be used to call up the foreign record that made the imprint – i.e. the one to

which the child record is linked. More on this later. For now, it's enough to remember that the key field is the identifying field of the record, whereas the foreign key is a *foreign* record's "fingerprint."

Hierarchical connections and relational joins

For a quick recap of the different types of links between file formats, refer to the chapter entitled "File Connections." Fig. 8 shows the underlying principle behind 'Many-to-One' connections. The developer uses relational joins to generate the connections between the various records in the file formats and decides himself when the connected records should be read in. In all the subsequent examples, RSN is the identifying field for the different file formats. In the "relational join" examples, we assume that the value of RSN has been copied from the parent record into the foreign key fields in the child record. Hierarchical file connections are based on an internal "copying" and managing of the RSN, so most of the locating of the parent records is done automatically.

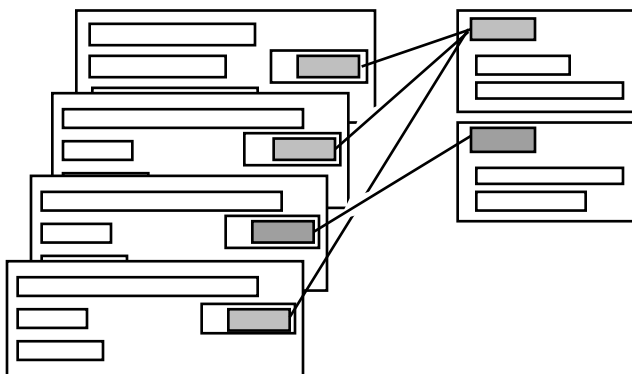


Fig. 8 File connections – a brief recap

Find parent record

Each child record has only one parent record in each connected file. When we have located a child record, connected records are usually read in automatically immediately thereafter. However, there are situations where this doesn't happen; in such cases we will want to read in the parent records ourselves.

Hierarchical

With a 'Single file find,' Omnis looks up only the file in question, even when connected file formats exist. Let's imagine a simple search with the Record Sequence Number of the file fPersons:

Single file find on P_RSN (Exact match) {67} (Proc 12)

RSN number 67 was chosen at random, merely as an example. When it comes time to find records that are higher up in the hierarchy, the next command will be:

Load connected records (fPersons) (Proc.I. 13)

In other words, we specify the name of the child file in Proc.I. 13. This file "reaches up" into the file structure and locates its parent record in all of its connected parent files.

Relational

When a given record has another record linked to it through relational joins, this tells us that one of the fields contains information capable of identifying the specific record it's connected to. In other words, this information is the foreign key. For the sake of simplicity, we can use 'Single file find' for every file to which the child file is connected. In Proc.I. 14, the parent file's key field is called P_RSN.

Single file find on P_RSN (Exact match) {C_FOREIGNKEY} (Proc 14)

When Proc.I. 14 has been carried out, the parent record will reside in memory.

Let's imagine a situation in which a person (fPerson) is connected to a company (fCompany) and to a specific customer category (fCategory), and that all these are stored in their respective files. In fPerson (the child file) there is one foreign key field for each of the two parent files, thus satisfying the conditions for a relational join. We have called the foreign keys P_CompanyKey and P_CatKey respectively, and they can be used as follows:

Single file find on CO_RSN (Exact match)

{P_CompanyKey}

Single file find on CA_RSN (Exact match) {P_CatKey} (Proc.I. 15)

CO_RSN and CA_RSN are key fields in the fCompany and fCategory, respectively. After Proc.I. 15 is finished, all connected records will have been read into the CRB.

Enable relational finds

If we wish to build up a table containing relationally joined records in two different file formats, we can do this automatically. First we set up a search format (or search calculation) that shows which fields are to have the same value. In other words, we set the key field in one of the file formats to be equal to the corresponding foreign key field in the other file format. (It's possible to narrow the table down even further by adding criteria to this search format – for example, periods of time and the like.) The connection is then established by the command 'Enable relational finds' containing the names of the file formats to be connected.

Once this has been done, we can generate a Find table using 'Find first,' or we can build a list directly by using 'Build list from file.' The generation of this list is independent on the Main file setting. Any sorting is set with the command 'Set sort fields.' Bear in mind, however, that this is not a method for locating records in a procedure one at a time. We have to think in terms of tables at this point. This is a practical way of doing things when you want to build lists – or print reports –

containing records from connected file formats.
Procedures 3 and 4 illustrate this.

'Enable relational finds' and Find tables

3

```
Set current list #L1
Define list {P_RSN,P_Name,CO_RSN,CO_Name}

Set search as calculation {CO_RSN=P_CompanyKey}
Enable relational finds {fPersons,fCompany}

Clear sort fields
Set sort field P_Name

Set main file {fPersons}
Find first (Use search,Use sort)
Repeat
  Add line to list
Next
Until flag false
```

'Enable Relational finds' and Find tables

3

CO_RSN is the Record Sequence Number in the file fCompany; the same is true for P_RSN and fPersons. Please note that in the 'Find first' command, 'Use search' and 'Use sort' must have been selected. It is completely controlled by the search format and the sort, no matter what the Main file is. If we neglected to include 'Use search,' list #L1 would have consisted of every conceivable combination of records in the two files.

'Enable relational finds' and 'Build list'

4

```
Set current list #L1
Define list {P_RSN,P_Name,CO_RSN,CO_Name}

Set search as calculation {CO_RSN=P_CompanyKey}
Enable relational finds {fPersons,fFCompany}

Clear sort fields
Set sort field CO_Name
```

'Enable relational finds' and 'Build list'

4

Here the list is sorted by company name. Instead of choosing the 'Use sort' option in the 'Build list from file' command, we could have sorted the list later using the 'Sort list' command. This would have been a quicker method partly because it would give you the option of preventing the 'Working message' from turning up.

Find connected records

When two file formats are connected, there will always be one or more records in one file linked to a specific record in the other one. We can call the former the "child" file and the latter the "parent" file. Let's imagine that the Main file has been set to the parent file and that we're in the parent file's window. What we want is to "scoop up" all the relevant child records. This is just the opposite of what we've been trying to do so far.

One-to-One

When we know that only one record is connected to each parent record, all we have to do is call up the first record that has the right value in the foreign key field (relational joins) or the "hidden copy" of the RSN field from the file above (hierarchical connection). In other words, we don't need to look for any other records in the child file that might also be linked to the current record in the parent file.

Find single child record – hierarchical

5

```
Set main file {fParent}
; Locate the desired parent record

Set main file {fChild}
```

Find on P_RSN (Exact match) Redraw windows

Hierarchical

5

First we let the developer find a parent record. In Procedure 5 we do a search in the fChild file. Strangely enough, however, we refer to an indexed field in the file above, i.e. fParent. The reason for this is that we aren't really looking at P_RSN, but at a sort of invisible, hidden copy of this field. When we set up a link from a child file to a parent file, Omnis generates a foreign key field in the child file that we never really get to see. Every time a new record is entered into the child file, Omnis automatically copies the RSN value of the parent record (which is currently in memory) over to this "invisible" field. The copy is thus stored together with each record in the child file. This field also has its own index. When we use the 'Find' expression in Procedure 5, where Main file is the child file and 'Find' still points to the RSN field in the parent file, Omnis understands that the search is to be carried out on the "invisible" foreign key field in the child file.

Find single child record – relational

6

Set main file {fParent} ; Locate the desired parent record Set main file {fChild} Find on C_ParentKey {P_RSN} (Exact match) Redraw windows
--

Relational

6

In this case, the logic is more "obvious" but perhaps harder to digest. We already have a record in the parent file, but we want to find a record in the child file that has a foreign key field value equal to the value of the key field of the parent file. To put it another way: we want to find the child record that bears the stamp of its parent. (In keeping with the modern trend toward high divorce rates, this example contains only one parent

file.) The foreign key field in the child file contains the parent's mark. To facilitate matters, we use the parent file's Record Sequence Number, P_RSN, as the identifying (i.e. key) field of the parent file. Thus the value of P_RSN is the parent's stamp.

Many-to-One

This is the most common situation when two files are linked. When searching for records that are connected, we often end up with more than one. These records can be placed in a list, to make it easy to display them in a window.

Find several child records – hierarchical	7
<pre> Format variable FoLs_ChildRecords (List) Local variable Lo_Forrige_P_RSN (Long integer) Set main file {fParent} Find first on P_RSN ;; Or find another record in fParent Calculate Lo_Previous_P_RSN as P_RSN Set current list FoLs_ChildRecords Define list {fChild} ;; All the fields in fChild Set main file {fChild} Find on P_RSN (Exact match) If flag true Repeat Add line to list Next on P_RSN (Exact match) Until flag false End If Set main file {fParent} Find on P_RSN {Lo_Previous_P_RSN} Redraw windows </pre>	

Hierarchical

Admittedly, Procedure 7 is a bit on the long side – but not without reason. We needed a list, so why not use one with a name that makes sense? Since this list is

probably going to appear in the window, it would have to be a format (not a local) variable. The loop that builds the list will continue to run until the 'Next' command fails. In view of what we know about unsuccessful searches, we should safeguard the RSN value of the current parent record. This will help us relocate the parent record when the loop has run its course. When the loop has finished, the CRB for fParent and fChild will be empty. This is easily remedied with a simple search, as shown at the end of the procedure.

Find several child records – relational

```
Format variable FoLs_ChildRecords (List)

Set main file {fParent}
Find first on P_RSN ;; Or find another record in fParent

Set current list FoLs_ChildRecords
Define list {fChild} ;; All the fields in fChild

Set main file {fChild}
Find on C_ParentKey {P_RSN} (Exact match)
If flag true
  Repeat
    Add line to list
    Next on C_ParentKey (Exact match)
  Until flag false
End If

Redraw windows
```

8

Relational

Even though the last 'Next' command in the loop fails here, we won't have the same problem as we did with hierarchical connections. When the last search clears the fields in fChild, nothing happens to fParent, because these have not been linked by means of Omnis' hierarchical connections. So we might as well leave the fields in the child file alone.

8

Many-to-Many

Many developers find that “many-to-many” linking is a fairly complicated business. What’s hardest to swallow, perhaps, is the fact that basically there is no easy way of generating links of this kind. What we really need is a trick up our sleeve. And, lo and behold, we’ve got one: a “link file.” Such a file contains the links between the two files to be connected. This might seem to be a cop-out, but it really isn’t. This method is often used and gives you very good control over your data. It would have been a lot harder to find your way around in the morass of data if the connections themselves had been hidden from the developer.

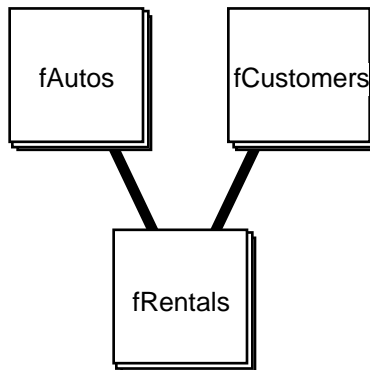


Fig. 9 Sample file structure for a car rental company, represented hierarchically.

The mechanics of linking

The link file is connected to both of the files we want to connect. Basically, when we have a ‘many-to-many’ connection, there are three ways of viewing the data: First, we can trace all the links between both files (for a given period of time, for example). Second, we can get hold of a specific record within one of the files and see which records in the “neighboring file” it has been connected to at some point or other (with the aid of the linking file). We can also do this by using the other file as our point of departure. Third, and lastly, we can still locate records that are connected directly

(i.e. fRentals – fCustomers, and fRentals – fAutos), just as we’ve always been able to.

Example: Car rental company

The following example models its procedures on a hypothetical application for a car rental company. The company has a stable of cars (fAutos) and of customers (fCustomers) who do business with the company. Each rental registration (fRentals) is linked to one auto (fAutos) and one customer (fCustomers). In other words, fRentals serve as a link between fAutos and fCustomers. This gives us a many-to-many connection between fAutos and fCustomers. Figure 9 shows us the links between the files, and the field names are shown below in Figure 10.

fAutos	fCustomers	fRentals	Comments
A_RSN A_Make etc.	C_RSN C_Name C_Address etc.	R_RSN R_From_Date R_To_Date (R_CarKey) (R_CustomerKey)	Key fields Foreign key (relational) Foreign key (relational)

Fig. 10 Field names in the car rental company application

Count rental registrations for a specified period: Search on linking file

The owner of the car rental company will naturally want to print a monthly budget for the accounts. To do this, we do a search on fRentals and thus call up whatever records are to be found within the desired time period.

Generate a list of rentals for November, 1993 – hierarchical
Format variable FoLs_Records (List)
Set current list FoLs_Records
Define list {fRentals,fAutos,fCustomers}
Set main file {fRentals}

9

Set search as calculation <pre>...{R_From_Date>=dat("011193")&R_From_Date<=dat("301193")}</pre> Build list from file (Use search)
--

Hierarchical

When the link file fRentals is linked to fAutos and fCustomers by means of hierarchical connections, the records in fAutos and fCustomers will appear when they should, automatically. All we need to be concerned with is getting hold of the right records in fRentals. We have selected a search here and built a list of records from fRentals. Since the fields in fAutos and fCustomers are also included in the definition of the list FoLs_Records, we will receive all of the information at once.

9

Print many-to-many report from linking file – hierarchical	10
Set main file {fRentals} Set search name sPeriod ;; Decide on an appropriate period Set report name rPeriodOverview Print report (Use search)	

Reports

Printing reports is a snap. We just manipulate fRentals. We set up a search for the particular period for which we want a data printout. Connected records in fAutos and fCustomers automatically appear in the CRB. We place selected fields from these files in the report, and otherwise treat the report as if it had contained fields from only one file. This is an example of what happens when Omnis’ hierarchical connections come into their own.

