

Prepare for Update 2

Once a delicate flower that might wither at the slightest hint of programming impropriety, PFU is now a robust mode that won't quit unless you make it. Now there are new ways of getting into trouble.

Format and Local Variables 6

Subroutines just aren't what they used to be. Format and local variables open up new possibilities for structuring your applications. We'll explore a few.

lookup() Function 12

This is vying for a place among my favorite OMNIS functions. Access to other datafiles is just the beginning!

Storing Lists 16

Lists aren't just temporary repositories for collections of values anymore. We'll explore storage, retrieval, and reporting techniques for Lists stored in the datafile.

OMNIS 7 and SYBASE® 22

SYBASE has some special features not shared with other SQL platforms that make it a great choice if you need a client/server database. Here are the ways OMNIS 7 takes advantage of what SYBASE offers.

SQL Reports 26

There is no direct reporting capability for SQL databases built into OMNIS 7, but there are a number of ways to generate reports on data retrieved from the server through OMNIS.

Tips and Techniques 30

Currency conversion, "fancy" report methods, and much more...

Class Schedule 36

If you need help using OMNIS 7 to its fullest potential and have a limited budget, this course offering is for you. An explanation of courses offered and my schedule through the end of 1992.

Example Disk Program 37

Seeing is believing. These example disks are designed to supplement the articles in OmniScience and supply working proof of the techniques I present.

Products 37

A growing list of OMNIS-based example disks, auxiliary software and publications.

Fax Survey and Order Form 39

This is your opportunity to give me feedback about this issue as well as to order example disks and other products and to register for classes. One handy form for any and all uses.

Doing It the Right Way

by David Swain

OMNIS 7 allows us tremendous flexibility in performing database tasks. There are very few processes that can be performed only one way — and there are often a number of “right” ways to accomplish a task.

There are also many “wrong” ways of using OMNIS that seem to entice many OMNIS programmers. As we are given more tools by the creators of OMNIS, there are as many opportunities to get entangled in a web of

fixes to fixes to misused features we try to implement as there are occasions to do better programming. I hope to help you avoid these pitfalls through this newsletter and the classes that I offer around the country. I believe that most programmers will make sound choices when given good explanations of the tools they are using.

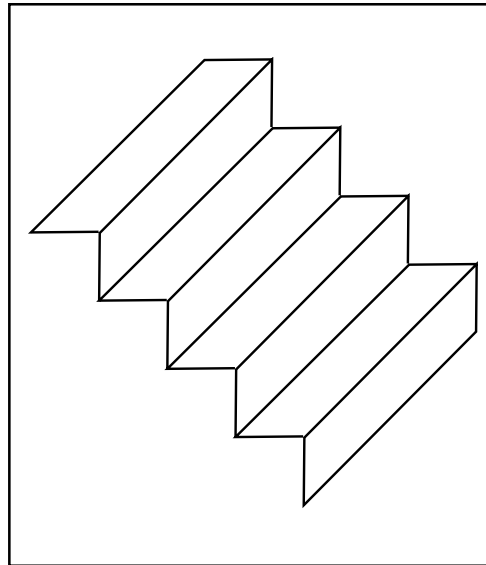


Figure 1 There is often more than one correct way to view or solve a problem in OMNIS 7.

There is an easy path to follow when you know how the various features of OMNIS work, separately and in concert. Designing an application is simply a matter of formulating your solution to take advantage of those features. A little extra time spent exploring possible alternative methods before beginning

to build an application will help you avoid the certain frustration of backtracking — or even having to start over — that you *will* face if you just jump in and start coding.

continued on page 40

Clarifications on Who to Contact

Circumstances surrounding the distribution of the first issue of OmniScience Volume II fostered a number of misconceptions in many of the people who received it. A desire for efficiency prompts me to clarify a few basic issues here.

Although I do all the writing and layout work for OmniScience and produce the original camera-ready copy, the printing is done by a service contracted by Blyth Software's office in Foster City, California. A number of printing anomalies were reported to my office about the first issue. I could only refer those callers to Blyth who, in turn, resolved the problem

continued on page 38

Prepare for Update

The *Prepare for update* command is coming of age. In older versions of OMNIS, the *Prepare for update* mode was easily aborted by a number of operations and programmers had to find work-arounds to perform what seemed on the surface to be simple tasks while in that mode. In OMNIS 7, the *Prepare for update* mode is much more durable, but this durability can lead to a few problems if mishandled.

The general purpose of the *Prepare for update* mode is to organize the various buffers maintained by OMNIS at the current workstation for a process that will update the datafile. The setup processes vary depending on whether a new Main File record is to be inserted or the current one is to be edited. In either of these cases, *any other CRB Files* that are in *read/write* mode may be *simultaneously* edited.

There are three *Prepare for update* modes: Edit, Insert, and Insert with Current Values. Let's examine these first to understand their purposes and how they operate. Then we will look at other commands and modes that may enter into the picture and possible pitfalls of various methods you may choose to employ.

Prepare for Insert

This command does two basic things: It clears the Main File record from the CRB and sets up a new blank record (think of it as a copy of the *null record* for the Main File) with the potential for being

written to disk. If the application is being used in multi-user mode, *Prepare for insert* does two more things: It re-reads *all* records that are in the CRB for non-Main Files and it issues a *request for lock* to the network operating system for all non-Main read/write Files.

The re-reading of these records is performed to insure that the most recent version of each record is available to this workstation for the update process. To further insure that certain records will *remain* unaltered by any but the cur-

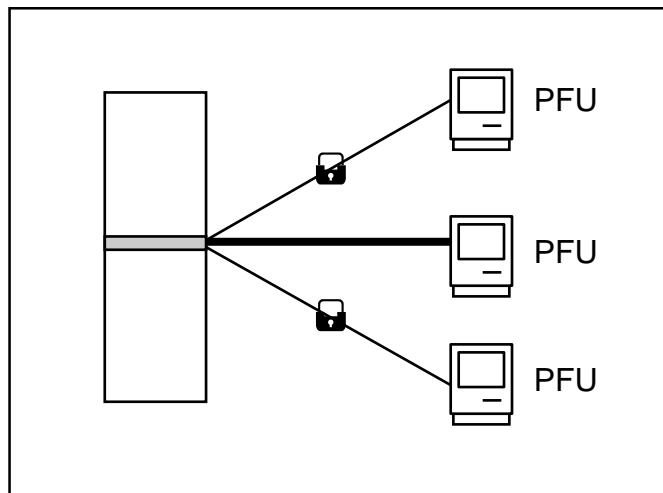


Figure 2 Only one workstation can check out a specific record for editing at a time.

rent workstation, the Files for those records should be set to read/write mode before issuing the *Prepare for insert* command.

If the application is operating in multi-user mode on an individual workstation and there is no network operating system in force (that is, the *workstation* is stand-alone), the request for lock is not sent.

A *Prepare for insert* followed by an *Enter data* can potentially cause problems on a multi-user system

under certain circumstances. If the application maintains a global record in what I call a System Constants File (a common technique) that is used as a source of unique sequential identification numbers for records in various Files (Customer ID, Invoice Number, etc.), that record also needs to be locked during the insert process to ensure consistency for these numbers. If this is done, only one person at a time can perform data entry for inserting a record *to any of the Files requiring access to this set of seed numbers*. Even fast data entry operators may require several minutes to complete an insert — and leaving the workstation while it is

in *Enter data* mode could cause lengthy delays in company-wide operations. This is not considered good multi-user programming practice. A more efficient method is given in the next section.

The *Prepare for insert* command does not generate default values for fields. If we would like to generate such values, we must calculate them in our procedure *after* the *Prepare for insert*.

Prepare for insert does not automatically update the current window after it clears the Main File record (and rereads the others in multi-user). If a data entry process is to be part of the insert, this command should be followed by a *Redraw windows* command — otherwise, the field values displayed on the window will seem to “dissolve” in front of the cursor as they will only be shown as cleared as the cursor moves into each entry field. If we have calculated default values, the redraw should be performed after these calculations.

Insert Current Values

The *Prepare for insert with current values* command performs almost the same functions as the *Prepare for insert* command. The difference is that instead of *clearing* the Main File portion of the CRB, it makes a *copy* of the Main File portion of the CRB with the potential for being written to disk. It will do this even if there is no actual record in the CRB, but only values in Main File fields that have been put into the null record through calculations or data entry. We can take advantage of this fact to solve the excessive record locking during data entry problem posed in the previous section.

We can emulate most of what *Prepare for insert* does with other procedure commands. For example we can clear the Main File portion of the CRB with the *Clear main file* command. We can follow that code with our *Enter data* command and carry on with the process *without locking any records during data entry*. There is also no possibility of encountering a deadly embrace situation with an *Automatic find* field. (This subject is covered later in this article.)

If the operator chooses to cancel, we can *Quit procedure* with no harm done. If the operator accepts the data entry by clicking OK, we then issue a *Prepare for insert with current values*. This copies the Main File values entered during *Enter data* into a record with the potential for being written to disk, re-reads all other records in the CRB, and locks all those in *Read/write* mode. Calculations involving those records, such as generating the Invoice or Customer ID Number and updating the Constants File seed value, can now be performed. Finally, the *Update files* command

is issued — and the records were only locked for a split second.

Here is a simplified version of such a procedure:

```
Clear main file
;calculate defaults
Redraw windows
Enter data
If flag false
    Quit procedure
End if
Prepare for insert w current values
;perform calculations involving
;locked record values
Update files
```

The Constants record is only locked for a split second since *no one* has it locked during data entry. Even if two workstations attempt to lock that record at almost the same time (a very unlikely situation), the one that must retry will probably be successful on the first retry.

But I haven't told you about that yet. It's time for another definition.

Semaphores

A *semaphore* is a Boolean indicator associated with a record that shows that someone has "checked out" that record for editing purposes. Only one workstation can edit a record at one time. An alternate term for a semaphore might be *record lock indicator*.

Semaphores are only set in multi-user mode on a network. They are actually set by the network software, not directly by OMNIS, which only issues a *request for lock*. Semaphores only come into play if another workstation requests a lock on a record that has already been checked out. They are cleared by *Update files*, *Cancel prepare for update*, and *Quit all procedures*. Here is how they are dealt with.

Wait for Semaphores

By default, OMNIS applications are in a *Wait for semaphores* mode. This means that if a record lock fails, indicating that at least one of the required records is being edited by another workstation, repeated attempts are made to lock this suite of records until they are all available. The icon for the mouse pointer becomes a padlock while this waiting continues.

The operator can still abort this process by issuing either a control-break (Windows) or command-period (Macintosh) from the keyboard. This halts all record locking attempts and returns control to the procedure command *after* the *Prepare for update* with a *flag false* condition. A *cancel trap* (trap for exceptional condition, usually an *If flag false* conditional block) should follow the *Prepare for update* command to accommodate the possibility of this occurring.

The perceived problem with this systems is that repeated record locking attempts cause extra network traffic. If we had used a *Prepare for insert* followed by an *Enter data* in the procedure above, this would certainly be the case. But proper programming, as in the example above, minimizes these repeated attempts and doesn't cause much extra network traffic. Some of the proposed alternatives I have seen can cause a nightmare of programming work-arounds.

We have the option of setting a *Do not wait for semaphores* mode. This may be necessary for extra control in some cases, but be advised of the extra things you must concern yourself with if you use this feature.

In this mode, control returns directly to the current procedure af-

ter a *Prepare for update* command (and some other commands) with either a *flag true* or a *flag false*. The Flag is set to “true” if the record locking sortie was successful and to “false” if it wasn’t. The procedure must appropriately deal with either situation, but it isn’t as simple as before.

One of the options we may want to give the operator is to wait until the records are all available. If *Do not wait for semaphores* is set, this requires that we place the *Prepare for update* command in a *Repeat* loop so it can be attempted again. Good examples of such procedures are given in the OMNIS 7 manuals, so I won’t repeat them here.

In a multi-user application, the *Update files* command locks the entire Main File briefly to insert the value of each indexed field in its proper position in that index. This is transparently done if *Wait for semaphores* mode is in force and normally causes no or relatively minor delays.

But if *Do not wait for semaphores* has been issued and one of the Files it needs to update is already locked by another workstation, control simply returns to the next command in the procedure with a *flag false* condition — *without performing the update*. To be certain that the update will be performed, we must place the *Update files* command in a *Repeat* loop with an *Until flag true* condition.

This actually causes a longer delay because more time is required for the procedure commands to be executed between attempts, creating a larger window of opportunity for other workstations to sneak in first. Adding a working message to explain the cause of the delay only compounds the problem.

Here is a personal opinion that you can process as you will. Some remedies can cause more problems than the original disease if not administered correctly. A drug or other treatment will create symptoms that require other drugs or treatments that require other drugs, ad infinitum. In hindsight, it would have been better to have eaten properly and avoided the junk food or coffee or cigarettes that caused the problem in the first place.

The use of *Do not wait for semaphores* can be viewed in this way — it certainly is by me. There are times when it may be appropriate — if used properly — but these are few and far between. All tools have their uses. In the right situation, this is a good tool to use. I feel it is not a good tool for general use.

By the way, even if *Do not wait for semaphores* has been invoked, *automatic find* fields still wait for semaphores. That is, if your application is in a *Prepare for update* mode during *Enter data* and the operator tabs out of an *automatic find* field from a *read/write* File, if the required record is locked by another workstation, the padlock cursor will appear and data entry will only resume once that record has again become available for locking. The “current values” method above solves this problem.

Prepare for Edit

The *Prepare for edit* command differs from the insert commands in that it doesn’t clear *any* records. Instead, it rereads the current records from *all* Files in the CRB in multi-user mode. It also allows the Connection for the Main File record to be modified. For more information on this, see the article “Relationships and Connections” in Volume I Number 1, pp. 6-13.

Since *Prepare for edit* rereads the Main File record, any changes made to that record during a prior *Enter data* would be wiped out. We can’t use the same technique to avoid record locking during *Enter data* here. On the other hand, our reason for needing the Constants record locked on insert doesn’t apply to editing — we don’t need access to those seed numbers.

But there still may be records in other Files that need modification as a result of a change to our Main File record. If it would cause problems to have those records locked during the data entry phase of the procedure, we would make the *read only* during data entry. We can then set those Files to *read/write* and change the values in those records on a subsequent editing pass with no data entry, updating them with the new information in the Main File record.

If we need to update any records with the *difference* between old and new field values from the Main File record, we must store the old values in temporary variables before performing the data entry.

Graceful Exit

There are two appropriate ways to complete the *Prepare for update* mode: invoking the *Update files* command in its most basic form (not checking the *Do not cancel pfu* option) or invoking the *Cancel prepare for update* command.

A third method is to use the *Quit all procedures* command. I consider using this command to be the “method of last resort” to canceling the *Prepare for update* mode.

There used to be many more ways this mode could terminate — *ungracefully*. Here’s a brief overview.

Old Problems

In previous versions of OMNIS (3, 3 Plus, Quartz, and 5), the *Prepare for update* mode would be aborted under a number of conditions. I found these to generally be very logical reasons to abort the *Prepare for update* mode (like removing the record being edited), but many OMNIS programmers were legitimately frustrated by rules that they felt were too restrictive. Blyth Software responded by making this mode virtually bulletproof.

Some of the commands that would abort this mode were: *Find*, *Next*, *Previous*, *Set main file*, *Clear main file*, *Build list from file*, and *Single file find* (on the Main File). These difficulties have all been remedied in OMNIS 7 and an amazing number of operations can be performed while in the *Prepare for update* mode these days.

Change Main File

The *Set main file* command can now be used while in the *Prepare for update* mode. This allows the procedure and its subroutines to go off and perform other actions on other Files involving *Find*, *Next*, *Previous*, and *Build list from file* commands (among others) without having to subsequently re-establish the *Prepare for update* mode on the original Main File.

When an *Update files* command is encountered, OMNIS 7 remembers what the Main File was when the most recent *Prepare for update* was issued and performs appropriately.

Locate Other Record

If a different Main File record is read into the CRB during a *Prepare for update* process, OMNIS continues trying to do the right

thing. Its interpretation of the “right thing” is really interesting and worth knowing.

If we are editing a specific record and a different record is located while in the *Prepare for edit* mode, OMNIS will simply edit the record that was read in. Any changes to the old record are wiped out by the record location command (*Find*, *Next*, etc.) that brought in the new record. Any subsequent change to a Main File field applies to the record in the CRB. Try this procedure to demonstrate the effect:

```
Begin reversible block
  Set main file {Customer file}
  Set read/write files
    {Customer file}
End reversible block
Single file find on CU_ID_NUMBER
  (Exact match) {'0000001'}
Prepare for edit
Calculate CU_FIRST_NAME as
  'Changed'
Single file find on CU_ID_NUMBER
  (Exact match) {'0000002'}
Calculate CU_LAST_NAME as 'Changed'
Update files
```

The first name field value will not be changed in either record. The last name field value will be changed in the record for Customer number 2.

The effect is much different if this happens during a *Prepare for insert* process. If a Main File record is read into the CRB during *Prepare for insert*, only a *copy* of the record is actually read in. That is, the values from that record are placed into the fields in the “blank record with the potential for being written to disk” that is in the process of being entered. Any Main File field values entered before the record location process are *replaced* by the corresponding field values from the located record.

To demonstrate this effect, consider the following procedure.

```
Begin reversible block
  Set main file {Customer file}
  Set read/write files
    {Customer file}
End reversible block
Prepare for insert
Single file find on CU_ID_NUMBER
  (Exact match) {'0000002'}
Calculate CU_LAST_NAME as 'New'
Update files
```

This has *exactly* the same effect as performing a *Prepare for insert with current values* with Customer number 2 already in the CRB. OMNIS remembers that it is *inserting* and doesn't allow that process to be interrupted by a Main File record location process.

