CLARION 5

Learning Clarion

TopSpeed Corporation
150 East Sample Road
Pompano Beach, Florida 33064
(954) 785-4555

Printed in the United States of America (0998)

TABLE OF CONTENTS

INTRODUCTION

Welcome to Learning Clarion!

The *Getting Started* book briefly introduced you to Clarion programming at its highest level. *Learning Clarion* now teaches you how you can use all the rest of the tools Clarion provides to create real-world applications. It contains *two* tutorials, on two very different levels:

- An **Application Generator** tutorial which familiarizes you with all the tools in the Clarion development environment.
- A **Clarion Language** tutorial which introduces the Clarion programming language and familiarizes you with the type of code generated for you by the development environment.

What You'll Find in this Book

Application Generator Tutorial

Chapters One through Twelve: introduce all the main Clarion development environment tools. It starts at the application planning stage, walks you through creating the data dictionary with the Dictionary Editor, then walks you through creating a complete application with the Application Generator. By the end of this tutorial you'll have created a complete order-entry application.

You'll use the Application Generator and work with Procedure, Control, and Code templates to produce an Order Entry application. You'll work with the Window Formatter to design windows. You'll work with the Report Formatter to design reports. You'll use the Text Editor to embed Clarion language source code into the code generated by the templates.

Clarion Language Tutorial

Chapter Thirteen: introduces the Clarion programming language at the hand-coding level. It starts at *Hello World*, then walks you through creating simple forms of the most typical types of procedures used in Clarion applications, all while explaining the functionality of the hand-code you're writing and relating it to the type of code you'll see generated for you by the Application Generator.

Documentation Conventions

Typeface Conventions:

Italics	Indicates what to type at the keyboard, such as *Enter This*.
SMALL CAPS	Indicates keystrokes to enter at the keyboard, such as ENTER or ESCAPE, or to CLICK the mouse.
Boldface	Indicates commands or options from a menu, or text in a dialog window. Note: this style can also utilize a different typeface to match the helvetica bold face which Windows uses as the system font.
`LETTER GOTHIC`	Used for diagrams, source code listings, to annotate examples, and for examples of the usage of source statements.

Keyboard Conventions:

F1	Indicates a single keystroke. In this case, press and release the F1 key.
ALT+X	Indicates a combination of keystrokes. In this case, hold down the ALT key and press the X key, then release both keys.

Anatomy of a Database

This section briefly describes the fundamentals of database design. It is meant only to provide an overview of the subject for those who are not already thoroughly familiar with standard database design concepts and issues. Experienced developers may want to move right on to the next chapter and skip this section.

Definitions

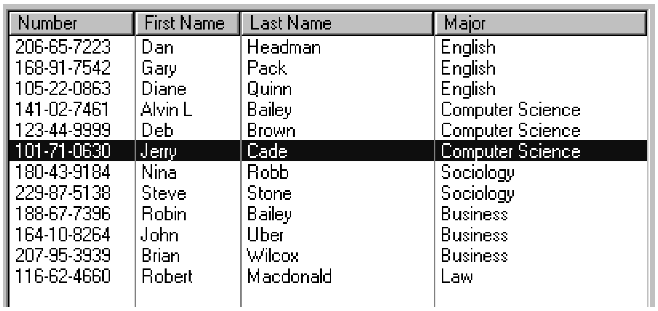
A **database** is a collection of information (data) in a system of files, records, and fields. The database is maintained by one or more computer programs or applications.

The basic unit of data storage is a **field**. A field is a storage place for information of a similar type. For example, one field might store a name and another field might hold a telephone number.

A group of different fields that are logically related make up a **record**. A record contains all the information related to one subject. For example, all the fields containing information concerning one student (name, address, telephone number, student number, etc.) makes up one student's record. This would be similar to a file folder a school might keep for each student.

A collection of logically related records make up a **file**. Using the same example, a collection of all students' records makes up the student body file. This would be similar to the file cabinets where students' folders are kept.

Another way of looking at this is as a table or spreadsheet:



Number	First Name	Last Name	Major
206-65-7223	Dan	Headman	English
168-91-7542	Gary	Pack	English
105-22-0863	Diane	Quinn	English
141-02-7461	Alvin L	Bailey	Computer Science
123-44-9999	Deb	Brown	Computer Science
101-71-0630	Jerry	Cade	Computer Science
180-43-9184	Nina	Robb	Sociology
229-87-5138	Steve	Stone	Sociology
188-67-7396	Robin	Bailey	Business
164-10-8264	John	Uber	Business
207-95-3939	Brian	Wilcox	Business
116-62-4660	Robert	Macdonald	Law

In this format, the entire table is a file, each row is a record, and columns represent fields.

A **database** is a collection of related files (tables). This is similar to a bank of file cabinets where the entire school records are kept. One file cabinet might hold the files with students' personal data, another with class enrollment information, and another with faculty information.

A database is a collection of tables with defined relationships between them. Effective database design breaks the data into related files that are joined together through linking fields. This will be covered in detail later in this section.

Summary:

One or more fields combine to form a record.

One or more records combine to form a file.

A collection of related files is a database.

File Systems and File Drivers

There are several data file formats used on PCs. These are the actual physical storage formats written to disk by programs that maintain the data files. Using TopSpeed's file driver technology, Clarion supports many of them. File drivers enable Clarion programs to read these different file formats.

The TopSpeed, Clarion, ASCII, BASIC, and DOS file drivers come with Clarion' Standard Edition. The Professional and Enterprise Editions also include Btrieve, ODBC, Clipper, dBase III & IV, and FoxPro file drivers. When you need to read data from another file system, you can add new file drivers to the Professional and Enterprise Editions. Call TopSpeed's Sales department at (800) 354-5444 or (954) 785-4555 to inquire about the availability of any specific file driver you need.

Each file system has its own idiosyncrasies and limitations. See the *Database Drivers* Appendix in the *User's Guide* for more information.

Data Types

Fields can store many different types of data, but each individual field may hold only one type. When a field is defined, its data type is specified. This enables it to efficiently store that type of data. For example, to store a number from 0 to 100, using a field defined as a single BYTE takes less space than one defined as a decimal number field (a byte can hold an unsigned whole number between 0 and 255).

Clarion supports the following data types (all fully documented in Chapter 3 of the *Language Reference*):

Alphanumeric

STRING	A field that holds a specific number of alphanumeric and other ASCII characters.

PSTRING	A field that holds a character string with a leading length byte containing the number of bytes in the string. This is the data type used by the Pascal language and the “LSTRING” data type of the Btrieve Record Manager.
CSTRING	A field that holds a character string terminated by a null character—ASCII zero (0). This is the data type used in the “C” language and the “ZSTRING” data type of the Btrieve Record Manager.

Integers (whole numbers)

BYTE	A one-byte field that holds an unsigned (positive) integer from 0 to 255.
SHORT	A two-byte field that holds a signed integer from -32,768 to 32,767.
USHORT	A two-byte field that holds unsigned (positive) integer from 0 to 65,535.
LONG	A four-byte field that holds a signed integer from -2,147,483,648 to 2,147,483,647.
ULONG	A four-byte field that holds an unsigned (positive) integer from 0 to 4,294,967,295.

Floating Point (Real) Numbers (numbers with fractional portion)

REAL	A field that holds an eight-byte signed floating point number with 15 significant digits.
SREAL	A field that holds a four-byte signed floating point number with 6 significant digits. An SREAL field uses the Intel 8087 short real (single precision) format.
BFLOAT8	A variation on the REAL data type, a BFLOAT8 field holds an eight-byte signed floating point number using the Microsoft BASIC (double precision) format.
BFLOAT4	A variation on the SREAL data type, a BFLOAT4 field holds a four-byte signed floating point number using the Microsoft BASIC (single precision) format.

Packed Decimal Numbers

DECIMAL	A field that holds a signed packed decimal value from -9,999,999,999,999,999,999,999,999,999,999 to 9,999,999,999,999,999,999,999,999,999,999.
PDECIMAL	A field that holds a signed decimal number in the Btrieve and IBM/EBCDIC packed decimal type of format. The only difference between a DECIMAL and a PDECIMAL is the place it stores the sign.

Other Data Types

DATE	A four-byte field that holds a date variable in Btrieve Record Manager format.
TIME	A four-byte field that holds a time variable in Btrieve Record Manager format.
GROUP	A single logical field containing multiple component actual data fields. For example, a GROUP field called PhoneNumber could contain two fields: AreaCode and Phone. A Group Field gives you the option of using the entire group or any of its components.
PICTURE	Although PICTURE is not a data type, when selected, PICTURE declares a STRING data type with a picture token (such as @P###P) defining the number of characters in the string. The picture token used for the field declaration is entered in the Record Picture field. This is useful for data fields whose storage picture and display picture must be different.

Sorting Data: Keys and Indexes

One of the most powerful aspects of a computerized database is the ability to sort data in many different ways. To do this manually requires multiple copies of record forms, many file folders, and many file cabinets. It would also require a lot of time spent filing each copy in different places for each sort order.

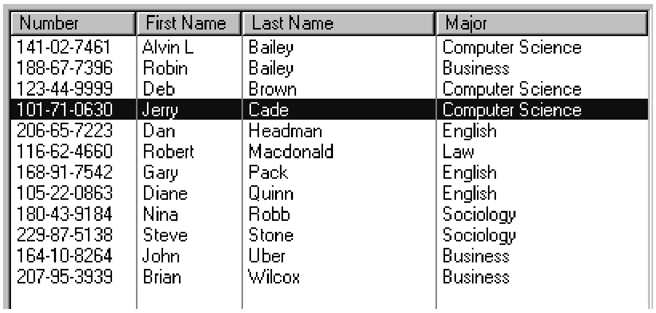
Sorting computer records in a database merely requires the definition of keys or indexes. Keys and indexes declare sort orders other than the physical order of the records within the data file. In some file systems the keys are kept in separate disk files, in others they are contained within the same file as the data. TopSpeed's file driver technology handles these differences transparently.

Keys and indexes are functionally equivalent. The only difference is the way they are maintained by an application.

- A **key** is dynamically maintained by the application. Every time a record is added, modified, or deleted, the sort sequence is updated, if necessary. Keys are useful for frequently used sort-orders.
- An **index** is not dynamically maintained, it is only built when needed. Indexes are useful to create sort sequences that are infrequently used, such as month-end or year-end processes.

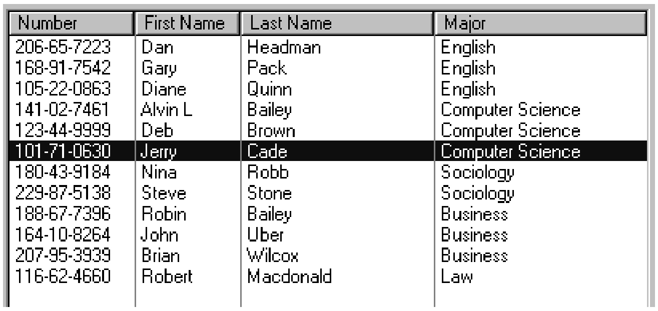
Using the student file example discussed earlier, suppose you wanted to sort the students' records two ways: by name alphabetically, and by name within each major. This produces two alternate sorts.

This example uses a one-component key on the student's name.



Number	First Name	Last Name	Major
141-02-7461	Alvin L	Bailey	Computer Science
188-67-7396	Robin	Bailey	Business
123-44-9999	Deb	Brown	Computer Science
101-71-0630	Jerry	Cade	Computer Science
206-65-7223	Dan	Headman	English
116-62-4660	Robert	Macdonald	Law
168-91-7542	Gary	Pack	English
105-22-0863	Diane	Quinn	English
180-43-9184	Nina	Robb	Sociology
229-87-5138	Steve	Stone	Sociology
164-10-8264	John	Uber	Business
207-95-3939	Brian	Wilcox	Business

The next example has two components in the key: major and student name. **A key can contain one or more fields, allowing sorts within sorts.**

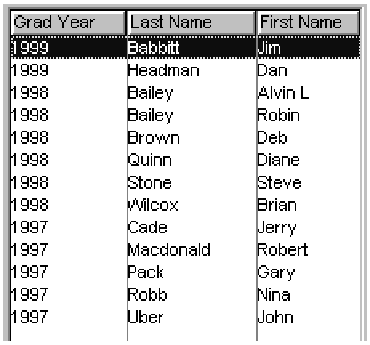


Number	First Name	Last Name	Major
206-65-7223	Dan	Headman	English
168-91-7542	Gary	Pack	English
105-22-0863	Diane	Quinn	English
141-02-7461	Alvin L	Bailey	Computer Science
123-44-9999	Deb	Brown	Computer Science
101-71-0630	Jerry	Cade	Computer Science
180-43-9184	Nina	Robb	Sociology
229-87-5138	Steve	Stone	Sociology
188-67-7396	Robin	Bailey	Business
164-10-8264	John	Uber	Business
207-95-3939	Brian	Wilcox	Business
116-62-4660	Robert	Macdonald	Law

Ascending and Descending Sort Orders

Some file systems support both ascending and descending order for keys or components of keys (for example, the TopSpeed file system). Other file systems only support ascending order, which means the data can only be sorted from lowest to highest (for example, the Clarion file system).

The next example has two components in the key: Graduation Year (descending), and student name (ascending).



Grad Year	Last Name	First Name
1999	Babbitt	Jim
1999	Headman	Dan
1998	Bailey	Alvin L
1998	Bailey	Robin
1998	Brown	Deb
1998	Quinn	Diane
1998	Stone	Steve
1998	Wilcox	Brian
1997	Cade	Jerry
1997	Macdonald	Robert
1997	Pack	Gary
1997	Robb	Nina
1997	Uber	John

Using Keys as Range Limits

Suppose you want to create a class enrollment report from a database containing records for the past fifteen years. All you are interested in (for purposes of this report) is the last three years. You can dramatically reduce processing time if you use a subset of the data file that only contains the records from the last three years. It takes two steps to accomplish this:

- First, define a key to sort the data by the date of each course.
- Next, define the range limits you are interested in. A range limit specifies a subset of the entire file to process. Only those records that fall within the range limits are considered.

In this example, only one-fifth of the records are processed (assuming that each year's course offerings are the same). Reducing the number of records to consider by 80% reduces processing time by the same amount.

Relationships Between Files

One goal of relational database design is reducing data redundancy. The basic rule is that data should be located in only one place. This is beneficial in two ways. First, it reduces storage space requirements. Second, it makes the database easier to maintain. To reach this goal, data files are broken up into separate, related files through a process called data normalization.

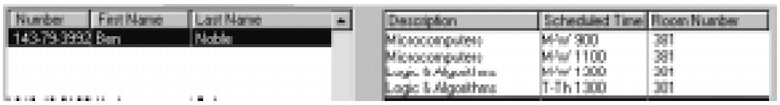
The first step is to move any repeating groups into separate files. For example, if a student could take a maximum of six classes, you could design the student file to contain six class fields (class1, class2, class3, etc.). But not all of these fields would be used in each record. If one student is taking six courses, all would be used, but if another student only took one class, there would be five empty fields in his record. For that reason, the class fields are moved into a separate file, eliminating the need to reserve space for empty fields. This creates a *One to Many* relationship (*One* student takes *Many* classes) between the student file and the classes file.

The next step is to move redundant data into separate files. Every field in the student file must be dependant on the primary key (Student Number). The student's name, address, and phone number remain in the student file. But the student's major description could be moved to a separate file. This eliminates the need to repeat the Major's Description for each student with that Major. To do this, add a Major number field to the student file and the Majors file. This creates a *Many to One* relationship (*Many* students have *One* major) between the student file and the majors file.

Once data storage is "normalized," related information is linked by ensuring a field in one file is identical to a field in the other. These common fields create the links between related files. A linking field could be a student number, a course code, or a classroom number. Any field (or group of fields) that uniquely identifies a record in the primary file can be used as a link.

Examples of these relations can be found in a school database:

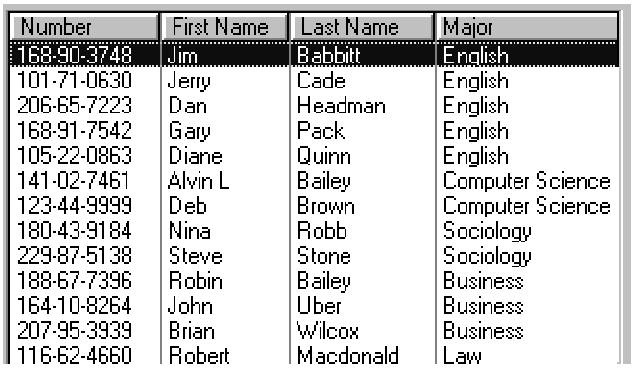
- **One** teacher teaches **many** classes (One-to-Many).



Number	First Name	Last Name
143-79-3992	Ben	Noble

Description	Scheduled Time	Room Number
Microcomputers	M/W 900	381
Microcomputers	M/W 1100	381
Logic & Algorithms	M/W 1300	301
Logic & Algorithms	T-Th 1300	301

- **Many** students would have **one** major (Many-to-One).



Number	First Name	Last Name	Major
168-90-3748	Jim	Babbitt	English
101-71-0630	Jerry	Cade	English
206-65-7223	Dan	Headman	English
168-91-7542	Gary	Pack	English
105-22-0863	Diane	Quinn	English
141-02-7461	Alvin L	Bailey	Computer Science
123-44-9999	Deb	Brown	Computer Science
180-43-9184	Nina	Robb	Sociology
229-87-5138	Steve	Stone	Sociology
188-67-7396	Robin	Bailey	Business
164-10-8264	John	Uber	Business
207-95-3939	Brian	Wilcox	Business
116-62-4660	Robert	Macdonald	Law

Database Summary

- A database is a collection of information (data) in a system of fields, records, and files.
- Fields can store many different types of data, but each individual field is specified to hold only one type.
- Each data item should be stored in only one place.
- One or more fields makes up a record. One or more records make up a file. A collection of related files make up a database.
- Clarion programs can access many different file systems through the use of file drivers.
- Keys and indexes declare sort orders other than the physical order of the records within the data file, and can contain more than one field, allowing sorts within sorts.
- Range Limits enable you to process a subset of records, which reduces processing time.
- Files are related through common fields containing identical data, which allows you to eliminate redundant data.

1 - PLANNING THE APPLICATION

As a general rule, every minute you spend planning your application beforehand saves you ten later. This chapter informally describes the planning process for the application you'll create in the subsequent chapters. In the real world, you'll probably create a bona fide functional specification for your important applications. This informal description defines:

- The tasks the application performs.
- The data the application maintains, and how it stores it.

As a starting point, this **Application Generator** tutorial application uses the data dictionary from the applications you created in the *Getting Started* manual. It extends the concept to a simple Order/Entry system, using the data dictionary for keeping track of customers.

Defining Application Tasks

This application will maintain the customer and billing files for a manufacturing company. The first task in planning just what the application will do is to assess what the company expects it to do.

For the purposes of this tutorial, the application we'll create is a simple order entry system. Customers typically phone in orders for one or more products at a time. A salesperson takes the order. The billing department prints out an invoice that night.

The application therefore must provide:

- Entry dialogs for taking the order, or modifying the data in it later.
- Access to the customer list from within the order entry dialogs. The customer list is the one you created with the Quick Start Wizard, stored in the Customer file.
- Access to the list of part numbers (items) that the company manufactures, from the order entry dialogs.
- Browse windows for listing sales transactions.
- Procedures that will maintain the Products list and Customer information.
- Printed reports.

Designing the Database

The first task in planning the file structure is to assess what data the application needs, and how to store it with the minimum amount of duplication.

Good database management maintains separate data files (also called “tables”) for each “entity” or group of discrete data elements. The data “entities” this application maintains are:

Customer	Customer name and address data that changes only when a customer moves. Created in the Quick Start tutorial, along with its related Phones file.
Orders	Basic information needed for assembling the data needed to print an invoice. It “looks up” information from the other files, such as the customer name and address. When a sales person takes a new order, they add a record to this file.
Detail	Product, price, and quantity ordered for an item on a given invoice: the variable information for each order. Though this duplicates price information in the Products file, you must maintain the price at the time of the sale here. Otherwise, when you increase the price in the Products file, it would cause the balance in the Detail file to change.
Products	Information on the products sold by the company, including product number, description and price. This data changes only when a price changes or a new product is added.

The Customer File

The Customer file stores “constant” data such as customer names and addresses. It’s most efficient to store this data in one place, allowing for a single update when the information changes. This also saves space by eliminating redundant customer information in the Orders file; otherwise, if there were 1000 orders for company XYZ, the address information would be repeated 1000 times. Reducing storage requirements by storing the data only once is called *normalization*.

The customer data requires a field to uniquely identify the customer. The company name is unsuitable because there could be duplicates. There may be, for example, multiple records for a customer called “Widget Depot,” if it has multiple locations. The Quick Start application *already* specified that the

CustNumber field is an auto-number key which automatically creates and stores unique customer numbers.

The CustNumber field serves as the *primary key* for the data file. Any other data files which are related to the Customer file must declare the CustNumber field as a *foreign key*. A *primary* key is a field, or combination of fields, that uniquely identifies each record in a data file. A *foreign* key is a field, or combination of fields, in one file whose value *must* match a *primary* key's value in another related file.

Because there may be many orders for each customer number, the relationship between the Customer file and the Orders file will be a *one to many* (1:Many) relationship. We say the customer data file is the *parent* file, and the Orders data file is the *child* file.

The Phones File

The Phones file stores telephone numbers—each customer could have several. Each record includes a CustNumber field to relate back to the Customer file.

The Phones file also includes a text field in which we can indicate whether the phone number is an office, fax, mobile or home number. Using the data dictionary, we'll specify that the control for entering data for this field should be a drop-down list with the choices already loaded.

The Orders File

The Orders data file gathers information for each sales transaction from all the other data files (such as the customer data). Because much of the basic data in this file prints in the "header" area of the invoice, this is sometimes called the Order Header.

Every sales transaction requires one record in the Orders file. The record relates to the customer information by referencing the unique customer number. Because some order records may reference one product, and others may reference ten, you'll create a separate Detail file which relates back to a unique order number. This creates a one to many relationship, with the Orders file as parent and Detail as child. The actual products ordered are identified by their product codes, in the Detail file.

The Orders record thus holds a customer number to relate back to the customer data (the foreign key), and a unique order number to relate to the Detail. You'll create a multi-component primary key on the two fields, so that you can easily create a browse sorted by customer and invoice number.

The Detail File

The Detail file stores the products ordered by their product codes (a foreign key into the Product file), their individual prices, the quantity of each, and the tax rate. An additional field holds an invoice number, which relates back to the Orders file in a *many to one* relationship.

The Detail file duplicates the price information with the fields in the product file; this is because prices may change. It's important to store the price field within the detail file record because if the price increases in six months, today's paid in full invoice would reflect a balance due.

The Product File

The Product file stores unique product numbers, descriptions, and prices. When the sales person looks up a product by name, the application inserts the product number into the Detail record. The product code is the primary key—no two items can have the same code, and every product must have a code. An additional field contains the tax rate for the product.

Referential Integrity

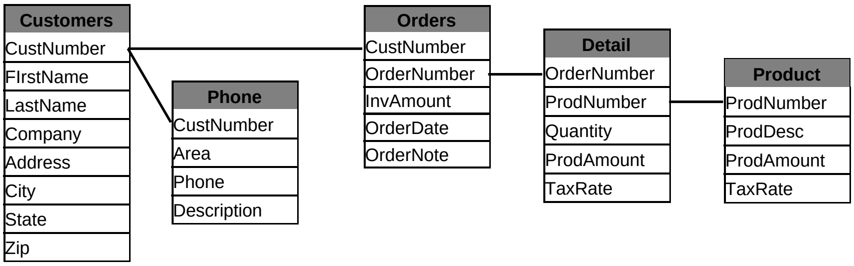
Referential Integrity refers to the process of checking all updates to the key field in a given file, to ensure that the validity of parent-child relationships is correctly maintained. It also refers to ensuring that all child file records always have associated parent records so that there are no "orphan" records in the database.

Because the data for a given transaction resides across several files, this application must enforce referential integrity. This is critical, yet many database application development tools require you to hand code procedures to enforce this. The Application Generator's templates implement this automatically in your generated source code when you select a few options in the Data Dictionary.

It is essential that the application not allow an update to a record that leaves a blank or duplicate value in a primary key field. For example, we need to restrict the ability of the end user to update a record in a way that could cause a duplicate Customer number. If two different companies shared a duplicate Customer number, you could send a bill to the wrong company.

The Complete Database Schematic

The schematic below provides an overview of the entire database. If you look at it from the point of view of the sales agent taking a phone order, the Orders file records who's ordering, the Detail stores what they're ordering, and the Customer and Product files store constant information about the customers and products.



The item code looks up the description and price. The customer code looks up the customer's name and address. Other data, such as the transaction date, fill in automatically (by looking up the system date, for example).

Finally, the tutorial will create a brand new data dictionary, and you will copy and paste the files that Quick Start defined for you into the new dictionary.

As for the actual application you create, because the tutorial is a teaching tool more concerned with showing what Clarion can do for you, it won't create a full-scale order entry system. However, you will find that some parts of the application will be very "showy," so that you can quickly learn how to do equivalent procedures in your applications.

Application Interface

The next major task before coding is to plan the user interface. For a business application like this, it's crucial that a salesperson quickly locate the data they need, so that they can record the sale and move on to the next phone call. Therefore, the application should put all the important data "up front" by default, and no entry or maintenance dialog should be more than a button or menu command away.

Additionally, the company uses many other Windows applications; therefore, it's especially important that the application have a standard windows "look and feel." End users learn a familiar interface more quickly.

To implement the tasks the application must execute in a consistent manner with our guidelines, we can plan for the items listed below. Though the following is no substitute for a real program spec, it should suit us for Tutorial purposes.

- Because the application will handle the maintenance for the customer, item, and billings files on different forms, the Multiple Document Interface (MDI) is necessary.
- The application should have a toolbar with buttons to load forms and browse windows, and to control the browsing behavior.
- To maintain a consistent “look and feel,” the main menu choices will be File, Edit, View, Window, and Help. The File menu accesses the printing and exit procedures. The Toolbar buttons call the form dialogs for editing a current record (if highlighted in a browse) or adding/deleting records, and for browsing through the files. The View menu calls the procedures for browsing data files. Window and Help perform standard actions.
- When adding new orders, the sales people should be able to pick customers and products from scrolling lists. Pertinent data in the order dialog—addresses, item descriptions and prices—should automatically “fill in” as appropriate.

OK, What Did I Just Do?

Here’s a quick recap of what you just accomplished:

- You defined the tasks the Application must accomplish.
- You designed the database that will allow the application to accomplish those tasks.
- You specified the user interface the application will use.

Now that the application description is fairly complete, we’re ready to begin work. The first step is to create the data dictionary.

2 - CREATING A DATA DICTIONARY

This chapter teaches you how to:

- Create a new data dictionary.
- Copy and customize file definitions from the Quick Start data dictionary to the new one.
- Relate the files and specify Referential Integrity constraints.
- Pre-format window controls for the fields.

This tutorial assumes that you have completed the *Quick Start Tutorial* in the *Getting Started* manual.

Tutorial Files

We recommend that you complete the entire tutorial, especially if the Clarion development environment is brand new to you (and even if you've used previous releases of Clarion). As you've already seen from the Quick Start tutorial in *Getting Started*, Clarion's template-driven Application Generator approach to programming is very different from other development environments for 3GL and 4GL languages. If you'll thoroughly immerse yourself in this tutorial, you will get the most out of your new tool.

The completed tutorial files reside in the \CLARION5\EXAMPLES\TUTOR directory. We provide these so you can see the end result of the tutorial and compare your application. There should only be cosmetic diferences.

If you're already an experienced Clarion programmer, you may want to just examine the completed tutorial files rather than step through the tutorial itself. However, we *do* recommend that you at least read through this tutorial, since there are methods of working in the development environment that the tutorial demonstrates that may not be obvious just by "plunging in" and working with the tools—this tutorial "shows off" some features of the development environment that may not be readily obvious at first glance.

NOTE: Wherever there are multiple ways to accomplish a single task, this tutorial will expose you to several of them to demonstrate the flexibility of the Clarion development environment.

Creating the Dictionary

Whenever you create a new application, you first define the data dictionary (.DCT file). From the data dictionary, the Application Generator obtains all its information about the data files your application uses, their relationships to one another, plus additional information such as predefined formatting for controls.