10

Generate a list of rentals for November, 1993 –
Relational

11

```
Format variable FoLs_Records (List)

Set current list FoLs_Records
Define list {fRentals,fAutos,fCustomers}

Set main file {fRentals}
Set search as calculation
{R_From_Date>=dat("011193")&R_From_Date<=dat("301193")}

Find on R_From_Date (Use search)
If flag true
  Repeat
    Single file find on A_RSN {R_BCarKey} (Exact match)
    Single file find on C_RSN {R_CustomerKey} (Exact match)
    Add line to list
  Next
Until flag false
End If
```

Relational

11

If we use relational joins between fExpenses and fAutos, and between fExpenses and fCustomers, we'll have to add a couple of extra commands to the procedure. The principle is the same as with hierarchical connections (see the preceding paragraph). The difference lies in the fact that connected records aren't read in automatically. But we do have the foreign keys to both of the connected files, which enables us to find the records by using simple searches.

R_CarKey is the foreign key field in fRentals; it tells us which car is linked to this rental. Likewise, R_CustomerKey tells us which customer the rental record is linked to. In other words, it tells us who has rented the car. (A_RSN and C_RSN are the Record Sequence Numbers of fAutos and fCustomers, respectively.)

Reports

The connected records appear in the report if we use 'Automatic Find' fields. These report fields must have

the identifying fields in the connected files as field names (A_RSN or C_RSN), and the corresponding foreign key field in the Main file (fRentals) as the comparison value. As for fAutos, we set A_RSN as the field for Omnis to search in; its value is found in R_CarKey.

See which cars a specific customer has rented:

Search using many-to-many connections

The owner, a curious man, would like to know what kinds of cars his customers like. He can do this by using Procedures 12 and 13.

See which cars a specific customer has rented –
Hierarchical

12

```
Set current list FoLs_Records
Define list {fRentals,fAutos,fCustomers}

Set main file {fCustomers}
Find first on C_RSN ;; <- Here the developer selects a customer.

Set main file {fRentals}
Find on C_RSN (Exact match)
If flag true
  Repeat
    Add line to list
    Next (Exact match)
  Until flag false
End If

Clear sort fields
Set sort field A_Make
Sort list
```

Hierarchical

12

In Procedure 12, the search is still carried out in fRentals, but we know that C_RSN (fCustomers) is supposed to have a fixed value. When we have determined which customer we want to peruse, we just carry out a simple search on fRentals. This is exactly the same thing that was done in the example under

One-to-One connections. Even though we refer to a field in a connected file (Find on C_RSN), the search is carried out in fRentals, with the index of the aforementioned “invisible field.” Finally, in this procedure we sort according to A_Make, which makes it easier to see which car the customer has used the most.

See which cars a specific customer has rented –
Relational

13

```
Set current list FoLs_Records
Define list {fRentals,fAutos,fCustomers}

Set main file {fCustomers}
Find first on C_RSN ;; <- Here the developer chooses a customer.

Set main file {fRentals}
Find on R_CustomerKey {C_RSN} (Exact match)
If flag true
  Repeat
    Single file find on A_RSN (R_CarKey)
    Add line to list
    Next (Exact match)
  Until flag false
End If

Clear sort fields
Set sort field A_Make
Sort list
```

Relational

13

When we use foreign keys, the real logic behind file connections becomes more apparent. When the customer has been found, we know what value R_CustomerKey is meant to have – that is, C_RSN. We run the searches on R_CustomerKey, which should be equal to C_RSN. Every time a new record in fRentals is found, we get the value of the foreign key R_CarKey, which in turn will guide us to the connected record in fAutos. We use this foreign key value to search fAutos, as shown in the ‘Repeat’ loop in Procedure 13.

See which customers have rented a specific car:
Search with the many-to-many link

But then something really annoying happens: One of the cars has suffered damage that wasn't discovered during the routine check between rentals. So the owner wants to find out who rented this car last and thus smoke out the culprit. In principle, this is the same thing we did in the previous example. This time around, however, the constant is fAutos, and we can find out which customers the corresponding records in fRentals are connected to.

Find out which customers have rented a car –
Hierarchical

14

```
Set current list FOLs_Records
Define list {fRentals,fAutos,fCustomers}

Set main file {fAutos}
;; <- A car should be chosen here.

Set main file {fRentals}
Find on A_RSN (Exact match)
If flag true
  Repeat
    Add line to list
    Next (Exact match)
  Until flag false
End If

Clear sort fields
Set sort field R_From_Date
Sort list
```

Hierarchical

14

This procedure calls up every incidence of this car's rental and everyone to whom it has been rented. The sort at the end ensures that the contents of the list will appear in chronological order.

Find out which customers have rented a car – Relational

15

```
Set current list FoLs_Records
Define list {fRentals,fAutos,fCustomers}

Set main file {fAutos}
;; <- Here a car should be chosen.

Set main file {fRentals}
Find on R_CarKey {A_RSN} (Exact match)
If flag true
  Repeat
    Single file find on C_RSN (R_CustomerKey)
    Add line to list
    Next (Exact match)
  Until flag false
End If

Clear sort fields
Set sort field R_From_Date(Descending)
Sort list
```

Relational

15

There's nothing revolutionary here, just a drill session on the concept of foreign keys.

The basics of many-to-many linking

Whether you use hierarchical connections or relational joins, the link file will always be your point of departure when carrying out a search. In practice it's enough to concentrate on this one file and get good results. A search often yields many records. The essence of many-to-many linking lies in having many connections. The same records in the "upper files" (fAutos and fCustomers) are often linked together many times by the link file (fRentals).

Constraining the number of records found

In some cases you might want to limit the result of the search, e.g. show only one connection where there are several connections between the same records. For example, we can imagine that the owner of the rental

company wants to see which cars a customer has tried out sometime or other in the past. Manipulations such as this are definitely easier to do on lists. The procedure for that is to carry out an ordinary, broad search as shown in one of the examples above, and to remove unwanted “duplicates” from the list later. Alternatively, we can print out a report, selecting the ‘Duplicates blank’ option for the report fields in question.

Searches Spanning Several Generations

Basically, the way to carry out searches that span several generations is to run one generation at a time, applying the same principles we have already described. In the examples below, the files have the following connections and fields:

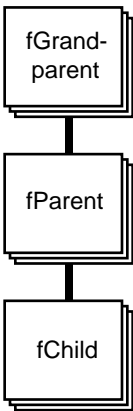


Fig. 11 The file connections in the examples

fChild	fParent	fGrand-parent	Comments
C_RSN C_Text (C_ParentKey	P_RSN P_Text (P_Grandparent Key	GP_RSN GP_Text	Key fields Foreign keys

Fig. 12 Field list for fChild, fParent and fGrandparent

Find Grandparent

Find grandparent – Hierarchical

16

Set main file {fChild}
Find first on C_RSN ;; <- Here the developer may choose a record.
Load connected records {fParent}

Hierarchical

16

As soon as a record in fChild has been found with the ‘Find’ (or ‘Find first’) command, the corresponding record in the parent file will be read in. To include the grandparent file as well, we can use the ‘Load connected records’ command. It calls up the connected record in the file above the one specified in the command. We may repeat this in each generation of files. In our case, however, we only need to do this with one generation, i.e. fParent – fGrandparent.

Any great-grandparents would be dealt with the same way: Let the “children” in each generation do the work (i.e. use a ‘Load connected records’ command with fGrandparent).

Find grandparent – Relational

17

Set main file {fChild}
Find first on C_RSN

Single file find on P_RSN {C_ParentKey}
Single file find on GP_RSN {P_GrandparentKey}

Relational

17

When we use relational joins, we have to search the files themselves. This is okay, because for each record searched, the foreign key to the record in the file above appears at the same time. C_ParentKey belongs to the file format fChild, but it contains the value that P_RSN in fParent is supposed to have. That’s why we use C_ParentKey in order to search fParent. When the connected record in fParent has been found, we once again get the foreign key of the fGrandparent record.

Since it is located in the P_GrandparentKey field, all we have to do is a search in fGrandparent on G_RSN, which should have the same value as P_GrandparentKey.

Find all grandparents - Relational finds	18
Set current list #L1 Define list {fChild,fParent,fGrandparent} Set search as calculation ...{C_ParentKey=P_RSN&P_GrandparentKey=GP_RSN} Enable relational finds {fChild,fParent,fGrandparent} Set main file {fChild} ;; Strictly speaking, Main file is superfluous Find first (Use search) While flag true Add line to list Next End While	

Enable relational finds

18

If we want to generate an entire table that shows all of the connections between the three files, Omnis provides us with an elegant built-in way of doing so. Procedure 18 is reminiscent of Procedure 5, except that it has one more file. Here the search calculation contains two criteria, each of which equals the foreign keys and the key fields in their respective files. The list thus generated shows all of the series of links that stretch from one end to the other. However, where the only connection is between fGrandparent and fParent, neither of these records will appear in the list. Detached records that are not connected to any specific record will also be omitted.

Find grandchild

One-to-One linking

If we have one-to-one connections at all levels, finding a grandchild will be straightforward enough. We find the record (in the

“descendant” file) which is connected to the current record. The process repeats itself for every level on down.

Find grandchild – Hierarchical

19

Set main file {fGrandparent}
Find first on GP_RSN ;; ...or any other record

Set main file {fParent}
Find on GP_RSN (Exact match) ;; Find connected parent record.

Set main file {fChild}
Find on P_RSN (Exact match) ;; Find connected child record.

Hierarchical

19

GP_RSN is the Record Sequence Number of fGrandparent, and P_RSN belongs to fParent. Each search moves down one level and finds the first record that’s connected to the record in the file above it.

Find grandchild – Relational

20

Set main file {fGrandparent}
Find first on GP_RSN ;; ...or any other record

Single file find on P_GrandparentKey {GP_RSN}
Single file find on GP_ParentKey {P_RSN}

Relational

20

The ‘Single file find’ command works very well with relational joins. Since the field P_GrandparentKey belongs to fParent, Omnis can find the first record in fParent where P_GrandparentKey matches GP_RSN. When the connected record in fParent has been found, we will know what value C_ParentKey must have – namely, P_RSN. We can use this value to find the connected record in fChild, as shown in the procedure.

Many-to-One

Normally, there will be many-to-one connections at each level when files are linked. If we search downward through the file structure, we'll get a long list of "descendant" records for each record in every generation. In other words, we'll get a fan-shaped structure and end up with a great many records on our hands. We have to view all the records that have been found, anyway. We should ask ourselves what type of information we are really after. The result of the searches will be so extensive that it would be advisable to use a report instead. Here, the records can be sorted by generation in the file structure. In some cases there might not be very many records that are connected at each level, and then such a search might be justified. Let's take a look at how to do this in the procedure that follows:

Find all grandchildren – Hierarchical

21

```

Format variable FoLs_Children (List)
Format variable FoLs_Parents(List)

Set current list FoLs_Children
Define list {fChild,fParent}
Set current list FoLs_Parents
Define list {fParent}

Set main file {fGrandparent}
Find first on GP_RSN    ;; ...or any other record

Set main file {fParent}  ;; Generate FoLs_Parents
Find on GP_RSN (Exact match)
If flag true
    Repeat
        Add line to list
        Next on GP_RSN (Exact match)
    Until flag false
End If

Set main file {fChild}   ;; GenerateFoLs_Children
For each line in list from 1 to #LN step 1
    Calculate P_RSN as Ist(P_RSN)
    Find on P_RSN (Exact match)
    If flag true
        Repeat
            Set current list FoLs_Children
            Add line to list
            Next on P_RSN (Exact match)
        Until flag false
    End If

```


Set current list FoLs_Parents End For
--

Generate FoLs_Parents; first Repeat loop

21

First we add all of the parent records that are connected to the grandparent record to a list (FoLs_Parents). When 'Next' in the first 'Repeat' loop finally fails, the fields in fParent and fGrandparent will be cleared from the CRB; but the FoLs_Parent list is retained.

Find all connected child records; second Repeat loop

21

After this we go through FoLs_Parents, one line at a time, and find the records in fChild that are connected to each and every parent record in the list. Any child records that we find we'll place in the FoLs_Children list. When the procedure comes to an end, we'll have gotten hold of all the records in fChild that are connected to the record in fGrandparent via fParent. In addition to all of the fields in fChild, the FoLs_Children list will also contain the parent file's RSN (P_RSN), which should make it possible to find out which parent records the various child records are connected to.

Find all grandchildren – Relational finds
Set current list #L1 Define list {fChild,fParent,fGrandparent} Set search as calculation... ...{C_ParentKey=P_RSN&P_GrandparentKey=GP_RSN... ...&GP_Text="Ancestor"} Enable relational finds {fChild,fGrandparent,fParent} Set main file {fChild} ;; Strictly speaking, Main file is superfluous Find first (Use search) While flag true Add line to list

22

Relational finds are the answer

22

We can circumvent the whole problem by using 'Relational finds.' This solution is a bit handier and more streamlind than using 'Find,' although it may be less expressive. Procedure 22 enlarges somewhat on Procedure 18. Please note the search calculation. Here we've added an extra search criterion. Our point of departure, the record in fGrandparent, is "GP_Text='Ancestor'." When the search has been set this way, all of the connected records will appear. Don't forget that there must be connections between every generation. This means that a parent record must be connected to the grandparent record, and a child record must be connected to this parent record before any of the three records can appear in the Find table.

Search Formats

Search formats are an integral part of Omnis. They greatly simplify the task of making a selective and successful search. Moreover, for the most part they are quite quick. The developer usually equips himself with a fair supply for his own use, but the user can also create his own. We aren't going to get into the nitty-gritty of search formats here; we'll take a closer look at some of their more functional aspects.

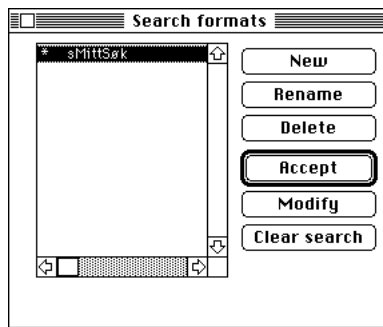


Fig. 13 Standard dialog box for 'Prompt for search format'

Prompt for search format

This command is a “go-ahead,” as it were, allowing the end-user to use his own search formats. Without any further programming on the developer's part, the user can generate, modify, and activate his own search formats. This is a powerful tool, and an especially handy one for executing a search with complex conditions.