Build List from File

If a procedure needs to be performed on some File other than the Main File during a *Prepare for update* process, we have absolutely no problems. We have already seen how changing the Main File has no effect on this mode.

If a procedure needs to build a list on the *Main File*, there is still no problem. The *Build list from file* command now operates in its own buffer rather than in the CRB. The Main File record is not even affected by this process.

Set Read/Write Files

Only records from Files in the *read/write* mode can be updated. The Main File must be in this mode for *any* of the *Prepare for update* commands to be successfully executed.

It is *not* a good idea to change a File to *read/write* mode while in *Prepare for update*. There is a chance
continued on page 34

Format and Local Variables

As I give classes around the country or receive calls for technical support on OMNIS 7, a considerable number of the questions OMNIS 7 users ask me involve the use of Format and Local variables. This article presents some basic explanations of the implementation of these variable types in OMNIS 7 and some guidelines for their use.

“Global” vs. “Local”

Let’s first discuss the basic concept of a *local variable*. To do this, we must juxtapose that concept with the idea of a *global variable*. Some definitions are in order.

A variable is *global* if access to it is available throughout the application. That is, its value can be used or changed from anywhere within the program. In OMNIS, this means that in any location or capacity a field name can be used, the name of any global variable can be used as well.

If we consider that a variable is just a pigeonhole in RAM for temporarily stashing a type of value, we can also differentiate a global variable as one whose memory allocation in RAM is “permanent”. That is, space in RAM is allocated for that variable name early in the life of the application and it remains allocated until that application is closed (when all memory allocated to the application is released).

For example, the hash variables (# variables) in OMNIS 7 are global variables. Their memory slots are

allocated when OMNIS is first opened and the most recent value is retained by each variable until we quit OMNIS — even if we switch to a different application.

Fields in Memory-only Files are also global — at least within the currently running application. Their memory slots are allocated when the application is first opened and released when the application is closed. The most recently assigned value is retained in memory. We’ll discuss the use of these in more detail later in this article.

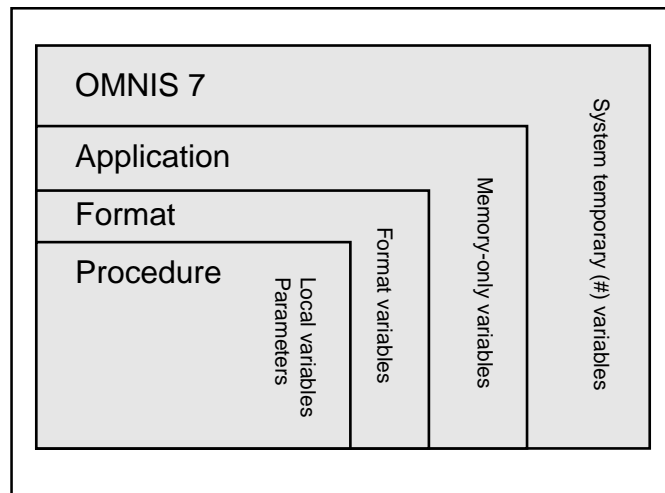


Figure 3 Different variable types have different spheres of influence.

In a sense, we can also consider the Current Record Buffer (CRB) to be a collection of global variables. In OMNIS, we differentiate fields from variables in that fields are gateways to information stored on disk while variable values remain *only* in memory.

A variable is *local* if access to it is restricted to a specific part of the overall program. Its value can only be used or changed in that part of the program — the variable doesn’t even exist anywhere else.

We can also differentiate a local variable as one whose memory allocation in RAM is “dynamic”. That is, space in RAM is allocated for that variable name “on the fly” as it is needed. When the variable is no longer needed, usually after the function or procedure that uses that variable has been executed, its memory allocation is released.

When a local variable is first used in a procedure, it has no value. It has just come into being and only a slot in memory has been assigned. For this reason, such a variable must first be assigned a value (with a *Calculate* statement in OMNIS 7) before it can be otherwise used.

The term “local” has more than one meaning in OMNIS 7. Please do not confuse *local variables* with the concept of *local fields* on a window. Local variables have nothing to do with the “local” attribute of a window field. The two concepts are completely independent of one another. A short explanation of a “local” field is that it, in some sense, “belongs to” the preceding field in Window Format field number order. Its value is automatically re-evaluated and redrawn when the cursor leaves the preceding field.

For more detailed discussions of the “local” attribute, see these OmniScience articles: “Computerized Bookkeeping” (Vol. I No. 2, pp. 14-15), “Data Entry Into Lists” (Vol. I No. 3, p. 34), and “Event Management Revisited” (Vol. II No. 1, p. 2). An older, but more comprehensive explanation of the local field attribute can also be found in Chapter 9 of my book *Unlocking Omnis 3 Plus — Version 3.3*.

OMNIS Local Variables

Many programming languages include the concept of local variables. Classic local variables in OMNIS 7 can be declared in the procedures of Menu and Window Formats.

There is another variable type, specific to the structure of an OMNIS application, that is somewhat local and somewhat global. This type is called a “format” variable. We will examine format variables a little later in this article.

A local variable in OMNIS 7 only exists during the execution of the procedure in which it is defined or declared. Memory is allocated for its value when the procedure begins and is released when that procedure ends.

Unlike some programming languages, a local variable declared in an OMNIS 7 procedure is *not* directly available in subroutines called by that procedure. This is because the definition of (and token for) each local variable is carried in its corresponding procedure. OMNIS procedures don't know they are subroutines until they are called by another procedure at runtime, so they can't be expected to contain tokens for all local variables in all procedures that might call them. We do, however, have a means of manipulating that variables value in a subroutine. It is called “parameter passing by reference”, and we will examine it in due course in this article.

OMNIS local variables can be assigned any File Format data type except Sequence. The reason for the exclusion of the Sequence type should be clear — local variables do not belong to a File, so Record Sequencing Numbers (RSNs) are not related to them in any way.

Parameters

A *parameter* is a special kind of local variable that can be declared in a procedure in OMNIS 7. When that procedure is called as a subroutine, these variables are assigned an initial value passed from the calling procedure instead of starting their current life with an empty value. Other than the initial value assignment, parameters are used in the same way as local variables and are treated as such by OMNIS.

Parameters can be assigned any of the data types that apply to local variables — that is, all OMNIS 7 data types except Sequence. But there is also a data type that is unique to parameters. This is the “field name” type. We will discuss this special parameter type at length later in this article, as well as the declaration and use of parameters in general.

Format Variables

The concept of a *format variable* is specific to OMNIS and does not have a direct counterpart in other programming languages. This type of variable is “local” to the Format in which it is declared, but “global” within that Format. The value in a Format variable cannot be accessed from any other Format in the application. As with local variables, format variables can only be declared in Menu and Window Formats, since those are the only Format types that allow procedures in the current (1.03) version of OMNIS 7. They have a variety of uses.

For example, this global accessibility within a Format means that we can create fields on a Window Format for any of its format variables if needed. Local variables can never be accessed by window fields.

The memory allocation for a format variable is made when that Format is first opened and is “permanent” unless the *Clear format variables when closed* command is invoked. This means that a Window or Menu Format can be closed and then reopened, and the format variable values associated with that Format will be as they were when the Format was closed — assuming that the application had been running continuously.

We will examine the nuances of clearing format variables below. First, let's see what the ground rules are for setting up local and format variables.

Variable Declarations

The *Local variable* command is the most common way to declare a local variable. Using this command, a local variable can be created for the current procedure. It can be given any OMNIS 7 field type except Sequence.

The name assigned to a local variable must follow the same rules as for field names within OMNIS 7. In the current (1.03) version, field names are limited to 15 characters. These characters must be upper case letters, numerals, and underscores. A field name cannot begin with a numeral, but *can* begin with an underscore. There is one exception to this rule for naming local variables: A local variable name can also begin with one or two percent (%) symbols. The use of this specialized notation will be explained below.

The *Local variable* command can be placed anywhere within the procedure, even after a *Quit procedure* command, and still have the effect of declaring the variable. This is because the requirement to allo-

cate memory for the variable is associated with (stored in) the procedure during the *modification* of the procedure, *not* during execution of the procedure at runtime. It is not possible to *use* the field name until after it has been declared (chronologically), since the field name would not yet exist, but the relative position of the declaration statement and procedure commands that use that variable within the procedure are unimportant.

As soon as the *Local variable* command has been completed, the variable declared there appears in the Field Names window under “Local variables” for that procedure. If you go back to the *Local variable* command at some later time and change the name of the variable, all references to that variable, including the Field Names list, are changed. The same holds true for changing the data type of the variable — and it also applies to format variables. This is because local and format variable names are tokenized in the same way as File Format field names, so all other references to the variable reflect the last name stored for that variable. (For a discussion on field name tokenization, see the article on the “eval() Functions” in OmniScience Vol. II No. 1, pp. 20-23.)

The name assigned to a local variable must be unique among the local variable (including parameter) names for that procedure — but it *can* duplicate the name of a File Format field or a format variable. OMNIS sees local variables and parameters as belonging to a different “File” than format variables or CRB fields and does not require uniqueness of names where local and format variables are concerned. If you choose to “take advantage” of this fact, there are a few other things you should know...

Whenever you refer to that name in that procedure, you will *always* be referring to the *local* variable as far as OMNIS is concerned. For example, you can never use the *Calculate* command to modify the value for the File Format field or the format variable that carries the same name as a local variable within that procedure. It doesn't matter whether you place the field name in the command attributes window by typing it or by using the Field Names window facility — when OMNIS 7 sees a field name placed that is also an active local or format variable, it uses and tokenizes the *most local* version of that field name.

If this causes you some distress or confusion — *don't do it*. (There are a lot of cliché jokes along the same line.) I strongly recommend giving a local variable or parameter a name that does not exist in the CRB or the list of format variables for the current Format. It doesn't take much creativity to do so and it can save a great deal of aggravation. On the other hand, it is perfectly OK and should not cause confusion to use the same local variable name in other procedures.

In practice, it is possible to declare a local variable with the *Local variable* command, use that variable in some way (most often in a *Calculate* statement), and then remove the declaration statement. The variable will remain. I do not altogether recommend this practice. If you want to avoid the scanning of the *Local variable* commands during execution of the procedure, just place all of them after a *Quit procedure* command at the end of the procedure. There are other means you could use, such as commenting out the declaration lines, but these would not keep the lines from being scanned at runtime.

Another method of declaring local variables is called “% notation”. This method is less flexible than using the *Local variable* command with regard to our choice of data type for the variable, but it doesn't require a separate declaration (*Local variable*) statement.

Any procedure command that requires a tokenized field name as an attribute can be used for this type of declaration. The most commonly used would be the *Calculate* and the *Set current list* commands. Simply supply a field name that begins with one or two percent signs and a new local variable is declared.

A name beginning with a single percent sign is a floating point numeric field and one that begins with two percent signs is a variable-length character field with a maximum 32,000 character capacity. Declaring a % field name with the *Set current list* command creates a local field of List type.

Subsequent use of the *Local variable* command can be used to change the data type of a % variable. The % sign(s) must remain part of the variable name if the variable was originally declared by this method, however.

The *Format variable* command is used to declare a format variable. This command has many of the same abilities and restrictions as the *Local variable* command.

For example, a format variable can be given any data type except Sequence, just like local variables. The naming restrictions are the same, except that there is no facility like % notation for declaring a format variable by another method. Two *Format variable* statements using the same variable name are not allowed in the same procedure.

Duplicate *Format variable* statements *are* allowed in different procedures in the same Format, though. These are assumed to refer to the same variable (memory location). Once a duplicate statement is created, changing the variable name in either one automatically changes the name of the variable throughout the Format. If you declare a different data type in such a duplicate statement, the last one created or modified (chronologically) determines the data type of the variable. It is good practice to avoid creating duplicate *Format variable* statements.

Format variables can be declared in any procedure within the Format — even one that is never executed. This is because the requirement to set up the format variable is associated with that Format the moment the statement is created in modification mode. The *Format variable* statement itself is never executed. A programming practice you might consider is to set aside a procedure for format variable declarations in each Format that requires them. This will give you a consistent place to modify their names and data types as you enhance your application.

None of these variable declaration commands is reversible or affects the Flag. Neither is the *Parameter* command we are about to discuss. They are not really executed at runtime, so we would not expect them to do either of these things.

Removing Variables

Removing the procedure command that defines a local, parameter, or format variable does not remove the requirement for OMNIS to allocate memory for that variable name. This requirement to allocate memory cannot be removed

until all tokenized references to that variable have been deleted. For local variables (and parameters), the references must only be removed from the one procedure. For format variables, all references in all procedures (*and all fields* in Window Formats) in the entire Format must be deleted. References such as using the variable name in a string (not in semicolon notation) or generating it using the *fld()* function or the *Calcluate (use fld) of name* procedure command are not tokenized references.

Once all reference to a variable have been deleted, the requirement for OMNIS 7 to allocate memory for that variable can be removed by selecting the *Remove unused variables* option from the Modify menu for that Format. You can check to see if the removal operation was successful by looking at the list of remaining format or local variables in the Field Names window.

Declaring Parameters

The *Parameter* command is used to declare parameter-type local variables. Unlike local or format variables declarations, though, the statements that declare these variables must remain in the procedure. The local variables that the *Parameter* commands declare are set up when the procedure is first called, but those commands must still be executed to fulfill their purpose. This is because their position in the procedure is fundamental to their use.

Up to nine parameters can also be declared implicitly in any procedure using a form of % notation. These implicit parameters are numbered 1 through 9 in a similar fashion to the numbering of the OMNIS temporary numeric variables (#1-#60). The implicit param-

eters' names are preceded by either one or two percent signs indicating floating point numeric (%1) or variable length character (%%1) field types respectively. More than nine parameters can be declared for a procedure, but only the *first* nine can be declared implicitly.

I can best explain the need for proper ordering of *Parameter* statements or implicit parameters by first explaining how parameters are passed.

Passing Parameters

To pass parameters to a procedure, it must be called as a subroutine of a running procedure. The procedures that can pass parameters are the *Call procedure*, *Call procedure with return value*, *Open window*, and *Install menu* commands. The last two of these commands send parameters to the zero procedure in the specified Format, which will automatically run as a result of that Format being opened or installed. We can consider these zero procedures to be subroutines since they will be executed before control is passed back to the command following the one that invoked their execution.

These commands allow for an optional list of parameter values to be passed to the subroutine. This list is enclosed in parentheses and the parameter values are delimited by commas. The parameter values in this list can be explicit values of the appropriate type for the corresponding *Parameter* command, field names containing such values (or used for passing by reference), or expressions that derive such values.

The values supplied in the parameter list of the calling procedure command are doled out to the pa-

rameters of the subroutine in the order that the *Parameter* statements are executed and/or the implicit parameters are encountered. As usual, OMNIS will do its best to fit the supplied values into the data types assigned to each parameter.

OMNIS 7's requirements for parameter passing are very precise. If fewer parameter values are supplied than are required by the subroutine (based on the number of *Parameter* statements in the subroutine), an error is generated during execution. This is because OMNIS makes the assumption that all the parameters are necessary to the operation of the procedure and that an initial value must be supplied for each.

There are times when this may not be true. One reason to use subroutines that are passed parameters is to serve as sort of a user-defined function to be called from various places within an application. Many functions have abbreviated forms that require fewer than the full set of parameters. (Consider the *lst()* function, for example.)

OMNIS 7 allows us to declare optional parameters in a procedure with the *Other parameters are optional* command. This command essentially turns off the requirement for values to be passed to subsequent parameters (in execution order). Parameters that are encountered *before* this command are still required and an error will still be generated if insufficient values are passed for all of them. Any parameter following the *Other parameters are optional* command that are not passed a value simply begin their existence as empty local variables.

While OMNIS 7 is very fussy about too few parameter values being

passed, no error is generated if *more* parameter values are passed than the number of *Parameter* statements or implicit parameters available to receive them. If we need to know that this has occurred, we can still write code that performs such a check. There is a special parameter named %0 that contains the number of parameter values passed from the calling procedure. Checking for too many parameters passed is only a matter of determining whether the value of %0 is greater than the number of parameters in the procedure. The programmer must hard-code this value since there is no variable that contains it. If there are only five parameters in a subroutine, the following conditional block can be used to halt execution.

```
If %0>5
    OK message {Too many parameters
                were passed to [sys(85)]}
    Breakpoint
End if
```

These problems should always be resolved while you are debugging your application and should never be encountered by your operators once the application is deployed.

Call With Return Value

OMNIS 7 allows us to call a procedure and receive a value generated by that procedure directly into a field that we specify. This is the purpose of the *Call procedure with return value* command. The return field can be of any type. The return value is generated using the *Set return value* command in the subroutine.

A question that often comes up when people ask me about this command is how to get more than one return value from the subroutine. There are a lot of clever meth-

ods that we can employ, and I will explain a couple of them here — but I will show you the best one in the next section.

One method is to concatenate a number of values into a long string and return that string value. The string can then be parsed into the separate values you require in the calling procedure.

Another method is to set up a List as the return value. Since a List is composed of a number of fields, this makes a natural package for transporting multiple values in a single field. You can even return multiple lines of values.

But the real method is not to rely on the return value at all. Instead we can use parameters to set up aliases for the fields we want to return. This is called “parameter passing by reference”.

Pass By Reference

We most often think of passing values to a number of parameters in a subroutine, processing those values in the subroutine, and perhaps returning a value to the calling procedure as though the subroutine were a function. Parameters passed by value is an easy concept to grasp.