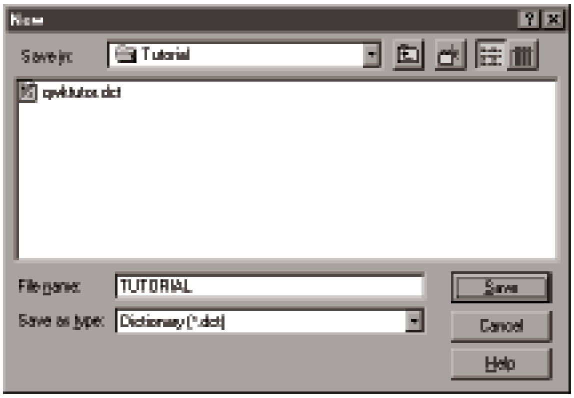
As a general rule, the more forethought and work you put into designing your data dictionary, the more Clarion's template-driven Application Generator can do for you. And, **the more work the Application Generator does *for* you, the less you have to do yourself!**

When you ran the Quick Start Wizard, though you didn't actually use the Dictionary Editor, you just defined the data file. This chapter introduces the Dictionary Editor.

Starting Point:
You should have the Clarion development environment open and the Pick dialog closed.

Fill in the name of the new data dictionary

1. **Choose** **File ➤ New** from the menu.
2. CLICK on the **Dictionary** selection.


Save in: Tutorial
qwiktutor.dct
File name: TUTORIAL
Save as type: Dictionary (*.dct)

The **New** dialog appears.

Name the new dictionary file

1. **Select** the */CLARION5/TUTORIAL* directory.
2. **Type** *TUTORIAL* in the **File Name** field.

Clarion appends the file extension; TUTORIAL.DCT is the full name for the dictionary file. You can use long filenames if you're using a 32-bit operating system (Windows 95 or NT). This tutorial does not use long filenames so it is valid for all users, including those using Windows 3.x.

3. **Press** the **Save** button to create the file.

   This creates an empty dictionary file. The caption bar shows the file name.

Specify a description for the dictionary

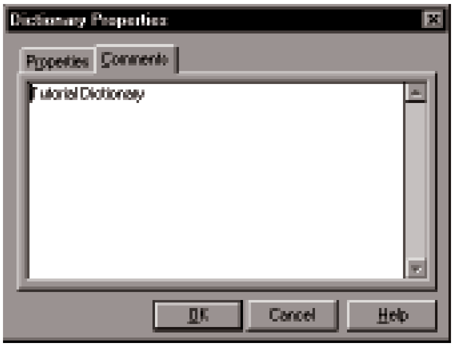
1. **Press** the **Dictionary Properties** button.

   The **Dictionary Properties** dialog appears.

2. **Select** the **Comments** tab then **type** *Tutorial Dictionary* in the text field.

   The **Comments** tab allows you to write free form text notes regarding the dictionary. It's optional, but extremely useful for programmers who may have to return to a project for maintenance after an interval of months.

   This dialog also provides a **Password** button, which allows you to prevent others from using this dictionary. There's no need to fill it in for the tutorial, but it's a useful feature to keep in mind.



Tutorial Dictionary

3. Close the **Dictionary Properties** dialog by pressing the **OK** button.

Copying Files From One Dictionary to Another

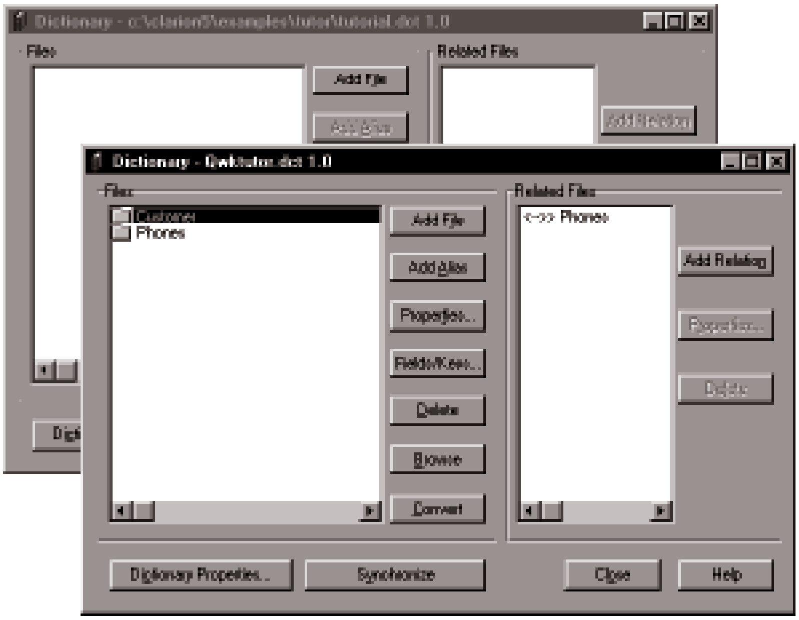
You can use the standard Windows copy and paste commands to copy file definitions from another dictionary (or to copy fields from one file to another). **In other words, once you've defined it once, why bother to re-define it when you can just copy what you've already done!**

Copy the Customer File Definition

Open the other data dictionary, select a file, and copy it

1. **Choose** **File ➤ Pick** from the menu then **select** the **Dictionary** tab.
2. **Select** the *QWKTUTOR.DCT* file from the file list, and **press** the **Select** button.

   Another Dictionary dialog opens, containing the file definitions from the Quick Start application. Move it and you can see that both .DCT files are open.



3. **Select** the **Customer** file from the **Files** list.
4. **Choose** **Edit ➤ Copy** (or **press** CTRL+C).
5. CLICK on the *TUTORIAL.DCT* **Dictionary** dialog.

   This makes the Tutorial dictionary the active dictionary.
6. **Choose** **Edit ➤ Paste** (or **press** CTRL+V).

The **Edit File Properties** dialog appears.

Not only does this operation copy the file definition; it copies the fields and keys as well.

7. **Press** the **OK** button to close the **Edit File Properties** dialog.

Copy the Phones File Definition

Now use the copy and paste commands to copy the other file definition.

Select the file and copy it

1. **CLICK** on the *QWKTUTOR.DCT* **Dictionary** dialog.
2. **Select** the **Phones** file from the **Files** list.
3. **Choose** **Edit ➤ Copy** (or **press** CTRL+C).
4. **Press** CTRL+F6, or **CLICK** on the *TUTORIAL.DCT* **Dictionary** dialog.

   This makes the Tutorial dictionary the active dictionary.
5. **Choose** **Edit ➤ Paste** (or **press** CTRL+V).

   The **Edit File Properties** dialog appears.
6. **Press** the **OK** button to close the **Edit File Properties** dialog.

Close the Quick Start dictionary file

1. **CLICK** on the *QWKTUTOR.DCT* **Dictionary** dialog.
2. **Press** the **Close** button, or **choose** **File ➤ Close**.

Relating the Files

You can copy file definitions (including their keys), but Clarion cannot copy the file relationships from other dictionaries. Therefore, you must re-define the relationship for these two files.

Define the first side of the relationship

We'll define this relationship differently than we did in the *Quick Start Tutorial* in *Getting Started*—from the Many (Child) side instead of the One (Parent) side, just to show you that we can.

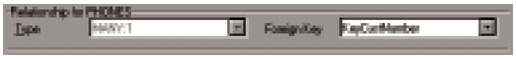
1. **Highlight** the *Phones* file then **press** the **Add Relation** button.

   The **New Relationship Properties** dialog appears. Because the last file selected was the *Phones* file, we'll set up the relationship from its perspective.

   Each Customer can have multiple phones, so Customer is the parent in a *parent-child relationship*. Therefore, from Phones file's perspective, it's a **MANY:1** relationship.

2. **Choose MANY:1** from the **Type** dropdown list in the **Relationship for Phones** group box.

3. **Choose** *KeyCustNumber* from the **Foreign Key** dropdown list.

   This is the key that matches a primary key in the Customers file; therefore, it's a foreign key.

Define the other side of the relationship

1. **Choose** *Customer* from the **Related File** dropdown list in the **Parent** group box.

2. **Choose** *KeyCustNumber* from the **Primary Key** dropdown list.

   In the *parent-child relationship*, the foreign key in the child must relate to a primary key in the parent. See *Anatomy of a Database* in Chapter 4 of this book, and the *Database Design* article in the *Programmer's Guide*, for more information on relational database theory.

3. **Press** the **Map by Name** button to link the fields.

   You can use this button because you named the linking fields the same in both files. This is a very good practice to make a habit of, since it makes application maintenance much easier over the long term. When you come back to make changes to a project that you completed a while ago, it is much easier to recognize the linking fields if they are named the same on both sides of the releationship.

Set Referential Integrity Constraints

By setting Referential Integrity constraints, you can specify how the Application Generator writes the source code that handles what happens if an end user attempts to modify a value in a primary key, or attempts to delete a parent that has children. If you don't set constraints, an end user can compromise the integrity of the database by creating "orphan" records one of two ways: by deleting a parent with children, or changing the parent's linking field value. In applications generated form Clarion's Application Builder Class templates, the actual code required to maintain your Referential Integrity is built into Application Builder Class (ABC) library.

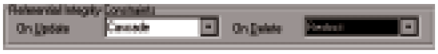
For the tutorial, specify that the application should *update* the foreign key record, if the primary key field value is changed. Also, specify that it should *not allow* the user to delete a parent with children.

1. In the **Referential Integrity Constraints** group box, **choose Cascade** from the **On Update** dropdown list.

   Cascading a change means that the application extends the change and updates the foreign key (child file) field, for all the child file records related to that one parent file record.

2. **Choose Restrict** from the **On Delete** dropdown list.

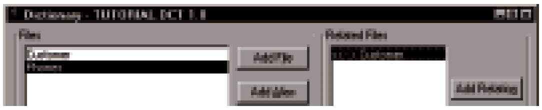
   Restricting a delete means that the application does not allow deleting a parent with children (the user must first delete all the children).

   Referential Integrity Constraints
   On Update: Cascade
   On Delete: Restrict

3. **Press OK** to close the **New Relationship Properties** dialog.

At this point, your **Dictionary** dialog looks like this:



Look at the small arrows to the left of the related file name in the **Related Files** list. It indicates the nature of the relationship between the two files. Two angle brackets (>> or <<) point to the many side from the one side. One angle bracket (> or <) points to the one side from the many side. Therefore, **Phones <<-> Customers** indicates *Many* phones file records can be related to *One* customer file record.

Save Your Work

It is a good habit to save your work frequently while developing your applications (power outages come without any prior notice). To do so, choose **File ➤ Save**, or press the *Save button* on the tool bar. This writes the dictionary file to disk.

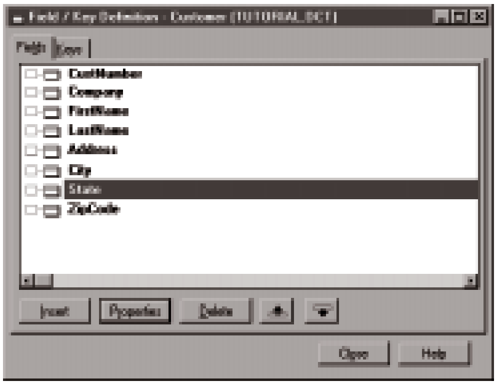
Pre-Defining Window Control Formats

Within the data dictionary, you can specify the default properties of the window controls which will update the fields you define. You can also specify certain Data Integrity rules by setting Validity Checks and the field's initial value. **These options are key factors in making the Application Generator do much of your work *for* you!**

Access the Field Properties dialog

1. **Highlight** the *Customer* file in the **Files** list.
2. **Press** the **Fields/Keys...** button.

   The **Field/Key Definition** dialog allows you to edit the properties for any field or key in the file.



3. **Select** the *State* field, and **press** the **Properties** button.

   The **Edit Field Properties** dialog appears, showing the options Quick Start filled in for this field.

Set Validity Checks

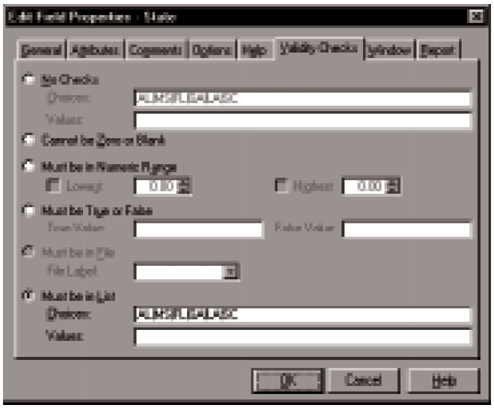
1. **Select** the **Validity Checks** tab.

   The **Validity Checks** tab allows you to set numeric ranges for number fields, specify that a field value must match another field in a related value, must be true or false, and in this case, that the field value must be in a list you specify in this dialog.
2. **Select** the **Must be in List** radio button.
3. **Type** the following in the **Choices** box:

   AL|MS|FL|GA|LA|SC

   A vertical bar (|) must separate each choice.

This defines the actual list of allowable choices. In this case, the dictionary specifies that only the state abbreviations for these six southern states of the United States is acceptable. You will specify that the default control for this field is a drop down list.

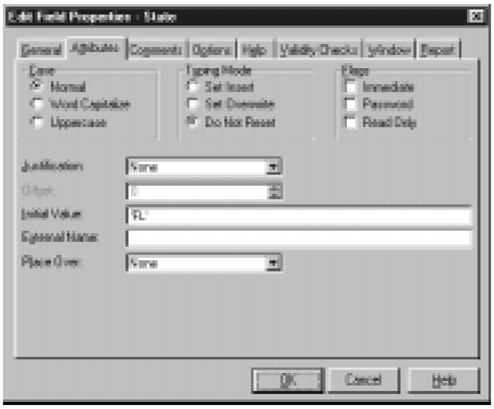


No Checks
Choices: AL|MS|FL|GA|LA|SC
Values:
Cannot be Zero or Blank
Must be in Numeric Range
Lowest 0.00
Highest 0.00
Must be True or False
True Value:
False Value:
Must be in File
File Label:
[✓ Must be in List]
Choices: AL|MS|FL|GA|LA|SC
Values:

Set a default value

1. **Select** the **Attributes** tab.
2. **Type** *'FL'* in the **Initial Value** field (including the single-quote marks).

   This specifies that anytime the control appears, its default value will be "FL." Initial values can be time savers for the end user; in this case, if most customers were located in "FL," it saves picking it from the list each time a new customer has to be added. The single-quote marks are necessary because you can also name a variable or function as the initial value of a field in a data file (be aware that there are slightly different rules for initial values of memory variables). In this case, the initial value is a string constant, as identified by the single quote marks around it.



Case: [✓ Normal] Word Capitalize Uppercase
Typing Mode: Set Insert Set Overwrite [✓ Do Not Reset]
Flags: Immediate Password Read Only
Justification: None
Offset: 0
Initial Value: 'FL'
External Name:
Place Over: None

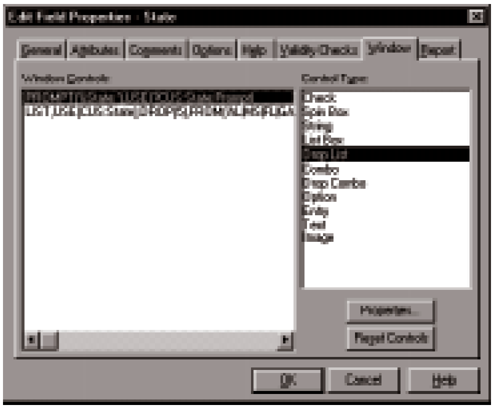
Specify a default window control

1. **Select** the **Window** tab.

   When you specify a **Must be in List** option, the default window control for the field is an OPTION structure with RADIO buttons. These appear by default in the **Window Controls** list.

2. **Select** **Drop List** from the **Control Type** list box.

   The **Window Controls** list now updates to show only a PROMPT and a LIST control with a DROP attribute.

3. **Press** the **OK** button to close the **Edit Field Properties** dialog.
4. **Press** the **Close** button to close the **Field/Key Definition** dialog.
5. **Choose** **File ➤ Save**, or **press** the *Save button* on the tool bar.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- You created a new, empty data dictionary (.DCT).
- You copied existing file definitions from one data dictionary to another (the easy way to work—never re-invent the wheel).
- You defined the relationship and Referential integrity constraints between the files.
- You pre-defined the window control format, data validity check, and initial value for one of the fields in the database.

In the next chapter, you'll learn how to add a file to the data dictionary, starting totally "from scratch"—without using Quick Load. You'll see just how quick and easy it is to do even without using a wizard.

3 - ADDING FILES AND FIELDS

Defining New Data Files

After copying and modifying the two files defined in the Quick Start application, you're ready to add a new file from scratch.

Starting Point:
The TUTORIAL.DCT file should be open.

Create the Orders File

Specify the label, prefix, and description

1. In the **Dictionary** dialog, **press** the **Add File** button.
2. When the **Add File** dialog appears asking whether you wish to use Quick Load, **press** the **No** button.

   The **New File Properties** dialog appears.

   The **Usage** radio buttons allow you to specify whether you're creating a data **File** definition, **Global** variable definitions, or a **Pool** of field definitions from which to derive new fields (we'll discuss deriving a little later). By choosing the **File** radio button, Clarion uses the information you fill in here for the FILE data structure declaration (see the *Language Reference*).
3. **Type** *Orders* in the **Name** field, then **press** TAB.

   The **Name** field only accepts a valid Clarion label, which uniquely identifies the data structure. A label may only contain letters, numbers, and the underscore(_) or colon (:) characters, and must begin with a letter or underscore. Executable code statements use this label to refer to the file.

After pressing TAB, "ORD" automatically appears in the **Prefix** field. The prefix is one way to uniquely identify fields of the same name in different data files. For example, *ORD:CustNumber* is the *CustNumber* field in the *Orders* file while *CUS:CustNumber* is the *CustNumber* field in the *Customers* file. You can also uniquely identify fields by using Field Qualification syntax (discussed in the *Language Reference)*.

4. **Type** *Order header file* in the **Description** field.

   This description appears next to the data file label in the *Dictionary* dialog list. If you select the **Comments** tab, you can type in a long text description. A description of what the file is for can be very helpful for when you return to the file for maintenance programming.

Choose the file driver

1. **Choose** **TOPSPEED** from the **File Driver** dropdown list.

   This declares the file format for the data file as the TopSpeed file format. This is the newer of the two proprietary file formats that TopSpeed Corporation has developed for use in Clarion (the older is "Clarion").

   The *Programmer's Guide* documents all the available file drivers and provides information about what data types each one supports, plus other useful information such as the default file extensions for data and/or index files. It also provides tips and tricks for choosing the right driver for the job, such as which drivers are best when your application must handle a very large database which is frequently updated, or which drivers are best when the quickest query time is the foremost concern.

2. **Press** the **OK** button.

   You can accept the defaults for all other options in the dialog. The dialog box closes, and the **Dictionary** dialog lists the Orders file, with "Order header file" listed next to it.

Name the Detail and Products Data Files

Create the Detail file

1. **Press** the **Add File** button in the **Dictionary** dialog.

   Choose **No** when asked if you wish to use Quick Load.

2. **Type** *Detail* in the **Name** field.
3. **Type** *Order detail file* in the **Description** field.
4. **Type** *DTL* in the **Prefix** field.

   By customizing the default prefix (changing it from "DET" TO "DTL"), you can make your code more readable. Three characters is the convention for file prefixes, but you are not limited to that.

5. **Choose** **TOPSPEED** from the **File Driver** dropdown list.

   Accept the defaults for all other options in the dialog.

6. **Press** the **OK** button.

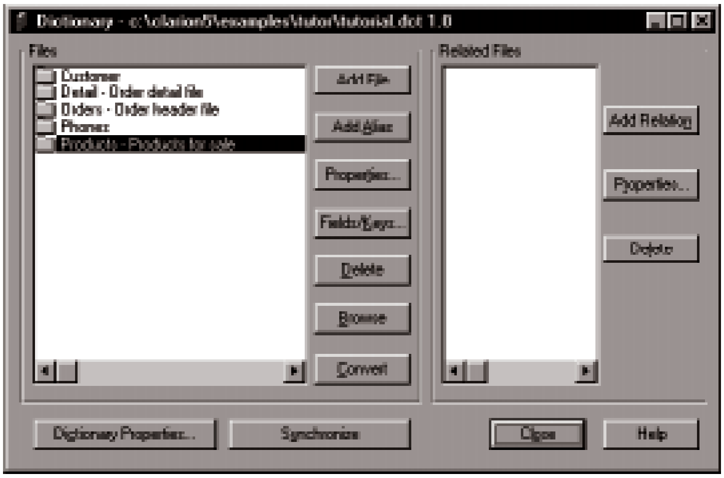
Create the Products file

1. **Press** the **Add File** button (don’t use Quick Load).
2. **Type** *Products* in the **Name** field.
3. **Type** *Products for sale* in the **Description** field.
4. **Type** *PRD* in the **Prefix** field.
5. **Choose** **TOPSPEED** from the **File Driver** dropdown list.
6. **Press** the **OK** button.

Save your work

1. **Choose** **File ➤ Save**, or **press** the *Save button* on the tool bar.

At this point, your Dictionary dialog looks about like this:



Defining the Fields

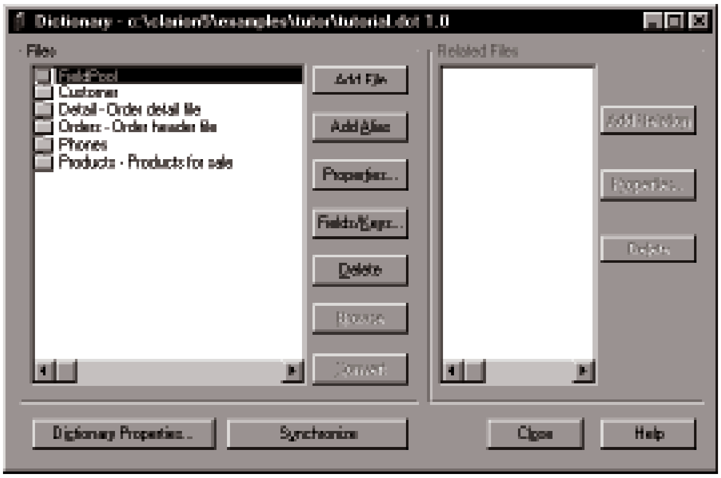
Define a Field Pool

At this point, we'll define several fields which will become the linking fields between the files in our database. We'll use a feature of the Data Dictionary called a "Field Pool" to ensure that all files that need these fields always define them exactly the same way.

1. **Press** the **Add File** button (don't use Quick Load).

   Field Pools are treated just like a file in the Data Dictionary, even though they do not generate any code into applications. You may have as many Field Pools in your data dictionary as you choose, but there is usually no need for more than one.

2. **Choose** the **Pool** radio button .
3. **Type** *FieldPool* in the **Name** field.
4. **Type** *POOL* in the **Prefix** field.
5. **Press** the **OK** button.



FieldPool
Customer
Detail - Order detail file
Orders - Order header file
Phones
Products - Products for sale

Open the Field / Key Definition windows

1. **Highlight** the **FieldPool** "file" in the **Files** list.
2. **Press** the **Fields/Keys...** button.

   This window displays a list of all the field and keys defined in the "file." Since this is a new file, there is nothing to display. We could simply start adding fields, but instead, we'll start by copying a field from the Customer file.

3. CLICK and drag on the title bar of the **Field / Key Definition** window and move the window so you can see the underlying **Dictionary** dialog.

4. **Highlight** the **Customer** file in the **Files** list of the **Dictionary** dialog.

   Notice that focus changes to the **Dictionary** dialog.

5. **Press** the **Fields/Keys...** button.

   Now another **Field / Key Definition** window appears containing the fields defined for the Customer file.

Select the field and copy it

1. **Select** the **CustNumber** field from the **Fields** list.

2. **Choose** **Edit ➤ Copy** (or **press** CTRL+C).

3. **Choose** **Window ➤ Field / Key Definition - FieldPool** (or just CLICK on the open window to give it focus).

4. **Choose** **Edit ➤ Paste** (or **press** CTRL+V).

   The **Edit Field Properties** dialog appears.

5. **Press** the **OK** button to close the **Edit Field Properties** dialog.

   This is the field that will be the linking field for the relationship between the Customer and Orders files. Linking fields in separate files are always defined the same, so copying the field definition is one way to get the existing field's definition into the Field Pool.

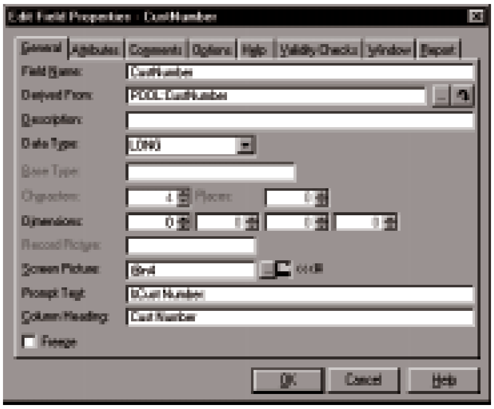
Derive the existing field

1. **Choose** **Window ➤ Field / Key Definition - Customer** (or just CLICK on the open window to give it focus).

2. **Highlight** the **CustNumber** field from the **Fields** list and **press** the **Properties** button.

   The **Edit Field Properties** dialog appears.

3. **Type** *POOL:CustNumber* in the **Derived From** field.



Edit Field Properties - CustNumber

Field Name: CustNumber
Derived From: POOL:CustNumber
Description:
Data Type: LONG
Base Type:
Characters: 4 Places: 0
Dimensions: 0 0 0 0
Record Picture:
Screen Picture: @n4
Prompt Text: &Cust Number
Column Heading: Cust Number
☐ Freeze

This means that the CUS:CustNumber field is now *derived from* the POOL:CustNumber field. The term "derived from" means that the POOL:CustNumber field's definition is the "parent" and all "children" fields which are "derived from" that field automatically share all the attributes of the parent.

Deriving field definitions from existing fields gives you the ability to make changes in only one place, then cascade those changes to all derived fields. For example, if the definition of the CustNumber field needs to change in all files using it, simply make one change to the POOL:CustNumber field definition, then cascade that change to all the derived CustNumber fields in all files. You can do this just by **choosing** **Edit ➤ Distribute Field** after changing the POOL:CustNumber field.

4. **Press** the **OK** button to close the **Edit Field Properties** dialog.

Add the rest of the fields to the Field Pool

1. **Choose** **Window ➤ Field / Key Definition - FieldPool** (or just CLICK on the open window to give it focus).
2. **Press** the **Insert** button to open the **New Field Properties** dialog.

   Once you begin the process of defining new fields, an empty **New Properties** dialog automatically appears after you add each successive field. This speeds up the process of adding multiple fields. After adding your last field, you just have to press the **Cancel** button on an empty **New Field Properties** dialog to return to the **Field / Key Definition** dialog.
3. **Type** *OrderNumber* in the **Field Name** field.

   This will be the linking field between the Orders and Detail files.
4. **Choose** **SHORT** from the **Data Type** dropdown list.

   This specifies a short integer (-32,768 to 32,767).
5. **Press** the **OK** button.
6. **Type** *ProdNumber* in the **Field Name** field.

   This will link the Detail file to the Products file.
7. **Choose** **SHORT** from the **Data Type** dropdown list.
8. **Press** the **OK** button.
9. **Press** the **Cancel** button.

   The **Field / Key Definition** dialog re-appears.
10. **Press** the **Close** button.

Define the fields in the Orders File

At this point, go back to the Orders data file and prepare to define its fields.

Open the Field / Key Definition windows

1. **Highlight** the **Orders** file in the **Files** list.
2. **Press** the **Fields/Keys...** button.
3. **Press** the **Insert** button to open the **New Field Properties** dialog.
4. **Type** *CustNumber* in the **Field Name** field.
5. **Press** the ellipsis **(...)** button to the right of the **Derived From** field.

   A **Select** window appears containing tree lists of all the fields already defined for all the files in the dictionary. You can derive new fields from any existing field—whether that field is in a file definition, global data, or a field pool.

6. **Highlight** the **CustNumber** field in the **FieldPool** file then **press** the **Select** button.

   The new field automatically becomes a perfect copy of the field from which it was derived—right down to the prompts and window control type. The button with the circular arrow icon right next to the ellipsis (...) button allows you to refresh the derived field from its parent's definition, if necessary (but the easier way is to **choose** **Edit ➤ Distribute Field** after changing the parent field).

5. **Press** the **OK** button to close the **New Field Properties** dialog.

   This is the field that will provide the link between the Orders and Customer files.

Derive the OrderNumber field

This provides a unique identifier for each order

1. **Type** *OrderNumber* in the **Field Name** field.
2. **Press** the ellipsis **(...)** button to the right of the **Derived From** field.
3. **Highlight** the **OrderNumber** field in the **FieldPool** file then **press** the **Select** button.
4. **Press** the **OK** button to close the **New Field Properties** dialog.