The standard pushbuttons in Omnis allow only for searches with one criterion. For example, there's no use in filling out more than one field in a window when a 'Prompted find' is carried out, because the only index used will be the last one filled out. Either the user must create his own search formats or you must apply tailor-made procedures.

Disadvantages with user-made search formats

There are some disadvantages, however. To the user, the process of creating search formats might appear unduly cumbersome for trivial search tasks. Not only that, but the user has to understand the meaning of the field names that have been used in the application. Often, the developer will have to use very long field names. It isn't possible to call up the field list (alt/cmdn-9) as long as the 'Design' menu isn't visible. If the developer uses a prefix in the field name, the user will also need to know this prefix in order to correctly type in the first couple of letters in the field name. Normally, the user has only to key in these few letters and a popup list of alternatives will appear. As an alternative to long, "self-explanatory" field names, the user can jot down a list of the field names and their meaning. However, this really isn't a very satisfactory solution, either.

The naming of search formats

Finally, you might discover that it is hard to find suitable names for search formats. Even experienced developers often assign names that are totally unrecognizable a few months later. Users are faced with the same problem, but they have the advantage of seeing the application and their own search formats every day. This makes it easier to memorize names – even cryptic ones. The standard list that 'Prompt for search' displays is very narrow, and doesn't show very many letters of the search format names. If the user chooses to use long, descriptive names, he'll have to scroll horizontally or visit the 'Rename' dialog box (where the full name is displayed) in order to recognize his search formats.

Conclusion

I believe that search formats created by end-users are very handy as an alternative to and possible enlargement upon existing search options in an application. However, the most advanced searches should be built-in in a way that makes them easier for the user to use. One way of doing this is to use your

own ‘Enter data’ to assemble the search criteria, and then generate lists that are displayed in separate windows. Ad hoc reports are yet another good alternative; short reports often serve the same purpose as searches.

Flexible search formats

When a developer uses search formats, there is little point in using fixed values in the criteria; it only limits the utility of the search format. A typical example of a search is to limit the period of time for which a report is written. If the user should want to see the figures for November, this doesn’t necessarily mean, of course, that he will always be interested in this particular month. This is why we need a variable as a comparative value.

Let’s take another look at our example of the car rental company again. fRentals contains R_From_Date and R_To_Date. To simplify this a bit, let’s concentrate on R_From_Date. This date is supposed to lie within a specific time period. We use two variables to set the period, e.g. gl_Period_Beginning and gl_Period_End. (These two must either be ‘Library’ variables or belong to a file format.) Both of them are date variables, with the same date format as R_From_Date. The search format will look like this:

```
R_From_Date >=      [gl_Period_Start]
AND
R_To_Date           <=      [gl_Period_End]
```

Note that the [gl_Period_Beginning] and [gl_Period_End] variables must be enclosed in square brackets. Strictly speaking, the “AND” line is superfluous. The user can assign values to these two variables in an appropriate window, and the subsequent report will thus be generated from records within the desired time period.

Search formats versus ‘Set search as calculation’

Before version 2.0 of Omnis 7, searches defined with ‘Set search as calculation’ were not analyzed with the thought in mind that one of the existing indexes in the file formats could be used during the

search. The search was therefore always much slower than when search formats were used. Now, however, search criteria are closely monitored, and the search is often just as quick with search formats. This is why the distinction between search formats and “search calculations” is no longer as clear-cut.

Pros and cons

When you’re in the middle of programming a procedure and suddenly realize that you need a search with two conditions (e.g. a time period), it’s very easy to set up the search as a calculation. It’s only when the conditions begin to proliferate that you need to resort to search formats. This gives you a better overview of all the conditions. Then again, search calculations can quickly become incredibly messy. However, they always have the advantage of allowing the developer to see the entire contents of the procedure. This way, nothing is hidden. A search format, on the other hand, is protected from the gazing eyes of the inquisitive developer. It has to be called up specifically if its contents are to be changed, and whatever is there you are obliged to remember when editing the procedure.

Speed

The difference in speed between the two should be tested in every single procedure. Search calculations are quickest on the simpler searches, but only as long as you avoid setting up calculations that have to be repeated for every index value that is checked. This dramatically favors search formats, which in certain cases avoid these “repetitive calculations.”

Creating search formats with notation

The notation system in Omnis lets you create search formats for the end-user. These formats are generated from whatever information was entered during what the user regards as an otherwise normal ‘Prompted find.’ This method absolves the user from having to work with search formats, making the application easier to use.

We’ll use a standard Omnis window as our point of departure, but change the ‘Find’ pushbutton to ‘User-defined,’ and put the following procedure under:

Find	23
Local variable Lo_Last_RSN (Long integer)	
If #CLICK	
Calculate Lo_Last_RSN as B_RSN ;;;...Your RSN in the Main file	
Clear main file	
Redraw windows	
Enter data	
If flag true	
Call procedure pMakeSearchFormat/1 {A•Main procedure}	
Else	
Single file find on B_RSN {Lo_Last_RSN}	
Redraw windows	
End If	
End If	

The Find procedure

The procedure clears the Main file and uses ‘Enter data’ to get the user’s search data. If the user interrupts, the procedure will look for the previous ‘current’ record. In order to achieve this, a copy of the Main file’s Record Sequence Number is used. In this procedure, the copy is known as Lo_Last_RSN. The RSN field here is called C_RSN. If the user presses OK, Procedure 24 will run:

pMakeSearchFormat/1 {A•Main procedure}	24
Local variable LoLs_FieldsInMainFile (List)	
Set current list LoLs_FieldsInMainFile	

```

Set search name sEmptySearchFormat
Build field names list (Clear list) {[sys(82)]}
Redefine list {Fo_Fieldname}

For each line in list from 1 to #LN step 1
  Load from list
  If len(fld(Fo_Fieldname))>0
    Call procedure pMakeSearchFormat/2 {A1•Add line to...
  End If
End For

Find (Use search)
If flag false
  Sound bell
End If
Redraw windows

Revert format {sEmptySearchFormat}

```

The basic search procedure

24

This procedure can be run from any window, provided the appropriate Main file has been set. With the ‘Build field names list’ command, the column’s name will automatically be #S5. To make the code clearer, we can change the list definition and insert Fo_Fieldname instead. To save time, sEmptySearchFormat, the search format we’re going to use, will already have been created. It’s empty and doesn’t contain any lines.

After this, the procedure goes through the LoL_FieldsInMainFile list and checks whether the fields contain any data. If a field turns out not to be empty, the ‘{Add line to search format}’ subprocedure is run. When every field has been checked, the search begins.

Finally, the search format is emptied of everything that had been added by the reloading into memory of the original empty version of sEmptySearchFormat.

```

pMakeSearchFormat/2 {A1•Add line to search format}

Calculate Fo_Fieldtype as [sys(82)].[Fo_Fieldname].$type

```

25


```

Switch Fo_Fieldtype
  Case "integer","number","boolean","sequence"
    If fld(Fo_Fieldname)=0
      Quit procedure
    End If
    Calculate Fo_Linetype as kSLeq

  Case "char"
    If mid(fld(Fo_Fieldname),1,1)="*"
      Calculate Fo_Fieldname as
        ...(Use fld() of name)
      Calculate Fo_Linetype as kSLcon
    Else
      Calculate Fo_Linetype as kSLbeg
    End If

  Case "date"
    Calculate Fo_Linetype as kSLeq

  Default
    Quit procedure
End Switch

Set reference Fo_LINE to $formats.sEmptySearchFormat.$objs.$add
...(Fo_Linetype,"","")
Calculate WASDONE as Fo_LINE.$linetype.$assign(Fo_Linetype)
Calculate WASDONE as Fo_LINE.$fieldname.$assign(Fo_Fieldname)
Calculate WASDONE as Fo_LINE.$text.$assign(fld(Fo_Fieldname))

```

Setting up single lines in the search

25

Each search line must be consistent with the field type of the field it contains. Therefore, the Fo_Fieldtype variable will be assigned the field type of the field in question. See the following:

Numbers

Number fields left empty by the user are usually given the value zero by Omnis. To avoid superfluous search criteria with the number zero, we simply refrain from setting up a search line. But if the number is higher than 0, we set the search criterion to 'Number field equals numerical value.' Why don't all you sharp developers out there rise to the challenge and come up with something better!

Text

With text fields, we usually have ‘Text field begins with specified text’ as the search criterion. If the user begins his search text with an asterisk (“*”), the search will be set to ‘Text field contains specified text.’ The asterisk will, of course, be removed before the search criterion (the search line) is added.

Other field types

Searching list fields, graphic fields, binary fields, etc. is a dead-end street.

Adding a line to the search format

Because of a little bug in versions 2.0–2.2, we had to go out of our way a bit. In principle, a notational expression using ‘add’ should have sufficed. But what we actually have to do first here is to create a line and then change its comparison type (\$linetype), current field name (\$fieldname) and comparison value (\$text). But this is no big deal.

Format variables

The following format variables were used in Procedures 23–25 (next page):

Format variables

Format variable WASDONE (Boolean)
Format variable Fo_LINE (Item reference)
Format variable Fo_Linetype (Character)
Format variable Fo_Fieldtype (Character)
Format variable Fo_Fieldname (Character)

26

These procedures have been slightly rewritten; but the idea and the original procedure is Tommy Obæk’s from Gatsoft A/S. Thanks, Tommy!

Speed Tests

The test method

The results of these tests should be interpreted philosophically. The tests were performed on a PowerBook 180 equipped with adequate RAM and a relatively fast hard disk. I have assumed that the differences between the various search methods will be the same on other machines, e.g. IBM compatibles, since Omnis applications are binary platform compatible. The procedure used was a variation on the following:

Single search: Search as calculation

Calculate Fo_Tot_Start as #CT

Calculate Fo_Number_Search as 100

Set search as calculation {B_TXT="Stallion-5"}

Calculate Fo_SearchStart as #CT

For #1 from 1 to Fo_Number_Search step 1

 Find on B_TXT (Use search)

End For

Calculate Fo_SearchEnd as #CT

Calculate Fo_Ticks as Fo_SearchEnd-Fo_SearchStart

Calculate Fo_Seconds as Fo_Ticks/60

Calculate Fo_ms_Per_Search as Fo_Seconds/Fo_Number_Search

Calculate Fo_Tot_Sec as (Fo_SearchEnd-Fo_Tot_Start)/60

27

The answer was recalculated in milliseconds, for the purpose of comparison with the hard disk's access time. However, the only thing that really counts here is *ticks*, since they form the basis of the test as a whole. This is why the procedures were run until their clock times stabilized, and then the lowest result was written down. The numbers in the graphs represent milliseconds per search and should be free from the influence of any preparatory work, e.g. the processing of search formats. The searches were carried out on a data file containing 412 records. All of these factors must be taken into account, and none of the results should be taken for granted.

Simple searches on an identified record

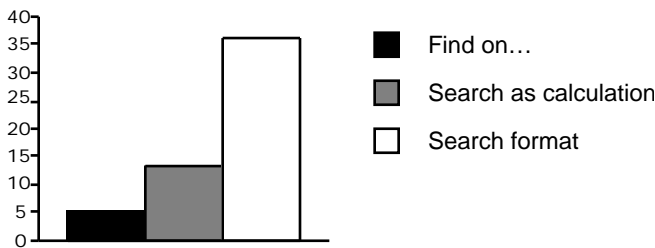


Fig. 14 Speeds tested with variations in single searches

Comments

With single searches there is little point in using more advanced methods. ‘Find’ with ‘Exact match’ is clearly the quickest, and that’s all you need in order to find an identified record.

Field larger than a certain value



Fig.15 Speeds tested with variations on searches for field values greater than a specific value

Comments

Surprisingly, ‘Search as calculation’ proved to be as quick as ‘Find.’ Once again, search format was the slowest.

(When I used ‘Find,’ I located the first record that fit the comparison value and then “paged” my way forward with the aid of ‘Next’ until the value exceeded the comparison value.)

Two fields, each with their respective values

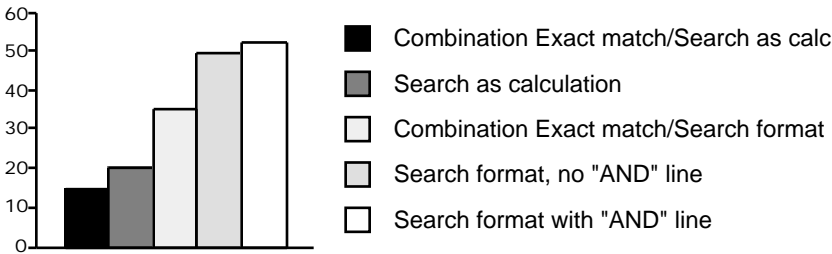


Fig. 16 Speeds tested with variations on searches with two search criteria

Comments

All sorts of combinations were possible here. The quickest in the test was a ‘Find’ with ‘Exact match’ and ‘Use search’ selected. The first search criterion was set by the designation of a value in the ‘Find’ command. The other one was in the search calculation in ‘Set search as calculation.’ Pure ‘Search as calculation’ searches proved to be quicker than a divided option with ‘Find (Exact match, use search),’ where the second of the search’s two criteria had been set with the aid of a search format. Finally, we see that inserting an ‘AND’ line in the search format appears to reduce the speed somewhat.

Conclusion

This test is not meant to provide a complete study of what types of searches are quickest at any given time. This is too comprehensive a subject for that. In any case, I would like to go out on a limb and suggest a few guidelines:

Simple searches

Simple searches are undoubtedly quickest using 'Find.' Relatively uncomplicated searches with few conditions appear to run more quickly when the search has been set with 'Set search as calculation' than when they've been set with a search format. Nevertheless, search formats do have a special mission, and in the more complicated searches it's hard to get along without them.