If we set up a parameter using the *Parameter* command and assign it the *Field name* data type, we are really setting up the parameter to act as an “alias” for the field whose name we passed. Any action performed using that parameter is actually an action performed on the field whose name was passed. Most importantly, if we use the *Calculate* command to act upon the parameter, the field whose name we passed receives a new value based upon that calculation.

We can pass any number of fields to a subroutine by reference. The only operation that doesn't seem to work by reference with the current (1.03) version of OMNIS 7 is the *Single file find* command.

If we need to be able to use a local variable from the calling procedure in one of that procedures subroutines, we can pass the local variable to that subroutine by reference. We can even give the parameter the same name as the original variable, although often we may pass different local variables to that subroutine from different calling procedures. Actions on the parameter will become actions on the original local variable.

We can even pass fields by reference to a number of subroutine levels! Consider the following set of procedures.

```
10 calling proc
Local variable TEST_NUMBER
  (Short number 0 dp)
Calculate TEST_NUMBER as 10
Call procedure 11 (TEST_NUMBER)
  {subroutine 1}
OK message (High position)
  {TEST_NUMBER=[TEST_NUMBER]}

11 subroutine 1
Parameter REFERENCE_1 (Field name)
OK message
  {REFERENCE_1=[REFERENCE_1]}
Call procedure 12 (REFERENCE_1)
  {subroutine 2}
OK message
  {REFERENCE_1=[REFERENCE_1]}

12 subroutine 2
Parameter REFERENCE_2 (Field name)
Calculate REFERENCE_2 as
  REFERENCE_2*25
```

Procedure 10 sets up a local variable, sets an initial value for that variable and then calls procedure 11, passing the local variable as a

parameter value. The parameter in procedure 11 is a *Field name* type, so the local variable from procedure 10 is passed by reference. The procedure then displays the parameter value in an OK message and calls procedure 12, passing the parameter value.

The parameter in procedure 12 is also a *Field name* type, so the parameter from procedure 11 and the local variable from procedure 10 are passed by reference to this parameter. A calculation then is carried out on the parameter in procedure 12. The OK messages in both procedure 11 and procedure 10 display the modified value. The value of the local variable from procedure 10 was "directly" modified by a command in a procedure two subroutine levels away!

If we pass a value other than a field name to a parameter that is a *Field name* type, an error will occur and OMNIS will abort processing. Again, this should be discovered during your testing and debugging phase and corrections should be made so that it never happens in your finished application.

Since I can pass fields by reference, I rarely use the *Call procedure with return value* command for any more than the simplest subroutine calls (when I'm too lazy to think about passing fields by reference). If I use it for complex subroutines that require multiple return values, I use pass by reference methods to generate the values and use the return value for passing back error numbers or messages.

Clear Format Variables

Local variable values and memory allocations are cleared when a procedure completes execution. We never have to worry about clearing

those values. But the memory allocation for format variables persists by default until the application is closed (or until the *Examine application file* option is selected from the Utilities menu in single user mode). The values in these memory slots also persist. There are times, however, when we may need to clear the values from those slots and release the memory allocation.

This is the purpose of the *Clear format variables* command. When it is invoked, the memory allocation for all format variables of the current Format is released. Memory is reallocated as each format variable is again referenced.

If we want to have OMNIS clear the format variable memory allocation for a specific Format automatically when that Format is closed, we must see to it that the *Clear format variables when closed* command is executed from within that Format. The default state is *not* to do this, so we have to explicitly tell OMNIS to perform this memory cleanup.

In my work to date, I have either *always* wanted to do this or *never* wanted to do this for a specific Format. I put a *Clear...* command in the zero procedure of Formats where I want to use this feature. We also have a *Do not clear format variables when closed* command, but I have never needed to use it with my simplistic programming style since this is the default state.

By the way, it is long procedure command names like those in the preceding paragraphs that help me to truly appreciate the tokenization of procedure commands in OMNIS 7 and the List navigation method of typing them in the procedure modification area.

continued on page 33

lookup() Function

The talents and capabilities of OMNIS continue to grow with the release of OMNIS 7. It has more extensive access to data from more sources than ever before. The new *lookup()* function adds to the growing list of OMNIS data access tools.

Access to Information

One can visualize OMNIS as a data-manipulating octopus with its tentacles reaching into many different sources of data at once (Figure 4). At the same time that an OMNIS 7 application has access to up to sixty Files in the associated datafile, it can maintain multiple open channels to multiple SQL databases and multiple interactive channels to other application programs and documents through Dynamic Data Exchange on Windows or through Publish and Subscribe on the Macintosh. OMNIS 7 can both read and write data along these channel types.

OMNIS 7 can also maintain multiple data access channels to multiple OMNIS datafiles *in addition to* the current datafile for the current application. These channels are read-only, hence the name “lookup”, but the very fact that OMNIS can do this is impressive. Lookup channels can also be used on the current datafile, which can give us additional data manipulation advantages in some cases.

This direct pipeline to information in other OMNIS datafiles can be a useful tool in your application design strategy. Relatively static in-

formation required by a number of applications, such as a translation table for State, Province, and Country codes and names, can be maintained in one datafile and accessed as needed from anywhere.

The Lookup Channel

To use the *lookup()* function, we must first set up a *lookup channel*. This channel is a direct line to an index in a specified File in the target datafile. The *lookup()* function is then used to perform the equivalent of a *Single file find* along that

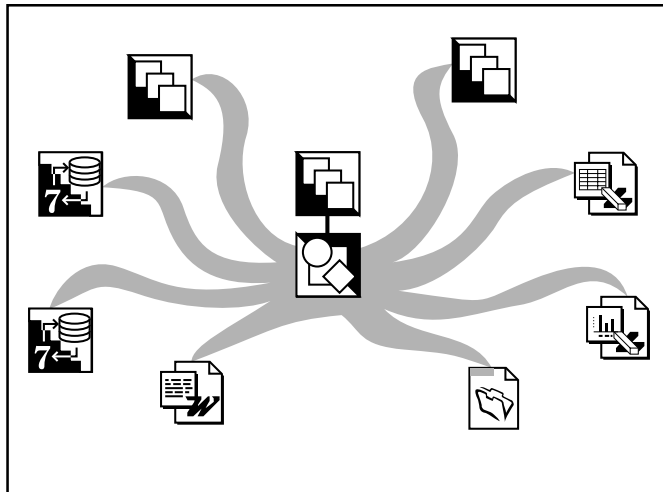


Figure 4 OMNIS 7 can access data through a number of channels.

index using a value supplied in one of the *lookup()* function parameters. The value of a specified field is returned and that value will be of the same type as the specified field. We can think of the *lookup()* function as being *replaced* by that value in an expression.

The lookup channel is given a name that allows us to identify our desired lookup path from among a number of paths that can be open simultaneously. We can think of this as an “alias” for a combination

of datafile pathname, File name, and indexed field. In use, we refer to a specific lookup “pipeline” by its name rather than by the detailed information required to describe that channel.

Open Lookup File

To set up a lookup channel, we use the *Open lookup file* procedure command. In its most general form, this command requires four parameters. These parameters are delimited by slash (/) characters. The parameter entry area expects to see a string (see “Strings and Expressions” in OmniScience Vol. I No. 2, pp. 18-21), so none of the

parameter values can contain a slash character. On the other hand, quotes are not needed to enclose the various string parameter values. Expressions (including simply field names) used to derive parameter values must be enclosed in square brackets to be evaluated. As with all strings in OMNIS, no syntax checking is done on these values (except for the “live” expressions in square bracket notation), so care should be taken in providing parameter values.

The four parameters recognized by the *Open lookup file* command are: the name (or alias) that we assign to the channel (called the “lookup reference” or “reference name” in the OMNIS 7 manuals), the pathname of the datafile the channel accesses, the name of the File Format that we wish to address within that datafile, and the number of the indexed field (in File Format field number order) that we wish to use for the *Single file find* operation. Each of these pa-

rameters has its variations and limitations as explained in the following paragraphs.

The channel is not opened and the lookup alias is not defined until the *Open lookup file* command is executed. The alias can be anything, since it is stored as a string, and it can be any length, can contain almost any character (except, of course, a slash), and is case insensitive. If an *Open lookup file* command is executed defining an alias that is already open, the old channel is closed and that alias now refers to the newly defined channel. It is a good idea to be consistent in your use of channel aliases.

If no alias is given (that is, if the first parameter of the *Open lookup file* command is omitted), OMNIS 7 assumes that you will only be opening one lookup channel and will not require the channel alias parameter in the *lookup()* function. The delimiting slash character *must still be included* in the parameter string for this command.

The datafile pathname parameter must, in general, be the *complete* pathname of the target datafile. In the special case that this datafile is in the same subdirectory or folder as the current application, only the filename of the datafile is required. In either event, we must hard-code the pathname into this parameter. This practice makes the assumption that our users will not move or rename that datafile — not always a safe assumption even if the end user is warned.

We can get by this restriction, and lend more flexibility to our application, by prompting the operator for the pathname of the “foreign” datafile using one of the useful file operation OMNIS extensions included in the OMNIS 7 Plus kit.

The *GetFile* routine presents a standard file location dialog. We can send parameters to it that supply a custom message for the window, a filter for the type of files to be shown, and the name of a string variable or field to receive the pathname of the selected file (which is passed by reference). The following procedure prompts the operator for a datafile names “State data” and stores its pathname in #S3. This was written to run on a Macintosh, so the file type for OMNIS 7 datafiles is “O7SD”. The first four fields in the State file are a Sequence field, the State code, the State name, and the name of the State capitol in that order.

```
Call external routine GetFile
    (#S3,'Locate the State data
    file...','O7SD')
If flag false
    OK message {GetFile not found}
    Quit procedure
End If
Open lookup file
    {/[#S3]/State file/2}
Calculate #S1 as 'TX'
OK message {[#S1] stands for
    [lookup('',#S1,3)].//It's
    capitol is [lookup('',#S1,4)].}
```

The OK message replies, “TX stands for Texas. It’s capitol is Austin.” The operation takes place very quickly. We could be even more clever and store that pathname in a field in a System Constants File. The next time this lookup path is required, our procedure could first check to see if the file still exists at the old pathname and only prompt the operator for the file if it is no longer there.

The datafile we specify in the *Open lookup file* command can be on *any* volume that the workstation can see, including volumes that other users on the network may have made public (assuming that your

network software supports such a facility). The datafile pathname parameter is case insensitive.

The File Format name parameter, however, is case sensitive. You must be very precise in specifying the File Format parameter. OMNIS 7 does not give us access to the Format and field names in another application, so you must know these ahead of time and hope they haven’t been changed since you last checked. A possible method to use is to print the File Format details from the application associated with the lookup datafile, either from the File Format modification area or from Utilities.

This method is also useful in determining the field number of the index for the lookup channel, the final parameter for the *Open lookup file* command. The field whose number you supply *must* be indexed for this to work at all — and it should be *uniquely* indexed to have all the records accessible to the *lookup()* function. The function performs the equivalent of an *exact match* find and there is no *Next* option.

The fact that we need to supply a field *number* instead of a field name here doesn’t really have anything to do with that field name not being in the current application, as the manual suggests. The real reason is that we are accessing the datafile *directly*, and *the datafile doesn’t know anything about field names*. It works strictly on the basis of field *numbers*. Field names are *only* contained in application files and are provided for the convenience of us programmers while we are developing and maintaining our applications — to give us something more meaningful than “field #8” to work with. The field names we supply for ourselves in the application file are simply bridges to their as-

sociated field numbers. OMNIS actually stores and works with the field number (and the File Format number) when referring to fields in window and report field definitions and in expressions. See the article on the “*eval()* Functions” in Volume II Number 1 for a discussion on this subject.

By the way, the field number parameter is optional. If it is not supplied, OMNIS 7 will assume it should use the first field in the File Format (*not* the first indexed field) as the index path for the lookup. The delimiting slash is still required in the parameter string.

The *Open lookup file* command both affects the Flag and is reversible. If the lookup channel can be established (that is, if the specified datafile, File Format, and indexed field can be found and the datafile can be opened), the Flag is set to “True”. Otherwise, the Flag is cleared to “False”. If the channel is set up in a reversible block, the channel name is essentially “local” to that procedure and its subroutines. The memory the channel requires is released when the procedure ends.

More than one channel using different indexes can be open to the same File Format in the same datafile simultaneously. In the example of the States file, I may need to use a channel that follows the Sequence field instead of the State code field. All I need to do is establish the channel with a different name than the first one.

Remember, the first and fourth parameters are optional under the rules outlined earlier in this article. The delimiting slash characters are still required. Valid forms of the *Open lookup file* parameter list are:

```
channelname/pathname/file/field#
/pathname/file/field#
channelname/pathname/file/
/pathname/file/
```

Function Parameters

To use the *lookup()* function, we must supply it with three parameter values that pinpoint a value in the target datafile. These parameters specify which lookup channel to use, what value to locate along that lookup index, and which fields value to return. Here are the details and limitations for the use of each *lookup()* function parameter.

The first parameter is the name of the lookup channel we wish to use. This name can be supplied as an explicit string, as the name of a string field that contains the lookup channel name, or as a string expression that derives that name. Remember that the *lookup()* function is only used in expressions, so explicit string values must be enclosed in single quotes. OMNIS can have no way to check this parameter for validity in the design mode when it does other syntax checking — it is up to the programmer to supply a valid value or an expression that derives one.

The lookup channel name parameter is optional. In the special case where we are only using one lookup channel and where we defined it without using an alias, that channel will be assumed. If we have named all our channels, but don't supply a channel name value for the *lookup()* function, OMNIS will use the name of the first channel that had been opened (and that is still open) in execution order. In most cases, it is probably not a good idea to rely on this default.

If we are excluding *only* the channel name parameter, we must still

include a null string (two adjacent single quote characters) in front of the first delimiting comma. When OMNIS 7 sees only two parameters being supplied in this function, it assumes that the *last* one has been dropped. Dropping the first parameter entirely while supplying the second and third will not give us the desired result. The valid forms for this parameter set are given at the end of this section.

The second parameter is the value to be used for the lookup. This parameter is definitely *not* optional — it is the key ingredient for this function. It must be of an appropriate data type for the indexed field specified for the channel. If only one parameter is supplied in the *lookup()* function, OMNIS assumes it is this one.

As with the first parameter value, this parameter must be either an explicit value of the proper data type, the name of a field containing such a value, or a more complex expression that evaluates to the proper type. String values that look like dates or times are converted as necessary. All other implied type conversions apply.

The final parameter is the number of the field whose value is to be returned in File Format field number order. The value returned will be of the same data type as that field. The parameter value can be supplied as an explicit number, the name of a field containing a number, or an expression that evaluates to a number.

This parameter is also optional. If it is not included, OMNIS 7 will assume you want the value of field number 2 from the File Format specified for the lookup channel you named. This parameter can be entirely dropped from the list sup-

plied in the function. If OMNIS sees the *lookup()* function using only two parameters, it assumes that the last one has been dropped.

To clarify how to *not* supply one or more of the optional parameters, the valid forms of the *lookup()* function parameter list are:

```
channelname,index value,field#
',index value,field#
channelname,index value
index value
```

Remember, if OMNIS encounters the *lookup()* function using only two parameters, it assumes that they are, respectively, the channel name and the index value. If OMNIS does not see a valid channel name in the first parameter slot in this case, an error will result. If only one parameter is supplied, it is assumed to be the index value. The default channel will be used and the value of field number 2 from that File will be returned.

No Error State

The only error generated by a problem with the *lookup()* function is that due to an invalid lookup channel name. If the *lookup()* function cannot locate a record in the lookup File based on the given index field value, it simply returns empty-handed. No error is generated in this case. Since *lookup()* is only a function that makes up part of an expression, and not a procedure command, it has no effect on the Flag. An unsuccessful lookup attempt *cannot* be detected with an *If flag false* command. This just means that we have to be more resourceful if we need to detect a state where a record is not found by the *lookup()* function.

We must perform our test separately in a conditional block (*If...*

statement) at some point before the expression where that channel and index value are used. The test is simply to see if a value exists in a field we *know* would not be empty for any record we might retrieve — like the index field itself. For most data types, we would check to see if the *length* of the returned value is non-zero. For numeric fields, we would check to see if the value itself is non-zero. Using the earlier example of the States channel and assuming that our index for the lookup is field number 2, our testing procedure segment would look like this.

```
Calculate #S1 as 'XX'
If not(len(lookup('States',#S1,2)))
  OK message {[#S1] is not valid}
  Quit procedure
End if
```

We could also check to see if the *lookup()* function returns the same value as was sent. The procedure segment would look like this.

```
Calculate #S1 as 'XX'
If lookup('States',#S1,2)<>#S1
  OK message {[#S1] is not valid}
  Quit procedure
End if
```

In either event, the failed lookup would be appropriately handled and our own error message would be generated.

Same File Lookup

The *Open lookup file* command is not limited to setting up lookup channels to “foreign” OMNIS 7 datafiles. It can also be used to arrange lookup channels to indexes in the *currently open* datafile! If we desire to do this, we can be certain of having the correct pathname to that datafile in the parameter list for the *Open lookup file* command by using *sys(11)* in square bracket

notation there. *sys(11)* contains the pathname of the first segment of the currently open datafile.

Besides the obvious advantage that we would always know where the datafile is located, there is a more subtle advantage to lookup channels on the current datafile. The *lookup()* function operating on a channel to a File in the current datafile *does not effect the CRB*. In addition to that, it only retrieves a single field value instead of an entire record. So we can retrieve a value in a record other than the current record in *any* File and not disturb the current record. (We *could* retrieve a value from the current record as well, but what's the point?) If we only need access to a few field values from a very large record (one with many fields), this can prove to be very efficient.

Two (or More) At Once

We can take this principle a step further. Since each use of the *lookup()* function is independent, we can use multiple instances of *lookup()* to access more than one record in the lookup File within a single expression.