Define the InvoiceAmount field

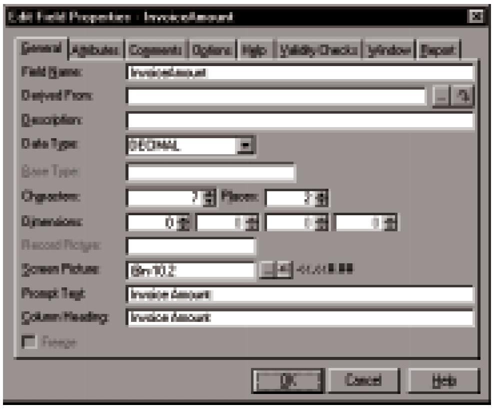
This field stores the total amount of the order.

1. **Type** *InvoiceAmount* in the **Field Name** field.
2. **Choose** **DECIMAL** from the **Data Type** dropdown list.
3. **Type** *7* in the **Characters** field.

   This specifies the total number of digits in the number (on both sides of the decimal point).

4. **Type** *2* in the **Places** field.

This specifies the number of digits to the right of the decimal point.



Edit Field Properties - InvoiceAmount

Field Name: InvoiceAmount
Derived From:
Description:
Data Type: DECIMAL
Base Type:
Characters: 7 Places: 2
Dimensions: 0 0 0 0
Record Picture:
Screen Picture: @n10.2 -n1,nn8.88
Prompt Text: Invoice Amount
Column Heading: Invoice Amount
Freeze

5. **Press** the **OK** button.

Define the OrderDate field

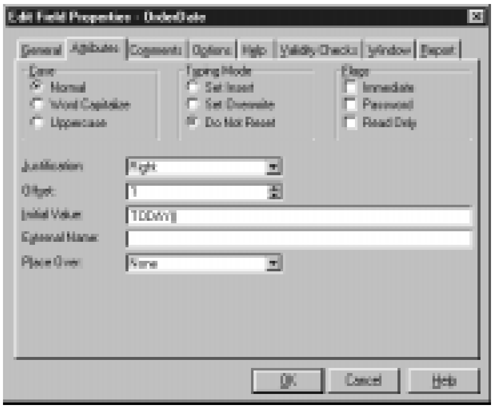
This field stores the date the order was placed.

1. **Type** *OrderDate* in the **Field Name** field.
2. **Choose** **LONG** from the **Data Type** dropdown list.

   LONG is the preferred date storage data type in the TopSpeed driver. This will contain a Clarion Standard Date value (see the *Language Reference* for more on Clarion Standard Dates and Times).
3. **Type** *@d1* in the **Screen Picture** field.

   The screen picture specifies the default “character” formatting for a field. In this case @d1 signifies MM/DD/YY format. The dialog box displays a representation of the formatting next to the field.
4. **Select** the *Attributes* tab then **type** *TODAY()* in the **Initial Value** field.

   The generated source code places today’s date in any control allowing a new record entry, using the built-in TODAY() function.



Edit Field Properties - OrderDate

Case: [✓ Normal] Word Capitalize Uppercase
Typing Mode: Set Insert Set Overwrite [✓ Do Not Reset]
Flags: Immediate Password Read Only
Justification: Right
Offset: 1
Initial Value: TODAY()
External Name:
Place Over: None

5. **Press** the **OK** button.

Define the OrderNote field

This allows for a short note for special handling instructions.

1. **Type** *OrderNote* in the **Field Name** field.
2. **Choose** **STRING** from the **Data Type** dropdown list.
3. **Type** *80* in the **Characters** field.

   This specifies 80 characters.
4. **Press** the **OK** button.

Close the dialogs

All the fields are defined, and a blank **New Field Properties** dialog should be active at this point.

1. **Press** the **Cancel** button to close the **New Field Properties** dialog.
2. **Press** the **Close** button to close the **Field / Key Definition** dialog.

Define the fields for the Detail File

At this point, we'll define the fields for the Detail data file.

Derive the linking field definition

1. **Highlight** the **Detail** file in the **Files** list then **press** the **Insert** button to open the **New Field Properties** dialog.
2. **Type** *OrderNumber* in the **Field Name** field.
3. **Press** the ellipsis **(...)** button to the right of the **Derived From** field.
4. **Highlight** the **OrderNumber** field in the **FieldPool** file then **press** the **Select** button.
5. **Press** the **OK** button to close the **New Field Properties** dialog.

   This is the field that will be the link between the Orders and Detail files.

Define the ProdNumber field

This field allows you to relate this file and the Products file.

1. **Type** *ProdNumber* in the **Field Name** field.
2. **Press** the ellipsis **(...)** button to the right of the **Derived From** field.
3. **Highlight** the **ProdNumber** field in the **FieldPool** file then **press** the **Select** button.

4. **Press** the **OK** button to close the **New Field Properties** dialog.

Define the Quantity field

This stores the number of each product ordered.

1. **Type** *Quantity* in the **Name** field.
2. **Choose** **SHORT** from the **Data Type** dropdown list.
3. **Press** the **OK** button.

Define the ProdAmount field

This stores the unit cost of the product as it was at the time of the order.

1. **Type** *ProdAmount* in the **Field Name** field.
2. **Choose** **DECIMAL** from the **Data Type** dropdown list.
3. **Type** *5* in the **Characters** field.
4. **Type** *2* in the **Places** field.
5. **Press** the **OK** button.

Define the TaxRate field

1. **Type** *TaxRate* in the **Field Name** field.
2. **Choose** **DECIMAL** from the **Data Type** dropdown list.
3. **Type** *2* in the **Characters** field.
4. **Type** *2* in the **Places** field.
5. **Press** the **OK** button.

Close the dialogs

All the fields are defined, and a blank **New Field Properties** dialog should be active.

1. **Press** the **Cancel** button to close the **New Field Properties** dialog.
2. **Press** the **Close** button to close the **Field / Key Definition** dialog.

Define the fields for the Products File

At this point, we'll define the fields for the Products data file. This is the last file.

Derive the linking field definition

1. **Highlight** the **Products** file in the **Files** list of the **Dictionary** dialog.

You may have to move some windows first to do this. Focus changes to the **Dictionary** dialog.

2. **Press** the **Fields/Keys...** button.

   Now another **Field / Key Definition** window appears containing the fields defined for the Customer file.

3. **Press** the **Insert** button to open the **New Field Properties** dialog.
4. **Type** *ProdNumber* in the **Field Name** field.
5. **Press** the ellipsis (**...**) button to the right of the **Derived From** field.
6. **Highlight** the **ProdNumber** field in the **FieldPool** file then **press** the **Select** button.
7. **Press** the **OK** button to close the **New Field Properties** dialog.

   This is the field that will be the linking field for the relationship between the Detail and Products files.

Define the ProdDesc field

This allows for a product description.

1. **Type** *ProdDesc* in the **Field Name** field.
2. **Choose** **STRING** from the **Data Type** dropdown list.
3. **Type** *25* in the **Characters** field.
4. **Press** the **OK** button.

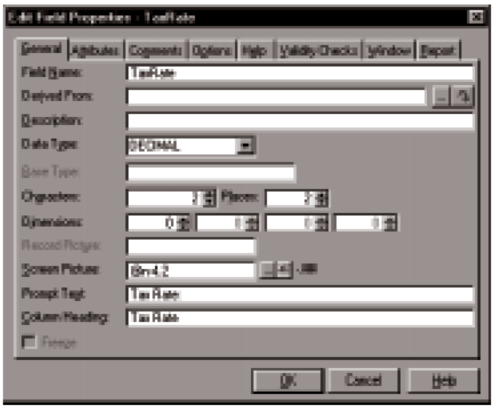
Define the ProdAmount field

This stores the unit cost of the product.

1. **Type** *ProdAmount* in the **Field Name** field.
2. **Choose** **DECIMAL** from the **Data Type** dropdown list.
3. **Type** *5* in the **Characters** field.
4. **Type** *2* in the **Places** field.
5. **Press** the **OK** button.

Define the TaxRate field

1. **Type** *TaxRate* in the **Field Name** field.
2. **Choose** **DECIMAL** from the **Data Type** dropdown list.
3. **Type** *2* in the **Characters** field.
4. **Type** *2* in the **Places** field.


Edit Field Properties - TaxRate

Field Name: TaxRate
Derived From:
Description:
Data Type: DECIMAL
Base Type:
Characters: 7 Places: 2
Dimensions: 0 0 0 0
Record Picture:
Screen Picture: @n4.2
Prompt Text: Tax Rate
Column Heading: Tax Rate
☐ Freeze

5. **Press** the **OK** button.

Close the dialogs and save your work

All the fields are defined, and a blank **New Field Properties** dialog should be active.

1. **Press** the **Cancel** button to close the **New Field Properties** dialog.
2. **Press** the **Close** button to close the **Field / Key Definition** dialog.
3. **Choose File ➤ Save**, or **press** the *Save* button on the tool bar.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- ♦ You created new data file definitions.
- ♦ You created a pool of field definitions from which new field definitions can be easily derived.
- ♦ You created the field definitions for all the files.

Now we'll go on to add keys and file relationships.

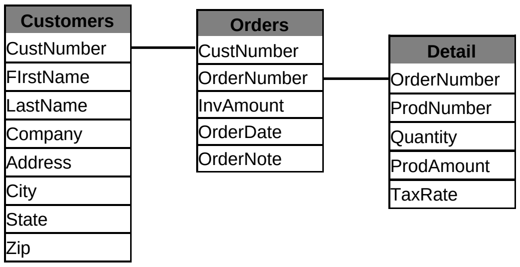
4 – ADDING KEYS AND RELATIONS

Now that all the files are defined, we can add keys then specify the file relationships. You already have defined the keys for the two files you created in the QwkTutor application in *Getting Started*. In this chapter, we'll define keys for the remaining files.

Starting Point:
The TUTORIAL.DCT file should be open.

Defining Keys for the Orders File

The fields in the Orders file that relate to other files in the database are the OrderNumber and CustNumber fields.



- The OrderNumber field relates to the Detail file.

  There should be no duplicate or null order numbers in the Orders file; this is a *primary* key.

  There may be multiple Detail records for a single matching Order Number. Therefore, this is a *One to Many* relationship—the Orders file is the "Parent" of the Detail file.

- The CustNumber field relates to the Customer file.

  There will be duplicate values in the CustNumber field that relate to records in the Customers file. The key we define in the Orders file is a *foreign* key. The Customers file key does not allow duplicates and nulls, and was defined as the primary key for that file.

  Multiple Order records can exist for each Customer, making this a *Many to One* relationship—the Orders file is the "Child" of the Customers file.

Create the Primary Key

Name the Key

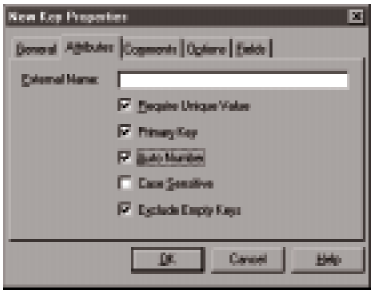
1. **Highlight** the *Orders* file in the **Files** list.
2. **Press** the **Field/Keys...** button.
3. **Select** the **Keys** tab.
4. **Press** the **Insert** button.

   The **New Key Properties** dialog appears.
5. **Type** *KeyOrderNumber* in the **Key Name** field.

   As an easy naming convention, we suggest incorporating both the word "key" and the field name in the key name (just as the Quick Start Wizard does).
6. **Select** the **Attributes** tab, **check** the **Require Unique Value** box, then **check** the **Primary Key** box.

   This specifies the key is a primary key. The generated source code automatically prevents the end user from inserting duplicate or null values.
7. **Check** the **Auto Number** box.

   The template-generated source code will increment the key field with each new record.



New Key Properties

General | Attributes | Comments | Options | Fields

External Name:

✓ Require Unique Value
✓ Primary Key
✓ Auto Number
☐ Case Sensitive
✓ Exclude Empty Keys

OK | Cancel | Help

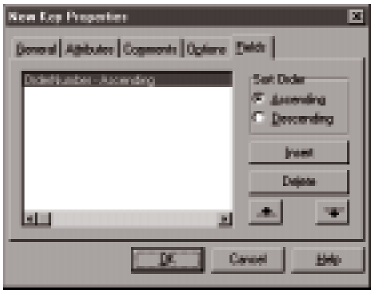
8. **Select** the **Fields** tab.

Specify the key field

1. **Press** the **Insert** button.

   The **Insert Key Component** dialog appears, ready for you to specify a field or fields for the key.
2. **DOUBLE-CLICK** on *OrderNumber*.

   This adds the field to the list of component fields for this key.



OrderNumber - Ascending

Sort Order
- [✓ Ascending]
- Descending

3. **Press** the **OK** button.

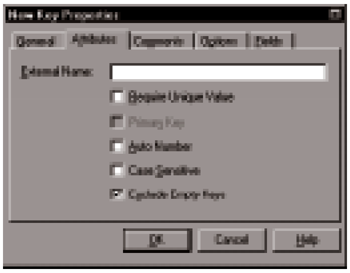
   A blank **New Key Properties** dialog appears, ready for you to specify another key.

Define a Foreign Key

Now you can define the CustNumber key. There may be duplicates within this file. It relates to the primary key in the Customers file, so therefore, this is a foreign key.

1. **Type** *KeyCustNumber* in the **Key Name** field.
2. **Select** the **Attributes** tab.

   The key *does* allow duplicates, so leave all the default settings.



External Name:
- Require Unique Value
- Primary Key
- Auto Number
- Case Sensitive
- [✓ Exclude Empty Keys]

3. **Select** the **Fields** tab.
4. **Press** the **Insert** button.

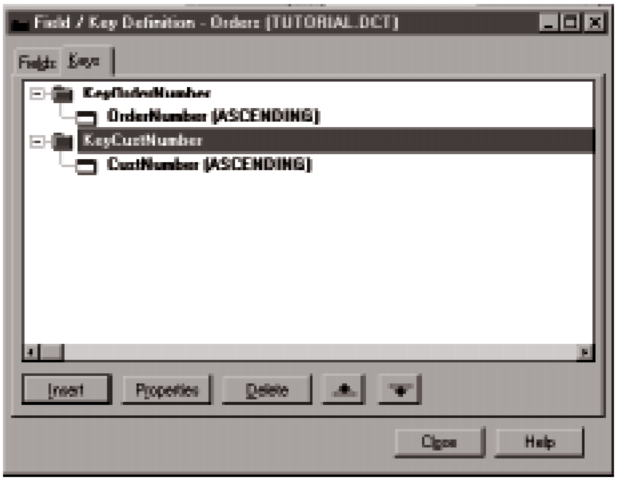
   The **Insert Key Component** dialog appears, ready for you to specify a field or fields for the key.

5. **Select** *CustNumber* then **press** the **Select** button.
6. **Press** the **OK** button.

   Each time you finish a new key, a blank **New Key Properties** dialog appears, ready for the next.

7. **Press** the **Cancel** button to close the blank **New Key Properties** dialog.

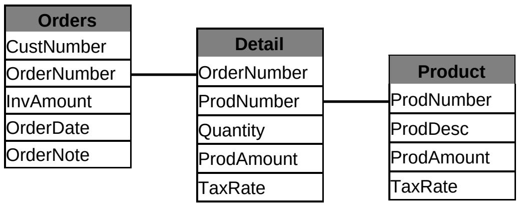
The **Field/Keys Definition** dialog for the Orders file now looks like this:



8. **Press** the **Close** button to close the **Field/Key Definition** dialog.
9. **Choose File ➤ Save**, or **press** the *Save button* on the tool bar.

Defining Keys for the Detail File

The fields in the Detail file that relate to other files in the database are the ProdNumber and OrderNumber fields.



- The OrderNumber field relates to the Orders file.

  There will be duplicate values in the OrderNumber field that relate to records in the Orders file. The key we define in the Detail file is another *foreign* key. The Orders file key does not allow duplicates and nulls, and was defined as a primary key.

  There may be more than one Detail record for a single matching Order Number. Therefore, this is a *Many to One* relationship, with the Detail file the “Child” of the Orders file.

- The ProdNumber field relates to the Products file.

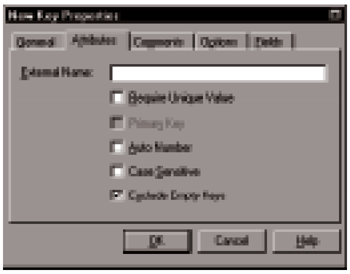
  There will be duplicate values in the ProdNumber field for the records in the Detail file. There may be more than one Detail record containing a single Product Number. Therefore, this is another *Many to One* relationship, with the Detail file the “Child” of the Product file.

Define the First Foreign Key

Define KeyProdNumber so that there may be duplicate ProdNumber values in this file.

1. **Highlight** the *Detail* file in the **Files** list.
2. **Press** the **Field/Keys...** button.
3. **Select** the **Keys** tab.
4. **Press** the **Insert** button.
5. **Type** *KeyProdNumber* in the **Key Name** field.
6. **Select** the **Attributes** tab.

   The key *does* allow duplicates so leave all the default settings.



- [ ] Require Unique Value
- [ ] Primary Key
- [ ] Auto Number
- [ ] Case Sensitive
- [x] Exclude Empty Keys

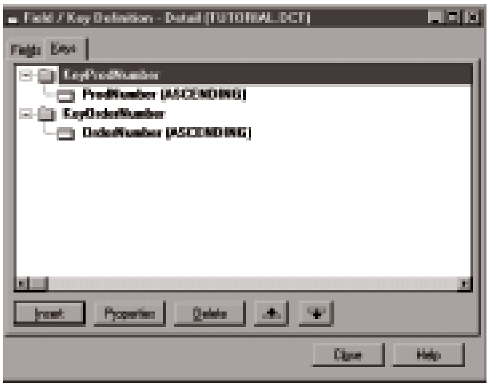
7. **Select** the **Fields** tab.
8. **Press** the **Insert** button.
9. **Select** *ProdNumber* then **press** the **Select** button.
10. **Press** the **OK** button.

    A blank **Key Properties** dialog appears, ready for you to specify another key.

Define the Second Foreign Key

1. **Type** *KeyOrderNumber* in the **Key Name** field.
2. **Select** the **Fields** tab.
3. **Press** the **Insert** button.
4. **Select** *OrderNumber* then **press** the **Select** button.
5. **Press** the **OK** button.
6. **Press** the **Cancel** button to close the blank **New Key Properties** dialog.

   The **Field / Key Definition** dialog for the Detail file now looks like this:



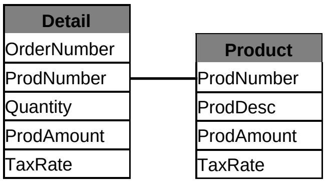
KeyProdNumber
ProdNumber (ASCENDING)
KeyOrderNumber
OrderNumber (ASCENDING)

7. **Press** the **Close** button to close the **Field / Key Definition** dialog.
8. **Choose** **File ➤ Save**, or **press** the *Save* button on the tool bar.

Defining Keys for the Products File

Only one field in the Products file relates to another file in the database: the ProdNumber field.

- The ProdNumber field relates to the Detail file.



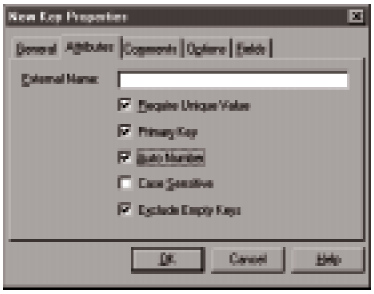
There should be no duplicate or null order numbers in the Products file; this is a *primary* key.

For each ProdNumber in the record there can be many Detail records. This is a *One to Many* relationship with the Products file a “Parent” to the Detail file.

Create the Primary Key

Name the Key

1. **Highlight** the *Products* file in the **Files** list.
2. **Press** the **Field/Keys...** button.
3. **Select** the **Keys** tab.
4. **Press** the **Insert** button.
5. **Type** *KeyProdNumber* in the **Key Name** field.
6. **Select** the **Attributes** tab.
7. **Check** the **Require Unique Value** box, then **check** the **Primary Key** box.
8. **Check** the **Auto Number** box.



External Name:

[✓] Require Unique Value
[✓] Primary Key
[✓] Auto Number
[ ] Case Sensitive
[✓] Exclude Empty Keys

9. **Select** the **Fields** tab.
10. **Press** the **Insert** button.
11. **Select** *ProdNumber* then **press** the **Select** button.
12. **Press** the **OK** button.

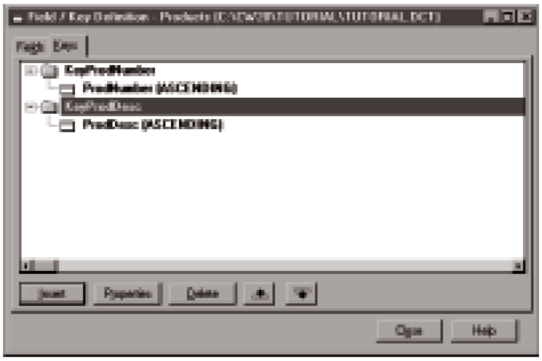
    A blank **Key Properties** dialog appears, ready for you to specify another key.

Define an Alphabetical Key

Users will probably want to see the list of Products in alphabetical order, so we'll add a key for that.

1. **Type** *KeyProdDesc* in the **Key Name** field.
2. **Select** the **Fields** tab.
3. **Press** the **Insert** button.
4. **Select** *ProdDesc* then **press** the **Select** button.
5. **Press** the **OK** button.
6. **Press** the **Cancel** button to close the blank **Key Properties** dialog.

   The **Field / Key Definition** dialog for the Product file now looks like this:



KeyProdNumber
ProdNumber (ASCENDING)
KeyProdDesc
ProdDesc (ASCENDING)

7. **Press** the **Close** button to close the **Field / Key Definition** dialog.
8. **Choose** **File ➤ Save**, or **press** the *Save button* on the tool bar.

Defining File Relationships

Defining Relationships for the Orders File

Now that all the keys are defined, we can add the relations. Once you have defined relationships, you can add Validity Checks for the fields that should only contain values that exist in another file. These are the last steps to completing the data dictionary.

- KeyOrderNumber relates the Orders file to the Detail file in a *One to Many* relationship.
- KeyCustNumber relates the Orders file to the Customer file in a *Many to One* relationship.

Define the first relationship

1. **Highlight** the *Orders* file in the **Files** list.
2. **Press** the **Add Relation** button.

   The default relationship **Type** is *1:MANY*, which you should accept.
3. **Choose** *KeyOrderNumber* from the **Primary Key** dropdown list.
4. **Choose** *Detail* from the **Related File** dropdown list.
5. **Choose** *KeyOrderNumber* from the **Foreign Key** dropdown list.
6. **Press** the **Map by Name** button.

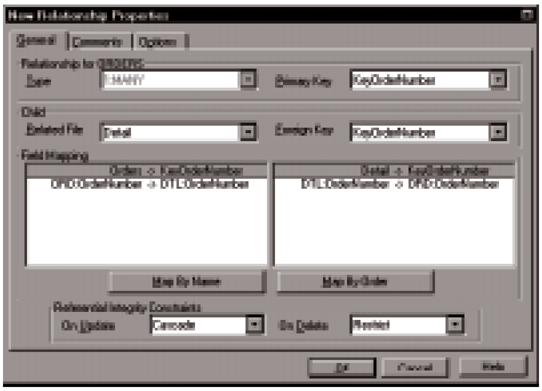
   This establishes the relationship by linking all the fields in the two keys that have the same name.

Set up the Referential Integrity constraints

1. **Choose** *Cascade* from the **On Update** dropdown list.

   This tells the templates to generate code to automatically update all related “Child” records when the “Parent” key field value changes.
2. **Choose** *Restrict* from the **On Delete** dropdown list.

   This does not allow the user to delete a “Parent” record that has related “Child” records.


New Relationship Properties

General | Comments | Options

Relationship for ORDERS
Type: 1:MANY
Primary Key: KeyOrderNumber

Child
Related File: Detail
Foreign Key: KeyOrderNumber

Field Mapping

Orders -> KeyOrderNumber	Detail -> KeyOrderNumber
ORD:OrderNumber -> DTL:OrderNumber	DTL:OrderNumber -> ORD:OrderNumber

Referential Integrity Constraints
On Update: Cascade
On Delete: Restrict

3. **Press** the **OK** button.

Define the second relationship

1. **Highlight** the *Orders* file in the **Files** list.
2. **Press** the **Add Relation** button.
3. **Choose** *MANY:1* from the **Type** dropdown list.

   Notice that, when you chose *MANY:1*, the prompts for the **Primary Key** and **Foreign Key** fields switched places. This happens because we are now defining this relationship from the "Child" file's viewpoint; the opposite side of the relationship to what we did previously. A Primary Key is always in the Parent file, while a Foreign Key is always in the Child file.
4. **Choose** *KeyCustNumber* from the **Foreign Key** dropdown list.
5. **Choose** *Customer* from the **Related File** dropdown list.

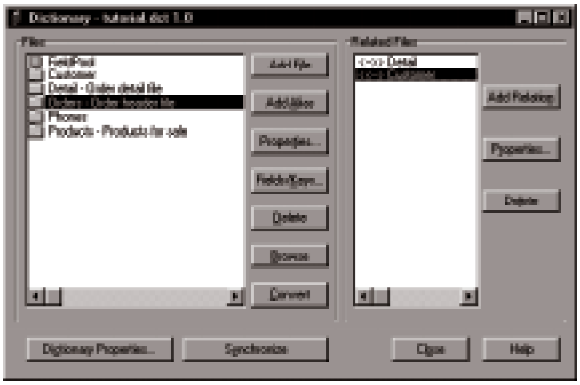
   This establishes the Customer file as the "Parent" in this relationship.
6. **Choose** *KeyCustNumber* from the **Primary Key** dropdown list.
7. **Press** the **Map by Name** button.

Set up the Referential Integrity constraints

1. **Choose** *Cascade* from the **On Update** dropdown list.

   Although we are defining this relationship from the "Child" file's viewpoint, the Referential Integrity constraints are still set on the "Parent" file actions.
2. **Choose** *Restrict* from the **On Delete** dropdown list.
3. **Press** the **OK** button.

   Your **Dictionary** dialog should now look like this:



4. **Choose** File ➤ Save, or **press** the *Save button* on the tool bar.

Defining Relationships for the Detail File

Each time you define a relationship in the Dictionary Editor, you define it for both files at the same time. Therefore, since you have defined all the relationships for the Orders file, there is only one relationship left to define in this data dictionary.

The last relationship is for the Detail file.

- KeyOrderNumber relates the Orders file to the Detail file in a *One to Many* relationship. You have already defined this.
- KeyProdNumber relates the Detail file to the Products file in a *Many to One* relationship.

Define the relationship

1. **Highlight** the *Detail* file in the **Files** list.
2. **Press** the **Add Relation** button.
3. **Choose** *MANY:1* from the **Type** dropdown list.
4. **Choose** *KeyProdNumber* from the **Foreign Key** dropdown list.
5. **Choose** *Products* from the **Related File** dropdown list.
6. **Choose** *KeyProdNumber* from the **Primary Key** dropdown list.
7. **Press** the **Map by Name** button.

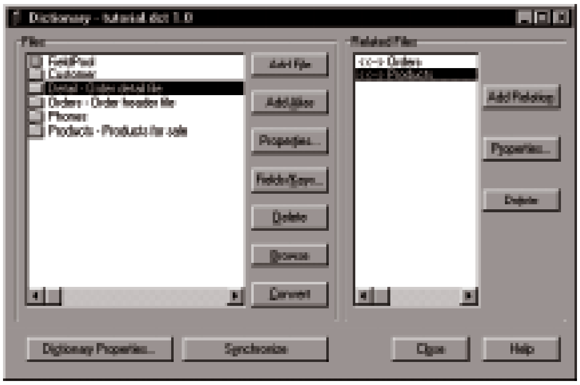
Set up the Referential Integrity constraints

1. **Choose** *Restrict* from the **On Update** dropdown list.

   We won't allow any changes to the product numbers.

2. **Choose** *Restrict* from the **On Delete** dropdown list.
3. **Press** the **OK** button.

   Your **Dictionary** dialog should now look like this:



4. **Choose File ➤ Save**, or **press** the *Save* button on the tool bar.

Defining Relationship-Dependent Validity Checks

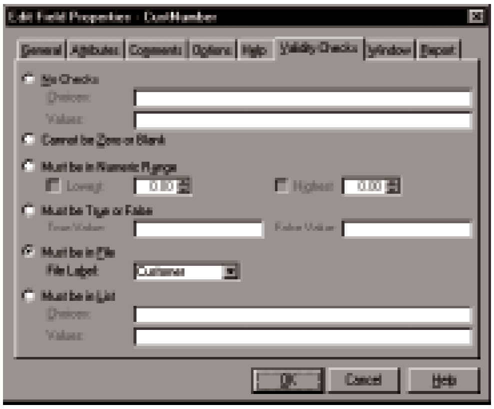
Now that all the file relationships have been defined, we can set the Validity Checks for two fields that we expect to put on update forms.