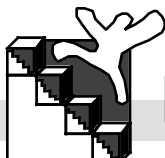
Rational use of searches

The odd search seldom involves any significant delay for the end-user. It is when lists are built, or when many searches are carried out in a similar manner, that you should think about speed. This is particularly the case in multi-user environments (when SQL is not used), where all of the information is exchanged over a network. The transmittal speed here is low compared to the local link between computer and hard disk. Basically you'll only have to deal with the most vulnerable procedures. Here you can try out the options available for 'Find,' 'Search as calculation' and search formats; and, naturally, it's a good idea to try the simplest one first. Then you take down the time, as shown in Procedure 25. And remember: don't delete any of the test procedures before you've found out which one is the quickest.

Section 8: 3rd. Generation Programming

Chapters:

1. External Routines
2. Introduction to Notation



External Routines

Introduction.....	2
What Is an External Routine?	3
Examples of External Routines	4
Ergohygeon	
pc-ORDERLY BOOK	
Plan info	
Can I Create an External Routine?	6
Where Do External Routines Have to	
Be Before I Can Have Access to Them?	7
When Should I Use External Routines?	8
Which Functions in Omnis Are Used in	
Connection With External Routines?	10
External routines	
Event handlers	
What Functions May I Use In an External Routine?.....	12
What Does the Code in an External Routine Look Like?	14
LibMain	
WEP	
The Stack Problem.....	17
What If I Want to Know More?.....	18
Difficult Words.....	19
An Example with Source Code	21
AB_INI.C	
AB_INI.DEF	

Introduction

External routines – A familiar concept to most Omnis programmers, but not everyone knows what it really means, or what external routines can be used for. And when the subject turns to C-programming and “Syntax errors,” there’s no denying it: many get chills down their spine. In this chapter we’ll attempt to demystify the concept somewhat, because it isn’t as difficult as it sounds. In our considered opinion, external routines are just what the doctor ordered; you will need them sooner or later, even though at present you may not even know they exist!

What Is an External Routine?

Briefly put, the ultimate aim of an external routine is to enhance the existing functionality of Omnis. We may not care to admit it, but there are times when Omnis falls short with respect to traditional programming languages. Some things are impossible to solve with Omnis alone; other things can be handled more effectively by an external routine than by creating a lot of complex code in Omnis.

We can take this a step further and say that it is a matter of writing special functions in C (or some other programming language) and subsequently compiling these in a library (DLL in Windows, Extension for Mac). This library can then be accessed from Omnis by calling functions that are stored in the library.

NOTE: It's important to realize that external routines, unlike Omnis itself, are platform-dependent; that is, you can't just take an external routine written for Windows and use it on your Macintosh, and vice versa. Throughout this chapter, the discussion and all our examples will take the Windows version of Omnis 7 as their point of departure, where the external routines go by the common name "DLLs."

Examples of External Routines

Nowadays a number of major commercial programs use external routines, albeit in different ways. See the three examples that follow:

Ergohygeon

This is a system for a company health service, developed at the Dept. of Computer Science, University of Bergen, Norway. Here external routines are used for drawing special kinds of graphics for job satisfaction profiles, health curves, etc. based on data stored in the Omnis database.

pc-ORDERLY BOOK

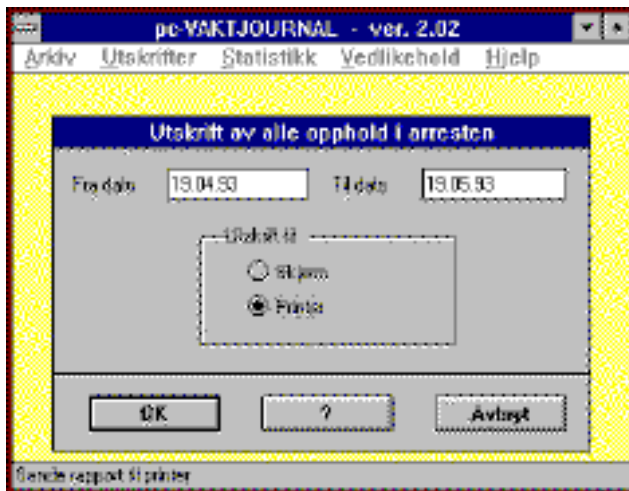


Fig. 1 Screen dump from the application "pc-ORDERLY BOOK"

This is an orderly book system for use by the municipal police department, developed by GAT - Soft as. It uses external routines to load information from a flat file database over into Omnis 7.

Plan info

This is a system for city planning, developed by the School of Architecture in Århus, Denmark. The external part utilizes geographical information stored in Omnis for the drawing of maps.

Can I Create an External Routine?

In principle, anyone can create his or her own external routine for enhancing functionality in Omnis.

To begin with, Omnis 7 and C programming are about as much alike as strawberries and pickled herring. It's just not possible to make syntax errors in Omnis when entering code; and it's easy to test any changes you may have made. When programming in C, however, you have to insert code in a number of different text files. Furthermore, you are not assisted by syntax checking, each command covers only small tasks, and all changes must be compiled (i.e. translated to a language the computer understands) before they can be tested.

So our conclusion is: Unless you are an experienced programmer already, you should steer clear of external routines to begin with. (Instead, get in touch with your local dealer; he'll help you!)

For another thing, you need some extra software. As we mentioned, an external routine has to be compiled before it can be utilized, and for this you will need a compiler. Blyth recommends Microsoft C, version 7.0; but other compilers, such as Borland C++, version 3.1, may also be used. A compiler also contains a number of other tools for program development.

Where Do External Routines Have to Be Before I Can Have Access to Them?

We get a number of inquiries about how to localize external routines so that they will be accessible for all Omnis applications. The simplest thing is to place all external routines just where Omnis 7 expects them to be, i.e. in the External Catalogue under Omnis 7 for Windows (for example, C:\OMNIS7\EXTERNAL\). If you are a Macintosh user, all external routines should be placed in the folder entitled “Omnis Extensions.”

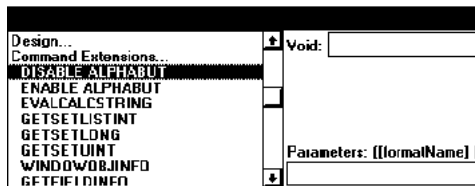


Fig. 2 Screen dump showing the Commands Extension list.

All external routines that lie in the External Catalogue will be read by Omnis at startup. Available functions will be included in Omnis' commands list (under the Command Extensions group), so that you can use these when developing the application.

You may also place external routines elsewhere; but then you must take care to include the Path to the routine when it is called. The disadvantage of this is that it leads to relatively inflexible solutions.

When Should I Use External Routines?

External routines have a wide variety of uses, and it's difficult to be categorical about when you should use them. There are times when you will have no other recourse than to create external routines, simply because Omnis alone cannot meet all your needs; at other times you will be faced with situations in which you can resolve a problem both in Omnis and by using external routines. Deciding which option is best is a matter of weighing the work involved in creating an external routine against the benefits provided. What follows is a short list of scenarios you could encounter; in all of them, external routines should figure as viable options in your decision.

Speed

You are doing major calculations or loop operations, and are finding it to be slow going.

Complexity

Difficult calculations with many if-tests and function shells can often be done simpler (and quicker) with an external routine.

External equipment

Omnis does not have built-in protocols for communicating with all kinds of peripherals. You must resort to an external routine if, for example, you want to import pictures directly from a video camera.

Graphical pushbuttons

Omnis does not provide predefined pushbuttons of its own, even though you can produce nice effects by placing a pushbutton area on top of a drawing with a button. But among the Omnis examples there is an external routine called 'COOLBUTN.DLL,' which contains graphical pushbuttons that you can enlarge to include your own buttons. Check this one out!

Windows API functions

The function library in Omnis is not as extensive as the API library in Windows. If you wish to make use of functions from here, you must create an external routine to get to them. Typical examples of API functions not found in Omnis are the treatment of .INI files, drawing on screen, and the startup of .HLP files.

Drawing on screen

The drawing of (for example) graphs using Omnis alone is a major undertaking. Consequently, many have developed external routines for this very thing, for example in the “PlanInfo” and “Ergohygeon” systems (both described earlier in this chapter). You can use the “Graph-It” library routine included in the Omnis 7 Plus package to draw your own graphs!

.INI files

In Omnis there is no way to store initializing information (i.e. remember the settings for different users each time) in .INI files like other Windows programs do. (Of course, the information can be stored in the datafile, but this entails a lot of extra programming work to determine which user had which settings.) We have worked out a nice little routine for this, which is called by specific parameters. The routine is given at the end of this chapter.

Which Functions in Omnis Are Used in Connection With External Routines?

Omnis provides a total of 6 different functions for dealing with external routines. A brief description follows. These commands are described in detail in the Omnis documentation. The functions are divided into two groups: one for external routines and one for event handlers.

External routines

Call external routine

Calls a specific function (also with parameters) in a specific library, for example 'Call external routine' "MYROUTIN/ShowAnalogClock (#T)." This results in the function ShowAnalogClock in the routine library MYROUTIN being called and carried out. The value of #T (i.e. the computer's system clock) is used as the function parameter.

Call external with return value

Like the function above, but here you can get the function to return a value to a field in Omnis 7, e.g. #S1.

Load external routine

Establishes a permanent link to an external routine library. Can be used if the routine library will be called many times and you want greater speed. When the routine has been loaded the first time, the next 'Call external routine' will go a lot faster.

Unload external routine

Removes a permanent link (established with 'Load external routine') to a routine library.

Event handlers

‘Event handlers’ are external routines that are a bit out of the ordinary. As opposed to an external routine that is called once and then runs to completion, an event handler will lay low and “listen” for special events. If one should occur (for example, the cursor moving from one field to another) a function in the event handler will spot it before the field gets #BEFORE. In such cases you can specify whether you want Windows or Omnis to handle the event.

Load event handler

Loads an event handler into memory and asks it to begin listening for events.

Unload event handler

Deletes any event handlers from memory and ceases to listen.

What Functions May I Use In an External Routine?

In principle there are two types of functions that are accessible from your external routine:

Windows API functions

All the standard functions that are used in traditional Windows programming (a considerable number!). You will find a complete rundown of these in the documentation that accompanied your purchase of a compiler. Some examples of API functions:

CreateWindow (...)

Create a new window.

GetDC (...)

Get the DC (Device Context) of a window.

EnableWindow (...)

Enable / Disable a window.

Omnis callback functions

A set of functions Blyth has compiled to enable a “normal” Windows library (DLL) to communicate with an Omnis application. Most of these functions will be familiar to you from traditional Omnis programming. Which functions are available, and which parameters they need is all explained in the documentation for external routines that is part of Omnis 7 Plus. Some examples of callback functions:

DoListOp (...)

Execute a list of commands.

DoPrepare (...)

Run a ‘Prepare for Edit / Insert.’

ShowWorking (...)

Shows an Omnis “Working” message.

What Does the Code in an External Routine Look Like?

It's hard to be specific about what the code in an external routine ought to look like (depending, as it does, on the programming language, functionality, and the number and type of functions); nevertheless there are certain rules you must follow.

There are two functions that are obligatory. No matter what type of external routine you are creating, the content of these two functions should remain very nearly the same each time.

LibMain

The first one is called 'LibMain()' and is essential if Omnis is to be able to establish and maintain contact with the external routine.

```
int FAR PASCAL LibMain (HANDLE hInstance, WORD wDataSeg, WORD
...cbHeapSize, LPSTR lpszCmdLine)
{
    if ( hLibInstance == NULL )
        hLibInstance = hInstance;
    if (cbHeapSize != 0 )
        UnlockData (0);
    return (hLibInstance) ? TRUE : FALSE;
}
```

WEP

The second function, 'WEP(),' is actually a mirror image of 'LibMain().' This function sees to it that the internal memory is cleared after the external routine has finished running.

```
int FAR PASCAL WEP (int bSystemExit)
{
    return 1;
}
```

You also need at least one function that can be called from Omnis. This function also has a specific "shell," but the content will vary each time.

```
void FAR PASCAL ReadInteger (int mode, HANDLE far *ref, HWND
mWind,...
HWND tophwindow, HWND instance, FARPROC callback)
{
    switch (mode)
    {
        case ext_load:
            ;; The DLL is called using the 'Load external routine' command.
            < Your commands>
            break;

        case ext_unload:
            ;; The DLL is called using 'Unload external routine.'
            < Your commands >
            break;

        case ext_call:
            ;; The DLL is called using 'Call external routine (with return value).'
            < Your commands >
            break;
    }
}
```

NOTE: Functions that are called from Omnis (e.g. the function above, for example) must be exported in the DEF file so that Omnis

knows the name of the functions in the DLL! (See the accompanying example.)

In addition to the functions above, you are free to create your own within the external routine, just as you would in regular programs.

The Stack Problem

Those of you who are familiar with the problems involved in handling memory (computer memory, that is!), also know that the size of the stack and the heap are important factors in external routines.

External routines have a rather special way of dealing with stacks. In regular programs, the size of the stack is apportioned at startup. External routines (DLLs) are less fortunate; they are obliged to share the stack with the program from which they are called.

For us programmers, this can cause some problems with, among other things, the declaration of variables and the call of certain API functions. The nature and extent of the problem varies, depending on what's wrong, and can include everything from strange output to a total crash of Omnis.

So you should be careful about which functions you use (try to avoid C-functions from 'stdio' and 'stdlib,' for example), and about how variables are declared (don't use static variables).

What If I Want to Know More?

Neither Rome nor Omnis were built in a day, so don't be surprised if it takes you a while to get the hang of external routines.

This chapter has given you a bare-bones understanding of what external routines are, how you use them, and a few simple examples of how to make them.

If you can see the usefulness of external routines and should wish to learn more about them, we have a few suggestions:

1. Read Blyth's external routines documentation.
2. Check out the examples that are done in the Omnis 7 Plus package. They contain a lot of good code you can use as a point of departure for your own routines.
3. If you get stuck, get in touch with your dealer. There's always someone there who can help you; this will also help us to keep abreast of difficult situations and pass the benefits on to others in the same boat.

Good luck!

Difficult Words

API

Application Programmers Interface. A set of defined functions available for a special program, for example Windows.