Consider this situation. Suppose we need to perform some operation that requires numeric field values from two records in the same File. The typical way we would do this is to locate the first record and store the field value in question in a temporary variable, then locate the second record and do the same. Finally, we would perform the calculation. It would look like this.

```
Single file find on INDEX {value1}
Calculate #1 as NUMBER_FIELD
Single file find on INDEX {value2}
Calculate #2 as NUMBER_FIELD
Calculate AVERAGE as (#1+#2)/2
```

continued on page 35

Storing Lists

OMNIS 7 now lets us declare a field in a File Format to be a List-type field. The immediate advantage this gives us is that it allows us to create more than eight temporary (RAM-based) lists by adding List-type fields to a Memory-only File.

But this facility also allows us to *store and retrieve* Lists in records on disk. There are a number of interesting possibilities that come to mind for taking advantage of this feature. Here are a few of the more common ones.

Lists as Data

The concept of a List being a data *item* is an intriguing one. My operational definition of a List is “a collection of combinations of field values” and I have always used them as a temporary collection of *record images* until OMNIS 7 came along. Viewing a List as “a partitioned, multi-valued field” takes a little practice, but it leads me in the direction of some very practical uses.

Multi-Valued Fields

In the last issue of OmniScience, I mentioned storing multiple telephone numbers in a List-type field. This is still my most frequently used example of a multi-valued field. The concept is that, in our modern telecommunications and consumer electronics age, many people now have a variety of telephone numbers for different purposes. Simply using work and home telephone fields in a database ap-

plication is often not sufficient anymore. We now have fax numbers, car phone numbers, pagers, and the like to keep track of. (I have a few friends and clients with eight or even ten important telephone numbers of various kinds!)

As long as these telephone numbers are only auxiliary information to the personal information stored in this File, a List-type field is a very useful way of storing as many telephone numbers as are necessary for each record. By “auxiliary” I mean that the values stored

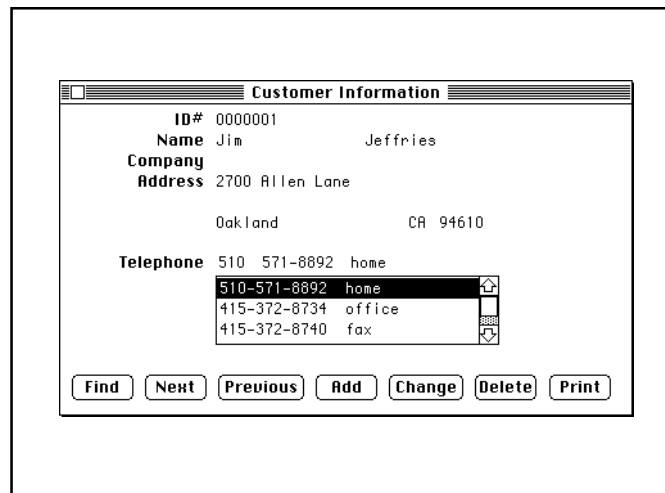


Figure 5 Telephone numbers stored in a List.

in this field are not used to *identify* the record as a uniquely indexed value. The reason I qualified this statement is that there is no facility for indexing List-type fields or their contents in the current (1.03) version of OMNIS 7, so we cannot locate a record using *Find*, *Next*, or *Previous* using this field or any of the information it contains. Searching records for a List that contains a certain value is also laborious at best. We should only store items not needed for locating records in a List-type field.

Given these restrictions, a field with multiple line and multiple columns can have some great uses once we learn how to manage it. Let's first set out the ground rules.

What Gets Stored

When we store a list value in a record, a lot of information is stored in that field. Obviously, the field values that make up the List are stored in the List-type field. In addition to that, the definition of the List is stored *for each record*. On the surface, this would seem to state that we could have different definitions for the contents of the same List-type field in different records. Although this statement is true, this variance of definitions from record to record would normally not be done in practice — and it takes a little extra work to be able to display the record-by-record contents if we choose to be inconsistent with those definitions. What we are being told here is that *we must define the List to be stored in each new record before each insert*.

In addition to the field values and the List definition, a number of other statistics are stored with a List. The values of #L, #LN, and #LM that applied to that List when the *Update files* command was issued are also stored in a List-type field. Furthermore, the *selected and saved states* of each line in that List are stored. All this information is retrieved and re-established for the List when it is read into the CRB as part of a record. If that List value is eventually displayed in a List field on a Window, the appropriate lines are highlighted.

Defining Stored List

All Lists are defined by the names of the fields that head the columns of that List. This is done using the *Define list* command.

In the last issue of OmniScience, I posed the question of what fields or variables to use for the column definitions of a stored List. After more work on the subject, I have found that the most convenient source of such fields for most of the situations I have encountered is a Memory-only File. These fields are globally accessible within the application, so they can be used on any window, in any procedure, or in any Report or Search Format. I will examine import and export considerations in a section later in this article.

The columns of a List must be defined before contents can be inserted by any means (other than calculated transfer — see below). If we clear the current record buffer for the Main File or invoke the *Prepare for insert* command, any List-type fields in the Main File area of the CRB are undefined. So we must first define these Lists before the operator can successfully perform data entry.

Data Entry

OMNIS 7 has a very clever way of allowing for data entry directly into the List *bypassing* the CRB. (Remember that the List is an area in RAM, *not* the List field that represents it on a window.) There are many ways of entering data into a List, but I feel that this method is the most appropriate for multi-val-

ued fields in a record. The workflow for the operator seems the most natural.

Let's return to our telephone number example and explore how we can set up a window for direct data entry of multiple telephone numbers. Figure 6 shows such a window in both field names and field numbers modification mode. Please refer to this figure for the following explanation of this method.

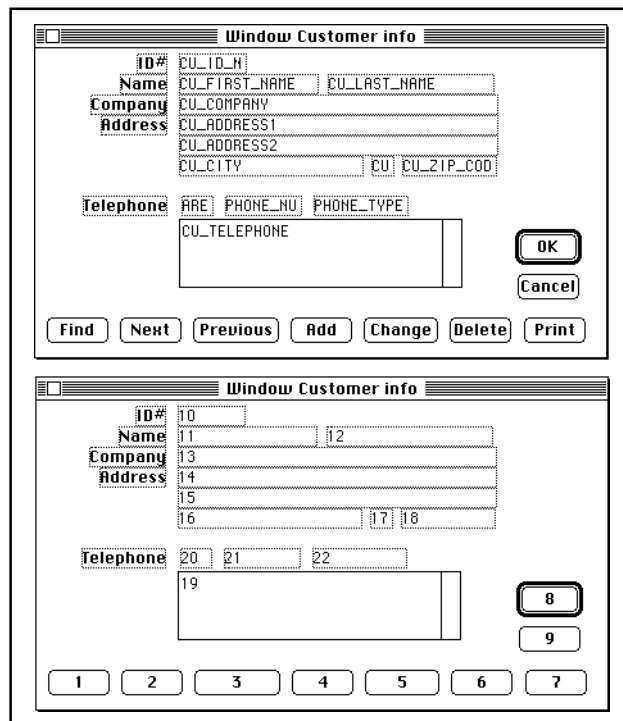


Figure 6 Data entry window using direct List entry method showing field names and field numbers.

If we place data entry or display fields on a window to represent the field names of a List, and make the *local* to a List field that represents that List, those fields become direct data entry points to that List in RAM, bypassing the CRB slots for those field names. Furthermore, when the cursor leaves one of those fields, the List field is automatically redrawn and will display the value that was just entered if it is included in the calculation for that List field.

In our example, the CU_TELEPHONE field is defined to contain the fields AREA_CODE, PHONE_NUMBER, and PHONE_TYPE from a Memory-only File. The List field that represents CU_TELEPHONE on this window is field number 19. The fields representing the columns in the List are field numbers 20 through 22 and are all given the "local" attribute. Those fields have now become data entry and display fields for the current line of CU_TELEPHONE.

When a List is cleared, or when it is defined, it contains no lines. The variables #L and #LN for that List are both set to 0. In order to perform our data entry task, our procedure must first *Add line to list* and then set the #L pointer to 1 so that there is a current line in which to enter data. *This must be done for each List-type field in the Main File.*

Although the value of each field is automatically redrawn in the List field as we tab from one data entry area to the next, there is no automatic facility for going from one line to the next. We must set up field procedures to perform this task.

There are three basic issues that these procedures must address. First, if we tab forward from the last field in a line, we want to go to the first field on the next line — and if there is no next line, the procedure needs to create one. Second, if we tab backward from the first field on a line, we want to go to the last field on the previous line — unless we are already on line number one, in which case we would like the cursor to go to the entry field that precedes the List entry area. Finally, we need to

determine what signal should indicate that the cursor should tab out of the List entry area to the following entry field.

We also need to find a way to keep the List field from becoming the current field during the insert or edit process since the operator can't enter anything there and might get confused.

The following procedure demonstrates a solution to our first problem: tabbing down a line in the List. It reacts to a #TAB message. If the current line is also the last line of the List, a new line is added. This line will be blank because our insert or edit procedure would have first cleared these memory variables in the CRB and our data entry processes never load values from the List into the CRB. The procedure then sets the line pointer to the next line, sets the current field to the first field on the line, and redraws the window. The redraw moves the highlight bar on the List field.

```
22  PHONE_TYPE
If #TAB
  If #L=#LN
    Add line to list
  End If
  Calculate #L as #L+1
  SNA set current field
    {[nam(AREA_CODE)]}
  Redraw windows
End If
```

The following procedure demonstrates the second and third of our issues from above. It is triggered by the cursor leaving the first List-entry field. Notice that the #CANCEL message must be excluded. #AFTER is generated by a Cancel, but we wouldn't want to go to all this bother if the process is being aborted. There are two cases addressed inside this field procedure.

The first shows a method of determining that all telephone numbers have been entered and that it is time to move on to another entry area on the window. My signal in this example is that the AREA_CODE field is left empty on tabbing forward. If this case is true, the cursor is put in the appropriate field. Here the cursor is put in the first entry field on the window since there are no entry fields following the List-entry area.

The second case concerns what happens if the operator shift-tabs from this field in either the insert or edit mode. (The INSERT and EDIT variables are from my Memory-only File and are explicitly set in my insert or edit procedure.) A further determination is made as to whether the current line of the List is line number 1. If it is, the cursor is placed in the entry field preceding the List-entry area. If it is not, the current line is set to the previous line and the cursor is placed in the last entry field in the List-entry area, and the window is redrawn to show the proper line of the List highlighted.

```
20  AREA_CODE
If #AFTER&not(#CANCEL)
  If #TAB&not(len(lst(AREA_CODE)))
    SNA set current field
      {[nam(CU_FIRST_NAME)]}
    Quit to enter data
  Else If INSERT|EDIT|#STAB
    If #L=1
      SNA set current field
        {[nam(CU_ZIP_CODE)]}
    Else
      Calculate #L as #L-1
      SNA set current field
        {[nam(PHONE_TYPE)]}
      Redraw windows
    End If
    Quit to enter data
  End If
End If
```

The next procedure is used to disallow the List field itself from becoming the current field during data entry. We can't just disable the field because the local redraw would not function properly. It is also possible that the operator may need to go to a specific line to make a correction, so this is the assumption that is made in this procedure. If the operator clicks or double-clicks on the List while in the data entry mode, the cursor is placed in the first field in the List-entry area. Since field attributes are always satisfied before field procedures come into play, the line on which the operator clicked has already been made the current line.

```
19  CU_TELEPHONE
If #CLICK|#DCLICK|#EDATA
  SNA set current field
    {[nam(AREA_CODE)]}
  Quit to enter data
End If
```

Cleaning Up

Once all the data has been entered for a record, there are a few items regarding the List we must clean up before writing the record to disk.

If you remember from earlier in this article, OMNIS stores *all* the statistics about the List in the record, including the current line number and the selection states of each line. We also are likely to have a blank line at the bottom of our List value as the result of the method I chose to signal the completion of the List-entry phase of data entry. This blank line should be removed as a gesture of good house-keeping. I always put my "cleanup" procedure steps in the main procedure for a process after the "cancel trap" and before the *Prepare for insert with current values* or *Update files* statement for inserts and edits respectively. (For a definition

and explanation of the term “cancel trap”, see page 184 of *Unlocking Omnis 3 Plus — Version 3.3*.)

As a matter of style, I reset #L to 1 before updating the List value. To remove a blank line from the end of a List, we must first determine if that line is truly blank, then delete it if it is. The last line number value equals #LN. One test is to concatenate all field values from that line and check for a zero length. Here is the finished procedure.

```

Begin reversible block
  Set read/write files
    {Customer file,System file}
  Calculate INSERT as 1
End reversible block
Clear main file
Clear range of fields
  AREA_CODE to PHONE_TYPE
Set current list CU_TELEPHONE
Define list {AREA_CODE..PHONE_TYPE}
Add line to list
Calculate #L as 1
Redraw windows
Enter data
If flag false
  OK message {Record Addition
    Cancelled at Your Request}
  Clear main file
  Clear range of fields
    AREA_CODE to PHONE_TYPE
  Redraw windows
  Quit procedure
End If
Calculate #L as 1
If not(len(con(lst(#LN,AREA_CODE),
  lst(#LN,PHONE_NUMBER),
  lst(#LN,PHONE_TYPE))))
  Delete line in list {#LN}
End If
Single file find on SY_SEQNO
  (Exact match) {1}
Prepare for Insert w current values
Calculate SY_NUM_CLIENTS as
  SY_NUM_CLIENTS+1
Calculate CU_ID_NUMBER as
  jst(SY_NUM_CLIENTS,'-7P0')
Update files
Redraw windows

```

Remember that all the List variables and procedure commands apply to the *current* List only. If your record contains more than one List, you must set the current List and perform these cleanup operations for each one. Your field procedures will also have to insure that the proper List is current so that commands like *Add line to list* are performed on the correct List.

Retrieving Lists

There is no magic for retrieving Lists stored in records, and no special techniques are normally required to properly use or display them. They are simply read into the Current Record Buffer and a specific List can be manipulated by setting it as the Current List.

There will be occasions, though, when you need the selected line of a retrieved List to correspond to some other value in the CRB. There are two basic methods we can employ for this.

The first method requires that the List contain a “key” field that exactly corresponds to the value from the CRB that is to be matched. The values of this field must be unique within the List. We can use the *Set search as calculation* command to set up matching criteria and then *Search list (from start)* to locate the proper line. Keep in mind that reference in the search calculation to any field name included in the List definition applies to that column in the List. If you need to locate a line in a List that corresponds to the current CRB value of the same field name, that value must first be transferred to a field name *not* included in the List definition. For example, locating a line in a List of State codes based on the current CRB value of ST_CODE would be done like this:

```

Calculate #S1 as ST_CODE
Set search as calculation
  {ST_CODE=#S1}
Search list (from start)

```

On the other hand, if we needed to locate the line in the List where the State code was the same as the State code in the Customer record, we don't need the intermediate calculation.

```

Set search as calculation
  {ST_CODE=CU_STATE}
Search list (from start)

```

The second method assumes that we have a static List (one that never changes its contents) used for lookup purposes and a corresponding field in a record that stores an integer value relating the record to a line in that List. This might be the case for a List of titles to be applied to a Customer record (Ms., Mr., Mrs., Dr., etc.). The List is stored in our System Constants File for easy and consistent retrieval. The following code segment redraws the List with the proper line selected.

```

Set current list SY_TITLES
Calculate #L as CU_TITLE
Redraw list

```

Transfer by Calculation

I haven't mentioned it before, but there is a very simple way of transferring information into a specific List. Calculating a List as the value of another List copies *all* the information from the second List into the first. This includes the definition of the List, the values of #L, #LN, and #LM, and the selected and saved states of each line.

This can be used as a means of quickly transferring data between a stored List in a record and a RAM-based List from either a

Memory-only File, a local or format variable, or one of the #Lx temporary List variables.

For static Lists that are stored in a System Constants record, this method can be used for quickly bringing those Lists into RAM. This frees the System Constants record from the responsibility of always being in the CRB and available for reference. There are many other uses for this technique.

Redefine List

There may be occasions when the fields in the definition of a List don't exist in the CRB. You may have defined the List using format or local variables, or you may have imported the record that contains the List using the *Omnis data transfer* method and the File Format or Memory-only fields that the old application used are not represented in the new application. However it happens, there is no need to be alarmed. We can use the *Redefine list* command to make the List value accessible again.

Redefine list assigns new names to the columns in a List *without changing the data types that already exist*. The columns are redefined in the same order they were originally defined. Fields that are to retain their current names can be skipped, but a delimiting comma must be provided for each one in the parameter list for the *Redefine list* command.

If the CRB data type of the new field name for a column doesn't match the data type of the column, OMNIS will do its best to map values between the two when values are transferred from the CRB to the List of vice versa. This usually goes smoothly, but there can be some unresolvable mismatches

if you aren't careful. For your own sanity, do your best to match data types for redefined column headings whenever possible.

Managing Definitions

There may be occasions when you must store List values with different definitions in different records in the same File. (This doesn't come under the heading of methods I prefer to use, but there may be a real-world need for this, so it must be explored.) OMNIS has no problems with storage and retrieval of these List values, but *you* may have some problems *displaying* these Lists in a List field on a window.

The problem is: How can we determine a calculation for the List field to display the List contents *no matter what fields define the List*.

To perform this feat, we need to know how to do three things. First, we need to be able to determine what columns the List value contains. Second, we need to be able to determine how wide a column must be (in number of characters) to display its maximum value length. Finally, we need to be able to dynamically tell the List field how to display values based on those column definitions and widths.