- When entering a new Orders file record, we can specify that the CustNumber must match an existing record in the Customer file.
- When entering a new Detail file record, we can specify that the ProdNumber must match an existing record in the Products file.

Define the Validity Check for Order Records

1. **Highlight** the *Orders* file in the **Files** list.
2. **Press** the **Field/Keys...** button.
3. **Highlight** *CustNumber* and **press** the **Properties** button.
4. **Select** the **Validity Checks** tab.
5. **Select** the **Must Be In File** radio button.
6. **Choose** *Customer* from the **File Label** dropdown list.

   This requires that the field can only contain values verified by getting a matching record from the Customer file. This is validated using the file relationship information, which is why this Validity Check cannot be set until the relationships have been defined.


Edit Field Properties - CustNumber

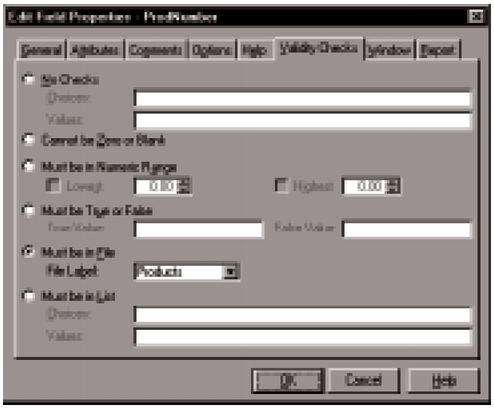
General | Attributes | Comments | Options | Help | Validity Checks | Window | Report

No Checks
Choices:
Values:
Cannot be Zero or Blank
Must be in Numeric Range
Lowest: 0.00 Highest: 0.00
Must be True or False
True Value: False Value:
(●) Must be in File
File Label: Customer
Must be in List
Choices:
Values:

7. **Press** the **OK** button.
8. **Press** the **Close** button to close the **Field / Key Definition** dialog.

Define the Validity Check for Detail Records

1. **Highlight** the *Detail* file in the **Files** list.
2. **Press** the **Field/Keys...** button.
3. **Highlight** *ProdNumber* and **press** the **Properties** button.
4. **Select** the **Validity Checks** tab.
5. **Select** the **Must Be In File** radio button.
6. **Choose** *Products* from the **File Label** dropdown list.



7. **Press** the **OK** button.
8. **Press** the **Close** button to close the **Field / Key Definition** dialog.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- You created the keys for all the new file definitions.
- You defined the relationships between the new files.
- You defined two relationship-dependent validity checks to require that foreign key field values always have related primary key records in a parent file.

The data dictionary is now complete. In the next chapter, we will import some existing data from another application, to show you just how simple it is to accomplish.

5 - IMPORTING EXISTING DATA

Data File Conversion

You may have existing data from legacy applications that you want to save and use in your Clarion applications. Therefore, this chapter shows you:

- How to import a file definition from an existing data file.
- How to browse and edit a data file using the Database Manager.
- How to convert data from one file format to another.

Starting Point:
The TUTORIAL.DCT file should be open.

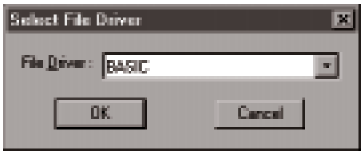
Importing a .CSV File Definition

One easy way to convert data files is to export your old data from the previous application to Comma Separated Values (.CSV) files. This is the file format originally used by the Basic language—the data is contained in double-quotes, fields are separated by commas, and records are separated by a Carriage Return/Line Feed. Clarion's BASIC file driver will easily read from and write to these .CSV files.

We will import the definition of an existing .CSV file containing Customer data, then generate a simple data file conversion program (to show you just how easy it is to do) to place the data in a TopSpeed data file.

Import the file definition

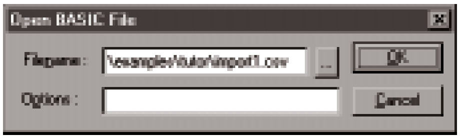
1. **Choose** **File ➤ Import File.**
2. **Select** **BASIC** from the dropdown list then **press** the **OK** button.



File Driver: BASIC

The **Open BASIC File** dialog appears.

3. In the **Filename** field, **type** *c:\clarion5\examples\tutor\import1.csv* then **press** the **OK** button.



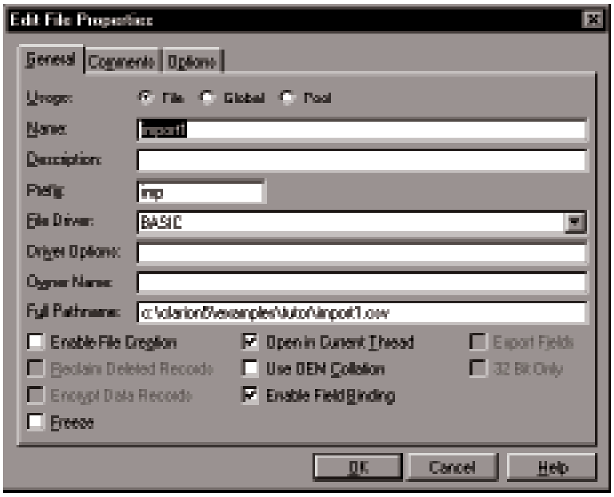
Open BASIC File

Filename: \examples\tutor\import1.csv

Options:

The **Edit File Properties** dialog appears.

4. **Press** the **OK** button.



Edit File Properties

Usage: [✓ File] Global Pool

Name: import1

Description:

Prefix: imp

File Driver: BASIC

Driver Options:

Owner Name:

Full Pathname: c:\clarion5\examples\tutor\import1.csv

☐ Enable File Creation ☑ Open in Current Thread ☐ Export Fields

☐ Reclaim Deleted Records ☐ Use OEM Collation ☐ 32 Bit Only

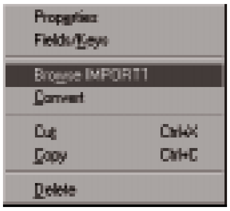
☐ Encrypt Data Records ☑ Enable Field Binding

☐ Freeze

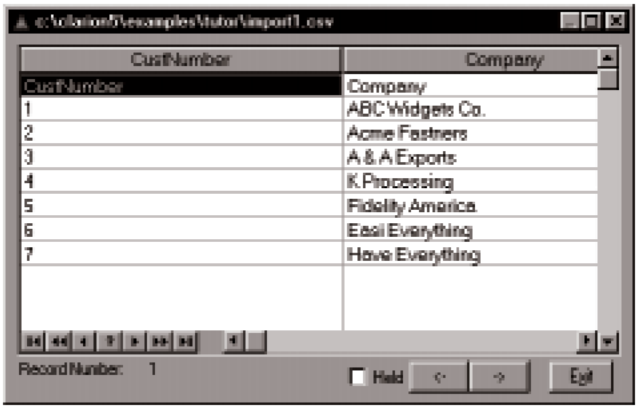
Now you have the IMPORT1.CSV file's definition. The next step will be to look at the data in the Database Manager.

Edit the data

1. **RIGHT-CLICK** on the IMPORT1 file, **highlight** **Browse IMPORT1** then **CLICK** to call the Database Manager.



The Clarion Database Manager allows you to directly edit the data in your files. This is a programmer's tool, designed to allow you to do whatever is necessary to change the actual data contained in your files. This means that there are no safeguards against violating your database's Referential Integrity or Data Integrity rules. Therefore, you must take care when you use this tool.



c:\clarion5\examples\tutor\import1.csv

CustNumber	Company
CustNumber	Company
1	ABC Widgets Co.
2	Acme Fastners
3	A & A Exports
4	K.Processing
5	Fidelity America
6	Easi Everything
7	Have Everything

Record Number: 1

Notice that the first record contains the field names, not actual data. This is a standard way that .CSV files are constructed. Also, these field names are exactly the same as the field names in the Customer file definition (this will make the data conversion much easier).

Converting a Data File

At this point, you’re looking at the .CSV file’s data in Clarion’s Database Manager utility. Next, you need to move that data into a TopSpeed file so your Clarion programs can use it.

Generate a file conversion program

1. **Choose** **File ➤ Convert File** (or **press** CTRL+V).

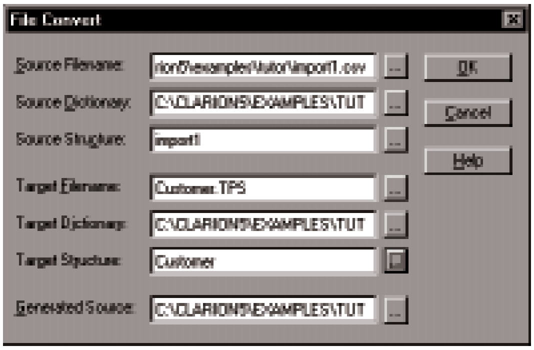
   The **File Convert** dialog appears.

2. In the **Target Filename** field, **type** in *Customer.TPS* as the name of the new file (eliminating all the default text that was in the field).

3. **Press** the ellipsis button (...) next to the **Target Structure** field.

4. **Highlight** *Customer*, then **press** the **Select** button.

   The **File Convert** dialog should now appear like this:



File Convert

Source Filename:	rion5\examples\tutor\import1.csv
Source Dictionary:	C:\CLARION5\EXAMPLES\TUT
Source Structure:	import1
Target Filename:	Customer.TPS
Target Dictionary:	C:\CLARION5\EXAMPLES\TUT
Target Structure:	Customer
Generated Source:	C:\CLARION5\EXAMPLES\TUT

When you press the **OK** button, this will generate all the Clarion source code necessary to take the data in the **Source Filename**, and copy it into a new **Target Filename** file, using the file format specified by the **Target Structure**.

The best reason to generate Clarion source code for the data conversion is to provide you the opportunity to modify the code before you compile and execute it to handle any special data conversion needs you may have. This makes the file conversion process completely flexible to handle any situation that can occur.

5. **Press** the **OK** button.

   A message box appears telling you the source code has been generated.

6. **Press** the **OK** button to return to the Database Manager.
7. **Press** the **Exit** button to return to the Data Dictionary Editor.

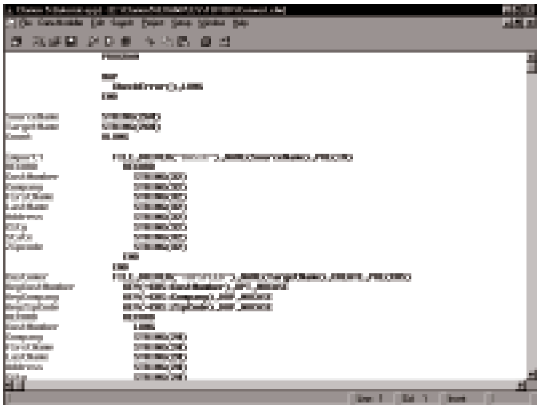
Delete the IMPORT1 file definition

The only purpose this file definition served was to allow the Database Manager to generate file conversion source code for you. Therefore, we can delete it from the Data Dictionary right now.

1. With IMPORT1 still highlighted, **press** the **Delete** button.
2. **Press** the **Yes** button when asked to confirm the deletion.
3. **Press** the **Close** button to exit the Data Dictionary Editor, and **press** the **Yes** button to save your changes as you exit.

Compile and execute the conversion program

1. **Choose** **File** ➤ **Open** (or **press** CTRL+O).
2. **Select** *Clarion source (*.clw)* from the **Files of type** dropdown list.
3. **Select** the *C:\CLARION5\TUTORIAL* directory.
4. **Highlight** the *CONVERT.CLW* file, then **press** the **Open** button.

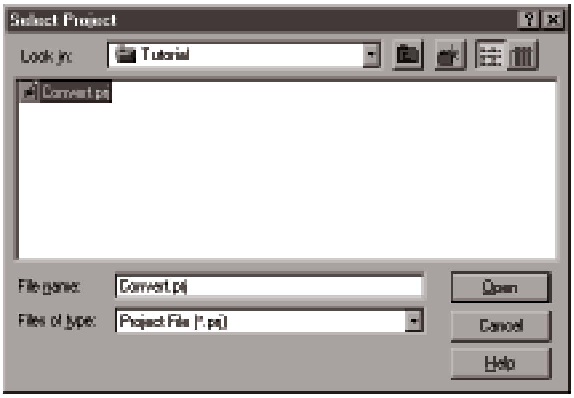


Clarion's Text Editor appears with the file loaded, ready to edit.

The Database Manager created the conversion program code in this file. This contains all the Clarion language source code necessary to read the data from the BASIC (.CSV) file and copy it to the TopSpeed data file.

3. **Choose** **Project ➤ Set...**.

The **Select Project** dialog appears.



4. **Select** *Project file (*.prj)* from the **Files of type** dropdown list.
5. **Select** the *C:\CLARION5\TUTORIAL* directory.
6. **Highlight** the *CONVERT.PRJ* file, then **press** the **Open** button.

The Database Manager also generated the CONVERT.PRJ file for you at the same time it created the CONVERT.CLW file.

Every Clarion program has a Project that controls the options for compiling the source code and linking to create the resulting .EXE file. For hand-coded programs (and file conversion programs generated by the Database Manager), these settings are contained in a .PRJ file. There is no .PRJ file necessary when you use the Application Generator—the .APP file itself contains all the Project settings.

At this point, you could modify the generated source code to perform any special data conversion you require (see the *How to Convert a File—Generate Source* and *How to Make a Field Assignment* topics in on-line Help for more information on customizing data file conversion code). However, there is nothing we need to do in this project, so you simply compile and run the program.

7. **Choose** **Project ➤ Run** (or **press** CTRL+R).

This compiles the program, links it into an .EXE, then runs the resulting executable to perform the file conversion. A status window appears as the program runs, letting you know the progress of the file conversion. Since there are only a few records to convert in this case, you probably won't be able to read it (it'll go by too fast).

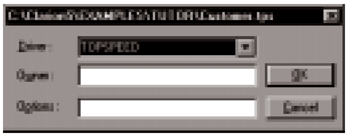
8. **Choose** **File ➤ Close** to close the file and exit the Text Editor.

Check it out

Now you can check the data in the new file by opening it with the Database Manager and browsing through the records.

1. **Choose** **File ➤ Open** (or **press** CTRL+O).
2. **Select** *Database file* from the **Files of type** dropdown list, **highlight** the *CUSTOMER.TPS* file, then **press** the **Open** button.

   A dialog opens that asks for the File Driver and any password and options required to open the file.

   Driver : TOPSPEED

3. **Select** *TOPSPEED* from the **Driver** dropdown list, then **press** the **OK** button.

   This demonstrates another way to open the Database Manager, other than from within the Data Dictionary Editor.

   Notice that the first record contains the field names and not actual data, just as the IMPORT1.CSV file did. You don't need these field names in this file, so just delete this "junk" record.

4. With the **highlight** bar in the first record, **press** DELETE, then **press** the **Yes** button when asked to confirm the deletion.
5. **Press** the **Yes** button when asked to make a backup file.

   The Database Manager always asks if you want to make a backup of the file the first time you perform any type of editing action on the data. It is always a good idea to let it make a backup (just in case).

6. **Press** the **Exit** button.

   A message box appears asking you whether to save the changes you made to the CUSTOMER.TPS file. This confirmation dialog provides you an additional opportunity to "roll back" any changes you made to the data if you decide it was a mistake to make any changes at all—*Yes* saves your changes and exits, *No* reverts the file to the state it was in before you entered the Database Manager and exits, and *Cancel* returns you to the Database Manager.

7. **Press** the **Yes** button to save the changes you made.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- ♦ You imported a file definition from an existing .CSV file.
- ♦ You used Clarion's Database Manager utility to examine the contents of the .CSV file.
- ♦ You used Clarion's Database Manager utility to generate code to create a data file conversion program.
- ♦ You compiled and executed a data file conversion program to import data from a .CSV file into a TopSpeed file.

Now you've converted some valuable existing data to the TopSpeed file format so your Clarion applications can use it. In the next chapter, we will begin building an application "from scratch" using the Application Generator.

6 – STARTING THE APPLICATION

Using the Application Generator

With the Data Dictionary complete, you now can use the Application Generator to create your application. This chapter shows you:

- How to create the .APP file, which stores all your work for the project.
- How to define the first (Main) procedure to create an MDI application frame, and how to call procedures from the application's menu.

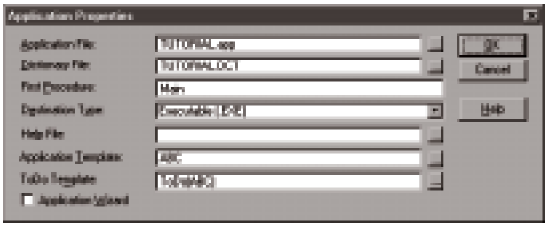
Starting Point:
Just the Clarion environment should be open.

Creating the .APP File

1. **Choose** **File ➤ New ➤ Application**.
2. **Select** *Application (*.app)* from the **Save as type** dropdown list.
3. **Select** the *C:\CLARION5\TUTORIAL* directory.
4. **Type** *TUTORIAL* in the **File name** field.
5. **Clear** the **Use Quick Start** box, then **press** the **Save** button.

   The **Application Properties** dialog appears.
6. **Type** *TUTORIAL.DCT* in the **Dictionary** field.
7. **Clear** the **Application Wizard** **check** box.

   This Application Generator Tutorial will not use any Wizards so that it can demonstrate how to use all the other tools that Clarion provides.



Application File: TUTORIAL.app
Dictionary File: TUTORIAL.DCT
First Procedure: Main
Destination Type: Executable (.EXE)
Help File:
Application Template: ABC
ToDo Template: ToDo(ABC)
☐ Application Wizard

8. **Press** **OK** to close the **Application Properties** dialog.

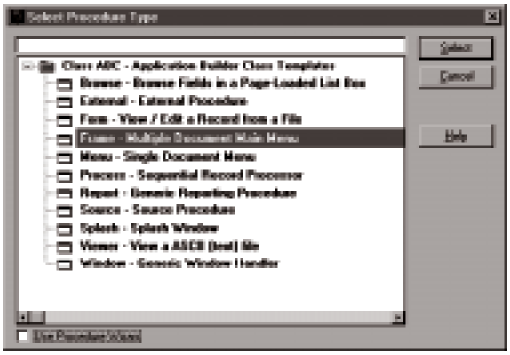
Creating the Main Procedure

The **Application Tree** dialog appears. This lists all the procedures for your application in a logical procedure call tree which provides a visual guide to show the order by which one procedure calls another. You previously saw it in the Quick Start tutorial.

The *Main* procedure is the starting point. The tutorial application will be an MDI (Multiple Document Interface) program. Therefore the natural starting point is to define the *Main* procedure using the **Frame** Procedure template to create an application frame.

Select the procedure type for Main

1. With *Main* highlighted in the **Application Tree** dialog, **press** the **Properties** button.
2. **Highlight** *Frame* in the **Select Procedure Type** dialog, **clear** the **Use Procedure Wizard check** box, then **press** the **Select** button.

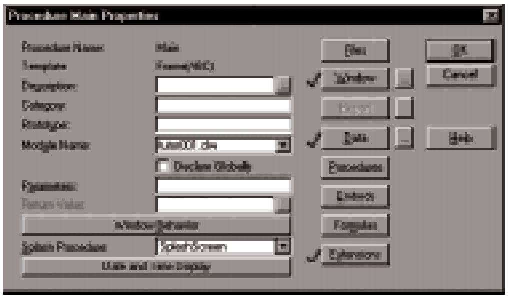


Class ABC - Application Builder Class Templates
- Browse - Browse Fields in a Page-Loaded List Box
- External - External Procedure
- Form - View / Edit a Record from a File
- Frame - Multiple Document Main Menu
- Menu - Single Document Menu
- Process - Sequential Record Processor
- Report - Generic Reporting Procedure
- Source - Source Procedure
- Splash - Splash Window
- Viewer - View a ASCII (text) file
- Window - Generic Window Handler

The **Procedure Properties** dialog appears. It defines the functionality and data structures for the procedure.

3. **Type** *SplashScreen* in the **Splash Procedure** field.

   This names the procedure containing an opening screen that will appear for just a brief period when the user first opens the application.



Procedure Name: Main
Template: Frame(ABC)
Description:
Category:
Prototype:
Module Name: tutor001.clw
Declare Globally
Parameters:
Return Value:
Splash Procedure: SplashScreen

Usually the first task when creating a procedure is to edit the main window. You can place controls, or if the procedure template has controls already, you can customize them.

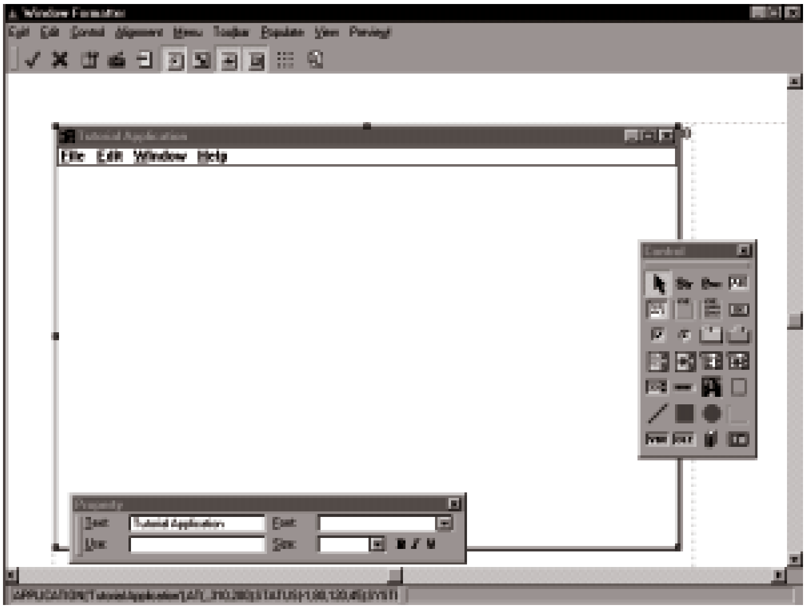
The application frame itself *never* has controls. Windows doesn't allow it. We will, however, customize the window caption (the text that appears on its title bar). Then we will add items to the predefined menu, which is also built into the *Frame* Procedure template, and create a toolbar for the application (a toolbar *can* have controls).

Edit the Main window

1. **Press** the **Window** button.

   The Window Formatter appears. Here are all the tools which allow you to visually edit the window and its controls.

2. **Choose View ➤ Show Propertybox** to display the **Property** toolbox (if it is not already present).



You'll notice under the **View** menu that there are several toolboxes available—all of these are fully dockable and resizable. This means you can configure your workspace as you want. Because of this configurability, throughout the rest of this tutorial some screen shots taken within the Window Formatter may not appear the same as on your computer.

There is one thing to note about dockability: if the toolbox is floating and you drag it by its title bar, it won't dock. This is a feature, because it allows you to float your toolboxes outside the workspace, if you choose. To dock, you must drag the toolbox using any open space in the toolbox or the “grab handle” (the double lines along the left or top edge). When the shadow outline you drag changes from a thick outline to a thin one you may drop the toolbox and it will dock where you’ve placed it. To undock a docked toolbox, just drag it with the “grab handle.”

You also have the same ability to create dockable toolboxes in your Clarion applications (see the DOCK attribute in the *Language Reference*). When you do, they will behave in this same manner.

3. CLICK on the sample window’s title bar to make sure it has focus (that is, the red “handles” are on the whole window).

4. **Type** *Tutorial Application* in the **Text** field of the **Property** toolbox, then **press** TAB.

   This updates the caption bar text in the sample window. Be sure that handles appear inside the sample application frame window when you execute this step.

Editing the Menu

From the Window Formatter’s menu, you can call the Menu Editor, which allows you to add or edit menu items for the application frame window. As you add each menu item, you can select the **Actions** tab to name the procedure to call when the user chooses that menu item.

For each new procedure you name for the menu to call, the Application Generator automatically adds a “ToDo” procedure to the Application Tree. You can then define that procedure’s functionality, just as you are now defining the application frame procedure’s functionality.

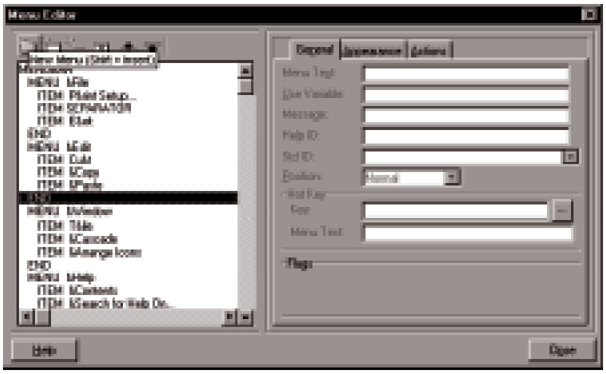
When the Application Generator generates the source code for your application, it automatically starts a new execution thread for each procedure you call from the main menu (this is required for an MDI application).

Add menu items

1. From the Window Formatter menu, **choose** **Menu ➤ Edit Menu** (or just DOUBLE-CLICK on the menu action bar in your window design).

   The Menu Editor appears. It displays the menu in hierarchical form in a list box at the left. The fields at the right allow you to name and customize the dropdown menus and menu items.

   This template already provides you with a “standard” menu. It contains basic window commands such as an **Exit** command on a **File** menu, the standard editing **Cut**, **Copy**, and **Paste** commands, and the standard window management commands commonly found in an MDI application.



```
MENUBAR
  MENU &File
    ITEM P&rint Setup...
    ITEM SEPARATOR
    ITEM E&xit
  END
  MENU &Edit
    ITEM Cu&t
    ITEM &Copy
    ITEM &Paste
  END
  MENU &Window
    ITEM T&ile
    ITEM &Cascade
    ITEM &Arrange Icons
  END
  MENU &Help
    ITEM &Contents
    ITEM &Search for Help On...
```

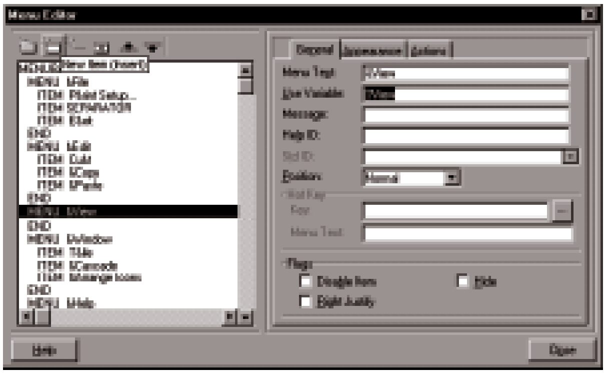
2. **Highlight** the second END statement (see the illustration above).

   The Menu Editor inserts new items immediately *below* the currently highlighted selection. The menu you'll add will be called **View**. It will contain three items: **Products**, **Customers**, and **Orders**. It will appear on the menu bar just before the **Window** menu.

3. **Press** the **New Menu** button in the top-left corner (or **press** SHIFT+INSERT).

   This inserts a new MENU statement, and its corresponding END statement.

4. **Type** *&View* in the **Menu Text** field then **press** TAB.



Menu Text: &View
Use Variable: ?View

```
MENUBAR
  MENU &File
    ITEM P&rint Setup...
    ITEM SEPARATOR
    ITEM E&xit
  END
  MENU &Edit
    ITEM Cu&t
    ITEM &Copy
    ITEM &Paste
  END
  MENU &View
  END
  MENU &Window
    ITEM T&ile
    ITEM &Cascade
    ITEM &Arrange Icons
  END
  MENU &Help
```

This defines the text that appears on the menu to the end user. The ampersand (&) indicates that the following character (V) has an underlined and provides keyboard access (the user can press ALT+V to drop down this menu).

Add the first menu item

1. **Press** the **New Item** button (or **press** INSERT).

   This updates the list on the left side of the dialog, changing the text of the menu you just added to "&View." It adds a new menu ITEM—a command on the dropdown menu—under &View, and before the END statement that goes with the &View menu.

2. **Type** *&Customers* in the **Menu Text** field then **press** TAB.

   ?ViewCustomers appears in the **Use Variable** field. This is an equate for the menu item so code statements can reference it. The leading question mark (?) indicates it is a field equate label (see the *Language Reference*).

3. **Select** the **Actions** tab.

   The prompts allow you to name a procedure to execute when the end user selects the **View ➤ Customers** menu item.