C

Programming language. The most used language for the development of Windows-based software. All examples of external routines that are packaged with Omnis 7 for Windows are written in C.

C++

An object-oriented programming language. Used, among other things, for the development of Windows-based software. Omnis 7 itself was written in C++!

Callback functions

Collective name for those Omnis functions you may use in your external routines. These functions are described in a separate document that is packaged with Omnis 7 Plus.

DLL

Dynamic Link Library, a collection (library) of external routines in a file.

External areas

A new type of field included in Omnis 7, v1.2. The fields are partly controlled by Windows (an external area handler) and partly by Omnis. Areas of use include graphical presentations, specially constructed fields, live video, and multi-media applications.

External routine

A function programmed (in a 3rd-generation language) with a view to enhancing Omnis's functionality.

Event

The event during the running of a program, e.g. pressing the left mouse button, maximizing a window, etc.

Event handler

A routine that spots events and processes them. Events that are not processed are returned to Omnis to be handled there.

.INI

A type of file used in Windows to remember a specific program setup. The files are stored, as a rule, in the Windows catalogue (as pure text files), and you can view the contents in, for example, Notepad.

Compiler

A program for translating the code we have written (source code) to a language your computer understands (object code).

LibMain()

Library Main, an obligatory function in an external routine for Windows.

PASCAL

A programming language. Can be used to develop Windows-based software and external routines.

WEP()

Windows Exit Procedure, an obligatory function in an external routine for Windows.

An Example with Source Code

We have chosen to conclude this short introduction into external routines with a complete code example written with Borland C, v3.1. The code on the following pages is also available on disk from AlphaBit (Norway), or Blyth; this will save you from having to type it all in yourself.

Our code example is called 'AB_INI' ; it is used in the manipulating of .INI files in Windows.

CALLBACK.C	Omnis callback functions
CALLBACK.H	Definition of Omnis callback functions
AB_INI.C	Source code for separate functions for the manipulating of .INI files
AB_INI.DEF	Definition file with export of functions
AB_INI.DLL	A compiled external routine
AB_INI.PRJ	Project file for Borland C, v3.1
AB_INI.APP	The application

See the listing of the files we have created (AB_INI.C and AB_INI.DEF) in the pages that follow.

AB_INI.C

```
/******
```

AB_INI.DLL - External routine for handling .INI files in Windows.
This DLL can be called from Omnis by the following 4 functions:

```
ReadInteger()  
ReadString()  
WriteInteger()  
WriteString()
```

See a description of the various functions and their related parameters below.

VERSION 1.1 March 28, 1993, Amund Haldorsen

```
*****/
```

```
#include <windows.h>  
;; Standard Windows API functions; must be included.  
#include "callback.h"  
;; Omnis callback functions for external routines
```

```
/******Global pointer and variables******/
```

```
HANDLE hLibInstance;  
;; Library Instance, a pointer to the external routine.  
char szFilename[100];  
;; Variable for the name of the .INI file.  
char szSectionName[100];  
;; Variable for [SECTION_NAME] in the .INI file.  
char szKeyfield[100];  
;; Variable for KEY_FIELD= in the .INI file.  
char szInitstring[100];  
;; Content of the initializing string if the .INI is empty.  
char szValue[100];  
;; The value read from or written to the .INI file (text).  
int nInitvalue;  
;; The content of the initializing value if the .INI file is empty.  
int nValue;  
;; The value to be read from or written to the .INI file (number).  
int get_parameter_info(FARPROC callback);
```

;; Function for ascertaining file name, section name and key name on the basis of the parameters sent from Omnis.

```
/*=====
ReadInteger()
```

This function reads in an integer from an .INI file.
The function has the following four parameters:

szFilename	The name of the .INI file.
szSectionName	The name of the section in the .INI file. [SectionName]
szKeyfield	The name of the key field from which the value is to be read.
nInitvalue	The value to be returned if this line doesn't exist in the .INI file.

NB! This function must be called using the 'Call external with return value' command, where you designate the field to which the value is to be returned.

```
===== */
void FAR PASCAL ReadInteger (int md, HANDLE far *ref, HWND mWind,
...HWND topwindow, HWND instance, FARPROC callback)
{
    switch (md)
    {
        case ext_load:
            ;; Load external routine; not in use.

        case ext_unload:
            ;; Unload external routine; not in use.

            break;

        case ext_call:
            ;; Call external routine.

            if (GetFldInt(ref_parmcnt, callback) != 4)
            {
                MessageBox (mWind, "Invalid numbers of parameters!
\n\nReadInteger ... (filename, section, key, standard value)",
"AB_INI.DLL -ReadInteger()",
...MB_OK|MB_ICONINFORMATION);
                return;
            }
    }
}
```

```

;; Test to see whether four parameters have been given in the call to the
external routine. If not, give error message and cancel the external
routine.

}
get_parameter_info(callback);
;; Select file name, section and key field from among the parameters.
nInitvalue = GetFldInt (ref_parm4, callback);
;; Read the standard value from parameter no. 4.
nValue = GetPrivateProfileInt ((LPSTR)szSectionName,
(LPSTR)szKeyfield, ...nInitvalue, (LPSTR)szFilename);
;; Read in value from the .INI file, or insert the standard value if the .INI
file is empty.
SetFldInt (ref_returnval, nValue, callback);
;; Return the retrieved value to the correct field in Omnis.
}
}

```

```

/*=====
ReadString()

```

This function reads in a string from an .INI file. The function has the following four parameters:

szFilename	The name of the .INI file.
szSectionName	The name of the section in the .INI file
[SectionName].	
szKeyfield	The name of the key field from which the value
	is to be read.
szInitstring	The string to be returned if this line doesn't exist
	in the .INI file.

NB! This function should be called using the 'Call External with return value' command, where you designate the field to which the value is to be returned.

```

=====*/
void FAR PASCAL ReadString (int md, HANDLE far *ref, HWND mWind,
HWND ...topwindow, HWND instance, FARPROC callback)
{
    switch (md)
    {
        case ext_load:
            ;; Load external routine; not in use.

```

```

        case ext_unload:
;; Unload external routine; not in use.
            break;
        case ext_call:
;; Call external routine.

if (GetFldInt(ref_parmcnt, callback) != 4)
{
    MessageBox (mWind, "Invalid numbers of parameters!
        ...\\n\\nReadInteger(filename, section, key,
        ...standard value)", "AB_INI.DLL -ReadString()",
        ...MB_OK|MB_ICONINFORMATION);
    return;
}
;; Test to see whether four parameters have been given in the call to the
external routine. If not, give error message and cancel the external
routine.

get_parameter_info(callback);
;; Select file name, section and key field from among the parameters.

GetFldVal (ref_parm4, fmt_cstring, 100, szInitstring, callback);
;; Read the standard value from parameter no. 4.

GetPrivateProfileString ((LPSTR)szSectionName, (LPSTR) szKeyfield,
... (LPSTR)szInitstring, (LPSTR)szValue, 100, (LPSTR)szFilename);
;; Read in value from the .INI file, or insert the standard value if the .INI
file is empty.

SetFldVal (ref_returnval, fmt_cstring, szValue, callback);
;; Return the retrieved string to the correct field in Omnis.
}
}

```

```

/*=====
WriteInteger()

```

This function reads in an integer from a .INI file. The function has the following four parameters:

szFilename	The name of the .INI file.
szSectionName	The name of the section in the .INI file
[SectionName].	
szKeyfield	The name of the key field to which the integer is
to be	written.

nValue The integer to be written to the .INI file.

NB! This function should be called using the 'Call external routine' command in Omnis.

```
=====*/
void FAR PASCAL WriteInteger (int md, HANDLE far *ref, HWND
    mWind, HWND topwindow, HWND instance, FARPROC callback)
{
    switch (md)
    {
        case ext_load:
            ;; Load external routine, not in use.

        case ext_unload:
            ;; Unload external routine, not in use.

            break;

        case ext_call:
            ;; Call external routine.

            if (GetFldInt(ref_parmcnt, callback) != 4)
            {
                MessageBox (mWind, "Invalid number of
parameters!\n\nWriteInteger...      ... (filename, section, key,
value)", "AB_INI.DLL- WriteInteger()",
                ...MB_OK|MB_ICONINFORMATION);
                return;
            }
            ;; Test whether there are four parameters in the call to the external
            routine. If not, display error message and cancel the external routine.

            }

        get_parameter_info(callback);
        ;; Select file name, section and key field from parameters.

        nValue = GetFldInt (ref_parm4, callback);
        wsprintf ((LPSTR)szValue, "%d", nValue);
        ;; Read in the number from parameter 4, and convert it to a string.

        WritePrivateProfileString ((LPSTR)szSectionName,
            (LPSTR)szKeyfield, (LPSTR)szValue, (LPSTR)szFilename);
        ;; Write the string containing the number to the .INI file.

    }
}

/*=====
WriteString()
```

This function writes a string to an .INI file. The function has the following four parameters:

szFilename	The name of the .INI file.
szSectionName	The name of the section in the .INI file
[SectionName].szKeyfield	The name of the key field to which the value is to be written.
szValue	The string to be written to the .INI file.

NB! This function should be called using the 'Call external routine' command in Omnis.

```

===== */
void FAR PASCAL WriteString (int md, HANDLE far *ref, HWND
    mWind, HWND topwindow, HWND instance, FARPROC callback)
{
    switch (md)
    {
        case ext_load:
            ;; Load external routine; not in use.

        case ext_unload:
            ;; Unload external routine; not in use.

            break;
        case ext_call:
            ;; Call external routine.
            if (GetFldInt(ref_parmcnt, callback) != 4)
            {
                MessageBox (mWind, "Invalid numbers of
parameters!\n\nWriteString      ... (filename, section, key,
standard value)", "AB_INI.DLL -WriteString()",
                ...MB_OK|MB_ICONINFORMATION);
                return;
            }
            ;; Test to see whether four parameters have been given in the call to the
            external routine. If not, give error message and cancel the external
            routine.
        }

    get_parameter_info(callback);
    ;; Select file name, section and key field from among the parameters.

    GetFldVal (ref_parm4, fmt_cstring, 100, szValue, callback);
    ;; Read the value to be stored from parameter no. 4.

    WritePrivateProfileString ((LPSTR)szSectionName,
        (LPSTR)szKeyfield, (LPSTR)szValue, (LPSTR)szFilename);
    ;; Write the string to the .INI file.

```

```

    }
}

```

```

/*=====
get_parameter_info()

```

This function selects the content of the first three parameters to be sent to the functions handling the .INI file. The parameters are:

```

szFilename           The name of the .INI file.
szSectionName        The name of the section in the .INI file
[SectionName].
szKeyfield,          The name of the key field under the section.
=====*/

```

```

int get_parameter_info (FARPROC callback)
{
    GetFldVal (ref_parm1, fmt_cstring, 100, szFilename, callback);
    GetFldVal (ref_parm2, fmt_cstring, 100, szSectionName, callback);
    GetFldVal (ref_parm3, fmt_cstring, 100, szKeyfield, callback);
    return 1;
}

```

```

/*=====
LibMain()

```

Library Main. This function must be included in all external routines. It establishes a kind of link between Omnis and itself and manages the pointers of the various segments.

```

=====*/
int FAR PASCAL LibMain( HANDLE hInstance, WORD wDataSeg, WORD
    cbHeapSize, LPSTR lpszCmdLine)
{
    if( hLibInstance == NULL)
        hLibInstance = hInstance;

    if(cbHeapSize != 0)
        UnlockData(0);

    return (hLibInstance) ? TRUE : FALSE;
}

```

```

/*=====

```

WEP()

WindowsExitProcedure. Must be included in the external routine. This function "cleans up" in the memory after the external routine has been called and carried out.

```
=====*/  
int FAR PASCAL WEP (int bSystemExit)  
{  
    return 1;  
}
```

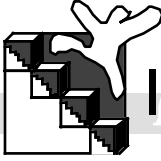
AB_INI.DEF

LIBRARY AB_INI
EXETYPE WINDOWS
STUB 'WINSTUB.EXE'
DESCRIPTION 'AB_INI.DLL - The handling of .INI files from Omnis 7'

CODE MOVEABLE DISCARDABLE PRELOAD
DATA MOVEABLE SINGLE PRELOAD
HEAPSIZE 1024

EXPORTS

ReadInteger @1
ReadString@2
WriteInteger @3
WriteString @4
LibMain @5
WEP @6 RESIDENTNAME



Introduction to Notation

What is Notation?	2
Some areas of use	
Notation in Window formats	
The Branched System	4
The crude command structure	
The different parts	
The relation between the attributes	
How to Write Notational Expressions	10
Where to put them?	
What can we derive from a notational expression?	
Altering the values of attributes	
Abbreviating long expressions	
“Current” attributes	
Square brackets	
Syntax and Debugging	18
Notations can be cranky	
“Context-sensitive” help for attributes and objects	
Continuous testing of notational expressions	
Error handler procedures	
Windows and Notation	25
Background objects and other objects	
Open and closed windows	
A word about colors	

What is Notation?

Version 2.0 of Omnis introduced a whole new system of commands called “metatools” (or notational commands). There are many of them – 398 to be exact – and there is a special way of writing them. What they all have in common is the fact that they allow the developer to give commands in his procedures that once had to be given directly using Omnis menus and various dialog boxes. An example of such a command is ‘\$hasborder,’ which can be used to determine whether a field shall have a frame or not. In earlier versions this was something we had to set when building the windows; after that it couldn’t be changed unless we returned to Design mode. This is no longer the case.

Some areas of use

In principle it is now possible to get procedures to do everything that can be done in Omnis, including the generation and modification of all kinds of formats. In fact, this is what notation is all about. What advantage is this to us? Well, among other things, we can allow procedures to do things that the developer would otherwise do himself. By and large, procedures are a whole lot quicker than we are, no matter how adept we may be with the mouse. We can get procedures to alter our applications for us, resulting in applications that resemble CASE tools, or we can create our own “templates” for regular formats.