The first task is relatively easy. There is a command called *Build list columns list* that puts the names of the columns from a specified List into the current List on separate lines. In its simplest form, this command builds a List of column names from the target List. If a second string field is supplied in the List that is to hold the column names, the *data type* of each column is included in the second field. This can all be very useful information for our second step if we can manage to parse it out.

Consider a List defined to contain #S1 and #S2, in that order. #S2 on each line will contain all the information we need to determine display width and other characteristics of the field whose name is in #S1. All we have to do is process its value to extract what we need. There are only a limited number of data types an OMNIS 7 field can have and we can make assumptions about the widths of most of these. Also, there are some of these data types that we can't display in a List field on a window. With the exception of *Short number*, *Short date*, and *Short time*, the first two characters of each available data type are unique.

Short numbers have a maximum of nine significant digits (plus a decimal point for the 2 dp variety) and the standard short date format (D m Y or m D Y) requires a maximum of nine characters. Short times require even fewer characters. And all three of these data types line up best in columns if they are right justified, so we can unify the way we handle fields with a data type that begins with "Sh".

Character and *National* data types include the maximum number of characters in their definitions, so we can extract that information from #S2. The values that are loaded into are List are of the form "Character 20" and "National 35". We could extract the numeric part of this string and place it into, say, #1 with the following statement.

```
Calculate #1 as mid(lst(#S2),10,6)
```

The space that would be extracted in the case of a *Character* field is ignored by OMNIS.

Integer fields can be assumed to have a maximum of three characters and should be right justified.

Boolean fields also contain a maximum of three characters. *Sequence* fields are the same as short numbers with zero decimal places, so we can assign them a maximum of nine characters and right justify such a column. *Date and time* fields can pose some additional problems if we want to use their assigned formats, but we can make the assumption (for brevity here) that they should use the standard #FD format for the List display.

Picture, List, and Binary data types can't be displayed in a List field, so we'll ignore them and not include them in our List field calculation.

To redraw the List field on the window with the proper columns, we need to use the *eval()* function for the calculation attribute of that field and we have to supply a string value for that function that contains a syntactically correct *jst()* function. To do this, we will use the string variable #S5 as a buffer in which to build up the complete *jst()* expression and perform the operation in a *Repeat* loop. Each iteration of the loop will parse the next line of the List columns List until there are no more lines.

```
Clear range of fields #S1 to #S5
Set current list #L1
Define list {#S1,#S2}
Build list columns list {TARGET}
Calculate #S5 as 'jst('
Calculate #L as 1
Repeat
  Load from list
  If pos(mid(#S2,1,2),'ChNa')
    Calculate #1 as mid(#S2,10,6)
    Calculate #S5 as
      con(#S5,#S1,',',#1+1,',')
  Else if
    pos(mid(#S2,1,2),'ShDaNu')
  If pos('um',#S2)
    Calculate #1 as
      mid(#S2,pos('dp',#S2)-3,2)
    Calculate #S5 as
```

```
      con(#S5,#S1,',',chr(39),
        '-9N',#1,chr(39),',')
  Else
    Calculate #S5 as
      con(#S5,#S1,',',chr(39),
        '-9D',chr(39),',')
  Else if mid(#S2,1,2)='In'
    Calculate #S5 as
      con(#S5,#S1,',',-3,',')
  Else if mid(#S2,1,2)='Bo'
    Calculate #S5 as
      con(#S5,#S1,',',-3B',',')
  Else if mid(#S2,1,2)='Se'
    Calculate #S5 as
      con(#S5,#S1,',',-9',',')
  End if
  Calculate #L as #L+1
Until #L>#LN
Calculate #S5 as
  con(mid(#S5,1,len(#S5-1),'))
Redraw windows
```

The expression for the calculation attribute for the List field representing TARGET is simply:

```
eval(#S5)
```

Redrawing the window re-evaluates this expression and remaps the values from the columns in the List for display in the List field.

Reporting Lists

Another challenge is including a List-type field value in a record in a report. Currently, OMNIS 7 has no direct facility for including the contents of a List-type field in a report as a unit. Report Formats do not have a List field as do Window Formats and List-type field values do not map directly to any other Report Format field type.

Two methods for performing this task were detailed in the last issue of *OmniScience* on page 13. After more experience with these methods, I find the second one mentioned to be superior in most cases. In this method, the lines of the List

are loaded into a long string field. Each line is formatted using the *jst()* function and the lines are delimited in the string with carriage returns, expressed as *chr(13)*.

The only drawback to this method is that the report must be procedure-based. The List-type values must be processed by a procedure on a record-by-record basis.

The facility for directly reporting List-type field values is such an important one that I am sure the people at Blyth intend to include it in some future release of OMNIS 7.

Stored Tables

So far, we have discussed Lists associated with individual records. Another use of Lists is to store lookup tables in a System Constants File. These might be lists of shipping charges, tax tables, etc. that we need to access for operations within the application.

In their simplest form, these Lists would contain two fields: a lookup range cutoff value and a return value. For example, a List of shipping charges might begin with a line that indicates that a package weighing up to 1 lb. cost \$1.35 to ship. This would be followed by a line that indicates that a package weighing up to 2 lb. cost \$1.85, etc. The final line would have to contain a cutoff value much larger than would ever be expected. The table could also be cast with this value being at the low end of the range for the return value. More complex tables might have multiple columns of return values or of selection values or both.

For efficient use of such Lists, they should be stored in properly sorted order. If the cutoff is at the high

continued on page 35

OMNIS 7 and SYBASE®

When this article was first conceived, SYBASE had a number of features that made it unique in the SQL database world. Recently, other vendors, notably Oracle, have announced versions of their products that contain some of the features mentioned in this article. Needless to say, Sybase, Inc. plans some technological leapfrogging as well. OMNIS 7 has special SQL features that are designed to take advantage of SYBASE features.

Integrity Above All Else

Talk with anyone from Sybase, Inc. about the purpose of any feature of their product and the same phrase comes up again and again — data integrity on the server. That is the primary focus of this company and the rationale behind the design of their product and the special features they provide in it.

In the SYBASE environment, Client/Server features are taken to the extreme. Not only is the *data* managed at the server level, but much of the *procedural programming* normally associated with an application at the client workstation can also reside on the server. This cuts down on network traffic and on the number of steps required to perform SQL tasks.

SYBASE is also designed with the rest of the world in mind. The Open Client application programming interfaces (API) allows non-SYBASE tools and application programs (like OMNIS 7 applications) to communicate with SQL Server.

The Open Server API gives SYBASE access to non-SYBASE data sources and services.

Virtual Server

SYBASE SQL Server™ is a *database operating system* running on top of the serving computer's native operating system. As such, it handles many of the multi-user functions allocated to the native system by most other products, including scheduling, task switching, disk caching, and locking. SQL Server is optimized to perform these

running on each processor. On other systems, this responsibility falls on the native operating system, which can only manage tasks at the process level.

Stored Procedures

This is one aspect of SYBASE's unique *programmable server*. Stored procedures are written in Transact-SQL, SYBASE's set of extensions to ANSI-standard SQL. They are then compiled and stored in SYBASE's data dictionary on the server. To use a stored procedure, the application just issues a procedure call and passes the necessary parameters. Stored procedures can call other stored procedures, both on the current server and on other systems.

A single procedure call becomes the equivalent of many individual SQL statements. This enhances performance in two ways. First, network traffic is drastically reduced from that required for traditional SQL queries. Second, since stored procedures are already compiled, they can be executed five to ten times faster than an equivalent sequence of single SQL statements.

Stored procedures are used to set up enterprise-wide business rules, transactions and parameterized queries that can be accessed by any application residing on any client workstation, making the job of writing *consistent* front end applications for multiple platforms a lot easier. Making the database itself more intelligent has many advantages at all levels of database use. The MIS department can maintain better control over the database

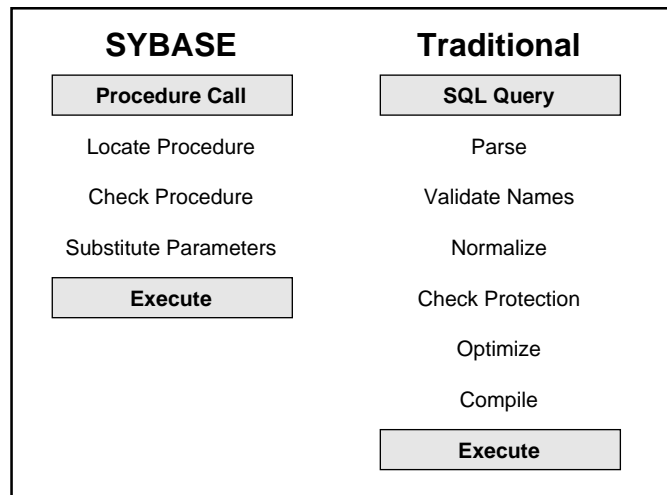


Figure 7 A compiled stored procedure with multiple SQL statements is more efficient than the traditional approach.

tasks in the best manner for SYBASE, so most of the inefficiencies and operating system overhead that cause problems on other systems are eliminated.

The SYBASE Virtual Server Architecture™ allows SQL Server to take full advantage of all the processors on a multi-processor computer system. SQL Server provides the task management services for dynamic and transparent load balancing *at the SQL statement level* with only one SQL Server process

because most processes are centrally located. Departments and end users can use a wide variety of front end tools and user interfaces while avoiding most performance and reliability problems.

Triggers

Triggers are a special type of stored procedure used to enforce complex business rules and referential integrity at the server level. Unlike normal stored procedures that must be explicitly called, triggers are *automatically* invoked by actions on the database.

There are three types of trigger: Insert, Delete, and Update. They are associated with a specific column of a table and are stored in the data dictionary with the other column attributes. This is a much safer and more secure arrangement for monitoring database integrity than passing that responsibility to the various application programs on client workstations. It also ultimately cuts down on the amount of code that must be written for the overall system because it eliminates the need for redundant integrity code to be included in every application module.

Triggers are generally used to monitor integrity issues that involve multiple fields or multiple tables. *Referential* integrity means that a trigger can maintain consistency between the primary and foreign keys of a database. A statement that embodies a possible trigger function of this type is "Do not delete inventory items for which there are open orders".

If a multi-record updating process is launched and even one of those updates violates a trigger, the entire transaction is rolled back. Pretty powerful stuff!

Defaults

Default values may also be assigned to columns at the server or data dictionary level. This operates a little differently than those of us who are used to personal computer database applications would expect. A default value in SYBASE is assigned to a field *when a new record is created if not value is explicitly entered*. A default value does not automatically appear on the screen of a front end application when an operator begins the data entry process for a new record. This is a result of the data being managed by the server instead of by the application. There are ways, however, that a front end application can access the default information on the server to create a more user-friendly interface.

Coupled with the concept of a default value is the concept of a NULL value. NULLs are different from blank or empty values. They are used to indicate that no entry has been made to a field. Stating that a field has had an empty value entered is different from stating that no entry has been made at all. The current (1.03) version of OMNIS 7 does not have the concept of a NULL value. All field types receive some value when a record is inserted even if no value has been assigned. The closest that an OMNIS 7 datatype comes to using a NULL is with the Boolean field type which is neither YES nor NO is no value is entered. A column in SYBASE can be defined to either accept or not accept NULL values.

Rules

Rules are like triggers in a way, but they are much simpler and are field-specific. They specify the domain of values allowed for a specific field. The domain can be a

range of values, a set of values, or a format for a value. For example, a United States Social Security Number could be restricted to nine numerals with a dash placed between the third and fourth and between the fifth and sixth (that is, to match the pattern 999-99-9999). The rule doesn't place the dashes in the string, but it enforces the requirement for their existence and proper placement.

The definition of a rule is an expression that defines a restriction on a field or a user-defined data type. It is stored in the data dictionary and can be bound to any column. The same rule can be bound to more than one column or user-defined data type.

As with triggers, if a rule is violated by a single field in a multi-record process, the entire process is rolled back. Of course, error messages are created by the system to aid in correcting the problem.

Two-Phase Commit

This rollback is made efficient by a sophisticated capability of SYBASE called a *two-phase commit*. This service guarantees consistency of the data in the event of any failure — hardware or software.

A transaction is the basic work unit of SQL Server. It may consist of several SQL statements that update and retrieve information stored on a server. In distributed data environments, a single transaction may need to affect data on many SQL Servers. An extra effort is required to maintain data consistency across multiple machines — especially if failures occur during the execution of a transaction.

When a multi-record or multi-table update is required and the data is

spread over more than one server, the application at the client workstation designates one of the SQL Servers as the *commit server*. This server acts as a recordkeeper for the transaction and decides whether to commit that transaction or roll it back once the first phase has been completed.

In the first phase, each SQL statement is processed by its server. That server then sends a message to the commit server indicating its readiness to commit its portion of the overall transaction. In the second, the commit server broadcasts a message to all involved servers to commit their portion and record the transaction. At this point, the transaction is committed regardless of subsequent failures.

If any server fails during this process, the commit server cancels the entire transaction and tells each participating server to roll back its portion. If the commit server should fail, the transaction is automatically rolled back.

Transaction Log

SYBASE goes to great lengths to maintain system integrity even for single server transactions. Each server maintains a log of every transaction that is performed by every user as it occurs. The resulting *transaction log* serves two basic purposes. First, the log is used to roll back a transaction if it is cancelled at some stage. Second, it is used to rebuild the database from the last backup if there is some sort of crash or other catastrophe.

The transaction log is built on a *write-ahead* basis. This provides for complete recovery because each transaction is written to the log *before* the database update is written to disk. Transactions cannot be

performed and not logged. All changes to the database can be rerun from the log.

SQL Server uses a *physical logging method* for generating this log. In this method, only the offset position and byte stream are logged, rather than the logical record. Since SYBASE is managed on a *page* basis rather than a *record* basis, this method is more efficient.

RPCs

SYBASE also allows server-to-server communication between SQL Servers through Remote Procedure Calls (RPCs). The facility for *back ends* to communicate directly with one another is unique to SYBASE. Any server that supports the SYBASE Tabular Data Stream (TDS) RPC protocol can participate in this communication. This facility extends SYBASE's Open Client/Open Server architecture and gives an application a simple and clearly defined interface with a single SQL Server that can access data on a number of other servers transparently.

The first advantage to this system is that applications don't need to know where all the data is. Applications simply call stored procedures in the main SQL Server which, in turn, access other servers through the RPC mechanism.

The second advantage is even more powerful. Using the TDS RPC protocol, parameters can be sent to the remote procedure *in their native data format*. Standard SQL requires that data be converted to an ASCII string on the sending end and back to an appropriate data type on the receiving end. Not only are these translation steps eliminated, but many native data types are more compact (require fewer

bytes) than their ASCII form. Using RPCs results in greater throughput. Result sets can be sent via the RPC mechanism as well.

SQLSERVE Connect

So how does OMNIS 7 take advantage of these superior features? Well, first of all, most of these apply to *any* front end to a SYBASE database. We just need a mechanism to send SQL statements. The SQLSERVE connect is our link to SQL Server. It processes requests from OMNIS applications and calls the API functions in SQL Server.

As with all SQL access through OMNIS, the *Send command to remote* command is the gateway to SYBASE. For example, to execute a stored procedure, we just have to send the appropriate SQL string.

```
Send: <BEGIN>
Send: exec procedure name
Send: <END>
Send: <EXECUTE>
```

We'll discuss how to deal with the multiple select tables and multiple error messages that a stored procedure could return a little later.

We can *create* a stored procedure or a trigger as well as creating tables, column definitions, rules and defaults from OMNIS 7 in the same way. The magic here resides fully in SYBASE and SQL Server and is accessed through SQLSERVE.

OMNIS 7 has a great many SQL keywords that are used to smooth out the differences between different vendors' implementations of SQL and to act as a shorthand for many processes. These keywords are translated into a SQL statement in the proper syntax for the current host database platform. They work with SYBASE through

the SQLSERVE connect as they would with other platforms. For details of these keywords, refer to the *Omnis Connects Manual*.

One OMNIS 7 SQL keyword that can't be used with SYBASE is <GROUPCHANNEL>. The mechanism for building transactions that access multiple tables is different in SYBASE than in other SQL database managers and multiple open channels acting as a group (to be committed or rolled back as a unit) is a foreign concept. The two-phase commit process are much more efficient — as well as being the SYBASE way of doing the job.

SYBASE Keywords

Although standard SQL access from OMNIS 7 is no different for SYBASE as for any other product, gaining access to SYBASE through Remote Procedure Calls is a different matter. OMNIS 7 has a few SYBASE-specific keywords to streamline the process of calling and receiving output from RPCs.

The basic keyword for calling RPCs is <RPC>. This keyword is used to define and execute a stored procedure on the current SQL Server or a remote procedure on an Open Server application or any other DBMS that supports the TDS RPC protocol. The keyword must be sent with the name of the procedure to be executed.

In addition to the remote procedure name, up to eight parameters can be passed. Each parameter must include the SQL Server data type of the parameter and the OMNIS field name containing the value being sent. The name of the corresponding RPC parameter and the passing of a NULL value can optionally be included with any of these eight parameters.

Parameters can also be specified for RPC output. In this case, the RPC puts the value directly into the OMNIS field. Again, the data type and OMNIS field name are required as well as the option to indicate that this field is to be an *output* field — That is, These parameters are passed by reference. Further details on the proper syntax for this keyword are found in the document *RPC.doc* supplied with the OMNIS 7 Plus kit.

The only limitation is that only one RPC can be called at one time by an OMNIS application, regardless of how many SYBASE channels are open. An attempt to execute a second RPC before the first has been completed will generate an error.