4. **Choose** **Call a Procedure** from the **When Pressed** dropdown box.

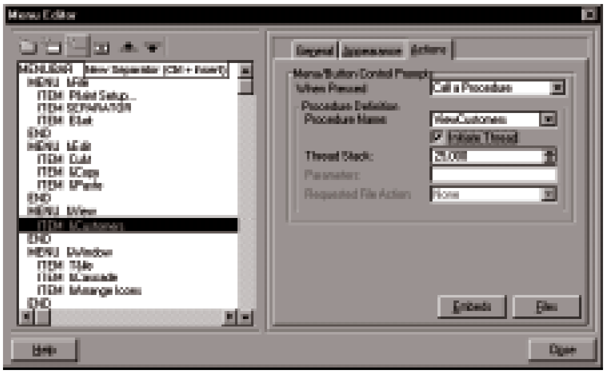
   New prompts appear to allow you to name the procedure to call and choose options.

5. **Type** *ViewCustomers* in the **Procedure Name** field.

   This names the "ToDo" procedure for the Application Tree.

6. **Check** the **Initiate Thread** box.

   The *ViewCustomers* procedure will display an MDI "child" window, and you must always start a new execution thread for any MDI window called directly from the application frame. The **Thread Stack** field defaults to the minimum recommended value.



```
MENUBAR
  MENU &File
    ITEM Print Setup...
    ITEM SEPARATOR
    ITEM E&xit
  END
  MENU &Edit
    ITEM Cu&t
    ITEM &Copy
    ITEM &Paste
  END
  MENU &View
    ITEM &Customers
  END
  MENU &Window
    ITEM T&ile
    ITEM &Cascade
    ITEM &Arrange Icons
  END
```

Menu/Button Control Prompts
When Pressed: Call a Procedure
Procedure Definition
Procedure Name: ViewCustomers
✓ Initiate Thread
Thread Stack: 25000
Parameters:
Requested File Action: None

Add the second menu item

1. **Press** the **Item** button.

   This updates the list on the left side of the dialog, changing the text of the item you just added to "&Customers."

2. **Type** *&Products* in the **Menu Text** field and **press** TAB.

   ?ViewProducts appears in the **Use Variable** field.

Normally, the next step is to define the action for the menu item—what happens when the end user executes it from the menu. We'll skip over this step for now, for this menu item only. Later, you'll create a procedure by copying it, then attaching it to this menu, just to show you this capability of the Clarion environment.

Add the third menu item

1. **Press** the **Item** button.
2. **Type** *&Orders* in the **Menu Text** field and **press** TAB.

   ?ViewOrders appears in the **Use Variable** field.
3. **Select** the **Actions** tab.
4. **Choose** **Call a Procedure** from the **When Pressed** dropdown box.
5. **Type** *ViewOrders* in the **Procedure Name** field.
6. **Check** the **Initiate Thread** box.

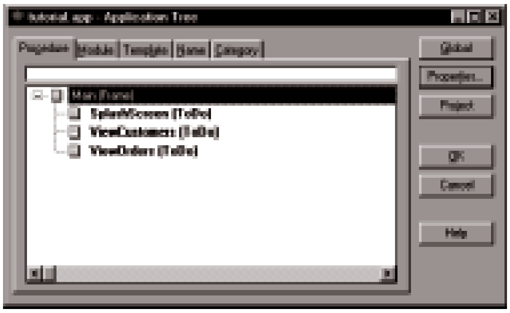
Close the Menu Editor and Window Formatter and save your work

1. **Press** the **Close** button to close the Menu Editor.

   This returns you to the Window Formatter.
2. **Choose** the **Exit!** menu selection and answer *Yes* when asked to save your window changes.

   This returns you to the **Procedure Properties** dialog.
3. **Press** the **OK** button to close the **Procedure Properties** dialog.

   This returns you to the **Application Tree** dialog. There are now three new procedures marked as "(ToDo)": ViewCustomers, ViewOrders, and SplashScreen. These were the procedures you named in the Menu Editor.

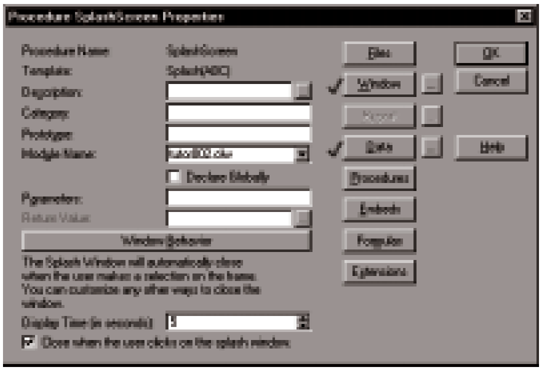
4. **Choose** **File ➤ Save**, or **press** the *Save* button on the toolbar.

Creating the SplashScreen Procedure

We named a Splash procedure, and now we'll create it.

1. **Highlight** the *SplashScreen (ToDo)* procedure and **press** the **Properties** button.
2. **Highlight** *Splash* in the **Select Procedure Type** dialog, **clear** the **Use Procedure Wizard check** box, then **press** the **Select** button.

   The **Procedure Properties** dialog appears. There's nothing we really have to do to this procedure for the tutorial except accept all the defaults.


Procedure SplashScreen Properties

Procedure Name: SplashScreen
Template: Splash(ABC)
Description:
Category:
Prototype:
Module Name: tutor002.clw
Declare Globally
Parameters:
Return Value:
The Splash Window will automatically close when the user makes a selection on the frame. You can customize any other ways to close the window.
Display Time (in seconds): 5
[✓] Close when the user clicks on the splash window.

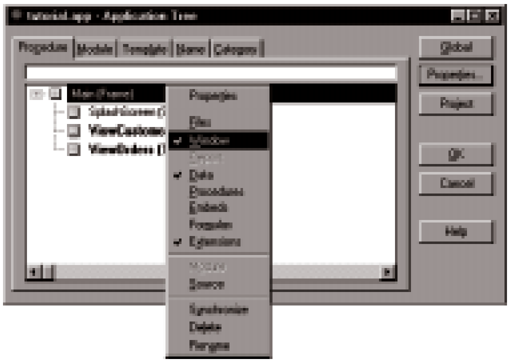
3. **Press** the **OK** button.

Adding an Application Toolbar

Call the Window Formatter and create the tool bar

1. **Highlight** the *Main* procedure.
2. RIGHT-CLICK to display the popup menu.

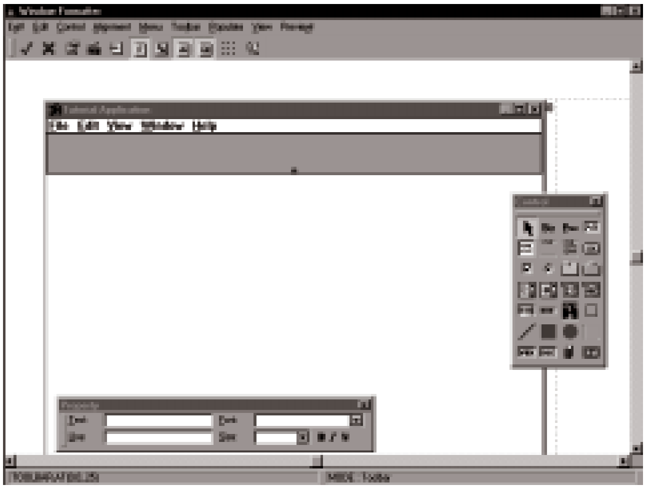
   Notice that this popup menu contains a set of menu selections that match the set of buttons down the right side of every **Procedure Properties** window. This popup menu provides you with direct access to all the Clarion tools that you use to modify existing procedures, so that you don't have to go through the **Procedure Properties** window every time.



3. **Choose** the **Window** menu item.

4. In the Window Formatter's menu, **choose Toolbar ➤ New Toolbar**.

   This adds the toolbar—always immediately below the menu—to the sample window. You can add any control to your toolbar by CLICKING on a tool icon in the floating **Controls** toolbox, then CLICKING in your toolbar.

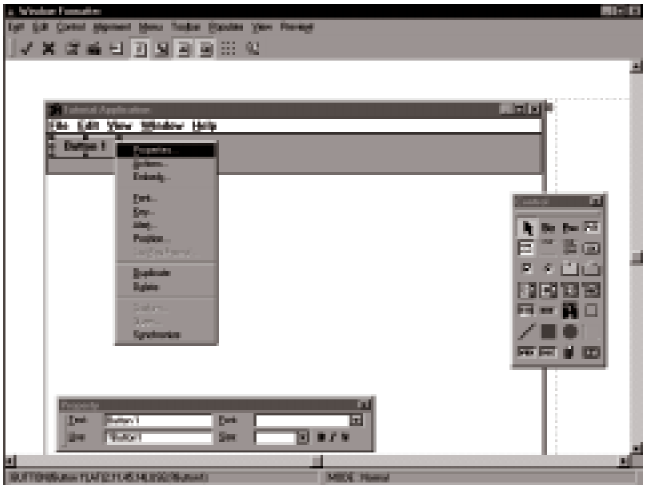


Place the first button

1. CLICK on the button tool (the one that looks like an "OK" button).

2. CLICK in the sample window toolbar area, just below the upper left corner.

3. RIGHT-CLICK the button you just placed, then **choose Properties** from the popup menu.



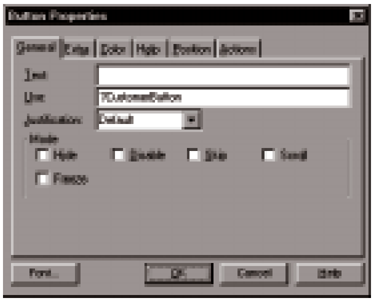
The **Button Properties** dialog appears.

4. **Clear** the **Text** field.

   We'll place images on these buttons, instead of text, to give the application a modern look.

5. **Type** *?CustomerButton* in the **Use** field.

   This is the field equate label for referencing the button in code. We included the word "button" for code readability.



6. **Select** the **Extra** tab.

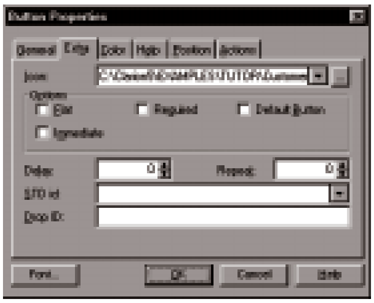
7. **Check** the **Flat** box.

8. Drop down the **Icon** list, scroll to the bottom and CLICK on **Select File...**.

   The **Select Icon File** dialog appears.

9. **Select** *GIF Files* from the **Files of type** droplist.

10. **Select** the *C:\CLARION5\EXAMPLES\TUTOR\CUSTOMER.GIF* file, then **press** the **Open** button.



11. **Select** the **Help** tab.
12. **Type** *Browse Customers* into the **Tip** field.

    This adds a tooltip to the button that will display whenever the mouse cursor hovers over the button.
13. **Select** the **Position** tab.
14. **Select** the **Fixed** radio buttons for both **Width** and **Height**.
15. **Set** the **Width** to *16* and **Height** to *14*.
16. **Select** the **Actions** tab.
17. **Choose** **Call a Procedure** from the **When Pressed** dropdown box.
18. **Choose** **ViewCustomers** from the **Procedure Name** dropdown box.

    This is the procedure name you typed for the **View ➤ Customers** menu item. Pressing the button will call the same procedure. Often, a command button on a toolbar serves as a quick way to execute a menu command.
19. **Check** the **Initiate Thread** box.
20. **Press** the **OK** button.

Place the second button

1. CLICK on the button tool.
2. CLICK in the sample window toolbar area, right next to the first button.

   A button appears, labelled “Button2.”
3. RIGHT-CLICK the button you just placed, then **choose** **Properties** from the popup menu.
4. **Clear** the **Text** field.
5. **Type** *?ProductsButton* in the **Use** field.
6. **Select** the **Extra** tab.
7. **Check** the **Flat** box.
8. Drop down the **Icon** list, scroll to the bottom and CLICK on **Select File...**.
9. **Select** *GIF Files* from the **Files of type** droplist.

10. **Select** the *C:\CLARION5\EXAMPLES\TUTOR\PRODUCTS.GIF* file, then **press** the **Open** button.
11. **Select** the **Help** tab.
12. **Type** *Browse Products* into the **Tip** field.
13. **Press** the **OK** button to close the **Button Properties** dialog.

    Normally you attach an action to the button at this point. Skip this step for now, for this button only. Later, we'll copy a procedure, then call it at the point in the generated source code which handles what to do when the end user presses the button to demonstrate using embed points.

Place the third button

1. CLICK on the button tool.
2. CLICK in the sample window toolbar area, right next to the second button.
3. RIGHT-CLICK the button you just placed, then **choose** **Properties** from the popup menu.
4. **Clear** the **Text** field.
5. **Type** *?OrdersButton* in the **Use** field.
6. **Select** the **Extra** tab.
7. **Check** the **Flat** box.
8. Drop down the **Icon** list, scroll to the bottom and CLICK on **Select File...**.
9. **Select** *GIF Files* from the **Files of type** droplist.
10. **Select** the *C:\CLARION5\EXAMPLES\TUTOR\ORDERS.GIF* file, then **press** the **Open** button.
11. **Select** the **Help** tab.
12. **Type** *Browse Orders* into the **Tip** field.
13. **Select** the **Actions** tab.
14. **Choose** **Call a Procedure** from the **When Pressed** dropdown list.
15. **Choose** *ViewOrders* from the **Procedure Name** dropdown list.

    This is the procedure name you typed for the **View ➤ Orders** menu item.
16. **Check** the **Initiate Thread** box.
17. **Press** the **OK** button.

Resize and align the buttons

The Window Formatter has a set of alignment tools that easily allow you to line up and resize your window controls.

1. With the *Orders* button still selected, **CTRL+CLICK** on the *Customers* button.

   This gives both buttons "handles" and the *Customers* button has the Red handles that indicate it has focus.

   CTRL+CLICK is the "multi-select" keystroke that allows you to perform actions on several controls at once. Once multiple controls are selected, you can move them all by DRAGGING on any one of the selected controls, or you can use any of the **Alignment** menu's tools on the entire group.

2. With both buttons still selected, **CTRL+CLICK** on the *Products* button.

   Now all three buttons have "handles" and the *Products* button has the Red handles that indicate it has focus and is the "key control" for the alignment actions.

3. **RIGHT-CLICK** and **choose** **Align Top** from the popup menu.

   When you have multiple controls selected, RIGHT-CLICK displays an alignment popup menu instead of a single control's popup menu. This action aligns all three buttons with the top of the *Products* button.

4. **RIGHT-CLICK** and **choose** **Spread Horizontally** from the popup menu.

   This spaces all three buttons apart equally. Make sure they're close together, so there's room for what's coming next!

Add the Browse Control buttons

1. **Choose** **Populate ➤ Control Template...** (or **CLICK** on the Control Template tool—the one at the bottom-right corner of the **Controls** toolbox).

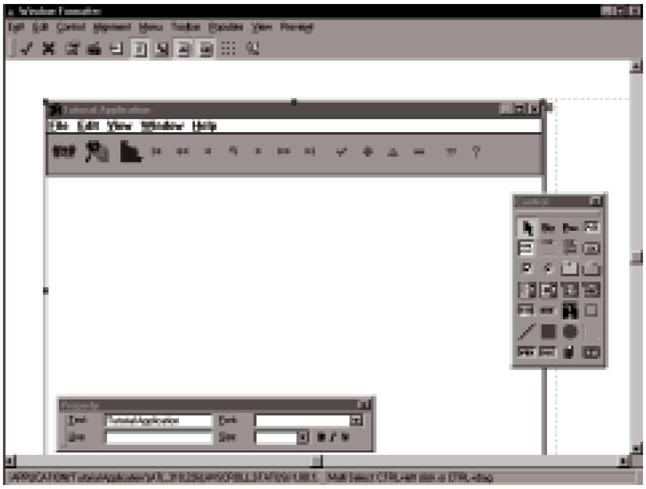
   The **Select Control Template** dialog appears.

2. **Highlight** the *FrameBrowseControl* template, then **press** the **Select** button.

3. **CLICK** in the sample window's toolbar, just right of the *Orders* button.

   The thirteen Browse control buttons appear on the toolbar. You've already seen these buttons on the applications you created in the *Quick Start Tutorial* chapter. These buttons can control the scrolling and update procedure call behavior of the Browse procedures you'll create (later in this tutorial).

   Your screen design should now look similar to this:


Window Formatter
Exit Edit Control Alignment Menu Toolbar Populate View Preview
Tutorial Application
File Edit View Window Help
Property

Close the Window Formatter and save your work

1. **Choose** the **Exit!** menu selection and answer *Yes* when asked to save your window changes.

   This returns you directly to the **Application Tree** dialog. It still contains the same two procedures marked as “(ToDo)”: *ViewCustomers* and *ViewOrders*.

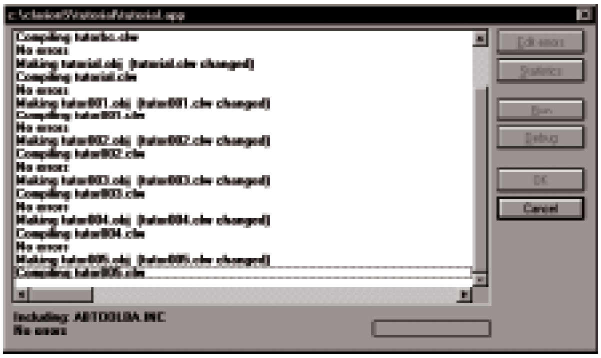
2. **Choose File ➤ Save**, or **press** the *Save* button on the toolbar.

Testing an Application under Development

1. With the **Application Tree** dialog open, **choose Project ➤ Run**, or **press** the *Run* button on the tool bar.

   The Application Generator generates the source code, displaying its progress in a message window, procedure by procedure.

   Next, the Make window appears, showing you the progress of the build as the compiler and linker do their work.

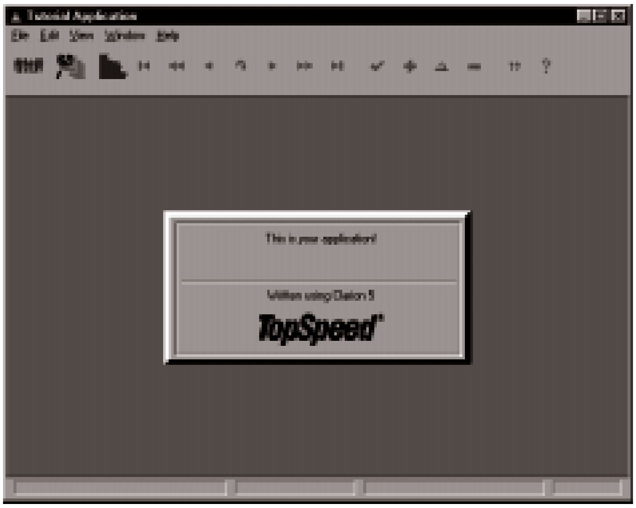


c:\clarion5\tutorial\tutorial.app

```
Compiling tutorbc.clw
No errors
Making tutorial.obj  (tutorial.clw changed)
Compiling tutorial.clw
No errors
Making tutor001.obj  (tutor001.clw changed)
Compiling tutor001.clw
No errors
Making tutor002.obj  (tutor002.clw changed)
Compiling tutor002.clw
No errors
Making tutor003.obj  (tutor003.clw changed)
Compiling tutor003.clw
No errors
Making tutor004.obj  (tutor004.clw changed)
Compiling tutor004.clw
No errors
Making tutor005.obj  (tutor005.clw changed)
Compiling tutor005.clw
```

```
Including: ABTOOLBA.INC
No errors
```

Then your Application window appears. It should look something like this when it first comes up with the splash screen:



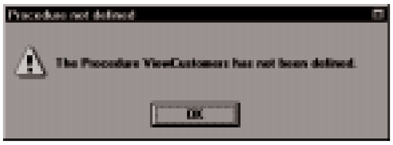
This is your application!

Written using Clarion 5

TopSpeed

2. **Press** one of the buttons on the toolbar, or **choose** one of the items on the **View** menu.

   The following message box appears:



Procedure not defined

The Procedure ViewCustomers has not been defined.

   This capability allows you to incrementally test your application, whether you have designed all the procedures or not.

   You'll fill in their functionality, starting in the next chapter.

3. **Press** the **OK** button to close the message box.
4. **Choose** **File ➤ Exit** to close the Tutorial application.

   Throughout the rest of this tutorial, feel free to *Make and Run* the developing application whenever the tutorial instructs you to save the file.

Look at the Generated Source Code

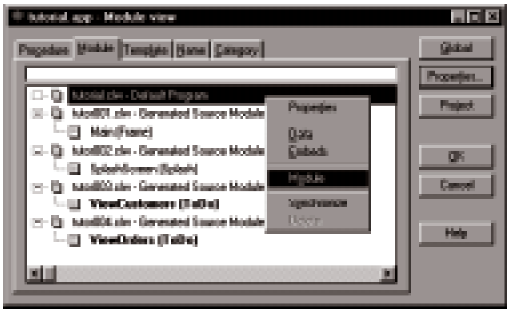
Let's take a quick look at what the Application Generator has done for you. The whole purpose of the Application Generator (and its Templates) is to write Clarion language source code for you. There is no "magic" to what the Clarion toolset does to create applications—it all goes back to the Clarion programming language.

1. With the **Application Tree** dialog open, CLICK on the **Module** tab.

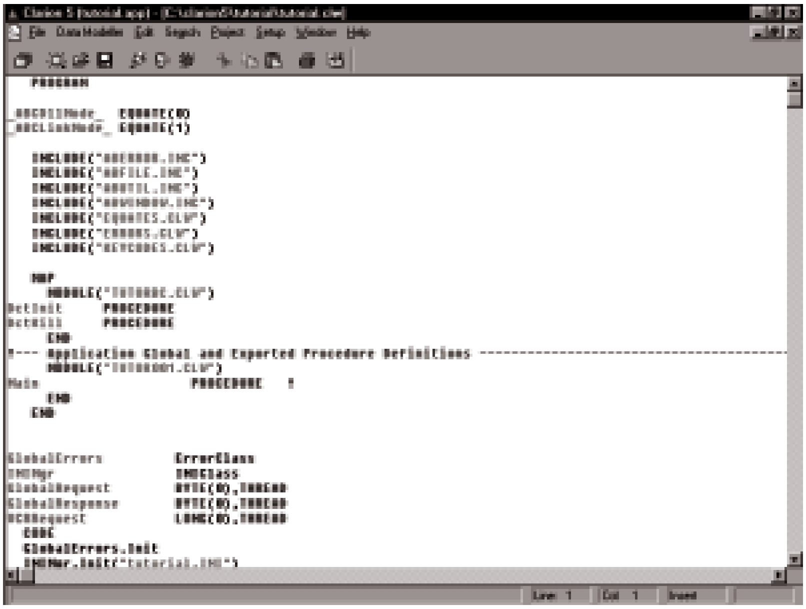
   This changes your view of the application from the logical procedure call tree to the actual source code modules generated for the application.

2. **Highlight** the *TUTORIAL.CLW* module, RIGHT-CLICK to display the popup menu then CLICK on **Module**.

   This takes you right into the Text Editor, looking at the last source code you generated (the last time you pressed the *Run* button). Any changes you made since the last time you generated code will not be visible here.


tutorial.app - Module view
Procedure
Module
Template
Name
Category
tutorial.clw - Default Program
tutor001.clw - Generated Source Module
Main (Frame)
tutor002.clw - Generated Source Module
SplashScreen (Splash)
tutor003.clw - Generated Source Module
ViewCustomers (ToDo)
tutor004.clw - Generated Source Module
ViewOrders (ToDo)
Properties
Data
Embeds
Module
Synchronize
Delete
Global
Properties...
Project
OK
Cancel
Help

TUTORIAL.CLW file is the main program module for this application, containing the Global data declarations and code. Don't be intimidated looking at all this code. After you've finished this tutorial, you can go on to the *Introduction to the Clarion Language* tutorial at the end of this book and become more familiar with the Clarion Language (it's actually very straight-forward).



```
   PROGRAM

_ABCDllMode_  EQUATE(0)
_ABCLinkMode_ EQUATE(1)

   INCLUDE('ABERROR.INC')
   INCLUDE('ABFILE.INC')
   INCLUDE('ABUTIL.INC')
   INCLUDE('ABWINDOW.INC')
   INCLUDE('EQUATES.CLW')
   INCLUDE('ERRORS.CLW')
   INCLUDE('KEYCODES.CLW')

   MAP
     MODULE('TUTOR_BC.CLW')
DctInit     PROCEDURE
DctKill     PROCEDURE
     END
!--- Application Global and Exported Procedure Definitions ----------------------------------------
     MODULE('TUTOR001.CLW')
Main                   PROCEDURE   !
     END
   END


GlobalErrors         ErrorClass
INIMgr               INIClass
GlobalRequest        BYTE(0),THREAD
GlobalResponse       BYTE(0),THREAD
VCRRequest           LONG(0),THREAD
  CODE
  GlobalErrors.Init
  INIMgr.Init('tutorial.INI')
```

3. When you have finished looking at the code, **choose** **File ➤ Close** to exit the Text Editor and return to the Application Generator.

   Be sure NOT to choose **File ➤ Exit**, otherwise you'll exit Clarion for Windows completely.

4. Now CLICK on the **Procedure** tab.

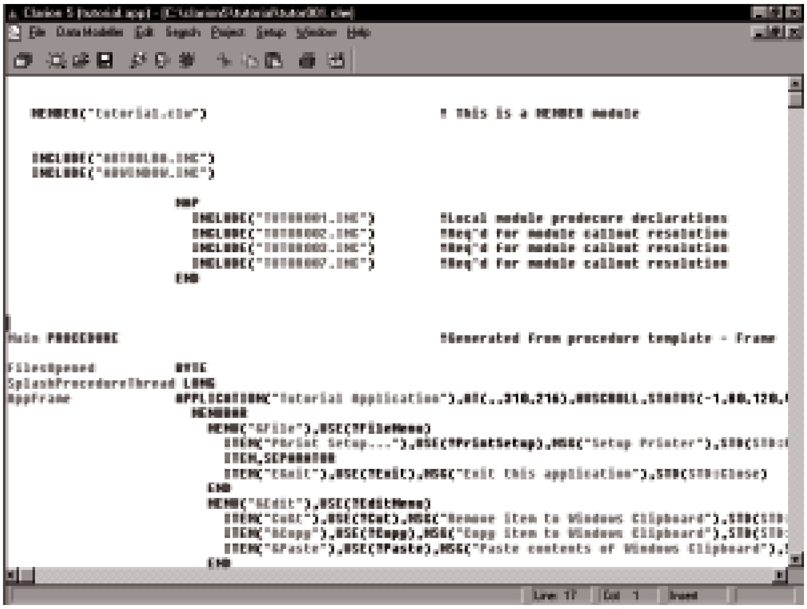
   This changes your view of the application back to the logical procedure call tree.

5. **Highlight** the *Main (Frame)* procedure, RIGHT-CLICK to display the popup menu, then CLICK on **Module**.

   This takes you into the Text Editor again, looking at the last source code you generated for the *Main* procedure. Again, any changes you made in the Application Generator since the last time you generated code will not show up in this code.

   You may have noticed that right below the **Module** selection was another called **Source**. Do not confuse these two, they do very different things. We will demonstrate the **Source** selection later in this tutorial.

   If you do make any changes to this code, you actually can compile and run the program to see what effect the changes make, however, **your changes will be lost the next time you generate source code.** Therefore, it is not a good idea to make any changes here.