Developer’s tools and automated programming

By adding new “tools” we can use notation to broaden the scope of the Omnis interface. These tools are based on the commands that already exist and greatly enhance functionality. This will help the developer improve the efficiency of all kinds of routine tasks. It’s a tempting thought to devise procedures that set Omnis just the way we want – for example, with respect to ‘Help’ options, ‘Long/mixed field names,’ etc. Moreover, we can use notation to make our own variations of standard windows, or to edit existing windows in some specific way. Some of you may

recall Omnis Express; it's now possible to create a similar application without the aid of external routines.

Forbidden and inaccessible areas

The notational commands give us an enormous potential for manipulating everything from the data in datafiles to fields in reports and graphics in windows. We can delete and create new file formats, indexes and fields, and modify nearly anything you can name. With power like this, you'd have thought it would be easy to get Omnis to crash; but the notational code is remarkably stable. In short, we can penetrate deeply into areas of programming that were once forbidden. Grasping the scope of the new potential may take some doing, but it's well worth the effort.

Notation in Window formats

With notation, Omnis has made it possible for the developer to program in detail, more like a 3rd-generation tool – particularly as far as windows are concerned. Windows can be more interactive. For example, drawn lines can move, point, change color, create an illusion of 3-D, etc. as a reaction to what the end-user does. Each developer can be inventive and devise clever and impressive ways of communicating with the end-user and lift the user interface to new heights. Much of what once had to be assigned to external routines can now be programmed directly in Omnis. And don't forget that we are no longer prevented from using the mouse, since we now have the functions `MOUSEOVER`, `MOUSEDOWN`, `MOUSEUP`, and 'drag and drop.' Those who've been burning with impossible but bright ideas for their applications have every reason to rejoice. Limitations are largely a thing of the past.

The Branched System

The crude command structure

In a normal programming language we are confronted with a long list of commands that are rather general in nature; to execute a specific action, we have to express it by writing in x number of parameters, and do so in some sort of order that will not always be that easy to remember. In notation, this problem is solved simply and elegantly. First we define *where* we want to do something, and then *what* we want to do. Once having gotten that far, the notational expression is very nearly complete. That's why syntax is often not a problem. As a rule, one constant or value at the end of the command is sufficient for the expression to attain full meaning.

The different parts

A notational expression looks like an incomprehensible series of words separated by a period and sprinkled with a few dollar signs and parentheses. First let's look at the individual words themselves. All the "words" in the notational system can be divided into two categories – roughly speaking: (a) groups and (b) attributes.

Groups

A group represents a collection of related objects (for example, windows, reports, or window fields). Together with the name of the object, this gives us the necessary information concerning *where* we wish to do something – as follows:

```
$windows ;; the group of window formats  
$windows.wCustomer ;; points at the window named  
"wCustomer"
```

(Prl. 1)

The group name is written in the plural, so it always ends in "s." If we wish to go deeper, we can combine groups in order to get at more detailed parts of the application. For example, we might wish to do some-

thing with a standard Pushbutton in a window. If the name is 'Edit,' the entire expression will be:

```
$windows.wCustomer.$objs.Edit
```

(Prl. 2)

Now we have found our way to this very pushbutton and are free to choose from among those commands which lie under Pushbuttons in general. (We could also have identified the Pushbutton with the window field number (#EF), but that would only make the procedures a lot more cumbersome to read.)

Attributes

The word “attribute” has such a broad meaning that many tend to use it willy-nilly. Nevertheless, what we *can* say is that it represents a characteristic that can be read or altered. For example, if we wish to modify the text in the 'Pause' pushbutton, we can write it like this:

```
Calculate $windows.wCustomer.$objs.Pause.$text as  
"Go" (Prl. 3)
```

Here the attribute is \$text; it is set to 'Go' with the help of the 'Calculate' command. In this way we can change a Pushbutton's function and allow the text in the Pushbutton to reflect this. (Read about the difference between \$winds and \$windows before you test this.)

The relation between the attributes

Now that we know the principal individual parts of the notational system, we can turn our attention to the structure. As the examples have illustrated, each object (“item”) has a number of characteristics called attributes, which we can either read or modify. In a window, an attribute might be the color of an object, the coordinates \$top and \$left, the type of object (Pushbutton, List field, etc.), the field name, and so on. And windows themselves have attributes of their own, for example: the type of window (Palette, No frame, Dialog, etc.), title, name, background color, etc.

General attributes

Each group and each object have their own attributes. In addition to the special attributes, a set of standard attributes is included which you should be able to use everywhere, i.e. on any item or attribute. These include the name of the attribute, identifier number, and other general information.

An object is the sum of its attributes

In principle, the system of attributes should be so developed that the sum of the attributes will comprise everything that can be said about an object. In fact, this is very often the case. If you change the attributes, you will get a completely different object (within certain bounds).

The groups

The “groups” are all-purpose terms for objects or attributes that are related and that are used together with the name of the object to indicate the exact part of the application we wish to work with.

The root of the system

The system is constructed in branched levels. At the top is \$root, from which all groups and attributes spring. This is the top of the pyramid. Under \$root we find groups such as the following:

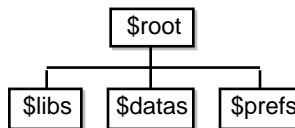


Fig. 1 The root of attributes

Main groups under \$root

The \$libs group contains the various libraries, \$datas is the group of open datafiles, and under \$prefs we find a number of attributes that apply to Omnis as a whole. (There are many more groups under \$root; those we have mentioned here are just a sample.) Let's see what we can find under \$lib:

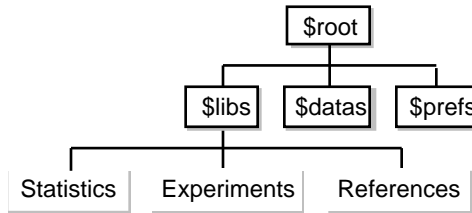


Fig. 2 Open libraries in a sample science application

Groups under a specific library

Now we have to decide which library we're going to work with. If we choose the "Experiments" library, the following group turns up (shown in Figure 3):

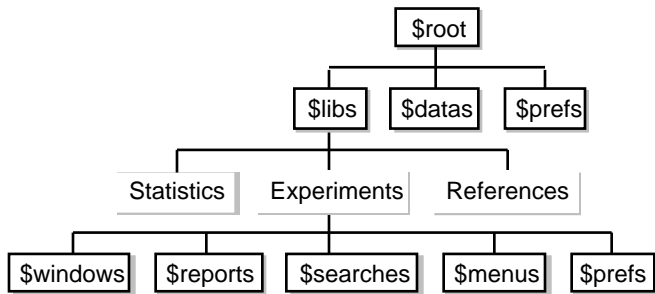


Fig. 3 Groups under a specific library

Afterward, if we wish to peruse one of the reports in “Experiments,” we follow the entire route as shown in Figure 4 below:

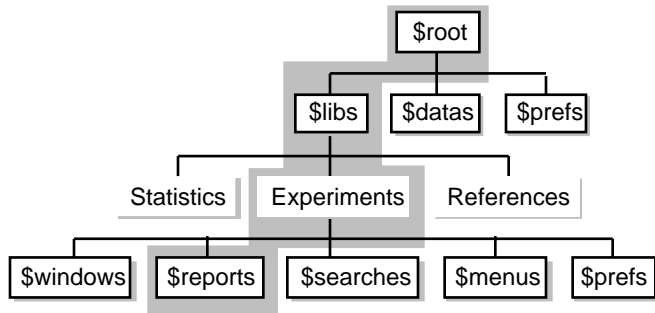


Fig. 4 `$root.$libs.Experiments.$reports`

Choosing a report

Then all we need to do is add the name of the report and continue working our way down in the system until we arrive at the desired attribute. For each level there is a set of attributes, each with its own meaning. Take a look now at the lists of all the attributes in the chapter on notation in “Reference 1.” That might give you some idea of the scope of the branched nature of the system.

The level determines the meaning

Several attributes can have the same name, but they take on different meanings as we place them in different levels. We can see from Figure 4 that the `$prefs` group under `$root` is not the same as that under a specific Library. If we look to see what is under each of these two groups, we will see that the corresponding attributes are totally unlike. The `$prefs` group under `$root` represents the preferences that encompass Omnis as a whole, whereas the `$prefs` group under a library pertain only to the library to which they belong.

Expandability and insight

The notational system is built to be expanded. As new functions are built into Omnis, the notations keep pace. The notational commands are like secret paths leading to the inner sanctum of Omnis: the code itself. By learning notation, you also learn a great deal about how Omnis is put together, and quite a bit about how limited or open-ended the functionality of the individual commands is.

How to Write Notational Expressions

Where to put them?

Technically speaking, notational expressions are the same thing as calculations, which means that we can insert them anywhere calculation fields can be filled in. We can put them in regular window fields, within procedures, in report fields, in search format lines, and in menus – in fact, we can put them virtually anywhere. And considering that it's possible to set off the entire notational expression in square brackets ([]), the list grows even longer. Go ahead and put some notational expressions enclosed in square brackets directly into free text in a window, in menu titles, or in reports.

Simple syntax

All attributes and groups begin with a dollar sign (\$). Between different attributes and the like, we insert a period, just as we separate file names and field names when these occur together. (You will no doubt have already noticed this.)

What can we derive from a notational expression?

A complete notational sentence can yield virtually anything and everything. What we derive depends entirely on what the expression consists of. If the last attribute in the sentence is a Boolean (yes/no) variable, the result will be either yes or no (or blank). This is no different from how we use Boolean variables in general. See Procedure line 4 (Prl. 4):

Calculate #F as \$windows.wCustomers.\$closebox (Prl. 4)

If the window has a 'closebox,' #F will be equal to 1 or Yes. If the last attribute is a text variable, the text will be set accordingly. See Prl. 5:

Calculate #S1 as \$root.\$libs.MyLib.\$pathname (Prl. 5)

For Macintosh, #S1 can be set to “Fat harddisk: Omnis applications:Testing and Experiments: Mylib.” For Windows machines, #S1, for example, can be set to “C:\DOCS\OMNISAPP\ TESTING\MYLIB.LBR.”

Constants

Certain numerical variables have values that are consistent with some of the predefined constants in Omnis. This means there’s a good chance we’ll see a reasonably understandable name for a constant, and not some cryptic number. This is true, for example, of \$objtype, which tells us which type of window field the object in question is. Instead of the number, we can derive the name of the constant by inserting the answer in a text variable, as shown in Prl. 6. Omnis will translate the number value and display the name of the constant without the prefix “k.”

```
Calculate #S1 as  
$windows.wCustomers.$objs.Edit.$objtype (Prl. 6)
```

In this case, #S1 gets the text “pushbutton.” The corresponding constant is called ‘kPushbutton.’

Altering the values of attributes

Most attributes can be altered in a variety of ways. We have already used the ‘Calculate’ command and placed the notational expression where we usually place the field name:

```
Calculate $winds.wPanel.$objs.Pause.$text as "Go" (Prl. 7)
```

The method shown in Prl. 7 is OK, but it has its limitations. Let’s approach it from a slightly different angle:

```
Calculate #F as  
$winds.wPanel.$objs.Pause.$text.$assign("Go") (Prl. 8)
```

Calculated with flag

The technique illustrated in Prl. 8 has a number of advantages. Firstly, the expression will yield a Boolean value, provided the command was successful. (The \$assign attribute is a command signifying “set to.”) In this particular case, the Boolean value ends up in #F. Secondly, the calculation field can be expanded (CMND/CTRL-U), which enables us to see the whole expression at a glance when we type it in. This makes it easier to edit big expressions. Thirdly, this is the only way there is to modify attributes outside procedures, i.e. when the notational expression occurs in an isolated calculation (not as a part of a procedure). As I see it, this will also make the expression clearer because it will not be divided up.

Abbreviating long expressions

Notations have an unfortunate tendency to grow rather long, demanding lots of tedious typing. It’s only near the end of the expression that things start getting interesting. Moreover, it is often the same object being referred to in each notational expression, which means we’ll be writing the same thing over and over again.

Item reference

Here is where ‘Item reference’ comes in. This is a new type of field that can be inserted in file formats or declared as Library variables, Format variables, or Local variables. An Item reference is a pointer that can be used as a legitimate replacement for the entire expression. We can say it works like an abbreviation for a lengthy notational expression. The effect is like inserting the whole expression every time the Item reference is used. The notational expression as a whole will read whatever is already written there, plus the “contents” of the Item reference.

Local variable Lo_PUSHBUTTON

(Prl. 9)

When an Item reference variable has been declared (as shown in Prl. 9), we must decide what the reference shall point to (Prl. 10).

Set reference Lo_PUSHBUTTON to
\$windows.wPanel.\$objs.Pause

(Prl. 10)

Now we can abbreviate the notational expression and quickly add a couple of new commands (Prl. 11–13):

Calculate #F as Lo_PUSHBUTTON.\$text.\$assign("Go") (Prl. 11)

Calculate #F as Lo_PUSHBUTTON.\$forecolor.\$assign(16) (Prl. 12)

Calculate #F as Lo_PUSHBUTTON.\$backcolor.\$assign(8) (Prl. 13)

Omit attribute

It's not always necessary to type in the entire row of groups and attributes. Some of them are apparent, based on the kind of name and subsidiary attributes that are specified. \$root, for example, rarely ever needs to be specified. Generally speaking, it is sufficient to write only as much as it takes to make the expression unambiguous within a specific application. If you find yourself wondering whether a part of an expression can be dispensed with, it's possible to test whether \$scanomit is true or false for that part of the expression. This attribute can be inserted after any other attribute. In Prl. 14, #F is equal to True.

Calculate #F as \$root.\$clib.\$scanomit

(Prl. 14)

“Current” attributes

The notational system also contains a couple of shortcuts in the form of attributes which take the drudgery out of writing an expression. As they all sort under \$root, there is only one of each in any given application. This means there is only one “current” window, one “current” procedure, and so on.

About “Current”

Concerning the attributes described here, ‘Current’ reflects the object the user is in contact with at all times. This needn't have anything to do with where the cursor happens to be blinking; what matters is where

the end-user clicks with the mouse, not where the cursor is positioned.