An RPC may return one or a number of select tables besides the return values specified for output fields. These select tables must be *entirely* processed by either <FETCH> or <BUILDLIST> calls before the return values can be processed. Sending the keyword <RPCRESULTS> processes the parameters that were passed by reference, placing the resulting values into the proper OMNIS fields.

The <RPCRESULTS> keyword can also be used to retrieve the return status after any stored procedure by invoking it with a *status* parameter. This is the name of a *Short number* field in OMNIS that will receive the return status value from that stored procedure. Again, all result rows must be processed first.

If your application needs to execute a stored procedure on the server that calls one on a remote server, the <RPCPASSWORD> keyword must be invoked for that remote server. To do this, you must already be logged onto the remote server. Any *new* channel to that

server can then be used for RPCs.

Error Handling

When stored procedures run, multiple error messages are generated by SQL Server. OMNIS 7 can handle these messages, but must do so in a separate procedure. The keyword <SQLEERROR> is used to specify that procedure. The Format name and number of the procedure are given as a parameter of this keyword. This procedure monitors errors in the same way that control procedures monitor events.

SQL Server also generates multiple messages which can be captured by a similar procedure specified using the <SQLMESSAGE> keyword. Examples of the implementation of these keywords, and of the procedures that process the incoming information can be found in the SQLEXPRESS application shipped in the OMNIS 7 Plus kit.

Multiple Result Tables

Stored procedures, as well as <BEGIN>..*END*> statement groups, may produce multiple result tables from SYBASE. The best method for retrieving these results is to read each set into a List using the <BUILDLIST> keyword. That keyword will return a *flag true* condition if the List is successfully built. A *flag false* condition exists between result tables. This should be cleared by issuing a <FETCH> between List builds. A *flag false* after a List build indicates that all results have been processed.

SYBASE is a registered trademark of Sybase, Inc. SYBASE SQL Server, Client/Server Interfaces, Transact-SQL, SYBASE Virtual Server Architecture, Open Client, and OpenServer are all trademarks of Sybase, Inc.

SQL Reports

One of the powerful features of OMNIS 7 — and a major reason many companies have invested in it — is its ability to access a wide range of remote, SQL-speaking databases. Although there are many SQL-specific features for data entry and retrieval, there are no commands, functions, or keywords for generating reports from these databases.

On the other hand, one of OMNIS 7's major strengths is its Report Generator. Every aspect, from mechanical details like the ability to sort and subtotal on a number of different levels simply to enhancements like the inclusion of text and graphic and the use of fonts, recommends this feature.

Perhaps there is some way we can mesh the report generating capabilities of OMNIS with the SQL access features to build reports directly from the host database. Here is an overview of some potential avenues.

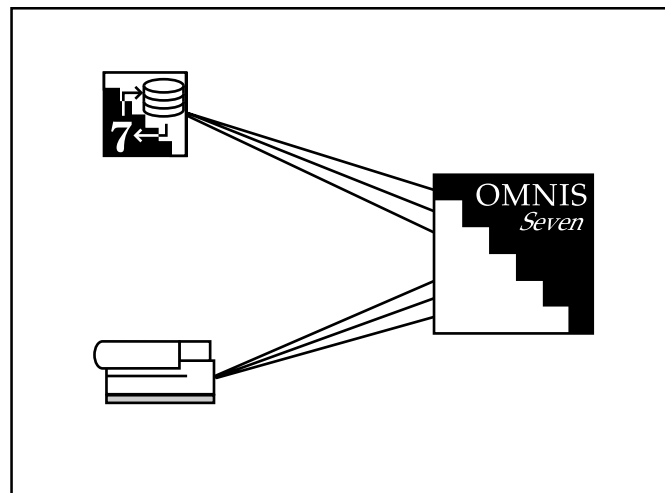


Figure 8 OMNIS-SQL data sharing paths.

Report Features

First, let's review the ways that OMNIS 7 can generate reports. There are three basic ways. The first method is to let the OMNIS Report Generator do all the work by issuing a *Print report* command. All that needs to be done is to set up a Report Format with the fields properly arranged, the subtotal levels organized, and the appropriate Search Format in force. OMNIS then manages the process of locating and ordering records from the datafile based on those criteria.

There are some limitations to this method. The primary one is that sorting and subtotal triggering can only be performed on Main File and Connected File fields. If a programmer decides not to use Connections, this can be *very* limiting.

In describing the *Print report* method above, I only mentioned printing a report from the datafile through the CRB. OMNIS 7 can also use the *Print report* command directly on the contents of a List if the Report Format is set up correctly. The Report Format must

use the names of the fields in the List. It is given the name of a List instead of a Main File in the appropriate place in the Sort Fields window. The *Base report on a list* option in the Sort Fields window must also be selected for this to be carried out properly.

This variation also has a few limitations, such as *only* being able to report values of fields that are defined in the List. But it can be used very effectively and is relatively easy to implement.

The second method requires more work and planning to implement, but allows the programmer much finer control over the reporting process. This is the method of using the *Print record* command. The Report Format must still be set up properly, but a procedure must also be created that locates the necessary records in the correct order and sends them to the Report Template one at a time. The Report Template still monitors page breaks and the printing of Subtotal Sections based on changes to the corresponding sort fields.

The extra work required to set up this type of report is usually greatly outweighed by the advantages it offers. Using a List as a sort buffer, the process followed by *Print report* can be emulated and extended. Whether or not Connections are used, sorting and subtotal triggering can be performed using fields from Files at many relationship levels. (This is still far easier if Connections *are* used.)

Finally, OMNIS 7 also includes an Ad Hoc Reports facility. Ad Hoc Reports work in much the same way as the *Print report* command — at least for the purposes of this discussion. That is, when the *Print ad hoc report* command is issued, the reporting facility “takes over” and generates the entire report — either from the datafile through the CRB or from a specified List. The same limitations apply to this method as to the one using *Print report*.

The Ad Hoc Reports facility has the advantage that the operator can be trained to create reports as they are needed. The disadvantage

here is that the operator also needs to know quite a bit about the structure of the datafile, Lists, and nuances of the AHR facility itself to successfully use it for reports of any complexity. SQL-based reports would certainly require such an understanding.

SQL Data Retrieval

Data retrieval from SQL databases is based upon the issuance of a *select* statement to the database server. This statement includes information on which fields are to be retrieved and can optionally include filtering (search) and ordering (sort) information.

SQL databases generally deal with *groups of records* called *selections* rather than individual records as OMNIS does, so the *select* operation will most often retrieve a number of record images at once in the following way.

If any records in the host database match the filtering criteria (which may not be included, indicating that *all* records are required), record images with the specified fields are retrieved as a *select table*, which resides in a scratch file on disk at the workstation. This scratch file is associated with the current *channel* and is used as a buffer for OMNIS to allow it to process each record image retrieved on that channel as it is able, since OMNIS isn't designed to handle all of them at once.

The *select table* is normally accessed and record images are read into the OMNIS application by issuing either the <FETCH> or <BUILDLIST> command. These commands are the data transfer links between the scratch file on disk and the corresponding area (CRB or Current List respectively)

in the RAM space allocated to the OMNIS application.

Another command, <RETRIEVE>, can be used to set up an *import file* containing the *select table*. It does not transfer values directly to the application.

Here is briefly how each of these commands is used.

The <FETCH> command reads the next record image from the *select table* into the CRB for the current application. The column values from the *select table* are read into the corresponding fields set up with the <DEFINE> command. Records are FETCHed, one record at a time, in exactly the same order they were located by the *select* statement on the server.

The <BUILDLIST> command reads all remaining record images from the *select table* into the current List, appending those lines to any that already exist in the List. The *Define list* command must already have been executed to set up appropriate column definitions in the List. The first column in the *select table* is mapped to the first column in the List definition, etc.

If some <FETCH> commands have already been issued for this *select table*, only those records that have not yet been read into the CRB are read into the List. To perform modification operations on these record images, they must still be read into the CRB one at a time using the *Load from list* command.

Records can also be imported from the *select table*, but not from the scratch file. Instead we can have the *select table* returned from the server sent directly to a tab-delimited import file by performing the following steps.

Assuming that the application already has a suitable File Format, we need to first assign an import file using the *Set import file name* command. This name is then available in *sys(22)* if we need to retrieve it.

We next associate this import file with the current SQL channel using the <FILENAME> command. Then we must use the <OPEN> keyword to open that import file. Once the import file is ready to receive the *select table*, we can issue our *select* statement, followed by the <RETRIEVE> command. This command reads the *select table* into the *import file* instead of the scratch file used by <FETCH> and <BUILDLIST>. Note that these two commands *cannot* be used after a <RETRIEVE> because the scratch file has no contents.

Finally, we must issue the <CLOSE> command to close the import file and perform other cleanup operations. This *must* be done before we can import any of these values into OMNIS.

Using our favorite means, we can now import the selected records. Here is a procedure segment that performs these operations.

```
Set import file name {Import file}
Send: <FILENAME> Import file
Send: <OPEN>
Send: <BEGIN>
Send: select * from tablename
Send: <END>
Send: <EXECUTE>
Send: <RETRIEVE>
Send: <CLOSE>
;import operation
```

An import file like this can also be shared with other application programs — Excel, for example — instead of (or in addition to) importing to OMNIS. An “import” of

data can also be done using <FETCH> from the scratch file and *Prepare for insert* in a *Repeat* or *While* loop, and this may prove to be a more straightforward method. But the data in the scratch file can't be directly shared with other application programs.

SQL Reporting

To generate a report based on data retrieved from a SQL database, we need to select an appropriate combination of SQL data retrieval method and OMNIS 7 reporting method. Some combinations will be better than others, and the ultimate choice will depend on how we configure our system.

Let's first eliminate the least likely combinations.

The Bad Choices

To use the *Print report* command in its normal mode would require the data to be pulled into a local OMNIS 7 datafile. If we chose to use a SQL database solution because we needed to manipulate more data than OMNIS 7 can handle, attempting to download all that data for reporting purposes is an obvious bad choice.

Even if we are dealing with a relatively small subset of the data, the number of records we might need to bring into OMNIS could still be massive. It would require a lot of hard disk storage and a lot of *time* to download the information. This approach is not a very direct one and is probably not practical for most cases.

Even if the number of records and the time involved in importing them were manageable, you may not want them to become a permanent part of the datafile. Some systems

might be designed for downloading from the host — although it is more likely that systems would be designed where the data from a number of satellite databases is *uploaded* to the host, either for consolidation or for permanent central storage. Importing just for the purpose of reporting requires that you also employ some method of flushing the data from the datafile once it is no longer needed there.

Fetch to CRB

A much more practical method is to read each record image from the scratch file using the <FETCH> command and perform a *Print record* for each <FETCH>. This is the method generally shown in the OMNIS 7 manuals. It offers a great deal of flexibility since the data can be further manipulated in the CRB before being written to the report. The basic form of a procedure for doing this (without all the error checking steps) would be:

```
Set report name  report
Send: <BEGIN>
Send: select * from  tablename
Send: <END>
Send: <EXECUTE>
Send: <DEFINE>  fieldnames
Send: <FETCH>
Prepare for print
Repeat
  Print record
  Send: <FETCH>
Until flag false
Print totals
```

The main problem with this method is that it relies on the order the record images are received from the server. We may very well want to sort them in some other or more complex order for our report.

True, we can include sorting criteria in our *select* statement, but that may not be adequate for our needs.

Buildlist for List Report

Another combination that may prove very useful and practical in many cases is to perform a <BUILDLIST> on the select table and then perform a *Print report* command using a Report Format that is designed to report directly from that List.

The Report Format would carry the sorting criteria we have specified, or we could override those sort fields by issuing a *Clear sort fields* command followed by one or a number of *Set sort field* commands. The record images in the List are automatically sorted for reporting purposes according to these criteria — without disturbing their actual order in the List.

The basic form of a procedure for doing this (again without all the error checking steps) would be:

```
Set report name  report
If necessary
  Clear sort fields
  Set sort field { sort level 1 }
  ;and so on
End if
Send: <BEGIN>
Send: select * from  tablename
Send: <END>
Send: <EXECUTE>
Define list { fieldnames }
Send: <BUILDLIST>
Print report
```

The only major drawback to this method is that all fields that are included in the report *must* reside in List. The *Print report* command does not have direct access to the datafile if it is set to *Base report on a list*. This is not normally a problem, since the List was retrieved from the select table and the select table is what we want to report on. But more complex system designs may find this a limitation.

For example, one very legitimate way that some developers are setting up OMNIS-SQL applications is to keep certain static tables in a local, single-user or “foreign”, OMNIS datafile. This is done to minimize network traffic by not requiring any but key field values relating to this File from being retrieved from the server. The local datafile can supply any required descriptive field values through *Single file find*, *Automatic find*, or *lookup()* methods.

In essence, we have created a hybrid system here and we can create *joins* of the data from the server and the static data from the local datafile in the CRB.

The problem with reporting from a List in such a system is that *Automatic find* does not function in List-based reports. Records from the datafile can't be read into the *List* in this way — and the CRB is a non-issue since it is not accessed from such a report.

It is possible to perform lookups on individual fields within such a report if the proper lookup channels have been established, but this method is still cumbersome in this scenario. If part of your database is in a local OMNIS 7 datafile (lookup tables, etc.), this may not be a good reporting method to use.

It is still desirable to have the sorting capabilities of the List at our disposal, so how can we manage that and still allow for the flexibility of manipulating data through operations in the CRB?

Load from List

The answer is to use the List for sorting purposes, then load those values from the List into the CRB, perform whatever manipulations

are needed, and *Print record* for each line of the List. This is more complicated than the other two viable methods described above, since it is very intensely procedure based, but it also offers the most flexibility of any of the methods discussed.

The basic form of a procedure for doing this (again without all the error checking steps) would be:

```
Set report name report
If necessary
  Clear sort fields
  Set sort field { sort level 1 }
  ;and so on
End if
Send: <BEGIN>
Send: select * from tablename
Send: <END>
Send: <EXECUTE>
Define list { fieldnames }
Send: <BUILDLIST>
Sort list
Calculate #L as 1
Prepare for print
Repeat
  Load from list
  ;perform manipulations here
  Print record
  Calculate #L as #L+1
Until #L>#LN
Print totals
```

Using this method, we can read in other records from the local datafile using the *Single file find* command. Our Report Format could perform the same task through the use of *Automatic find* fields. We have complete control over sorting and can sort on any nine fields retrieved from the server by the *select* command that was issued.

All we had to do was issue explicit commands for every action we wanted OMNIS to take. After retrieving the select table with <BUILDLIST>, we had to explicitly sort the List. We then had to go to each line in the sorted List and

load it into the CRB. The procedure could then process that record image in whatever manner we might require before sending it to the printer via the Report Template. It's not so complicated when it's broken down into steps!

Internal Statistics

In some reports, it is necessary to perform calculations (such as averages) or display values (such as “3 of 362”) based on the total number of records being reported. When printing *all* the records from a File in an OMNIS datafile, we can know the number of records *to be printed* by accessing the value of *sys(82)*, the number of records in the Main File. Is there, perhaps, some equivalent value we can access when printing SQL-based reports?

It so happens that *sys(135)* contains the number of rows retrieved from the select table. This *sys()* function value is of more benefit to us with some of the methods I've described than with others.

For example, if we decide to use the <FETCH> to CRB method, we are left out in the cold. We wouldn't know how many records were in the select table until the *last* <FETCH> is issued. *sys(135)* returns the number of records that have *currently* been retrieved. This could be useful for calculating an average in the Totals Section, but is completely useless to us if we need to show something like a *percentage of the total* figure on a record-by-record basis.

The <BUILDLIST> methods, on the other hand, yield this value to us *as soon as the record images are read into the List*. This is long before we need that value in the report itself. For that matter, we could also use the value of #L.

Tips and Techniques

This section is used to describe short, useful features or techniques that can make your OMNIS programming easier or more efficient. Sometimes a specific example may not be of direct benefit to your immediate projects, but the methods employed could still be helpful.

Your requests and suggestions for this column are welcome and will be properly credited if used. (It may take an issue or two to fit yours in.)

Currency Exchange

An increasing number of OMNIS developers in the United States are becoming involved in designing applications for international use. These applications often need to deal with differences in the currencies used in various countries. European, Australian, and Latin American developers have had to deal with this for a long time, but some new tools in OMNIS 7 can now make the job of currency conversion a lot easier. (Even when the EC unified currency goes into effect, there are still Japan Australia and the States to contend with.)

Programming currency conversion processes requires the use of an Exchange Rate File. This File will have a minimum of two fields: a uniquely indexed country code or name and an exchange rate relative to some base rate — the choice of base rate doesn't really matter, but one usually chooses one's country's own currency for convenience in obtaining the rates.

Other fields, such as the full name of the country, the name and symbol for that monetary type, etc., could also be included.

The traditional method of manipulating these records and performing the conversion operation is to locate the record containing the exchange rate of the currency we are converting *from* and store that rate in a temporary variable. We then locate the record for the currency we are converting *to* and do the same. The country codes would have already been captured in temporary string variables during a data entry process. Our procedure then calculates the numeric amount of money in the new currency by dividing the original amount by the *from* exchange rate to bring that amount to a common basis, then multiplying that result by the *to* exchange rate.

The method I intend to demonstrate here involves the use of the *lookup()* function to eliminate the record location steps employed above. I will also use parameter passing to a subroutine to generalize the process. We can then pass the appropriate information to this central subroutine from anywhere in the application.

Assuming that a lookup channel named "rate" has already been established to the Exchange Rate File in our datafile, we can call our conversion subroutine with the following three parameters:

```
(AMOUNT, lookup('rate', CODE1),
      lookup('rate', CODE2))
```

CODE1 and CODE2 are the temporary fields used to capture the from and to country codes respectively. If we pass the AMOUNT parameter by reference, it will contain the converted value when the subroutine has finished. This is how my subroutine will work here.