Clarion 5 [tutorial.app] - [C:\clarion5\tutorial\tutor001.clw]
File Data Modeller Edit Search Project Setup Window Help
Line: 17 Col: 1 Insert

```
  MEMBER('tutorial.clw')                             ! This is a MEMBER module


  INCLUDE('ABTOOLBA.INC')
  INCLUDE('ABWINDOW.INC')

                     MAP
                       INCLUDE('TUTOR001.INC')        !Local module procedure declarations
                       INCLUDE('TUTOR002.INC')        !Req'd for module callout resolution
                       INCLUDE('TUTOR003.INC')        !Req'd for module callout resolution
                       INCLUDE('TUTOR007.INC')        !Req'd for module callout resolution
                     END


Main PROCEDURE                                       !Generated from procedure template - Frame

FilesOpened          BYTE
SplashProcedureThread LONG
AppFrame             APPLICATION('Tutorial Application'),AT(,,310,216),HVSCROLL,STATUS(-1,80,120,
                       MENUBAR
                         MENU('&File'),USE(?FileMenu)
                           ITEM('P&rint Setup...'),USE(?PrintSetup),MSG('Setup Printer'),STD(STD:
                           ITEM,SEPARATOR
                           ITEM('E&xit'),USE(?Exit),MSG('Exit this application'),STD(STD:Close)
                         END
                         MENU('&Edit'),USE(?EditMenu)
                           ITEM('Cu&t'),USE(?Cut),MSG('Remove item to Windows Clipboard'),STD(STD
                           ITEM('&Copy'),USE(?Copy),MSG('Copy item to Windows Clipboard'),STD(STD:
                           ITEM('&Paste'),USE(?Paste),MSG('Paste contents of Windows Clipboard'),
                         END
```

6. When you have finished looking at the code, **choose** **File ➤ Close** to exit the Text Editor and return to the Application Generator.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- You created a new .APP file—without using a wizard.
- You created an application Frame procedure—without using a wizard.
- You created a menu in your application frame.
- You created a splash screen procedure for your application.
- You created a toolbar under your application's main menu and placed iconized, flat buttons on it.
- You used Clarion's resize and alignment tools to adjust your screen design.
- You used a Control Template to populate your toolbar with a set of standard navigation buttons.
- You compiled and ran your work-in-progress to test its functionality.

In the next lesson, we'll add a Browse procedure to the application.

7 – CREATING A BROWSE

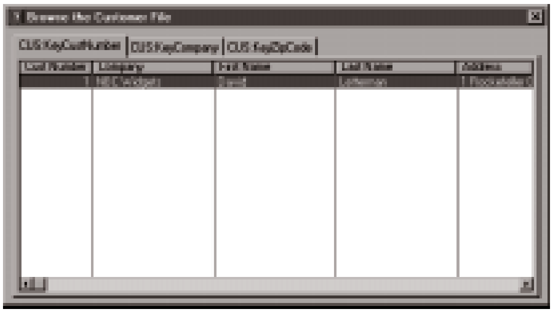
Creating a Browse Window

In this chapter, you'll create a browse window similar to the one created for you by the Quick Start Wizard. The Application Generator uses the same templates, which generate the same basic code—but doing it this way, you'll have a chance to "do it from scratch." This shows you just how much the Wizards do for you, and how you can do it all yourself, too. You'll start with the Customer browse window.

Starting Point:
The TUTORIAL.APP file should be open.

Creating the Customer Browse Window

You recall that the Quick Start Wizard created a window for the *Customer* file Browse procedure, that looked like this:



Now you'll create a similar one using the *Browse* Procedure template:

Select the procedure type for the ViewCustomers procedure

1. DOUBLE-CLICK on *ViewCustomers* in the **Application Tree**.
2. **Highlight** the *Browse* Procedure template in the **Select Procedure Type** dialog, **clear** the **Use Procedure Wizard** **check** box, then **press** the **Select** button.

   The **Procedure Properties** dialog appears.

Make the window resizable

1. In the **Procedure Properties** dialog, **press** the **Extensions** button.
2. In the **Extensions and Control Templates** dialog, **press** the **Insert** button.
3. **Highlight** *WindowResize* in the **Select Extension** dialog, then **press** the **Select** button.

   This Extension template generates code to automatically handle resizing and re-positioning all the controls in the window when the user resizes the window, either by resizing the window frame, or by pressing the Maximize/Restore buttons.
4. **Press** THE **OK** button to close the **Extensions and Control Templates** dialog.

Edit the Browse procedure

1. In the **Procedure Properties** dialog, **press** the **Window** button.
2. RIGHT-CLICK in the window's title bar and **choose** **Properties** from the popup menu.
3. In the **Window Properties** dialog, **type** *Browse Customers* in the **Text** field.
4. **Select** *Resizable* from the **Frame Type** drop-down list.
5. **Select** the **Extra** tab and **check** the **Maximize Box** box.

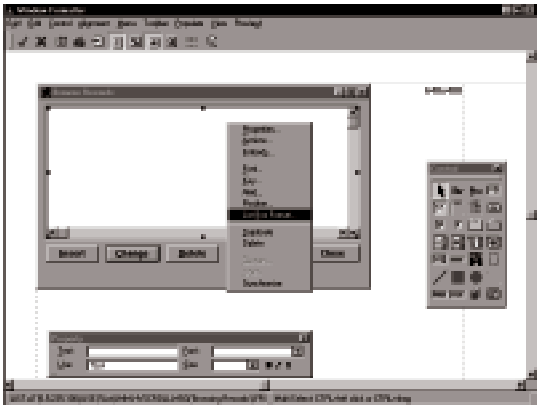
   These last two steps allow the user to resize the window at runtime.
6. **Press** THE **OK** button to close the **Window Properties** dialog.

Populating and Formatting a List Box Control

The List Box Formatter allows you to format the data in the list.

Prepare to format the list box

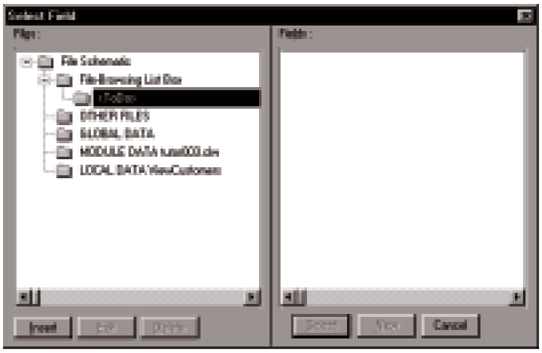
1. RIGHT-CLICK on the list box in the window, then **choose** **List Box Format...** from the popup menu.



The **Select Field** dialog appears. This provides access to the files defined in the data dictionary. The **Files** list displays all the files selected for use in this proceure in a hierarchical arrangement (the *File Schematic*), which includes the browse list box control.

Select the file and fields to place in the browse list box control

1. **Highlight** the “ToDo” item below the **File-Browsing List Box** and **press** the **Insert** button.



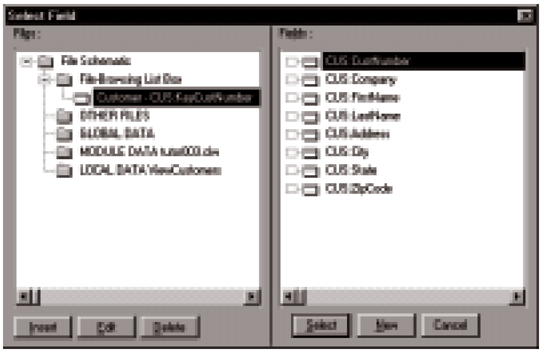
2. **Highlight** the *Customer* file in the **Insert File** dialog, then **press** the **Select** button.

   This adds the file to the File Schematic in the **Select Field** dialog, which now lists the file and its fields.

3. **Press** the **Edit** button.

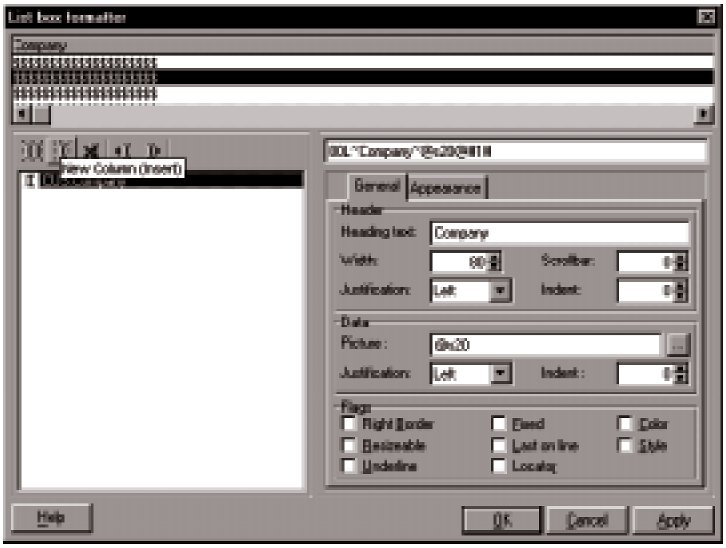
4. **Highlight** *KeyCustNumber* in the **Change Access Key** dialog and **press** the **Select** button.

   This is important, because it sets the display order for the records in the list. If you don’t specify a key, the records appear in (sort of) whatever order they were added to the file (also called “Record Order”).



5. **Highlight** *CUS:Company* in the **Fields** list, then **press** the **Select** button.

   This brings you into the List Box Formatter with the selected field added to the list. The tabs on the right allow you to format the appearance of the field highlighted in the list on the left.



80L~Company~@s20@#1#

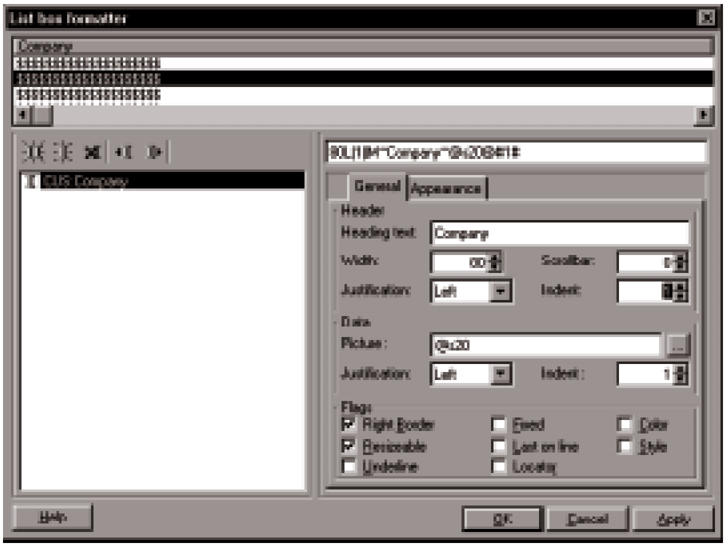
Heading text: Company
Width: 80
Scrollbar: 0
Justification: Left
Indent: 0
Picture: @s20
Justification: Left
Indent: 0

Apply special formatting to the first field

1. On the **General** tab, **check** the **Right Border** and **Resizeable** boxes.

   This adds a resizable right vertical border to the field at runtime.

2. CLICK once on the up arrow of both of the **Indent** spin boxes to slightly indent the heading text and the displayed data.



80L(1)|M~Company~@s20@#1#

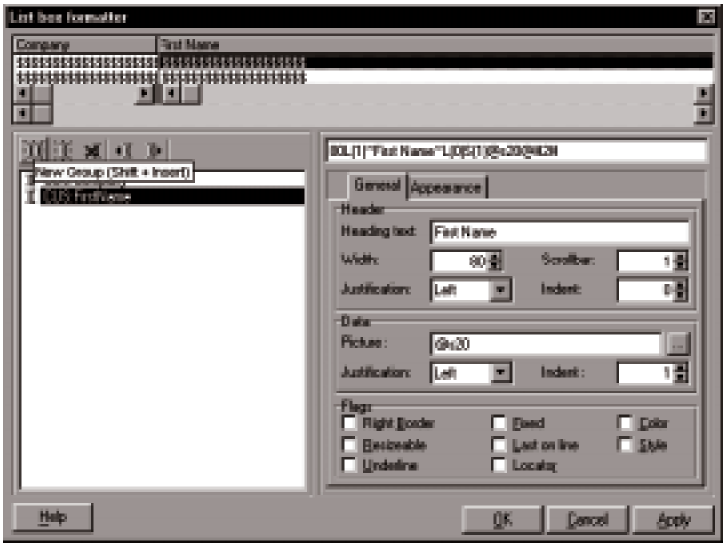
Heading text: Company
Width: 80
Scrollbar: 0
Justification: Left
Indent: 1
Picture: @s20
Justification: Left
Indent: 1
Flags: [✓ Right Border] [✓ Resizeable] Underline Fixed Last on line Locator Color Style

Populate the second field

1. **Press** the **New Column** button.
2. **Highlight** *CUS:FirstName* in the **Fields** list, and **press** the **Select** button.

3. **Clear** the **Right Border** and **Resizeable check** boxes.

   The List Box Formatter automatically “carries forward” these formatting options from the last field you added, making it very simple to add multiple fields with similar formatting options. In this case, clearing these check boxes deletes the column divider between this and the next column, which will be the *LastName* field.



#COL[1]|"First Name"L(0)|S(1)@s20@80L2N

Header
Heading text: First Name
Width: 80
Scrollbar: 1
Justification: Left
Indent: 0

Data
Picture: @s20
Justification: Left
Indent: 1

Flags
☐ Right Border ☐ Fixed ☐ Color
☐ Resizeable ☐ Last on line ☐ Style
☐ Underline ☐ Locator

Populate the third field

1. **Press** the **New Column** button.
2. **Highlight** *CUS:LastName* in the **Fields** list, then **press** the **Select** button.
3. **Check** the **Right Border** and **Resizeable** boxes.

   This once again adds the resizable column divider between this and the next column.

Populate the fourth field

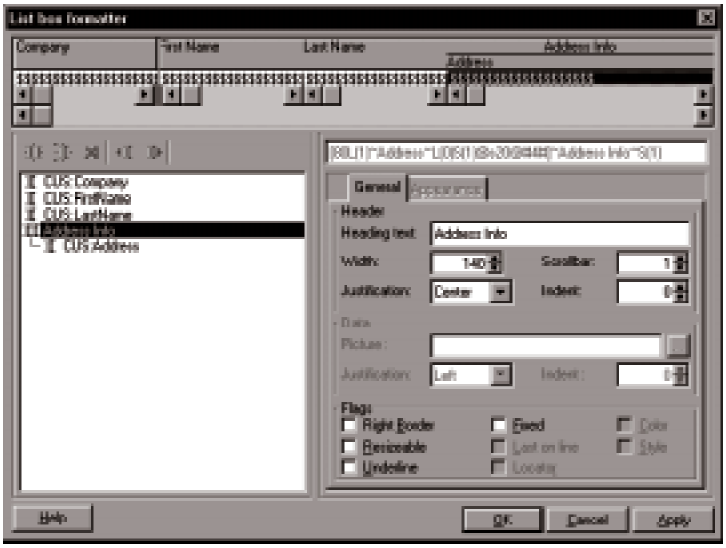
1. **Press** the **New Column** button.
2. **Highlight** *CUS:Address* in the **Fields** list, then **press** the **Select** button.

Group some fields

1. **Press** the **New Group** button.
2. **Press** the UP ARROW to highlight the new group.

   By creating a new group, in which you’ll place the address information, you can add a group header. This appears *above* the field headers, and visually links the data in the columns beneath. Notice that, as you add fields and make changes you can see the effects of your changes in the sample list box atthe top of the **List Box Formatter** dialog.

3. **Type** *Address Info* in the **Heading Text** field.



CUS:Company
CUS:FirstName
CUS:LastName
Address Info
CUS:Address

[80L(1)|~Address~L(0)S(1)|80L(20)@40#|~Address Info~S(1)]

Header
Heading text: Address Info
Width: 140
Scrollbar: 1
Justification: Center
Indent: 0

Data
Picture:
Justification: Left
Indent: 0

Flags
☐ Right Border
☐ Resizeable
☐ Underline
☐ Fixed
☐ Last on line
☐ Locator
☐ Color
☐ Style

This provides the text for the group header. Any fields appearing to the right of this one will be included in the group, until you define another group.

As you add fields, the List Box Formatter continually updates its sample window (at the top) to show you how your list will appear.

4. **Highlight** the *CUS:Address* field.

Populate the fifth field

1. **Press** the **New Column** button.
2. **Highlight** *CUS:City* in the **Fields** list, and **press** the **Select** button.

Populate the sixth field

1. **Press** the **New Column** button.
2. **Highlight** *CUS:State* in the **Fields** list, and **press** the **Select** button.

Populate the seventh field, and exit the List Box Formatter

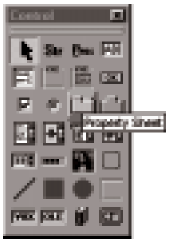
1. **Press** the **New Column** button.
2. **Highlight** *CUS:ZipCode* in the **Fields** list, and **press** the **Select** button.
3. **Press** the **OK** button to close the List Box Formatter.

Adding the Tabs

When the Quick Start Wizard created this procedure it had tab controls that changed the list's sort order depending on which tab was selected. Therefore, we'll add this functionality right now to show how easy it is to accomplish!

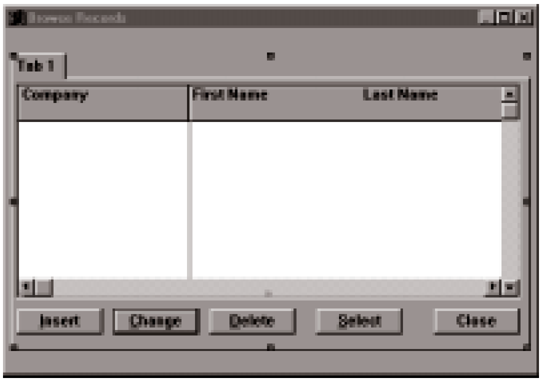
Add the Property Sheet and the first tab

1. CLICK on your window's title bar to place the red "handles" on your window design.
2. Place the mouse cursor directly over the middle handle on the top then DRAG it up to create some room at the top.
3. CLICK on the *Property Sheet* control in the **Controls** toolbox (it's the one that looks like several file folders).



4. CLICK above and to the left of the List box to place the property sheet and one Tab control.
5. DRAG the "red handle" at the bottom *left*-hand corner so that it appears just below and to the left of the Insert button.
6. DRAG the "red handle" at the bottom *right*-hand corner so that it appears just below and to the right of the Close button.

   This resizes the property sheet so that it *appears* as though the list box and buttons are on the tab. In fact, they are not, and we don't want them to be, since we want all these controls to be visible no matter which tab the user selects. Your window should now look something like this:

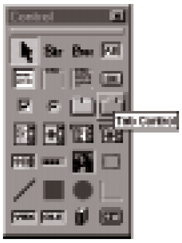


7. CLICK on the “Tab 1” text for the first tab in the sheet.
8. **Type** *KeyCustNumber* in the **Text** field of the **Property** toolbox, then **press** TAB.

   This changes the tab’s text. This tab text can be anything, but naming the key also names the sort order it will display.

Add the rest of the tabs

1. CLICK on the *Tab control* in the **Controls** toolbox (it’s the one that looks like a single file folder).



2. CLICK immediately to the right of the *KeyCustNumber* text to place the next Tab control.
3. **Type** *KeyCompany* in the **Text** field of the **Property** toolbox, then **press** TAB.
4. CLICK on the *Tab control* in the **Controls** toolbox.
5. CLICK immediately to the right of the *KeyCompany* tab to place the next Tab control.
6. **Type** *KeyZipCode* in the **Text** field of the **Property** toolbox, then **press** TAB.

Hiding the Buttons

When the Quick Start Wizard created this procedure it did not have Insert, Change, Delete, Select, and Close buttons—at least, you didn’t see them when you executed the resulting program! Actually, those buttons *were* all there, but they were hidden so the user would just use the toolbar buttons to update the file.

The “secret” here is that the toolbar buttons actually just tell hidden buttons in the Browse procedure to do what they normally do. Therefore, when you are designing a Browse procedure without using the Wizards, you do need to have the update buttons on the screen, but the user does not have to see them at runtime.

1. RIGHT-CLICK on the *Close* button in the sample window then CLICK on **Properties...** from the popup menu.
2. **Check** the **Hide** box, then **press** the **OK** button.

This adds the HIDE attribute to the control so you won't see it on screen at runtime. Of course, you can still see it in the Window Formatter.

3. **RIGHT-CLICK** on the **Select** button then **CLICK** on **Properties...**.
4. **Check** the **Hide** box, then **press** the **OK** button.
5. **RIGHT-CLICK** on the *Delete* button then **CLICK** on **Properties...**.
6. **Check** the **Hide** box, then **press** the **OK** button.
7. **RIGHT-CLICK** on the *Change* button then **CLICK** on **Properties...**.
8. **Check** the **Hide** box, then **press** the **OK** button.
9. **RIGHT-CLICK** on the *Insert* button then **CLICK** on **Properties...**.
10. **Check** the **Hide** box, then **press** the **OK** button.

Move the buttons and resize the list

There's no need to waste the space these buttons (which the user won't see) occupy on the window, so we'll move them out of the way.

1. **CLICK** on the *Close* button then **SHIFT+CLICK** and **DRAG** the button up into the list box.

   DRAGGING a control with the SHIFT key depressed allows you to move the control in only one direction; if you start moving it down it will only move up and down, but if you start moving it to either side it will only move side to side.

2. **CLICK** on the **Select** button then **CTRL+CLICK** on the *Insert*, *Change*, and *Delete* buttons to **select** them all, then **CLICK** and DRAG the buttons up into the list box.

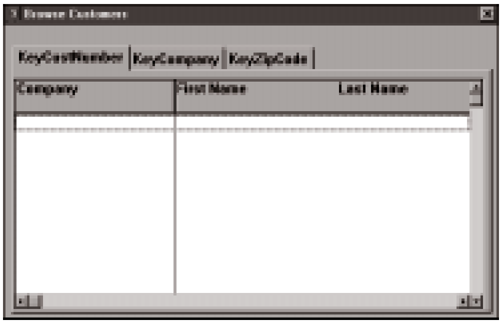
   DRAGGING multiple controls at once allows you to move the controls while maintaining the relative positions of the controls within the group. Now we'll use the space we just gained to make the list longer.

3. **CLICK** on the list box then DRAG its bottom-center handle down to make the list longer.

Testing the Customer Browse

The Window Formatter provides a test mode which allows you to preview just how your window looks on the desktop. Since this is an MDI window, it appears inside the Window Formatter frame.

1. **Choose** **Preview!** on the Window Formatter's menu bar.



2. **Press** ESC (or the **X** button) to return to the Window Formatter.

Setting the Sort Orders

Now that the tabs are there, we need to tell the list box what alternate sort orders to use and when.

1. **RIGHT-CLICK** on the list box then **choose** **Actions...** from the popup menu.

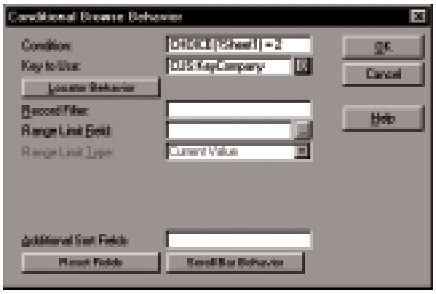
   The list box is actually a *BrowseBox* Control template that has been placed in the *Browse* Procedure template's default window design in the Template Registry (see the *Application Handbook* for more information on the Template Registry). This means that it has associated prompts which tell it how to populate the list and what actions to perform.

   The prompts that appear on the **Actions** tab come directly from the templates (in this case, the *BrowseBox* Control template). This is how you communicate to the templates exactly what code they need to generate to give you the behavior you ask for (and nothing else). These prompts, their meanings and uses, are all covered in the *User's Guide* and in the on-line help for each window in which they appear.

2. **Select** the **Conditional Behavior** tab.
3. **Press** the **Insert** button.
4. **Type** *CHOICE(?Sheet1) = 2* in the **Condition** field.

   This sets the condition under which the alternate sort order will be used. This expression uses the Clarion language CHOICE function (see the *Language Reference*) to detect when the user has selected the second tab on the sheet. The generated code will use this expression in a conditional statement that will change the sort order at runtime.

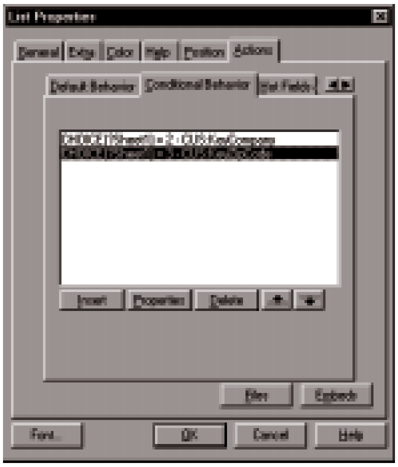
5. **Press** the ellipsis button (...) next to the **Key to Use** field.
6. **Highlight** *CUS:KeyCompany* then **press** the **Select** button on the **Select Key** dialog.



Condition: CHOICE(?Sheet1) = 2
Key to Use: CUS:KeyCompany
Range Limit Type: Current Value

Now, when the user selects the second tab, the *BrowseBox* Control template will generate code to switch to the key on the *Company* field. It doesn't need to know what to do for the first tab, because that always uses the Access Key we set in the File Schematic.

7. **Press** the **OK** button.
8. **Press** the **Insert** button.
9. **Type** *CHOICE(?Sheet1) = 3* in the **Condition** field.
10. **Press** the ellipsis button (...) next to the **Key to Use** field.
11. **Highlight** *CUS:KeyZipCode* then **press** the **Select** button on the **Select Key** dialog.
12. **Press** the **OK** button.



CHOICE(?Sheet1) = 2 - CUS:KeyCompany
CHOICE(?Sheet1) = 3 - CUS:KeyZipCode

13. **Press** the **OK** button to close the **List Properties** dialog.

Closing the Customer Browse

1. **Choose** **Exit!** on the Window Formatter's menu bar, and save your window changes when prompted to do so.
2. **Press** the **OK** button in the **Procedure Properties** dialog to close it.
3. **Choose** **File ➤ Save**, or **press** the *Save* button on the toolbar to save your work.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- ♦ You created a new Browse Procedure—without using a wizard.
- ♦ You added an Extension Template to automatically make the new procedure's window resizable.
- ♦ You used the List Box Formatter tool to design a scrolling list of records.
- ♦ You used the List Box Formatter tool to design a scrolling list of records.
- ♦ You added a Property Sheet and several Tabs to your screen design.
- ♦ You hid and moved buttons to provide a "cleaner" screen design.
- ♦ You used the Window Formatter's Preview mode to see your window design in action.
- ♦ You set dynamic sort orders for the user based on which Tab control they select.

Now that the first Browse procedure is complete, we'll go on and create its associated update Form procedure.

8 – CREATING AN UPDATE FORM

Creating an Update Procedure

In the last chapter, we formatted the Customer Browse procedure's list box and added tab controls to change the sort order. To finish the basic procedure, we name the Update procedure. This is the procedure that handles the action for the Insert, Change, and Delete buttons.

Starting Point:
The TUTORIAL.APP file should be open.

Add a "ToDo" procedure

1. **Highlight** *ViewCustomers* in the **Application Tree** dialog, then **press** the **Properties** button.

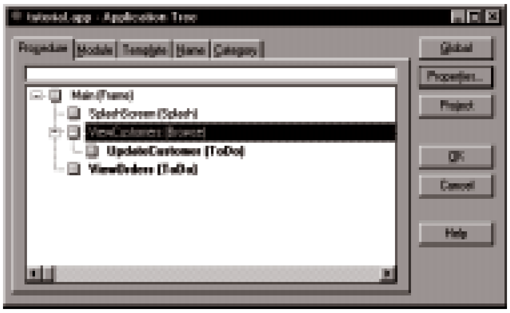
   The **Procedure Properties** dialog appears. There are actually three ways to get to this dialog (use the method that suits the way you work):

   - **Highlight** the procedure then press the **Properties** button.
   - DOUBLE-CLICK the procedure in the **Application Tree** dialog.
   - RIGHT-CLICK the procedure and select **Properties** from the popup menu.

2. **Type** *UpdateCustomer* in the **Update Procedure** entry box at the bottom of the **Procedure Properties** dialog.

   This names the procedure to update the records displayed in the browse. The new procedure appears in the **Application Tree** as a "ToDo."

3. **Press** the **OK** button to close the **Procedure Properties** dialog.



Notice that you didn't have to start a new execution thread for the update procedure. You want it to run on the same thread as the browse, so that the end user can't open a form window to change a record, then activate the browse window again, and open another form on the same record. In other words, you don't want an end user trying to change the same record twice at the same time!

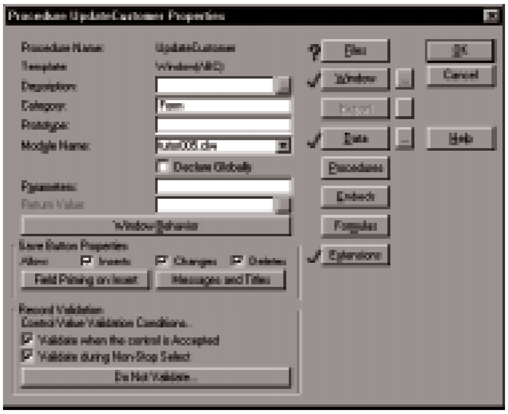
Creating the Update Form Procedure

The Update Procedure should use the Form Procedure template to create a procedure that the end user can use to maintain a record. It should provide a prompt and entry control for each field in the record.

Select the procedure type for UpdateCustomer.