\$cobj

The Current object is the window field that the user is in contact with in some way or another. In practice, this is frequently the very object that contains the field procedure the developer is working on. Procedure 1 shows how \$cobj may be used to address the field to be highlighted:

Field procedure, Entry field	Procedure 1
<pre>If #BEFORE Calculate #F as \$cobj.\$backcolor.\$assign(1) ;; Set to white Else if #AFTER Calculate #F as \$cobj.\$backcolor.\$assign(9) ;; Back to gray End if</pre>	

The field containing Procedure 1 is an Entry field with a gray background (the window also has a gray background). When the end-user enters the Entry field, the background shifts to white. When he exits the field, the background color reverts to light gray. This is a sterling way of showing where the cursor is.

\$cproc

This attribute points to the procedure that's currently running. It applies to field procedures, menu procedures, and all kinds of control procedures.

\$cwind

The \$cwind attribute refers to the window that contains the object the end-user is in contact with. In most cases this will be the topmost window on screen.

\$clib

The \$clib attribute (Current library) is the library that contains the format owning the field or object the user is in contact with, or the library that contains the procedure being run. In short, it's just the library you would expect to be "Current" library. This attribute is useful when more than one library is in use at a time.

\$dlib

The library that has been set to 'Design library,' i.e. the library the developer is most likely to be working with, is referred to as \$dlib.

Omitting parts of an expression

When no window has been designated, Omnis assumes that the expression applies to \$cwind. The same thing applies if no library in particular has been designated, in which case Omnis uses \$clib. The attribute \$canomit for those groups that are not written is actually not 'true'; the way in which Omnis interprets the expression nevertheless gives it meaning. So the developer can often, in good conscience, leave out the bulk of the full expression, which will save him a lot of unnecessary typing.

Getting the name of "Current" attributes

Let's look at a case involving some unusual syntax that many developers might have problems with. If, for example, we wish to know the name of the object that is "current," we must write the notational expression as follows:

Calculate NAME as \$cobj().\$name (Prl. 15)

Please note the parentheses after \$cobj in Prl. 15. It might seem that the natural thing would have been to leave them out; but we can't, and that's that. This way of writing also applies to the other "Current" attributes, i.e. \$clib, \$dlib, \$cdata, \$cwind, etc.

Square brackets

Variables and calculations set off by square brackets are accepted anywhere in the expression. This is one means of using the same notational expressions in a number of different situations. It is also a means of allowing us to use the same expression to manipulate more than one object (i.e. various fields or formats). Everything within square brackets will be “carried out” and the result put into the notational expression before the notation itself is performed.

Dissimilar object names

It is possible to insert a variable in square brackets instead of the name of window fields, format names, or the like, such as:

```
Calculate #F as  
$winds.[#TOP].$head.$title.$assign("Topmost!")
```

(Prl. 16)

In the rather supercilious example in Prl. 16, the window that lies at the top of the stack of open windows on screen is given the title “Topmost!”

Dissimilar attributes and values

If we want different actions based, for example, on the user’s selections, one variable can be made to contain the attribute’s name, and another variable what the attribute is to be set to.

Change action depending on value of #1

```
Local variable Lo_Attribute (Character)  
Local variable Lo_Value (Short integer)  
  
Switch #1  
  Case 1  
    Calculate Lo_Attribute as "$backcolor"  
    Calculate Lo_Value as 3  
  
  Case 2  
    Calculate Lo_Attribute as "$linestyle"  
    Calculate Lo_Value as 12
```

2

```
Default
    Calculate Lo_Attribute as "$forecolor"
    Calculate Lo_Value as 1
End switch

Calculate #F as $obj.[Lo_Attribute].$assign(Lo_Value )
```

Change action depending on value of #1

(2)

In this example, we imagine that the #1 variable is set in accordance with the user's actions. Then, when #1=1, the background color is set to 3 (= red by default); when it is 2, the line style is set to 12 (= a dash variant); and when it is 3, the foreground color is set to 1 (= white by default). The main point of Procedure 2 is that the notational expression can change function completely, depending on the values in Lo_Attribute and Lo_Value.

Syntax and Debugging

Notations can be cranky

One of the problems with notational programming lies in determining how to express what we want. This isn't always so easy, notwithstanding the strictly logical way in which the system is built up. Nor do we get all that much help from Omnis when the notational expression fails. There are many expressions which seem perfectly fine and yet don't accomplish what we want them to. Also, there are times when an otherwise acceptable notational expression just won't work, because as you test your application the way the user will encounter it, the settings the expression depends on will have changed. A typical example is the fact that a window is not regarded as "open" when in Design mode, and any references to \$wind (i.e. "current open window") will not be valid.

In the event of errors

A notational error will not cause the procedure to stop; the procedure will merely ignore the notation and continue to run. This is a positive thing as far as the stability of the procedure is concerned, but it makes debugging that much more difficult.

"Context-sensitive" help for attributes and objects

We aren't completely left to our own devices, however. Built into the notation is a comprehensive system of helps, where we can glean important information about attributes in measured doses. In many cases this will eliminate the need to consult the user's manual (or the Omnis Help window that briefly explains the Omnis commands).

\$desc

Accompanying every attribute is a short string of text that defines what the attribute stands for. This serves well as a kind of reminder, but it won't help you all that much unless you are already familiar with the

function. In Prl. 17 #S1 receives the description of \$closebox:

```
Calculate #S1 as  
$windows.wCustomers.$closebox.$desc
```

 (Prl. 17)

In this case, #S1 becomes: “True if the window has a close box or system menu.”

\$desc and \$add

When confronted with a \$add operation, most people wind up scratching their heads. They can’t quite remember what information was required for the particular group they wanted to place an object in, let alone in what order the information was to appear. Then it helps to describe the syntax like this:

```
Calculate #S1 as $cwind.$bobj.$add.$desc
```

 (Prl. 18)

After this, #S1 will contain the information you need for adding a new background object in an open window:

```
“$add(type,top,left,height,width,invisible,disabled) or  
$add(object) adds a new field or object to the window.”
```

\$cando

If you find yourself wondering whether it’s possible to realize a given notational expression, you can tack on a \$cando at the end. If all goes well, the answer should be ‘kTrue.’ But keep in mind that the circumstances should be as similar as possible under Design mode and when the application is running. The difference can cut both ways. For example, the notation may function well in Design mode but not when the application is running normally; but the opposite can also be the case. Later on we’ll take a look at a more reliable, less on-again-off-again method for testing notation.

What attributes are present?

At every level of the notational hierarchy there is a set of standard attributes, special attributes, and (where applicable) standard group attributes. These are all carefully described in the chapter on notation in “Reference 1.” But it can be useful to have a more direct overview, without having to look things up in the manuals. Procedure 3 fits the bill.

(Remember that \$att(n) is a pointer for attribute number “n,” \$attcount is the number of different attributes that may be used with the expression in question, and \$name is the name of the attribute.)

Make attribute list

```
Local variable Lo_No_Of_Attributes (Long integer)
Local variable Lo_Att_No (Long integer)
Local variable Lo_Name (Character)
Local variable Lo_Description (Character)
Local variable Lo_Notation (Character)
Local variable Lols_Attributes (List)

Calculate Lo_Notation as "$root"

Calculate Lo_No_Of_Attributes as [Lo_Notation].$attcount
Set current list Lols_Attributes
Define list {Lo_Name,Lo_Description}

For Lo_Att_No from 1 to Lo_No_Of_Attributes step 1
    Calculate Lo_Name as [Lo_Notation].$att(Lo_Att_No).$name
    Calculate Lo_Description as [Lo_Notation].$att(Lo_Att_No).$desc
    Add line to list
End For

Breakpoint {;; Lols_Attributes}
```

3

Compiling a list of attributes

(3)

The above is a fairly good general procedure for this purpose. Insert the desired notational expression in Lo_Notation. In Procedure 3 this is set to “\$root,” but you can alter it to suit yourself. Since the procedure uses its own local list (so as not to interfere with other

lists in the application), it must be halted before it has finished running, otherwise the list will disappear from memory. A ‘Breakpoint’ command has been inserted in the last line for this very purpose. Using OPT/RB, click on LoLs_Attributes in the line with the ‘Breakpoint’ command, and you may use the Popup menu to view the list contents.

Continuous testing of notational expressions

Earlier we used #F to check whether a notational expression had been successfully carried out. If we are to test for each line (which is a smart way to start out), the procedures will be inordinately long when we check the flag in this way. In fact, it will mean using three whole lines for each test, as shown in Procedure 4:

Test notation
Calculate #F as \$cwind.\$closebox.\$assign(kTrue) If flag false Breakpoint ;; In case of an error, the procedure stops here. End if

4

In any advanced application there are likely to be a good many notational expressions, and umpteen variations of the above procedure can be rough going indeed. But we can do something really clever instead:

1. Declare your own personal WASDONE.

Define a global Boolean variable in a file format or as a Library variable:

Library variable WASDONE (Boolean)

(Prl. 19)

(The name is immaterial; we could have called it SUCCESS, EXECUTED, FLAG, NOTATION, ASSIGN, or something similar.)

2. Put it in the ‘Break calculation.’

Then click with OPT/RB on the variable and select ‘Set calculation’ from the popup menu. (The calculation in question is the Break calculation.) We set it as shown in Prl. 20:

WASDONE=not(kTrue)

(Prl. 20)

The ‘Break on calculation’ option is turned on automatically, which gives us an “active” alternative to the cumbersome test shown in Procedure 4. In the notational expressions, we use WASDONE instead of #F. If the execution of the notational expression is not successful, WASDONE will equal ‘Empty’ and Omnis will stop and show you the right procedure! The new version of procedure 4 consists of only one line:

Calculate WASDONE as \$cwind.\$closebox.\$assign(kTrue) (Prl. 21)

Thus no further testing of flag is necessary. When the application has been successfully debugged, we can turn off ‘Break on calculation.’

Having continuous testing as a standard

Those who like this method might perhaps wish it were turned on automatically at each work session. Procedure 5 will execute what was just explained in the foregoing paragraph.

Initialize active notation testing

Library variable WASDONE (Boolean)
Set break calculation on WASDONE {WASDONE=not(kTrue)}
Field menu command: Set Break On Calculation

5

Automatic initialization

(5)

We can call this procedure when we boot the system. If your application already has password protection, you can run this procedure, for example, if #UL=0 (i.e. if “Master password” has been given). ‘Master user,’ as default, has all rights, so more than likely, ‘Master

user’ is the developer himself – in person! Thus you can make things ready for development work when the developer logs on, and prepare the way for running the application as usual at all other user levels.

Error handler procedures

The system of ‘Error handler’ procedures in Omnis also applies to notation. If the notation contains an error answering to one of the error messages here, it will be in the hash variable #ERRTEXT and the code number will be in #ERRCODE. One way to exploit this is to use your own Error handler procedure. This is an otherwise normal procedure that is called every time #ERRCODE is greater than 0 (null = no errors), alternatively within the error code range as specified when the ‘Load error handler’ command is used. These variables are evaluated for each command or action in your procedures and, as such, quickly disappear as far as the developer is concerned. Therefore we must “grab” the content just after the error has occurred, either in the next procedure line (which is tiresome in the long run) or with the aid of an Error handler procedure.

We decide which procedure is to be our Error handler procedure with the command ‘Load error handler’:

Load error handler STARTUP/500 {Error handler procedure}(PrI. 22)

As indicated in the procedure address in PrI.22, we have decided to put the procedure at the end of the STARTUP menu format. We may write the procedure itself in the following way:

Error handler procedure

```
Local variable Lo_ErrorMessage(Character)
Local variable Lo_ErrorCode(Character)

Calculate Lo_ErrorMessage as #ERRTEXT
Calculate Lo_ErrorCode as #ERRCODE
Breakpoint
```

6

Error handler procedure

(6)

If an error crops up in the procedure line, the Error handler procedure is run immediately. #ERRTEXT winds up in Lo_ErrorMessage, and #ERRCODE winds up in Lo_ErrorCode. The 'Breakpoint' command at the end enables us to read the local variables (i.e. they keep them from vanishing from memory); in addition, the address of the procedure line that caused the error will now appear in the 'Stack' menu.

Windows and Notation

We conclude this chapter by taking a closer look at windows and notation. In many ways, windows have enjoyed most of the advantages of the notational system. Although not everything is self-evident, you needn't despair! After a few points are clarified, a whole new universe of possibilities will open up and hopefully all you enthusiastic developers out there will start burning the midnight oil.

Background objects and other objects

Most of what is stored in a window format can be found in the following groups: \$objjs, \$bobjs, and \$procs, each of which have their own objects and their own personalized attributes.

\$objjs

The \$objjs group contains all Entry fields, Check boxes, Pushbuttons, Tables, Lists, and other normal fields we would expect to see in the foreground of a window. They can be identified by name (\$name), which is the text we find in the procedure title in the procedures that lie under the fields. (The text shown in the window isn't that important.) These objects can also be identified by their field number, which is handy if, for example, you know that the name itself is bound to change sooner or later.

\$bobjs

Background objects are all the rectangles, lines, free text, pictures, etc. that lie directly "on" the window. They are identified by their number. This number is assigned only once (just like the RSN) and is not reused when a background object is deleted. The numbers start at 1001. The easiest way to see them is to make a list with \$name and \$objtype, perhaps the coordinates (\$top, \$left) and other attributes as well, as shown in Prl. 23:

Calculate #L1 as...

```
...$windows.wWindow.$bobjs.$makelist($ref.$name,  
...$ref.$objtype,$ref.$top,$ref.$left,$ref.$width,$ref.$h  
eight)
```

(Prl. 23)

Let's say we discover that the rectangle we want to be green is no. 1032. The notation will be as shown in Prl. 24:

Calculate WASDONE as...

```
...$winds.wWindow.$bobjs.1032.$forecolor.$assign(4)
```

 (Prl. 24)

\$procs

In the \$procs group we find the procedures that are behind the fields in the window. Each procedure is identified by its title or field number (which is also the procedure number). The example shown in Prl. 25 copies Procedure 0 in the 'wWindow' over to #S1.