If we want the converted amount in a different field, then we can either supply that field as a *return field* or pass its name by reference and use the *Set return value* command for our conversion calculation. The AMOUNT would then be passed by value. We could also pass only AMOUNT and the two country codes, letting the subroutine's calculation perform the lookups.

The finished subroutine would look like this if we pass AMOUNT by reference:

```
Parameter AMT (Field name)
Parameter RATE1 (Number 4 dp)
Parameter RATE2 (Number 4 dp)
Calculate AMT as AMT/RATE1*RATE2
```

It would look like this if we pass AMOUNT by value and use a return field:

```
Parameter AMT (Number 2 dp)
Parameter RATE1 (Floating dp)
Parameter RATE2 (Floating dp)
Set return value {AMT/RATE1*RATE2}
```

If we pass only the country codes instead of the exchange rate lookup values, the parameter list for the subroutine call will be:

```
(AMOUNT, CODE1, CODE2)
```

The corresponding subroutine would look like this:

```
Parameter AMT (Field name)
Parameter FROM_CODE (Character)
Parameter TO_CODE (Character)
Calculate AMT as AMT/
      lookup(FROM_CODE)*lookup(TO_CODE)
```

Neither case is better or worse than the other — it's a matter of personal preference. The point here is that we can have direct access to more than one record from a File in the same expression using the *lookup()* function.

WIN Graphics Warning

Gavin Foster of Blyth UK faxed me a short time ago with a couple of suggestions he felt you should be aware of. The first involves Picture fields in the Windows product. When *Shared picture format* is active, black and white pictures pasted into picture fields go into “reverse video”. *This is not a bug.*

Shared pictures can be drawn in any two colors. After a great deal of experimentation, the programmers at Blyth decided to map white pixels to the text color and black pixels to the background color. This choice was based on their observation that most color pictures they tested look better this way — it just happens to reverse monochrome bitmaps.

The simple workaround is to reverse the text and background colors assigned to that Picture field on the window. A more creative workaround is to decide on some other pleasant and harmonious (or raucous and gaudy — it’s your choice) colors for this field. In any event, your choice of colors here doesn’t effect the stored picture — only the *display* of that picture. This is only a function of the drawing routine. It is also only a consideration for the Windows product.

AHR Labels

Gavin’s other suggestion deals with providing *No line if empty* fields in Ad Hoc label reports. As you may know, there is no such attribute available for Ad Hoc Report fields. Reports using the label template provided with OMNIS will show gaps when fields for Company or Second Address Line contain a blank value. This can be remedied by opening the original template Report Format and assigning *No line if empty* to both #1 and #2.

Page Numbering

It is sometimes desirable to reset page numbering after a subtotal break in a report. Consider a report that prints a long list of names and addresses broken down by country or by State where each subdivision begins a new page and may go on for many pages. It may be required to have page numbers that read like “Albania-1, Albania-2, Bosnia-1” and so on.

Although such a facility doesn’t directly exist in the current (1.03) version of OMNIS 7 — and probably shouldn’t — we can perform this operation fairly simply. All we need is a variable to hold the page number of the most recent subtotal break. This variable should be cleared just prior to printing the report so that its value is 0 for the first subtotal grouping.

In the Subtotal Section that corresponds to the Sort Field level that is triggering the break (most likely Sort Field number 1), we must place an invisible, calculated field for that variable that calculates its value to #P. This calculation should occur in a field at the very end of the subtotal section *after all other fields have been printed*. This is to avoid the possibility that subsequent line in that section might print at the top of the next page and throw our scheme off for the next subtotal grouping.

Let’s say that we decide to use #50 for this purpose. Instead of placing a field or square bracket expression to display the value of #P in the Heading Section, we will place a field or square bracket expression to display #P-#50. This could be concatenated with other text or field values, such as the country name in the example above, to enhance the page number display.

Undocumented Values

When Omnis 5 was in beta testing, I discovered some undocumented values for #ER that dealt with selecting items from menus. I demonstrated some of these at the Omnis 5 rollout meetings we gave across the United States during the summer of 1989, but never “officially” documented them because the people on the Blyth programming staff told me the values would probably change at some point.

One change was in selections made from user-defined menus. In Omnis 5, the menu number (in install order) times 100 plus the procedure number selected was added to a base number and assigned to #ER. Now, #ER is always given the value 14 without distinguishing the menu or procedure number.

Standard menu items each have a unique code with a value greater than or equal to 11000. You can view this code by selecting the *Call procedure* command and double-clicking on any of the standard menus in the displayed menu list.

There are two sets of numbers you can’t view there. The *Recent...* menu items have values of 16900 plus the line number selected. The *About...* item in the *Apple* menu has a value of 11010.

We can disallow menu selections during data entry using an application control procedure as follows:

```
If #ER=14|#ER>=11000&(INSERT|EDIT)
  OK message {Please finish entry}
  SNA remain on current field
End if
```

There are two exceptions. The basic *Edit* menu functions and selections from the *Apple* menu don’t get reported, so we can’t avoid them.

Tips and Techniques (cont.)

Percent Complete

For repetitive processes that require a significant amount of time to complete, it may be desirable to inform the operator what progress has been made toward completion. The typical way this is done in many programs is to display a “thermometer” that indicates the percentage of completion of the task as it is being performed. The current version of OMNIS 7 does not have such a feature, but we can still show the percentage of completion numerically in a working message.

First, we can only do this if we know beforehand how many iterations of our repetitive process constitutes completion. If there is no way of knowing how many iterations will be required, there is no way we can display what portion of them have been completed. For example, if we need to update a field in *each* record in a File, we can know the total number of records that must be updated by setting that File as the Main File and accessing *sys(82)*, the number of records in the Main File. On the other hand, if we need to update a *subset* of records from that File, we can only know how many records must be processed if we have loaded their images into a List. When that List is made the *current* List, #LN contains the number of records to be processed. In any event, we can establish a variable to store this number for reference.

We must then set up a variable to count the number of iterations already completed. It is usually most appropriate to use a numeric # variable or a local variable for this purpose. The variable should be cleared before beginning the re-

petitive process (a local variable would start out as cleared) and should be incremented at the end of each of our *Repeat* or *While* cycle.

We would display the progress made in a working message. The *Working message* command should be at the beginning of the loop. It will contain a square bracket expression that derives the percentage of completion based on the current values of our two variables. For good measure, we can use the *jst()* function to set the number of display decimal places and append a percent sign (%) to the figure.

Since the string supplied for a working message remains static unless it is explicitly redrawn, we must also include a *Redraw working message* command in the repetitive block *after the incrementation step for our counting variable*.

If we use #20 for the number of records in the Main File and #21 as the iteration counter, the finished procedure would take this form:

```
Set main file {Filename}
Calculate #20 as sys(82)
Calculate #21 as 0
Clear main file
Find on SEQNO
Repeat
  Working message {Now at
    [jst(#21/#20*100,'N0')%.}
  ;Process in question
  Calculate #21 as #21+1
  Redraw working message
Next
Until flag false
```

If, at some point in the future, OMNIS acquires a thermometer or some other type of dial indicator, this same process could be used to determine its display.

CrossRef Patch

Owners of our Omnis Cross Reference Tool will be pleased to know that a patch is available and we are able to ship new product again. Here is a little history.

The original program was written in BASIC by a friend of mine, John Crossley, who was then an employee at Apple. I convinced him to let me market the product for him. He moved on into management and didn't have time to maintain or upgrade the product, so I bought the code from him and made an agreement with another friend, Omnis developer Jim Treulich, to have him convert it to Pascal and enhance the product. I was to retain ownership, but we would split any profits from sales 50-50.

Jim did a great job, and quite a few people obtained the product — many claiming they couldn't live without it. Unfortunately, Jim died in the spring of 1991. His wife, Lydia, moved away and I haven't been able to successfully make contact with her. She has the source code (I hope!).

The bad news is that Jim left a kill date in the product from our beta testing phase. He simply set the date to some arbitrary date in the future, thinking that we would update the product well before that time. The date ended up being somewhere around the end of November 1991. Launching the program after that date caused it to destroy itself.

Fortunately, Kevin Doyle in Massachusetts had the presence of mind a while ago to contact Kelly Burgess in Montana to see if he could hack out a patch. Kelly was successful and contacted me so I could inform all of you and continue dis-

tributing the product. Here is what you must do to patch your copy.

Using a file editing program (FEdit, ResEdit, etc.), locate the following hexadecimal string:

```
A9F4 0CAE A56D 0BA1
```

This will be followed either by FFFF or FFFC. Directly after that there is a 6F. Change the 6F to 60. You should have no further problems.

The only problem is in using this product with OMNIS 7 procedure listings. OMNIS 7 didn't exist when the product was released, so it only works with Omnis 3 Plus and Omnis 5. The only problem is that the Cross Reference expects to see a procedure listing that begins with:

```
PROCEDURES FOR format name AS AT
time
```

OMNIS 7 begins its procedure listings with:

```
Procedures for format name as at
time AM/PM
```

To get the Cross Reference to work on OMNIS 7 procedure listings, simply use the text editing features of the Cross Reference (or any other text editor) and convert this string to upper case. If you also want the time to be read correctly by the Cross Reference Tool, convert it to a 24 hour format and remove the AM or PM. Yes, these changes would be simple enough for me to put into the product — if I had the source code to work with.

If you are a registered owner of the Cross Reference Tool for Omnis 3 and 5, but no longer have a usable original, I will replace it for \$5.00 (plus the necessary shipping surcharge for non-U.S. destinations). A patched version is now shipping.

Unique Index Check

On page 16-15 of the *Application Designers' Handbook*, it is mentioned that the *Unique index check* attribute of a window field is not adequate to ensure uniqueness in a multi-user system. The reason for this is that this attribute checks to see if that value exists *as the cursor leaves the field*, not when the record is being written to disk.

An example is given of a method to ensure uniqueness in a multi-user system. I give you an alternative method here that you may find to be more streamlined.

My assumption is that we want to check for uniqueness as the record is about to be written to disk, and that the user should be informed of a uniqueness failure and given a chance to enter a different value.

An appropriate place to do this is in a window control procedure. We can perform the uniqueness check after the operator has clicked the OK button and the procedure is about to write the new record. It is only a split second between the check and the actual disk write, so this should be pretty safe. It also doesn't keep other data entry people from doing the same operation as the manual example does.

Uniqueness checking on an edit is a different matter. The *Test for unique index value* command only checks to see if the value exists in the index. It doesn't check to see if any *other* record has this value, but if *any* record has it. An unmodified value that is still unique will return a *flag false* for this command. On edit, you should only perform the check if the index value has *changed*. To do this, calculate a temporary variable as the original value before data entry.

Local Variables

continued from page 11

Local Usage

As one of their many functions, reversible blocks can be used to set up "local usage" of otherwise more global variables. By calculating a new value for a global temporary or memory variable in a reversible block, its previous value is placed in storage for retrieval at the end of the procedure. This can be an alternative to creating a local variable in that procedure.

For example, I use this technique with my data entry state variables INSERT, EDIT and FIND. Since they are Boolean memory variables, they automatically revert to empty values after any procedure in which I set them to YES reversibly.

When to use what

I suppose that after all this explanation, you probably want to know why, where, and when we would want to use which type of variable.

I use global memory variables when I can identify a universal need for a specific variable. This is so I don't have to create a local variable every time I have this need. These variables also allow me to transport values to other Formats. I use them to name fields in stored Lists.

I use local variables when I need a temporary variable of a data type other than character or number, or when I want a name associated with a temporary. I wish I could define them in Report Formats.

I use format variables when I need to display a value on only one window. They make great radio button and check box fields for report selection windows.

Prepare for Update

continued from page 5

that the record in that File is locked by another workstation and this can lead to serious trouble (see the “Deadly Embrace” section below). The reason some programmers do this is to attempt to avoid locking certain records, like the Constants File record, until the last moment. We have already seen a safer method of doing this in this article.

But how can we make sure that we aren’t inadvertently locking unnecessary records? My recommendation is to set the default mode (the one at the File Format level) for *all non-Memory only* Files in an application to *read only*. At the beginning of a procedure in which such a File needs to be updated by an insert, edit or delete operation, that File should be set to *read/write* mode in a reversible block. No matter how the procedure is exited, the File will revert to its original *read only* state. In this way, only those records we need to have locked will be.

Update Without Cancel

The *Update files* command has an option *Do not cancel pfu*. This option allows the records to be written to disk, but retains *all* record locks for the current workstation. As more records from *read/write* Files are read into the CRB, they are locked as well.

Eventually, records not even in the CRB at this workstation will remain locked by it until an *Update files* without the option is issued. An example of this would be multiple inserts with the option turned on for each update.

There may be some cases where this is useful — and it *is* a good

option for the OMNIS language to have — but I must counter the example in the *Programmers’ Reference Manual* on page 4-414 with some real-world considerations.

This example inserts an Invoice Header record with multiple Line Items. The rationale for using the *Do not cancel pfu* option, beyond demonstrating it, is to ensure “that the complete transaction is locked until complete”. If this only uses the Invoice Header and Line Items Files, this may be worthwhile, although the chances of any other workstation attempting to change any aspect of a brand new transaction in the second or two it takes to write the line items to disk are pretty slim.

If an Inventory File is also involved, this is a very bad idea. It is likely that the Inventory records also need to be updated for recording cumulative sales information, the last sale date for that product, etc.

Allowing these records, which many other workstations in an active telemarketing environment also need to update, to remain locked for the entire transaction (instead of just updating and releasing them) can cause some serious delays system-wide.

For a company with only two or three users, the delay might not be noticeable. But for an enterprise with twenty or thirty data entry people actively adding invoices, it could be devastating!

One proper use for this feature is for performing multiple inserts based on calculated criteria. When we use the *Do not cancel pfu* option, it is not necessary to re-issue the *Prepare for insert* command for each new record. The following procedure illustrates this.

```
Begin reversible block
  Set main file {Customer file}
  Set read/write files
    {Customer file, System file}
  Single file find on SY_SEQNO
    (Exact match) {1}
  Calculate #1D0 as 1
End reversible block
Prepare for insert
Repeat
  Calculate CU_LAST_NAME as
    con('Number ', #1)
  Calculate SY_NUM_CLIENTS as
    SY_NUM_CLIENTS+1
  Calculate CU_ID_NUMBER as
    jst(SY_NUM_CLIENTS, '-7P0')
  Update files (Do not cancel pfu)
  Calculate #1 as #1+1
Until #1>10
Cancel prepare for update
```

Notice the use of the *Cancel prepare for update* command at the end of the procedure. If I had used an *Update files* command (without the option this time), I would have ended up with an extra record.

Deadly Embrace

A *deadly embrace* in database parlance is a situation where one workstation is waiting for access to a record that another workstation already has locked and that second workstation is waiting for one that the first workstation already has locked. They can both wait *forever* and not make any progress!

OMNIS is designed to avoid the possibility of a deadly embrace *if you use it in the simplest and most direct ways*. OMNIS 7 allows us the greatest number of ways so far to create a deadly embrace situation. Not only can we perform *Single file find*, *Load connected records* and *Set read/write files* commands while in a *Prepare for update* mode — both of which have deadly embrace potential — but we can now perform *Find*, *Next* and *Previous*

and *Set main file*. We can also perform *Update files* without releasing locked records, which has deadly embrace potential as well.

This is *not* a criticism of OMNIS 7, but a warning to be pristine in your programming methods. These features may have some potential benefits, but I strongly recommend that you avoid using any of the above-mentioned procedure commands while in a *Prepare for update* mode unless you absolutely *have to*. And be *very* careful how you use the *Do not cancel pfu* option of the *Update files* command.

Powerful and complex new features or options can sometimes be a Pandora's box instead of the panacea we would like them to be.

Planning for the Future

Occasionally I am told about upcoming features in OMNIS before many of you. Although I can't tell you details about these features, I have been asked by upper management at Blyth Software to mention items in this newsletter that may help you in planning for future versions if a specific technique would be wise to start using.

I have a suggestion along these lines. In the last newsletter, on page 4, I mentioned that radio buttons and check boxes only react to clicks — and only during *Enter data* — in the current (1.03) version of OMNIS 7. This would allow you to simplify field procedures for these field types. In the not-too-distant future, this will no longer be the case. These window objects will detect a much wider range of events than they do now. To prepare for future upgrades, make sure you trap for specific events in *all* field procedures.

lookup() Function

continued from page 15

We could perform the same operation this way.

```
Open lookup file {[sys(11)/File/2}
Calculate AVERAGE as
  (lookup('',value1,3)
   +lookup('',value2,3))/2
```

We didn't have to read two entire records into the CRB. We only had to retrieve two values. See the Currency Exchange example in the Tips and Techniques section of this issue (page 30) for a practical example of this technique.

Build list with lookup()

A student in one of my advanced classes recently asked if we could build a List of values from a "foreign" datafile using the *lookup()* function. In less than five minutes, we came up with this solution. See if you can follow along.

```
Begin reversible block
  Open lookup file
    {/State data/State file/}
End reversible block
If flag false
  OK message {open lookup failed}
  Quit procedure
End If
Format variable STATE_LIST (List)
Set current list STATE_LIST
Define list {#S1/20,#S2/30,#S3/2}
Calculate #1 as 1
Repeat
  Working message {Getting states}
  Add line to list
    {(lookup('',#1,3),lookup('',#1,4),
      lookup('',#1,2))}
  Calculate #1 as #1+1
Until not(len(lst(#1-1,#S1)))
Delete line in list {#LN}
```

This, of course, assumes that the Record Sequencing Numbers are continuous in the lookup File.