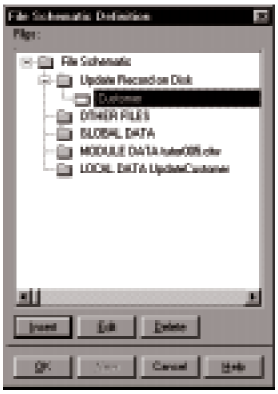
1. DOUBLE-CLICK on *UpdateCustomer* in the **Application Tree** dialog.
2. **Highlight** the *Form* Procedure template, **clear** the **Use Procedure Wizard check** box, then **press** the **Select** button.

   The **Procedure Properties** Window appears. Notice that this dialog looks different than the Splash, Frame, or Browse **Procedure Properties** dialogs looked, because the prompts at the left vary for each type of Procedure template. The *Application Handbook* and on-line help describes the customization options available on each **Procedure Properties** dialog.



3. **Press** the **Files** button to name the file the Form will update.

   The **File Schematic Definition** dialog appears.
4. **Highlight** the "ToDo" item below the **Update Record on Disk** and **press** the **Insert** button.
5. **Highlight** the *Customer* file in the **Insert File** dialog, then **press** the **Select** button.



6. **Press** the **OK** button to return to the **Procedure Properties** window.

Make the window resizable

1. In the **Procedure Properties** dialog, **press** the **Extensions** button.
2. In the **Extensions and Control Templates** dialog, **press** the **Insert** button.
3. **Highlight** *WindowResize* in the **Select Extension** dialog, then **press** the **Select** button.

   This Extension template generates code to automatically handle resizing and re-positioning all the controls in the window when the user resizes the window, either by resizing the window frame, or by pressing the Maximize/Restore buttons.
4. **Press** THE **OK** button to close the **Extensions and Control Templates** dialog.

Edit the Window Properties

1. In the **Procedure Properties** dialog, **press** the **Window** button.
2. RIGHT-CLICK in the window's title bar, then **choose** **Properties** from the popup menu.
3. **Select** *Resizable* from the **Frame Type** drop-down list.
4. **Select** the **Extra** tab and **check** the **Maximize Box** box.

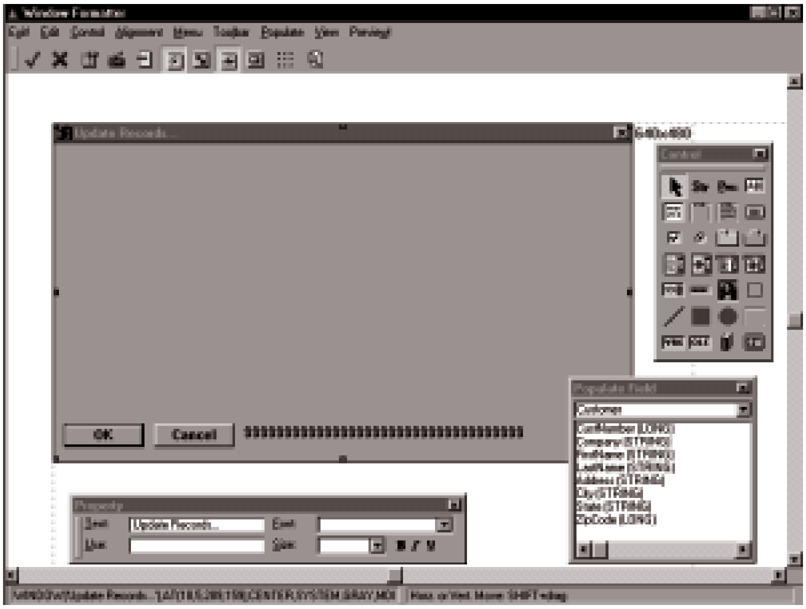
   These last two items allow the user to resize the window at runtime.
6. **Press** THE **OK** button to close the **Window Properties** dialog.

Populating the Fields

The default window design contains three fields for you already. The *OK* button will close the dialog, accepting the end user's input and writes the Customer file record to disk. The *Cancel* button closes the form without updating. The string field provides an action message to inform the end user what action they are taking on the record.

Placing the fields in a window is called *populating* it.

1. **Choose View ➤ Show Fieldbox.**



This displays a toolbox containing all the fields from all the files specified in your procedure's File Schematic. These fields are all ready to populate onto your window design.

2. CLICK on *CustNumber* in the **Populate Field** toolbox then move the cursor over the window design.

   The cursor changes to a crosshair and a "little book" which indicates the field comes from the data dictionary.

3. CLICK near the upper left corner of your window design.

   This places both the data entry control and its associated prompt. These controls default to whatever you specified in the data dictionary for the field.

4. DOUBLE-CLICK on *Company* in the **Populate Field** toolbox.

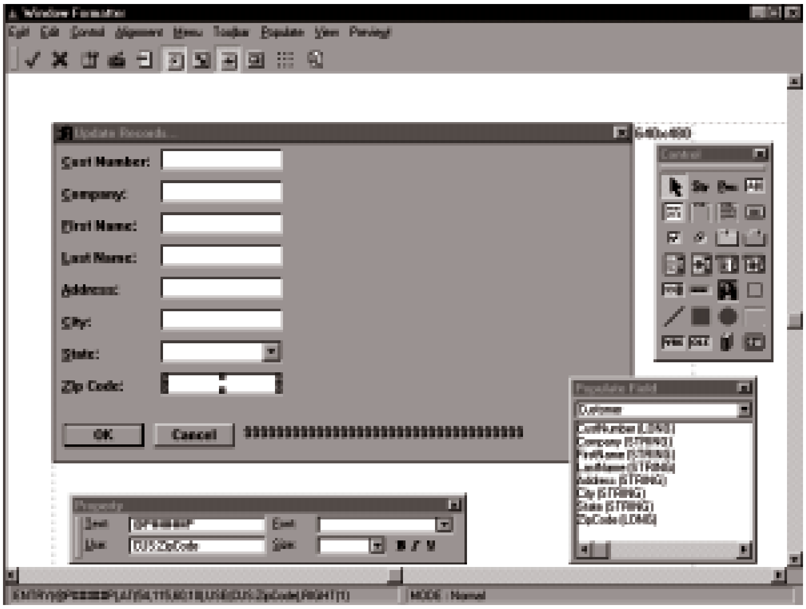
   DOUBLE-CLICK automatically places both the field and its prompt just below the first fields you placed. You could select and place each field as we did the first, but DOUBLE-CLICK is much faster.

5. DOUBLE-CLICK on *FirstName* in the **Populate Field** toolbox.
6. DOUBLE-CLICK on *LastName* in the **Populate Field** toolbox.
7. DOUBLE-CLICK on *Address* in the **Populate Field** toolbox.
8. DOUBLE-CLICK on *City* in the **Populate Field** toolbox.
9. DOUBLE-CLICK on *State* in the **Populate Field** toolbox.

This places the prompt and the drop list. We pre-defined this field as a LIST control with the DROP attribute in the data dictionary. Since it has a pre-defined set of valid entries, we don't have to format it.

10. **DOUBLE-CLICK** on *ZipCode* in the **Populate Field** toolbox.

The form window now looks something like this:



Moving and Aligning Fields

For a professional look, we need to move these fields around and align the sides and bottoms of all the fields in the screen.

Move the fields to their approximate positions.

1. **CLICK** on the *State* drop list.
2. **SHIFT+DRAG** the *State* droplist to the left, closer to its prompt.

   SHIFT+DRAG "contrains" the control's movement to the plane of it's first movement (either horizontal or vertical).
3. **CTRL+CLICK** on the *State* prompt.

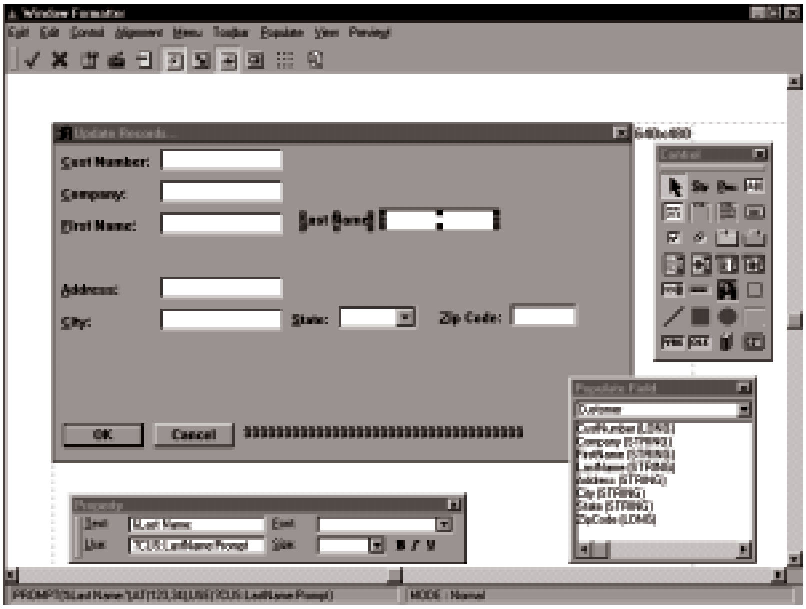
   When you CTRL+CLICK on a control, the previously selected control's handles turn blue while the newly selected control's handles are red. You now have two controls selected.
4. **Move** the *State* drop list and prompt (**CLICK** and **DRAG** on either of the two selected controls) up and to the right of the *City* controls.

Once multiple controls are selected, you can move them as a group the same way you would move one individual control.

5. CLICK on the *ZipCode* entry box.
6. SHIFT+DRAG the *ZipCode* entry box to the left, closer to its prompt.
7. CTRL+CLICK on the *ZipCode* prompt.
8. **Move** the *ZipCode* entry box and prompt up and to the right of the *City* and *State* controls.

   You may need to make the window a little wider to accomplish this.
9. CLICK on the *LastName* entry box.
10. SHIFT+DRAG the *LastName* entry box to the left, closer to its prompt.
11. CTRL+CLICK on the *LastName* prompt.
12. **Move** the *LastName* entry box and prompt up and to the right of the *FirstName* controls.

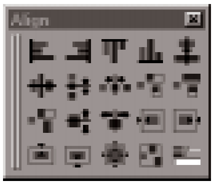
    Now your window should appear something like this:



Align the fields to their final positions

1. **Choose** **View ➤ Show Alignbox**.

   The Window Formatter has an **Alignbox** toolbox that contains the same set of alignment tools that are also available through the **Alignment** menu. Your **Alignbox** toolbox may not look exactly like the one pictured here. That's because you can reconfigure the shape of any of the floating toolboxes by DRAGGING any border—the toolbox will reshape itself to fit the new size. Try it and see how flexible it is. This allows you to configure your workspace the way *you* want it. Naturally, you also recall from previous lessons that all these toolboxes are dockable, too.



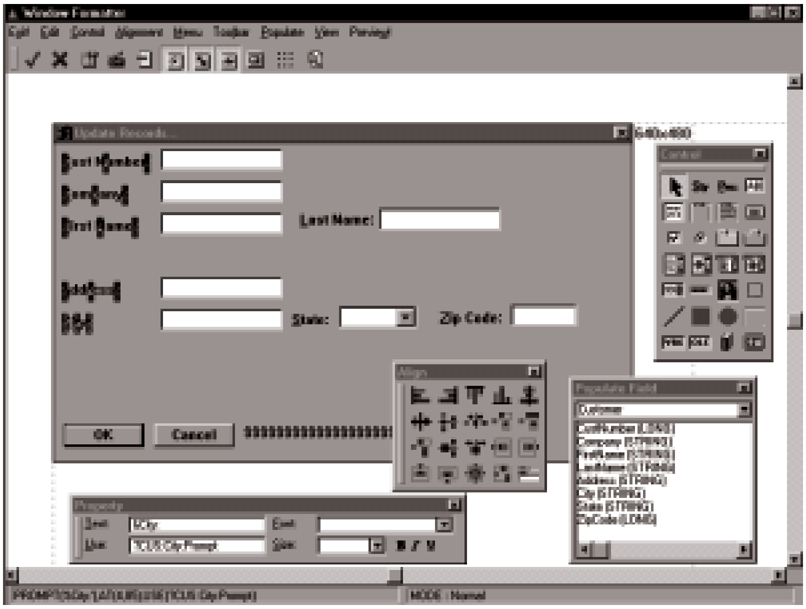
2. **CLICK** on the first prompt in the upper left corner.

   This should be the *CustNumber* prompt. Its handles should appear when you click it.

3. **CTRL+CLICK** on the four prompts immediately below the first.

   As you CTRL+CLICK on each control in turn, the previously selected controls' handles turn blue while the newest selected control's handles are red. The control with the red handles provides the "base point" for the alignment operation. All the other selected controls are aligned in relation to the control with the red handles.

   With all five prompts selected, it should look like this:



4. **Press** the **Align Left** button (the top left button) in the **Align** toolbox.

   The controls all line up along their left edges, based on the position of the last item selected (the one with the red handles).

To identify the controls in the **Align** toolbox, simply place the mouse cursor over the control and wait half a second for the tool tip to appear.

5. **Press** the **Spread Vertically** button in the floating **Align** toolbox.

   The controls all evenly spread themselves between the top and bottom controls selected.

6. CLICK on the first entry control (this should be the *CustNumber* entry field).

7. CTRL+CLICK on the entry controls immediately below it to **select** them all.

8. RIGHT-CLICK and **choose** **Align Left** from the popup menu that appears.

   Here is yet another way to get to the alignment tools. The alignment popup menu appears only when you have multiple controls selected.

9. RIGHT-CLICK and **choose** **Spread Vertically** from the popup menu that appears.

   This should align all the data entry controls with their respective prompts that we already **Spread Vertically**.

10. CLICK on the *LastName* entry box.

11. CTRL+CLICK on the three controls to its left (its prompt, the *FirstName* entry field and prompt).

12. **Choose** **Align Horizontally** from either the floating **Align** toolbox or the RIGHT-CLICK popup alignment menu.

    This aligns the controls in a neat row.

    There is one more way to select multiple controls in the Window Formatter: Lasso them.

13. **Place** the mouse cursor slightly above and to the left of the first control in the bottom row (this should be *City* prompt).

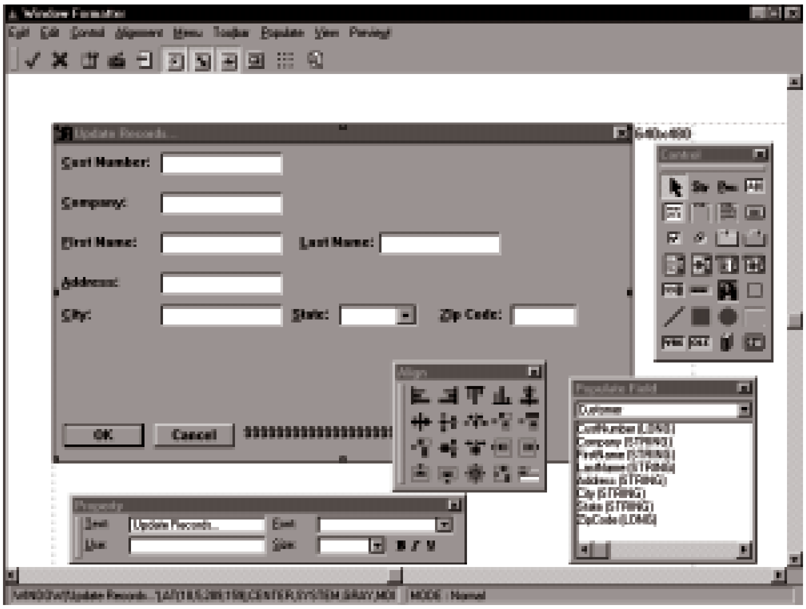
14. CTRL+CLICK AND DRAG slightly down and to the right until the box outline surrounds all five controls to the right (the prompts and controls for *City*, *State* and *ZipCode*) then release the mouse button.

    The red “handles” appear on the ZipCode entry control and the blue “handles” on the other controls. This is the “lasso” technique.

15. **Choose** **Align Horizontally** from either the floating **Align** toolbox or the RIGHT-CLICK popup alignment menu.

16. Use the **Align Horizontally** tool to align the *CustNumber*, *Company*, and *Address* entry boxes with their respective prompts.

    The window should now look something like this:



The form window is almost done. Now we will add a browse list box for the related Phones file records.

Adding a BrowseBox Control Template

Control templates generate all the source code required to create and maintain a specific type of control (or set of controls) on your window. All the entry controls we just placed on this window are simple controls, not Control templates, because they do not need any extra code to perform their normal function. Control templates are only used when a specific control needs extra functionality that the "bare" control itself does not provide. For example, the OK and Cancel buttons are both Control templates—the OK button's Control template saves the record to disk, while the Cancel button's Control template has all the "cleanup" code necessary to cancel the current operation.

Now you will place a *BrowseBox* Control template that displays all the records from the *Phones* file that are related to the current *Customer* record.

Place the Control Template

1. **Choose** **Populate ➤ Control Template**, or CLICK on the Control Template tool in the floating **Controls** toolbox (last tool icon on the right, bottom row).

2. In the **Select Control Template** dialog, **highlight** the *BrowseBox* Control template, then **press** the **Select** button.

   The cursor changes to a crosshair and "little book."

3. CLICK just below the *City* entry box to place the control.

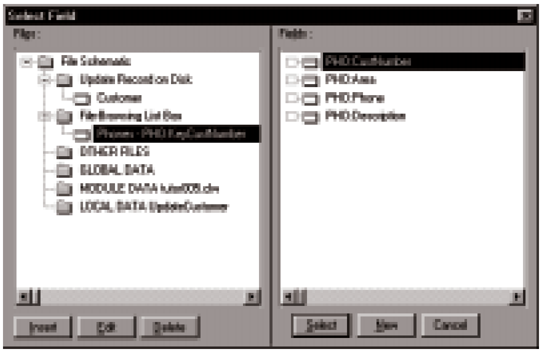
   The Select Field dialog appears, ready for you to choose the file this BrowseBox will display.

4. **Select** the "ToDo" item below the **File-Browsing List Box** and **press** the **Insert** button.

5. **Highlight** the *Phones* file in the **Insert File** dialog, then **press** the **Select** button.

6. **Highlight** the *Phones* file in the **Files** list, then **press** the **Edit** button.

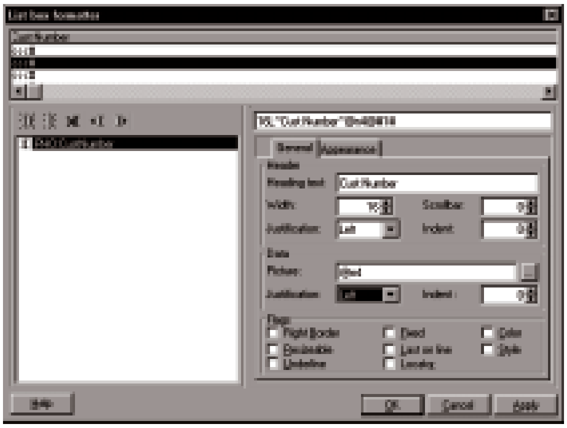
7. **Highlight** *KeyCustNumber* in the **Change Access Key** dialog, then **press** the **Select** button.



Place the Phones file fields in the List Box Formatter

1. **Highlight** *PHO:CustNumber* in the **Fields** list, then **press** the **Select** button.

   The List Box Formatter now appears, ready for you to choose the rest of the fields to display.



2. **Select** Left from the Data group box's Justification dropdown list.

   This changes the data justification from the default value (which is Right justification for numeric values).

3. **Press** the New Column button.

4. **Highlight** *PHO:Area* in the Fields list, then **press** the Select button.

5. **Select** Left from the Data group box's Justification dropdown list, then **press** the OK button.

6. **Press** the New Column button.

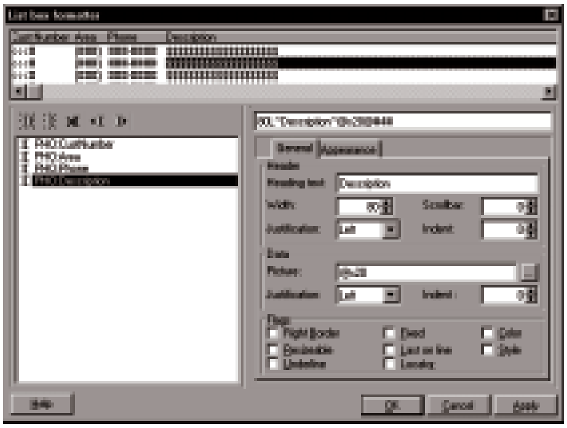
7. **Highlight** *PHO:Phone* in the Fields list, then **press** the Select button.

8. **Select** Left from the Data group box's Justification dropdown list, then **press** the OK button.

9. **Press** the New Column button.

10. **Highlight** *PHO:Description* in the Fields list, then **press** the Select button.

    As an optional step, resize the columns in the List Box Formatter's sample window (at the top of the dialog) to make them wide enough for the column headers. To resize the columns, just DRAG the left edge of the column with the mouse.



List box formatter

Heading text: Description
Justification: Left
Data
Justification: Left

11. **Press** the OK button to close the List Box Formatter.

    This places the formatted List box on the window at the position we specified. This may expand the window. If so, resize the List box by DRAGGING its handles, then move the OK, Cancel, and Message string controls down to the new bottom of the window.

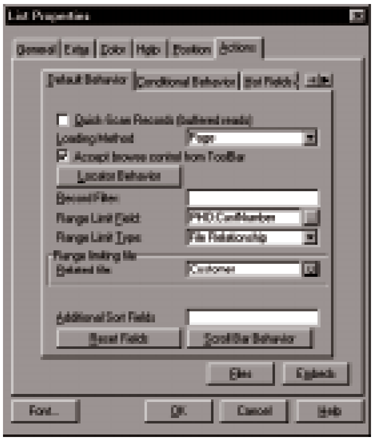
Set up the control template's record range limits

1. RIGHT-CLICK the list box you just placed, and **select** Actions from the popup menu.

2. **Press** the ellipsis (...) button next to the Range Limit Field.

3. **Highlight** the *PHO:CustNumber* field in the **Key Components** list, then **press** the **Select** button.
4. **Choose** **File Relationship** from the **Range Limit Type** dropdown list.
5. **Press** the ellipsis (...) button next to the **Related File** field then **select** *Customer* from the **Select File** dialog.

   This identifies the Customer file as the related file. These steps limit the records displayed in the list box to only those records related to the currently displayed Customer file record.



List Properties

General | Extra | Color | Help | Position | Actions

Default Behavior | Conditional Behavior | Hot Fields

☐ Quick-Scan Records (buffered reads)

Loading Method: Page

☑ Accept browse control from ToolBar

Record Filter:

Range Limit Field: PHO:CustNumber

Range Limit Type: File Relationship

Range limiting file

Related file: Customer

Additional Sort Fields:

6. **Press** **OK** to close the **List Properties** dialog.

Adding the BrowseUpdateButtons Control Template

Next we'll add the standard Insert, Change and Delete buttons for the list box control.

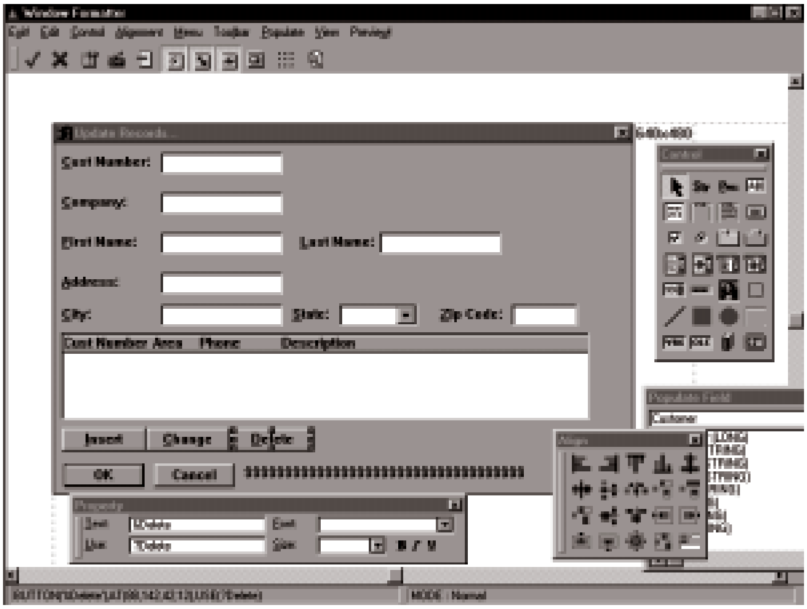
Place another type of Control Template

1. **Choose** **Populate ➤ Control Template**, or CLICK on the Control Template tool in the **Controls** toolbox (last tool icon on the right, bottom row).
2. **Highlight** the *BrowseUpdateButtons* Control template, then **press** the **Select** button.

   The cursor changes to a crosshair and "little book."
3. CLICK below the left end of the list box.

   The **Insert**, **Change**, and **Delete** buttons appear. Remember, these are the buttons that will allow the toolbar buttons to function, so they must be present in the window design. They do not have to be visible to the end-user, so you can hide them if you choose. However, since this BrowseBox is placed on an update Form procedure, for this application we'll leave this set of *BrowseUpdateButtons* visible. This will allow the user to use either set of buttons. The toolbar update buttons will only function for this list when the list box has focus—not when the user is inputting data into any other control—so keeping these buttons visible will ensure that the user can easily maintain the Phones file records.

At this point the window should look something like this:



Specify Edit in Place for Phones Update

1. **RIGHT-CLICK** on the **Delete** button and **choose** **Actions** from the popup menu.
2. **Check** the **Use Edit in place** box.

   Setting the **Actions** for one button sets them for all three buttons in the set, because they all belong to the same Control Template. Since the *Phones* file is a small file with just a couple of fields, there's no need for a separate Update Procedure.

3. **Press** the **OK** button.
4. **Choose** **Exit!** to close the **Window Formatter**.
5. **Press** the **OK** button to close the **Procedure Properties** dialog.
6. **Choose** **File ➤ Save**, or **press** the *Save* button on the tool bar.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- ♦ You created a new Form Procedure—without using a wizard.
- ♦ You learned just how quickly you can populate data entry controls onto a Form by using the **Fieldbox** toolbox.
- ♦ You learned to use the Window Formatter's tools to move and align controls.
- ♦ You used the List Box Formatter again and created a range limited listbox.
- ♦ You learned how to implement edit-in-place for simple updates which don't require a Form procedure.

Now you've created all the procedures necessary to maintain both the *Customer* and *Phones* data files. Next, we'll create the procedures which will maintain the *Products* data file.

9 – COPYING PROCEDURES

The Products File Procedures

Now that we've created the Customer browse, we can reuse much of that work for the next procedure by copying the procedure, then changing its fields. In this chapter, you'll copy the *ViewCustomers* procedure to create the *ViewProducts* procedure.

You will also use "Embed points" to write "embedded source code" to call the *Viewproducts* procedure from your application's menu and toolbar. This will introduce you to the numerous points at which you can add a few (or many) lines of your own source code to add functionality to any procedure.

Starting Point:
The TUTORIAL.APP file should be open.

Copy the Procedures

As you recall, when you created your **View ➤ Products** menu item, and the toolbar button labeled "Products," you didn't specify a procedure to call when the end user executed them. We'll start by creating the procedure to call.

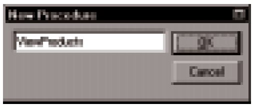
1. **Highlight** the *ViewCustomers* procedure in the **Application Tree** dialog.

   This is the procedure you will copy.

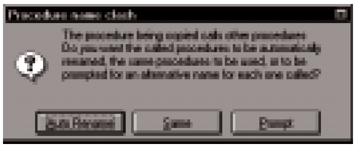
2. **Choose** **Procedure ➤ Copy...**.

   The **New Procedure** dialog appears.

3. **Type** *ViewProducts* in the entry box then **press** the **OK** button.



Because the *UpdateCustomer* procedure is nested under the *ViewCustomers* procedure (the one you are copying), the **Procedure name clash** dialog appears. This offers you options on how to handle the clashing procedures.

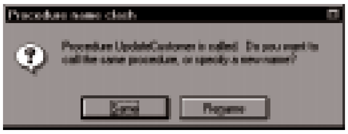

Procedure name clash

The procedure being copied calls other procedures. Do you want the called procedures to be automatically renamed, the same procedures to be used, or to be prompted for an alternative name for each one called?

4. **Press** the **Prompt** button.

   By pressing the **Prompt** button, you tell the Application Generator to let you have the opportunity to rename all the clashing procedures, or not.

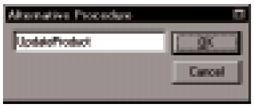
   Another warning message box appears to inform you of a specific duplicate procedure name.