Calculate #S1 as \$windows.wWindow.\$procs.0.\$proctext

 (prl.25)

A word about numbering and identifying

We already know that normal foreground objects can be identified by their field number (#EF) or name. The field name is also the name of the corresponding procedure, which means that the field name can be used to locate the procedure under the surface.

There is, however, a certain overlapping between the \$objs group and the \$bobjs group. If we refer to an object in the \$objs group by a field number greater than 1000, this is like pointing to a background object. Likewise, objects in the \$bobjs group with numbers smaller than 1001 will be interpreted as foreground objects. Now let's try to change the foreground color of a foreground field with the field number 5 and afterward do the same thing with a background field with the identification number 1023:

Front object, field number 5:

Calculate #F as \$cwind.\$objs.5 .\$.forecolor.\$assign(4) (Prl. 26)

Calculate #F as \$cwind.\$objs.5 .\$.forecolor.\$assign(4) (Prl. 27)

Background object, field number 1023:

Calculate #F as

\$cwind.\$objs.1023 .\$.forecolor.\$assign(4) (Prl.28)

Calculate #F as

\$cwind.\$objs.1023 .\$.forecolor.\$assign(4) (Prl.29)

(Here, Prl. 26 means the same as Prl. 27, and Prl. 28 means the same as Prl. 29.)

\$ident

It is important to keep the field numbers separate from the \$ident numbers, which follow another system. Omnis uses the latter in every group of attributes. They are assigned only once and start at 1001, just like the numbering of background objects. We get this identification number by using the attribute \$ident, as shown in Prl. 30:

Calculate #1 as \$windows.wTest.\$objs.Find.\$ident (Prl. 30)

So #1 contains the identification number of the object 'Find.' Here the name of the field is "Find," just like the name of the procedure underneath. Imagine that the field number (#EF) is 7 and that \$ident is 1014. If we want to identify an object with the aid of the \$ident number, we have to use the '\$findident' command. In Prl. 31-33 we use the various ways of identifying the 'Find' object to change its foreground color:

Identified by field name:

Calculate #F as

\$winds.wTest.\$objs.Find. \$.forecolor.\$assign(4) (Prl. 31)

Identified by field number (#EF):

Calculate #F as

\$winds.wTest.\$objs.7 .\$.forecolor.\$assign(4) (Prl. 32)

Identified by \$ident number:

Calculate #F as...

...\$winds.wTest.\$objs.\$findident(1014) .\$.forecolor.\$assign(4) (Prl. 33)

Open and closed windows

In Omnis we have two groups of windows; the one, `$winds`, are windows that are always open; the other, `$windows`, are the window formats, i.e. the collection of “recipes” for all the windows in a library. When a window is opened, Omnis uses the appropriate recipe to build the window such as it will appear on screen. So we can regard a window in the `$windows` group as a kind of list of instructions.

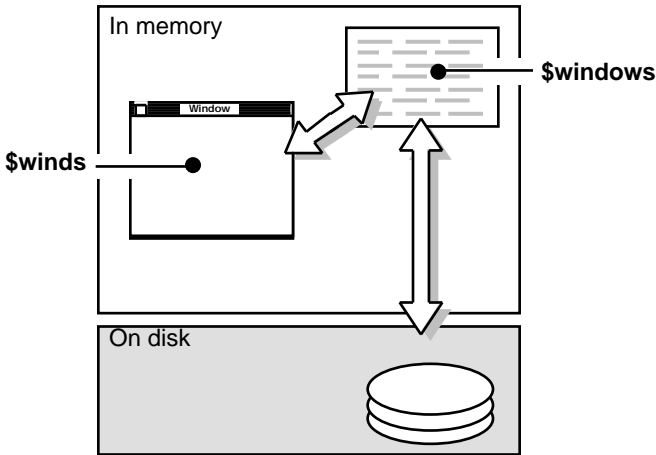


Fig. 5 Open window group (`$winds`) and window format group (`$windows`)

Altering a window's appearance by means of notation

This is one of the most exciting aspects of notation. If we wish to see the changes directly (for example, use a change of color to communicate with the end-user), the notation must work with objects or background objects in an open window. We find the window in the `$winds` group. The changes show up at once, without the need for redraw. If we want to keep the changes, ‘Store window’ must be executed before the window is closed; likewise, ‘Save format’ must be executed before Omnis, the application, or the library in question is closed. Try changing `$linestyle`, `$backcolor`, `$forecolor`, `$font`, `$fontsize`, and

\$backpattern for all the background and foreground objects in the open window. You'll love it!

Modifying procedures in a window

At present, it is not possible to modify procedures in an open window. You must turn to the window format as it appears in the \$windows group. If you wish to see the effect of your changes, you may open the window with the 'Open window' command. And if you want to make the changes permanent, a 'Save window' command must be executed before you quit for the day.

Saving the changes

When the developer applies the command 'Store window,' the open window's appearance will be translated into an appropriate set of instructions, which in turn are stored as the window format. The 'Save format' command saves the window format to disk (but it doesn't effectuate the translation as the 'Store window' command does).

Design mode

When a developer is working with a window in Design mode, this is considered a special state; it is not "open" at this point. You can tell that this is so, because all Pushbuttons and other fields are inactive. The sys(50) function never points to the window the developer is working with in Design mode. Still, other windows may appear to be active in the background; but to make their corresponding procedures run, you'll have to bring one of the windows forward by clicking on it. The Timer procedure, LCPs and WCPs are inactive. The window in Design mode is stored automatically and saved to disk as soon as the developer closes it.

A word about colors

When notation was introduced, numbers 1–16 (which identify the colors in a library) took on added interest. However, it is a hard job

to remember which color goes with which number. It pays to make your own constants, as shown in Procedure 7.

Color constants definitions
Library variable kBlack (Short integer) Library variable kWhite (Short integer) Library variable kLightGray (Short integer) Library variable kDarkGray (Short integer) Calculate kBlack as 1 Calculate kWhite as 16 Calculate kLightGray as 9 Calculate kDarkGray as 8

7

Calculate WASONED as
...\$cwind.\$bobjs.1056.\$forecolor.\$assign(kDarkGray) (prl.29)

The code gets clearer

In your procedures, the result is clearer code. Note that the values of these constants apply only to the default colors. If the color table is modified in a library, the developer will have to make adjustments by changing the name and value of the color constants.



Notation is the key that unlocks the Omnis treasure chest.

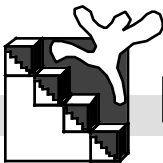


Section 9:

Special Topics

Chapters:

1. Keyboard Shortcuts (Macintosh)



Keyboard Shortcuts

(Macintosh)

Using Keyboard Shortcuts.....	2
Increasing the Number of Potential Hotkeys (v2.x only).....	3
Standard Hotkeys (2.x).....	5
Keys That Are Completely Unused.....	6
Word Processing Techniques.....	7
Hotkeys in v3.0.....	8

Using Keyboard Shortcuts

In Omnis the developer can allow the user to use keyboard shortcuts (hotkeys) for the most frequently used commands. Merely add a letter after a backslash (\) in the title of the menu procedure, and Omnis will interpret and execute these automatically. However, there are already a sizable number of hotkeys with CMND/CTRL combinations in the standard menus. Various restrictions apply: which letters can be used depends on which menus the developer grants the user access to.

Choice of characters

The problem with hotkeys is that they can be hard to remember at first. That's why it's important to choose keys that will be easy to remember, e.g. the first letter of a command. In the 'Commands' menu we already have "I" for 'Insert,' and "E" for 'Edit.' In addition to these tips, use your imagination.

Association with the command

This often means associating the letter (or number) with the command in some way. Most people will remember the first vowel. The number "1" can be used, since it can signify "first." Many will associate "0" with making something equal to 0; or perhaps the fact that "0" comes before "1" can be exploited. The number "9" is the last number in the series of digits, and as such can be made to signify "last." And so on. It's probably a good idea to tell the user why you chose the letters or numbers that you did. In any case, ease of remembrance is a key factor in user-friendliness.

Differing keyboard layouts

Hotkeys only work (in v2.x) on the main character of a key (not the optional character that some keys have). If the end-user has to hold down SHIFT or some other modifying key, the hotkey will not work. Bear this in mind, since different countries have different keyboard layouts.

Increasing the Number of Potential Hotkeys (v2.x only)

Instead of resorting to mediocre hotkeys, you will often find yourself wishing you could use some other modifying key than CMND/CTRL. Alas, it's not all that simple.

#KEY and #SKEY

When 'skeyevents' for a library is turned on, this means that Omnis is keeping a running track of which key was last hit. This key is stored either in #KEY or #SKEY, depending on whether it was a normal key or a special key. In themselves they cannot be used for regular hotkeys, since they receive either a normal or a special key – not a combination of the two. On the other hand, it is perfectly OK to use a special key alone as a hotkey, for example HELP, HOME, END, PAGEUP, PAGEDOWN, etc. The important point here is that the function in question should not be too unlike that suggested by the name printed on the key.

Combination with #OPTION or #SHIFT

One way of obtaining alternative hotkeys is by testing the variables #OPTION/#ALT or #SHIFT in conjunction with #KEY, for example in the Window Control Procedure. The disadvantage with this is that the character that winds up in #KEY will be an alternative character, and not the letter printed on the key that was pressed (for example “•” when hitting OPT- “A”).

List of #SKEY codes

Here is a list of the most frequently used special keys, their #SKEY values, and the accompanying constant:

#SKEY	Constant	Key name
17	kUp	Cursor UP
18	kDown	Cursor DOWN
19	kLeft	Cursor LEFT
20	kRight	Cursor RIGHT
21	kPUp	PAGEUP
22	kPDown	PAGEDOWN
25	kHome	HOME
26	kEnd	END
27	(#TAB)	TAB
28	(#RETURN)	RETURN
29	(#OK)	ENTER (numeric keypad)
30	kBack	BACKSPACE
31	(#CANCEL)	ESC / CLEAR (numeric keypad)
34	kFwdDel	Forward delete
35	-	HELP

The keys where the constants are marked with a hash (#) have a message variable sent when being used, so there is really no constant needed.

Standard Hotkeys (2.x)

To avoid duplication of hotkeys in the menus installed at any given time, we have compiled a list of all the predefined hotkeys in Omnis v2.x. It may help you find good candidates for hotkeys among those characters that are available in your application.

Key	Menu	Command active when...
'	Modify	Uncomment Selected Lines
0	Style	Other sizes...
1	Design	Cycle Formats
2	Design	Formats...
3	Modify	Window
3	Modify	Report window
3	Modify	Set access
4	Modify	Field List
5	Modify	Procedures
6	Modify	Tools to top
7	Modify	Parameters
7	Modify	Show Procedure Names
7	Modify	Window attributes...
8	Modify	Modify Specified Format
8	Modify	Reorder fields
9	Design	List Fields Names
;	Modify	Comment Selected Lines
A	Edit	Select All
B	Style	Bold
C	Edit	Copy
D	Commands	Delete Record
D	Modify	Delete Line
E	Commands	Edit Record
E	Modify	Execute procedure
F	Commands	Find Record
F	Design	Find and Replace
G	Design	Find Again
I	Commands	Insert Record
I	Modify	Insert Line
I	Style	Italic
J	Commands	Insert With Current Values
M	Modify	Install menu
N	Commands	Next Record
N	Modify	Next Procedure Line
N	Modify	Show Narrow Sections
N	Style	Normal

O	File	Open Application	file menu is not replaced
P	Commands	Previous Record	a window is open*
R	Modify	Print Report	editing report format
R	Modify	Show Window Actual Size	editing window format
S	Modify	Save	editing a format
T	Design	Replace Again	design menu is displayed
U	Style	Underline	editing window or report format
V	Edit	Paste	edit menu is displayed
W	Modify	Open Window	editing window format
X	Edit	Cut	edit menu is displayed
Y	Modify	Show Debug Menus	editing procedures
Y	Modify	Show Text Boundaries	editing window or report format
Z	Edit	Undo	edit menu is displayed
[Style	Down (font size)	editing window or report format
]	Style	Up (font size)	editing window or report format
+	Stack	Move up stack	debug menus are displayed
-	Stack	Move down stack	debug menus are displayed

* The 'Show commands menu' window option must be set.

Keys That Are Completely Unused

H, K, L

Word Processing Techniques

There are a number of hotkeys which, when used in text fields, are a boon to word processing, especially with a Powerbook or any other computer with few special keys. Bjørn Kjøseth has cooked up the list that follows.

Key	Modify key	Equivalent	Entry field, Calculation	Lists, Procedure lists
A	CTRL	Home	First	Line 1
A	SHIFT+ CTRL	Shift-Home	Select from cursor through first character	Select from chosen line through first line
D	CTRL	End	Last	Last line
D	SHIFT+ CTRL	Shift-End	Select from cursor through last character	Select from chosen line through last line
H	CTRL	Backspace	Delete previous character	N/A
K	CTRL	PageUp	One page back	One page up
K	SHIFT+ CTRL	Shift-PageUp	Select from cursor and one page back	Select from line and one page up
L	CTRL	PageDown	One page ahead	One page down
L	SHIFT+ CTRL	Shift- PageDown	Select from cursor and one page ahead	Select from line and one page down
C	CTRL	OK	SNA perform an OK (or ENTER)	N/A
I	CTRL	TAB	SNA perform a TAB	N/A
M	CTRL	RETURN	Press RETURN (carriage return in multi-line field)	N/A
->	OPTION		End of following word	N/A
<-	OPTION		Beginning of previous word	N/A

Hotkeys in v3.0

Most limitations concerning hot keys have been done away with in v3.0, which allows us to use any combination of `OPTION`, `SHIFT`, and `COMMAND` (Macintosh) or `SHIFT`, `CTRL` and `ALT` (Windows) as modifying keys directly in menu formats. Until now, we've only been allowed to use `COMMAND` or `CTRL` this way. Thus most of this chapter applies to developers who will be using v2.x.

Section 10: **Beyond the Tricky Bit**

ISBN 82-91465-01-0