Storing Lists

continued from page 21

end of the range for a line, the sort should be in ascending order on that column. If the cutoff value is on the low end of its range, the sort should be in descending order.

In use, these lookup Lists would be retrieved from the Constants record and copied into a temporary List by calculated transfer. A line is selected for use by locating it with a *Search list (from start)* command using appropriate search criteria and the value of the required field is extracted using the *Load from list* command or the *lst()* function.

Sets of Records

It is possible to store archived selections of record images in a File. Perhaps a client would like to keep lists of the top ten products sold each week in a record along with other sales information.

This is of limited utility because we can't index a List field or its contents, so we could not easily locate a specific record based on the List contents alone.

Import and Export

There will be occasions when we will need to export or import records that contain List-type fields. On export, we can transfer the List contents to a long string, supplying field and record delimiters. (We can't use return characters for record delimiters here.) On import, we can have a procedure parse such a string and map the values to the proper fields. There are string length limitations to consider here.

I will deal with this issue more in the next issue when I discuss importing and exporting techniques.

OMNIS 7 Class Schedule

For the past seven years, my primary commitment to the OMNIS user community has been in providing comprehensive and affordable education experiences across the United States. The articles and books I have written on OMNIS products, as well as this newsletter series, have all sprung from the needs expressed by the students I have helped. This tradition of assisting OMNIS users through traveling seminars continues.

There are many subjects I could cover. In the past, I have offered OMNIS-based database design courses, reporting techniques, and so on. Since OMNIS 7 is still relatively new, I am focussing on more basic issues for now. Here are the courses I am currently offering.

Professional Developer Training

This three-day course is the prerequisite for all others I offer. It is a detailed look at the basic programming tools in OMNIS 7, including Connections, the CRB, Lists, Reversible Blocks, Event Management, and other issues. \$750.00 (auditors — \$250.00)

Advanced OMNIS 7

This two-day course delves deeper into the tools available in OMNIS 7. The Debugger, format and local variables, advanced List manipulation, and Ad Hoc reports are among these advanced topics. \$550.00 (auditors — \$200.00)

Streamlining for Performance

This one-day course deals with issues of speed and efficiency in using OMNIS 7. Examples are taken from real world applications and students may pose their own problems in the afternoon session. \$300.00 (auditors — \$100.00)

Logistics

In each city to which I travel for these classes, I give one class of each course. Professional Developer Training is given Monday - Wednesday, Advanced OMNIS 7 is given Thursday - Friday, and Streamlining for Performance is given on Saturday. Each day is a full 8:30 am to 5:00 pm day. The classes are generally given in a hotel near a major airport for the convenience of students who fly in for the training. Classes are on a "bring your own computer, buy your own lunch" basis to save you money. You must also supply your own copy of OMNIS 7.

I offer a discount to people who take multiple full-paid classes in the same week *and* who pay for their classes two or more weeks in advance. This discount is currently \$100.00 per *additional* class. If I cancel a class, for whatever reason, a full refund is given. Most often, this is due to insufficient prepaid registrations for that class. If *you*

cancel more than a week before the class, a full refund is given as well. Cancellations within a week of the class receive a 50% refund. No refunds are given if you simply don't attend without telling my office.

Auditing

Students who feel the need to refresh their memory *for a specific course they have already taken* may audit that course at a reduced rate. That rate is roughly one third of the normal class fee. I have found over the years that auditors often learn more the second time around because of the repetition.

Registration

You may register by telephone, mail, or fax. Just send me the information requested on the Fax Survey on page 39, indicating which classes you wish to attend in which city from the schedule on the following page and include payment information. I hold all payments until I am certain that the class will be held. I need a minimum of eight full-paid people per class. (Auditors count one third.)

1992 Class Schedule

For more information, call 510-522-6107.

	Prof Dev	Advanced	Streamlining
Oakland, CA	Jul 13-15	Jul 16-17	Jul 18
Detroit	Jul 20-22	Jul 23-24	Jul 25
Boston	Aug 10-12 (follows MacWorld Expo)	Aug 13-14	Aug 15
Seattle	Sep 14-16	Sep 17-18	Sep 19
Oakland, CA	Sep 21-23	Sep 24-25	Sep 26
Toronto, ON	Oct 5-7	Oct 8-9	Oct 10
Atlanta, GA	Oct 19-21	Oct 22-23	Oct 24
Los Angeles	Nov 9-11 (follows MCN Conference)	Nov 12-13	Nov 14
Oakland, CA	Dec 7-9	Dec 10-11	Dec 12

Example Disk Program

Even the greatest verbal explanation is worth a lot more if accompanied by working examples. It is my intention to make such examples available with each issue of Omni-Science. I recognize that not everyone will feel the need for these, so I have made them optional.

Each issue will contain at least one disk offer and often two or three. These will be at different levels of complexity or commercial value and will be priced accordingly.

For example, each issue will at least offer a disk containing the examples from the Tips and Techniques section and other articles of that issue, as well as variations and extensions of those examples. The records contained in the associated data file(s) may also hold useful information. This will be an inexpensive disk, usually in the \$20.00 to \$30.00 range.

If there are more complex principles in some article that require a more comprehensive application for demonstration, a disk with such an application will be offered in the \$50.00 to \$100.00 range.

Finally, I have been working for over four years on a project code named the "Database Construction Set" for which I have received many requests. The project just got too big and there seemed to be no good pricing and licensing scheme for it. I have decided to release it in a "serialized" form (as in "tune in next time for the next exciting..."). These "Solution Packs" (as they will be called) will contain code and documentation on topics that cover the basic needs of business transaction database applications. For example, the first will focus on

General Ledger accounting systems, the next on invoicing systems, and so on.

Programming modules that you can patch into your applications *with no additional royalties* are included in these Solution Packs as well as a manual that explains the principles behind the code. A licensing agreement will accompany each Solution Pack basically stating that you will retain my copyright in each format, include the phrase "portions copyright by David Swain" on your About window and in any manuals. It will also state that you are not to distribute the module as open code or share it among your friends. The Solution Pack modules will generally fall in the range of \$200.00 to \$500.00, depending on their complexity and commercial value.

I hope these disks will be of use to many of you and that the extra time I spend preparing them for you will be appreciated.

This issue's disk contains an application demonstrating:

Prepare for Update Techniques

Format and Local Variable Use

lookup() Function Examples

Stored List Values

Currency Conversion

Ad Hoc Label Report

Report with Reset Page Numbers

Unique Index Check Example

Percent Complete Working Message

and many other related techniques.

This disk is priced at \$20.00 — order yours soon!

Products

As time passes, I will have a number of example programs, programmer aids, and back issues to offer you. Descriptions and prices will be noted in this column. For indicated items, be sure to specify with your order whether you need Macintosh or Windows version of the software package.

There is an additional shipping surcharge for Canada and other non-US deliveries.

Example Disk Volume II Number 2 **\$20.00**

As outlined in the preceding column. Specify Mac or Windows version

OMNIS 7 Shell Application **\$150.00**

A great starting point for any application. It will save you hours on each new project! Open code. Specify Mac or Windows version

OMNIS Cross Reference Tool **\$100.00**

This is a very useful programming aid. It lets you see where you have been using and updating variables and how you have been structuring your subroutine calls. Macintosh only

Data Dictionary **\$20.00**

Parses the entire File Formats listing (dumped from Utilities) into an OMNIS database of File Formats, Fields, and Connections. You'll be amazed at its simplicity. Open code. Specify Mac or Windows version

Black Boxes **\$30.00**

Modular procedures for data conversion that you can paste into any application. Open code. Specify Mac or Windows version

Data Generator **\$50.00**

When you need massive numbers of records for testing your applications, this application can give them to you. It creates real-looking random client records. (Don't mail to them!) Specify Mac or Windows version

continued on page 38

Clarifications

continued from page 1

with their printer. These problems should not arise again, but if you notice problems with the printing of the newsletter (blank, missing, or overprinted pages, etc.), contact Blyth Software so they can be aware of the problem and deal appropriately with their printer.

The first issue of OmniScience Volume II happened to be shipped in the same envelope as a disk entitled "Tips & Techniques". This is in no way a product of mine or of my company. It has no relationship to OmniScience. The original "master" for that disk is produced by Ray Sauers, another OMNIS 7 developer, and copies are then produced by a service contracted by Blyth Software. Blyth distributed the disk and my newsletter in the same envelope to save on postage and the extra labor of producing two mailings. We received a number of calls at Polymath from people who had problems opening the files on that disk. They called my office

since there was no address or telephone number on the disk and they could only assume the disk was part of my newsletter. I have been assured that, in the future, the "Tips & Techniques" disks are to be labeled appropriately with Ray's address and telephone number so you can contact him directly with any feedback. The example disks I offer to accompany OmniScience are separate products that are produced and distributed by my company, Polymath Business Systems. They are individually tested for viability before being mailed to the people who purchase them.

We also receive many calls at Polymath reporting bugs (real or imagined) and problems with OMNIS 7. Although I am always interested in what you may have discovered, *I am not the proper channel for submitting bug reports.* I have no special "pull" with Blyth and a bug will not get any more immediate attention if I report it instead of you. I *can* help you with work-arounds if what you are reporting really *is* a bug. I can also help you

gain more clarity when your "bug" is only a misunderstanding of OMNIS 7. But these fall under the heading of telephone consultations for which I must charge you. You may want to try Blyth Software first with these problems since you have already paid for technical support from them (when you bought the product). Historically, in cases where that avenue hasn't proven satisfactory, I have been found to be a valuable and worthwhile alternative source of technical support.

We all play an important role in you use of OMNIS 7. Blyth produces a phenomenal product and provides technical support, Ray provides innovative ideas on using that product, and I supply in-depth explanations of product features. There will naturally be some differences in the information you receive from these different sources, but tap into all of them. As with examining a sculpture, you can learn more about the intricate details of OMNIS by seeing it from different points of view.

Products

continued from page 37

Example Disk Volume I Number 1 \$20.00

Demonstrates basic features like reversible blocks, optimized searches, list used as a sort buffer for reports, etc. Also random and prime number generation, parsing number into words, rounding techniques. Originally written in Omnis 5 and converted to OMNIS 7.
Specify Mac or Windows version

Example Disk Volume I Number 2 \$20.00

Demonstrates Many-to-Many relationships, Flat File Image, fixed length export report, text blocks in reports, List minima and maxima, streamlined algorithm techniques, etc. Originally written in Omnis 5 and converted to OMNIS 7.
Specify Mac or Windows version

Example Disk Volume I Number 3 \$30.00

More complex than most other Example Disks, hence the increase in price. Primarily demonstrates multiple Main File reporting. Also demonstrates data entry into Lists, age in years calculation, flat file image of complex structures, etc. Originally written in Omnis 5 and converted to OMNIS 7.
Specify Mac or Windows version

Example Disk Volume II Number 1 \$20.00

Demonstrates event management techniques, advanced List manipulation, uses of the debugger, eval() function, DMS to decimal conversion, etc.
Specify Mac or Windows version

OmniScience Volume I \$40.00 per issue / \$200.00 for 7 issue set

The original series is still in production in parallel with Volume II. The first four issues are currently available. Tables of contents available on request.

OmniScience Volume II \$20.00 per issue / \$75.00 for 4 issue set

The original series is still in production in parallel with Volume II. The first four issues are currently available. Tables of contents available on request.

Unlocking Omnis 3 Plus \$49.95 (plus shipping)

Many people are still actively using this version of OMNIS. This is the only viable reference available if you are. About 70% of this material is still applicable to OMNIS 7.

Telephone Consultation \$20.00 per 10 minute time interval

This is a little lower than my standard hourly rate, but I feel I can't be quite as effective flying blind over the phone. On the other hand, there have been very few problems I couldn't solve.

Suggestions for OMNIS support products you would like to see in the future are greatly appreciated.

2nd Quarter Fax Survey

I can give you better information if I receive some from you. This survey is intended to find out what kind of information OmniScience readers want as well as what kind of OMNIS support products and services you need. I would also like your feedback on the articles presented in this issue.

Please mail or fax your completed survey to:

OmniScience
 1418 Park Avenue
 Alameda, CA 94501
 Voice: 510-522-6107
 Fax: 510-522-6110

Please rate the major articles from this issue of OmniScience.

	Useful	Clear	Too simple	Too advanced
PfUpdate	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Local Vars	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
lookup()	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Storing List	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Sybase	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
SQL Report	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
T&T	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

OMNIS 7 Support Products Order Form

Example Disk Volume II Number 2	\$20.00	_____
OMNIS 7 Shell Application	\$150.00	_____
OMNIS Cross Reference Tool	\$100.00	_____
Data Dictionary	\$20.00	_____
Black Boxes	\$30.00	_____
Data Generator	\$50.00	_____
Example Disk Volume I Number 1	\$20.00	_____
Example Disk Volume I Number 2	\$20.00	_____
Example Disk Volume I Number 3	\$30.00	_____
Example Disk Volume II Number 1	\$20.00	_____
OmniScience Volume I (7 issue set)	\$200.00	_____
OmniScience Vol. I No. 1	\$40.00	_____
OmniScience Vol. I No. 2	\$40.00	_____
OmniScience Vol. I No. 3	\$40.00	_____
OmniScience Vol. I No. 4 (available July 92)	\$40.00	_____
OmniScience Volume II (4 issue set)	\$75.00	_____
OmniScience Vol. II No. 1	\$20.00	_____
Unlocking Omnis 3 Plus	\$49.95	_____

Please include credit card number for billing. We will contact you with total amount, including shipping charges (and CA sales tax, if required) before billing or shipping product.

MCard Visa Am Ex

Expires _____

Name _____

Company _____

Address _____

City _____ State __ Zip _____ Country _____

Voice phone _____ Fax _____

Comments _____

List your 3 favorite O7 features

List the 3 you least understand

What problems are you having?

Please (circle one) send / fax me more information about

Classes available

OMNIS support products

The disk type I require is

Mac PC 3 1/2 PC 5 1/4

Right Way

continued from page 1

In my training and consulting travels, I often see examples of things being done in OMNIS just because they *can* be done in OMNIS, rather than using some other, and more appropriate, tool. Many times it seems that the issue of whether that process *should* be done in OMNIS, or done *that way* in OMNIS, or even done *at all*, was never addressed. An “OMNIS-macho” attitude (“OMNIS can do *anything*”) does not always lead to a successful OMNIS application.

On the other hand, I also often see people trying to create extensions to OMNIS to perform operations that OMNIS can do as well as, if not better than, the extension. The attitude that “OMNIS can’t do anything well” is also a problem that can lead to chaos. I have even seen a development team at one company using OMNIS as little more than a shell for calling extensions. Needless to say, these people had to deal with a *lot* of frustration in doing even some of the simplest data management tasks (inserts and edits) because they insisted in

“doing it themselves” rather than using OMNIS to do what it was *designed* to do.

This is not to say that OMNIS 7 is the ultimate programming language or that it currently provides all the tools we could want for any task we could ever imagine. Far from it! But OMNIS is constantly evolving — and the people at Blyth Software who create this product for us are constantly exploring potential new features. Their stated intention is to come out with one major upgrade and two interim enhancements *each year!* Version 1.1 of OMNIS 7, which will be released soon after you receive this issue of OmniScience, is the first of this years enhancements.

As part of my charter in serving the educational needs of the OMNIS-using community, I try my best to examine all sides of an issue and all means toward an end before making suggestions as to how processes should be performed. Often there is more than one viable alternative — or the proper choice of method depends on other situational requirements. I attempt to outline these as well.

In the Next Issue

There are many methods available for sharing information from an OMNIS 7 application with documents from other programs. Volume II Number 3 will detail many of these that have not already been covered in issues 1 and 2.

Import and Export

This traditional method of data transfer has been with us through all generations of the OMNIS product family. It is still a very practical means of sharing data with many programs.

Publish and Subscribe

For the Macintosh side of the aisle, this facility of System 7 is a very powerful tool for linking an OMNIS 7 application with a variety of analytical and data manipulating programs.

Dynamic Data Exchange

The Windows environment first gave us the ability to dynamically share data between two documents. The product has matured a great deal and presents us with a wealth of possibilities.

Apple Events

Exchanging data isn't the only link we can provide between programs in System 7 on the Macintosh. We can actually control other programs from our OMNIS 7 applications — and control our OMNIS 7 applications from other programs!

Connection Structures

There is still a lot of mythology and superstition guiding many developers' use (or non-use) of Connections. Here are some basic concepts to consider.

New Features in 1.1

Version 1.1 of OMNIS 7 will have shipped by the time issue number 3 is ready and there are some useful new features that will need explaining.

Tips and Techniques

More quick tidbits to make your programming life a little easier — or, at least, more interesting!



OmniScience is published quarterly by Polymath Business Systems, 1418 Park Avenue, Alameda, CA 94501. Address all editorial correspondence, requests for special permission, subscriptions, or requests for bulk orders to The Editor, *OmniScience*, 1418 Park Avenue, Alameda, CA 94501.

Domestic subscriptions (in US funds): 4 issues, \$75.00. Canadian subscriptions, \$80.00. Outside the USA and Canada, \$90.00. Single issues are available for \$20.00 domestic, \$22.00 Canada, and \$25.00 elsewhere. Send subscriptions, changes of address, and fulfillment questions to *OmniScience*, 1418 Park Avenue, Alameda, CA 94501.

Copyright © 1992, David Swain. All rights reserved. No part of this journal may be used or reproduced in any fashion (except in brief quotations used in critical articles and reviews) without the prior written consent of David Swain.

OmniScience and Polymath are trademarks of David Swain. OMNIS and OMNIS 7 are trademarks of Blyth Software, Inc. Microsoft and Windows are registered trademarks of Microsoft Corporation. Other software and hardware mentioned are trademarked by their respective manufacturers or distributors.