Procedure name clash

Procedure UpdateCustomer is called. Do you want to call the same procedure, or specify a new name?

5. **Press** the **Rename** button.

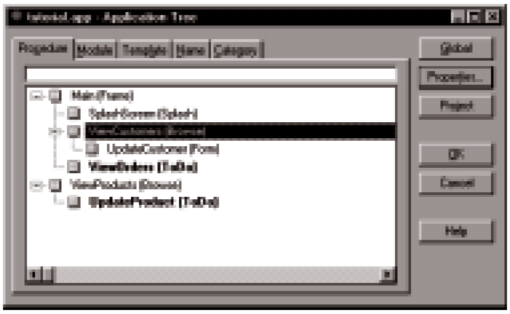
   The **Alternative Procedure** dialog appears.


Alternative Procedure

UpdateProduct

6. **Type** *UpdateProduct* in the entry box, then **press** the **OK** button.

   The *ViewProducts* and *UpdateProduct* procedures appear in the Application Tree. They look "disconnected" from the other procedures because no other procedure calls them (yet). We'll do that next.


tutorial.app - Application Tree

Main (Frame)
SplashScreen (Splash)
ViewCustomers (Browse)
UpdateCustomer (Form)
ViewOrders (ToDo)
ViewProducts (Browse)
UpdateProduct (ToDo)

Working with Embed Points

The Clarion templates allow you to add your own customized code to many predefined points inside the standard code that the templates generate. It's a very efficient way to achieve maximum code reusability and flexibility. The point at which your code is inserted is called an *Embed Point*. Embed points are available at all the standard events for the window and each control, and many other logical positions within the generated code.

In this example, you add embedded source code—using a Code template that will write the actual source for you—at the points where the end user chooses the **View ➤ Products** menu item, and at the point where the end user presses the **Products** button on the application's toolbar.

Name the procedure to call to View the Products

1. RIGHT-CLICK on the *Main* procedure in the Application Tree.

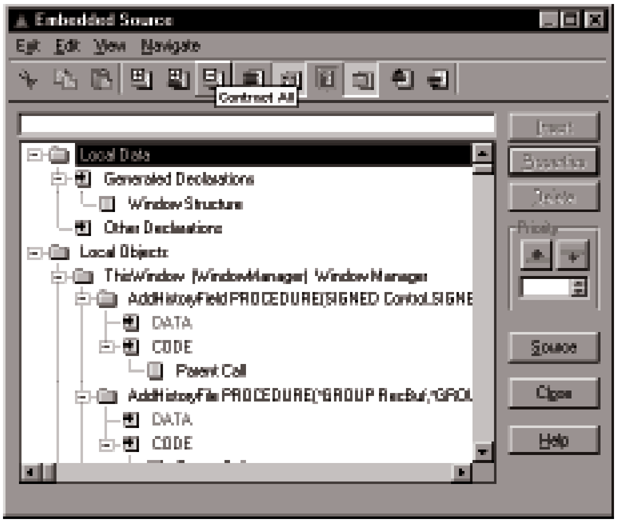
   There are several ways to access the embedded source code points within a procedure. Two of them appear on the popup menu that you now see.

   The first is the **Embeds** selection, which calls the **Embedded Source** dialog to show a list of all the embed points within the procedure.

   The second is the **Source** selection, which actually generates source code for the procedure and calls the "Embeditor" (the Text Editor in embed point edit mode) to allow you to directly edit all the embed points within the context of generated source. The generated source code is "grayed out" to indicate that you cannot edit it, and every possible embed point in the procedure is identified by comments, following which you may type your code.

   There are advantages to each method of working in embed points, so we'll cover both methods during the course of this tutorial. First, we'll use the **Embedded Source** dialog.

2. **Choose Embeds** from the popup menu.



The **Embedded Source** dialog appears, allowing access to all the embed points in the procedure. You can also get here from the **Embeds** button on the **Procedure Properties** window, but the popup menu is quicker. This list is either sorted alphabetically or in the order in which they appear in the generated source, depending on whether you have the **Sort Embeds Alphabetically** box checked in **Setup ➤ Application Options**.

3. **Press** the **Contract All** button.

   This will make it easier to locate the specific embed point you need.

4. **Locate** the **Control Events** folder, then CLICK on its **+** sign to expand its contents.

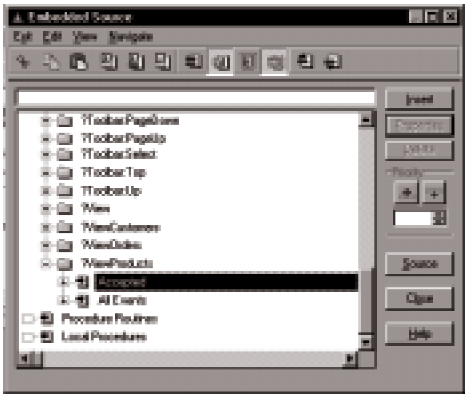
   The menu selection is a control, just as an entry box on the window is.

   You'll notice that there are some up and down buttons and a spin box at the right side of the window which allow you to select a **Priority**—these are important. The templates generate much of the code they write for you into these same embed points. Sometimes, the code you want to write should execute *before* any template-generated code, and sometimes it should execute *after*, and sometimes it should execute somewhere between various bits of generated code. The exact placement of your code within the embed point is determined by the **Priority** number. This provides you with as much flexibility in placing your embed code as possible. The **Priority** numbers themselves do not matter, but the logical position within the generated code does, and that's why this dialog also shows comments which identify the embed priorities. Don't worry, here's more coming on this issue later that'll help make it more clear.

5. **Locate** the **?ViewProducts** folder, then CLICK on it to expand it.

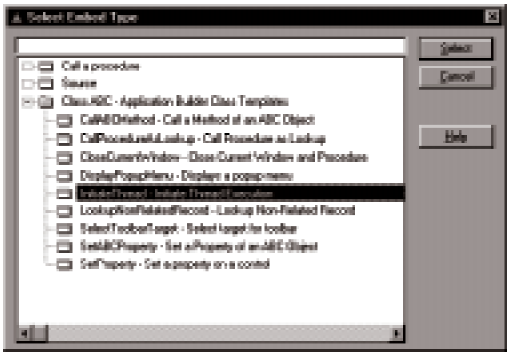
6. **Highlight** **Accepted** then **press** the **Insert** button.

   The "Accepted" event for this menu selection marks the point in the generated code that executes when the user chooses the menu command.



   The **Select embed type** dialog appears to list all your options for embedding code. You may simply **Call a Procedure**, write your own Clarion language **Source** in the Text Editor, or use a Code template to write the source code for you. This is one advantage to editing embed points from within the **Embedded Source** dialog—you can use Code templates to write the code for you instead of writing it yourself.

7. **Select** the **InitiateThread** Code template, then **press** the **Select** button.

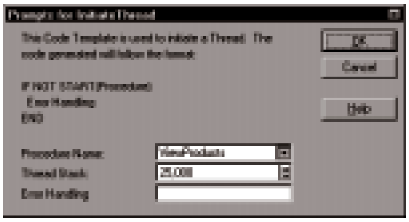


Call a procedure
Source
Class ABC - Application Builder Class Templates
CallABCMethod - Call a Method of an ABC Object
CallProcedureAsLookup - Call Procedure as Lookup
CloseCurrentWindow - Close Current Window and Procedure
DisplayPopupMenu - Displays a popup menu
InitiateThread - Initiate Thread Execution
LookupNonRelatedRecord - Lookup Non-Related Record
SelectToolbarTarget - Select target for toolbar
SetABCProperty - Set a Property of an ABC Object
SetProperty - Set a property on a control

A Code template usually provides just a few prompts and instructions on its use. It gathers the information it needs from you to write its executable code, which it then inserts into the standard generated code produced by the Procedure template directly into this embed point. This Code template is designed to start a new execution thread by calling the procedure you name using the START procedure.

8. **Choose** *ViewProducts* from the **Procedure Name** dropdown list.

   This names the procedure to call when the user chooses the menu item. This is the name of the procedure you previously copied.



This Code Template is used to initiate a Thread. The code generated will follow the format:

```
IF NOT START(Procedure)
  Error Handling
END
```

Procedure Name: ViewProducts
Thread Stack: 25,000
Error Handling:

9. **Press** the **OK** button.

Name the procedure to call for the Products toolbar button

At this point, you could do the same thing to call the *ViewProducts* procedure from the **Product** button. However, there's an easier way to write this code again—just Copy and Paste it from one embed point to another!

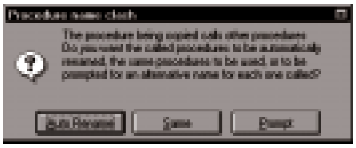
1. **CLICK** on the **Copy** button (the middle button of the button group at the left end of the toolbar).

   The Code template you just added should still be highlighted, so this will copy it to the Windows clipboard.

2. **Locate** the *?ProductsButton* folder (in the same **Control Event Handling** folder you are already in), then **CLICK** on it to expand it.

3. **Highlight** **Accepted** then **press** the **Paste** button (the far right button of the button group at the left end of the toolbar).

   The **Procedure name clash** dialog appears again to warn you that you've already called this procedure once.

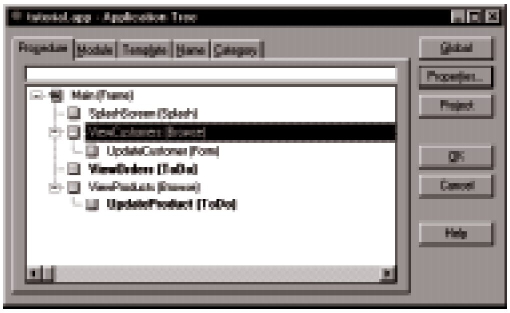

Procedure name clash

The procedure being copied calls other procedures. Do you want the called procedures to be automatically renamed, the same procedures to be used, or to be prompted for an alternative name for each one called?

4. **Press** the **Same** button.

   Now this embed point will generate the same code as the previous one.

5. **Press** the **Close** button.

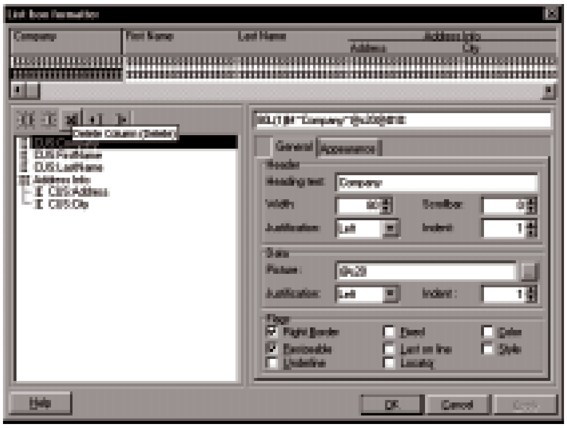


The *ViewProducts* procedure now “connects” to the *Main* procedure. Now you can customize the copied procedures for the Products file.

Modify the Browse

Change the file for the browse list control

1. RIGHT-CLICK the *ViewProducts* procedure and **choose Window** from the popup menu.
2. RIGHT-CLICK the listbox control and **choose List Box Format...** from the popup menu.



3. In the List Box Formatter, **press** the **Delete** button repeatedly until all the fields are removed.

   When you've removed the last field, the **Select Field** dialog automatically appears.

4. **Highlight** the *Customer* file in the **Files** list, then **press** the **Delete** button.
5. **Highlight** the "ToDo" which replaces the *Customer* file, then **press** the **Insert** button.
6. **Highlight** the *Products* file then **press** the **Select** button.

   The **Select Field** dialog now lists the correct file and fields for this procedure.

7. **Press** the **Edit** button, then **select** *KeyProdDesc* from the **Change Access Key** dialog.

   The **Select Field** dialog now lists the correct file and fields.

Re-populate the fields

1. **Highlight** *PRD:ProdNumber* in the **Fields** list, then **press** the **Select** button.
2. **Check** the **Right Border** and **Resizeable** boxes, and increment both the **Indent** spin boxes to one (1).
3. **Press** the **New Column** button.
4. **Highlight** *PRD:ProdDesc* in the **Fields** list, then **press** the **Select** button.
5. **Choose** *Left* from the **Data** group's **Justification** droplist.
6. **Press** the **New Column** button.
7. **Highlight** *PRD:ProdAmount* in the **Fields** list, then **press** the **Select** button.
8. **Increment** the **Data** group's **Indent** spin box to twelve (12).

   Since the default **Justification** is *Decimal*, this step is required to ensure the fractional values have room to appear.

9. **Press** the **New Column** button.
10. **Highlight** *PRD:TaxRate* in the **Fields** list, then **press** the **Select** button.

    Notice the default **Justification** is *Decimal*, and the **Data** group's **Indent** spin box is still set to twelve (12) fromthe previous field.

11. **Press** the **OK** button to close the List Box Formatter.

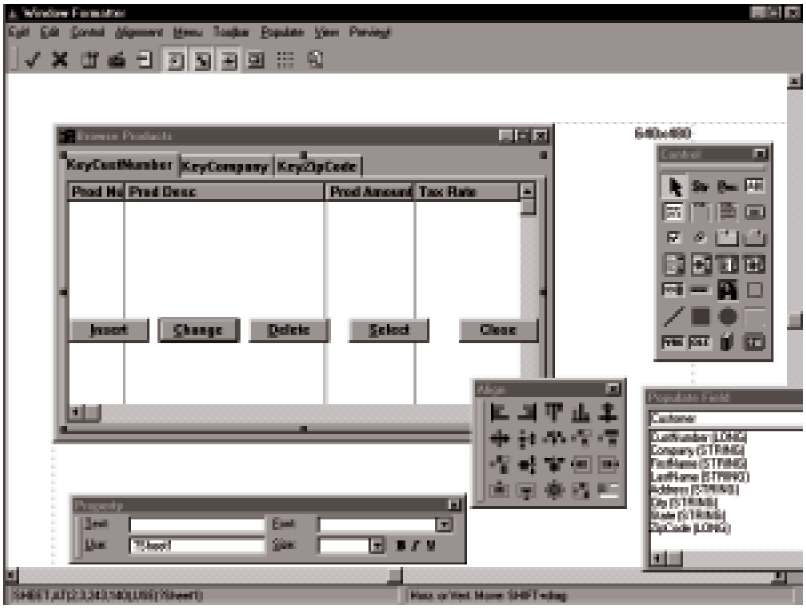
    Don't worry about the buttons on top of the list box. Remember, these are just the "hidden" buttons that the toolbar update buttons call.

Change the name of the window

1. **CLICK** on the sample window caption bar.
2. **Type** *Browse Products* in the **Text** field of the **Property** toolbox, then **press** TAB.

Remove all the tabs

1. CLICK immediately to the right of the *KeyZipCode* tab to **select** the entire property sheet.


Window Formatter
Edit Edit Control Alignment Menu Toolbar Populate View Preview!
Browse Products
KeyCustNumber KeyCompany KeyZipCode
Prod No Prod Desc Prod Amount Tax Rate
Insert Change Delete Select Close
Control
Align
Property
Populate Field
Customer

To be sure that you have CLICKED in the right place, look at the **Property** toolbox and make sure that its **Use** field displays *?Sheet1*. If it does not, try again until it does.

2. **Press** DELETE on the keyboard.

   All the tabs disappear.

Add an entry locator

If the list of Products to display is very long, the user can do a lot of scrolling before finding the specific Product they want. By default, all BrowseBox Control Templates have a “Step” record locator that allows the user to press the first letter of the value in the sort key field to get to the first record that begins with that letter.

Sometimes with large databases however, the user needs to enter the first several letters to get close to the record they want. An Entry locator provides that functionality by specifying an entry control for the user to type into—when they then press TAB to leave the entry box, the list scrolls to the first record matching the data the user entered. This only works with STRING keys.

1. If it’s not already present, **choose** **Options ➤ Show Fieldbox** to display the floating **Populate Field** toolbox.

If the **Populate Field** toolbox still shows the Customer file's fields, simply close the toolbox then re-open it to refresh the toolbox with the Products file fields.

2. DOUBLE-CLICK on *ProdDesc* in the **Populate Field** toolbox.

   This automatically places both the prompt and entry box for the field near the top left corner of the window. This is the Locator's entry box.

3. **Choose** **Exit!** from the menu to close the Window Formatter (save your changes as you exit).

Clean up the alternate sort orders

1. RIGHT-CLICK the *ViewProducts* procedure and **choose** **Extensions** from the popup menu.

   The **Extension and Control Templates** dialog appears. This dialog lists all the Control templates in the procedure and their **Actions** prompts.

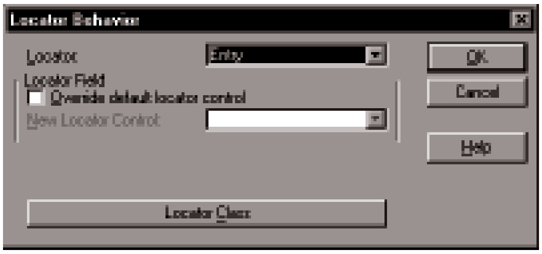
   This dialog also allows you to add and maintain Extension templates to the procedure. Extension templates are very similar to Control templates, in that they add specific functionality to the procedure, but an Extension template's functionality is not directly associated with any control(s) on the window. In other words, Extension templates add "behind the scenes" functionality to a procedure that don't directly affect the user interface.

   A very good example of Extension templates comes in Clarion's Internet Connect product. Clarion Internet Connect contains (among its other development tools) a set of Extension templates which automatically "translate" a Clarion Windows application into dynamic HTML pages. This allows the Clarion application to execute across the Internet. The user just needs to use any Java-enabled browser to access the Clarion program. TopSpeed sales can give you more information on Internet Connect, if you're interested.

2. **Highlight** **Browse on Products** then **press** the **Locator Behavior** button.

3. **Select** *Entry* from the **Locator** droplist then **press** the **OK** button.

   This completes the requirements for the Entry Locator. The key field of the sort order (in this case PRO:ProdDesc) is the default locator control.



Locator Behavior

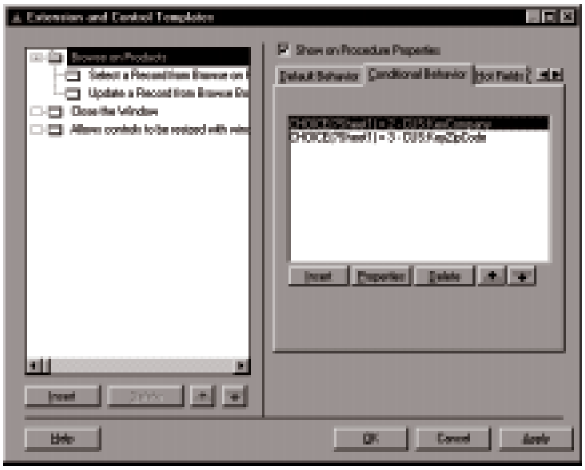
Locator: Entry

Locator Field

☐ Override default locator control

New Locator Control:

4. **Select** the **Conditional Behavior** tab.


Extension and Control Templates
Browse on Products
Select a Record from Browse on
Update a Record from Browse
Close the Window
Allows controls to be resized with win
Show on Procedure Properties
Default Behavior
Conditional Behavior
Hot Fields
Insert
Properties
Delete
Help
OK
Cancel
Apply

CHOICE(?Sheet1) = 2 - CUS:KeyCompany
CHOICE(?Sheet1) = 3 - CUS:KeyZipCode

5. **Press** the **Delete** button twice.

   This removes the two conditional expressions we entered for the *ViewCustomers* procedure.

6. **Press** the **OK** button.
7. **Choose** **File ➤ Save**, or **press** the *Save* button on the toolbar.

Creating the Form Procedure

When you renamed the reference to the *UpdateCustomer* procedure while copying *ViewCustomer* to *ViewProducts*, it made the *UpdateProduct* procedure a “ToDo” procedure. Therefore, we need to create a form to update the Products file.

Select the procedure type for UpdateProduct

1. **Highlight** *UpdateProduct*, then **press** the **Properties** button.
2. **Highlight** *Form*, **clear** the **Use Procedure Wizard** **check** box, then **press** the **Select** button.
3. **Press** the **Files** button in the **Procedure Properties** dialog.
4. **Highlight** the “ToDo” file, then **press** the **Insert** button.
5. **Highlight** the *Products* file then **press** the **Select** button.
6. **Press** the **OK** button to return to the **Procedure Properties** dialog.
7. **Press** the **Window** button to design your form.

Populate the fields

1. If it’s not already present, **choose** **View ➤ Show Fieldbox** to display the **Populate Field** toolbox.

2. **DOUBLE-CLICK** on *ProdNumber* in the **Populate Field** toolbox.

   This automatically places both the prompt and entry box for the field near the top left corner of the window.

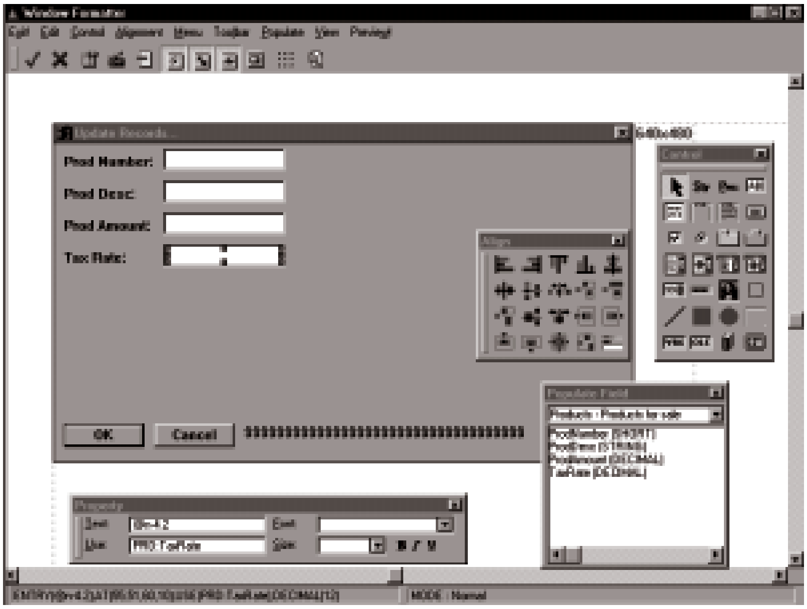
3. **DOUBLE-CLICK** on *ProdDesc* in the **Populate Field** toolbox.

   This automatically places both the prompt and entry box for the field immediately below the last field that was placed.

4. **DOUBLE-CLICK** on *ProdAmount* in the **Populate Field** toolbox.

5. **DOUBLE-CLICK** on *TaxRate* in the **Populate Field** toolbox.

   The form window now looks something like this:



ProdNumber (SHORT)
ProdDesc (STRING)
ProdAmount (DECIMAL)
TaxRate (DECIMAL)

Change the form window caption

1. **CLICK** on the caption bar of the sample window.
2. **Type** *Product Form* in the **Text** field of the **Property** toolbox, then **press** TAB.

Exit the Window Formatter, and save your work

1. **Choose Exit!** on the menu to close the Window Formatter (save your changes).
2. **Press** the **OK** button in the **Procedure Properties** dialog to close it.
3. **Choose File ➤ Save**, or **press** the *Save* button on the toolbar to save your work.

   The Products file update form window is done.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- ♦ You copied an existing procedure, renaming the subsequent procedures it called.
- ♦ You used a Code Template in the **Embedded Source** dialog to call the new procedure from the main menu.
- ♦ You modified the copied procedure to display from another file.
- ♦ You added an Entry Locator to the Browse procedure.
- ♦ You created an entire Form procedure very quickly by just using the **Populate Field** toolbox.

Now that you're thoroughly familiar with Procedure Templates, we'll go on to use some Control and Extension Templates.

10 – CONTROL AND EXTENSION TEMPLATES

For the *ViewOrders* procedure, you'll create a window with two synchronized scrolling list boxes. One will display the contents of the Orders file, and the other will display the related records in the Detail file. You'll use a generic Window procedure, and populate it using Control templates. The only reason for doing it this way instead of starting with a *Browse* Procedure Template is to demonstrate another way of building a procedure—using Control templates placed in a generic Window (the same way the Browse Procedure template itself was created).

Control templates generate all the source code for creating *and maintaining* a single control or related group of controls. In this case, placing a *BrowseBox* Control template allows the Application Generator to produce the code that opens the files and puts the necessary data into the structures which hold the data for display in the list box.

Starting Point:
The TUTORIAL.APP file should be open.

Creating the Procedure

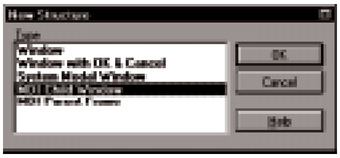
Select the procedure type

1. **Highlight** *ViewOrders* then **press** the **Properties** button.
2. **Highlight** *Window* in the **Select Procedure Type** dialog then **press** the **Select** button.

Edit the Window

1. In the **Procedure Properties** dialog, **press** the **Window** button.

   The **New Structure** dialog appears. The generic window procedure is like a blank slate in which you define your own window. Since the procedure has no predefined window, you choose the type of window for your starting point. In this case, you need an MDI child window.



2. **Highlight** *MDI Child Window*, then **press** the OK button.

   The Window Formatter appears.

3. **Resize** the window, making it more than twice its original size (in both directions).
4. RIGHT-CLICK in the window's title bar, then **choose** Properties from the popup menu.
5. **Type** *Orders* in the Text field.
6. **Select** *Resizable* from the Frame Type drop-down list.
7. **Select** the Extra tab and **check** the Maximize Box box.
8. **Press** the OK button to close the Window Properties dialog.

Placing the BrowseBox Control Template

1. **Choose** Populate ➤ Control Template, or CLICK the *Control Template* tool in the Controls toolbox (last tool on the right, bottom row).

   The Select Control Template dialog appears.

2. **Highlight** the *BrowseBox* Control template, then **press** the Select button.

   The cursor changes to a crosshair and "little book."

3. CLICK near the upper left corner of the sample window.

Place the Orders file fields in the List Box Formatter

1. **Highlight** the "ToDo" item below the File-Browsing List Box and **press** the Insert button.
2. **Highlight** the *Orders* file in the Insert File dialog, then **press** the Select button.
3. **Press** the Edit button and **select** *KeyOrderNumber* from the Change Access Key dialog.
4. **Highlight** *ORD:CustNumber* in the Fields list, then **press** the Select button.
5. **Press** the New Column button.
6. **Highlight** *ORD:OrderNumber* in the Fields list, then **press** the Select button.
7. **Press** the New Column button.
8. **Highlight** *ORD:InvoiceAmount* in the Fields list, then **press** the Select button.
9. **Increment** the Data group's Indent spin box to twelve (12).

   Since the default Justification is *Decimal*, this step is required to ensure the fractional values have room to appear.

10. **Press** the **New Column** button.
11. **Highlight** *ORD:OrderDate* in the **Fields** list, then **press** the **Select** button.
12. **Press** the **New Column** button.
13. **Highlight** *ORD:OrderNote* in the **Fields** list, then **press** the **Select** button.
14. **Choose** *Left* from the **Data** group's **Justification** droplist.
15. **Resize** the columns in the sample list box.
16. **Press** the **OK** button to close the List Box Formatter.
17. **Resize** the browse list box control to make it wider by dragging the handle in the middle of the right side (almost as wide as the window).

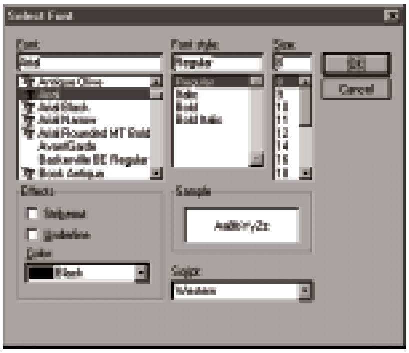
Format the list box appearance

1. **RIGHT-CLICK** the list box, and **select** **Properties** from the popup menu.
2. **Select** the **Extra** tab and **check** the **Vertical** and **Horizontal** boxes in the **List Properties** dialog.

   This adds vertical and horizontal scroll bars to the list.
3. **Press** the **Font** button.

   Because one field (the description field) is long, you can specify that the list box should use a smaller font, displaying more information without requiring the end user to scroll.
4. **Choose** a font (your choice), and **set** the size to 8 points.

   See the *User's Guide* for tips on topics like choosing the right fonts for controls. In general, you want to stick with the fonts that ship with Windows; otherwise, you can't be sure your end user has the same font on their system. The illustration below sets the font to Arial, which is a font that ships with Windows.



Font: Arial
Font style: Regular
Size: 8
Color: Black
Script: Western

5. **Press OK** to close the **Select Font** dialog.
6. **Press OK** to close the **List Properties** dialog.

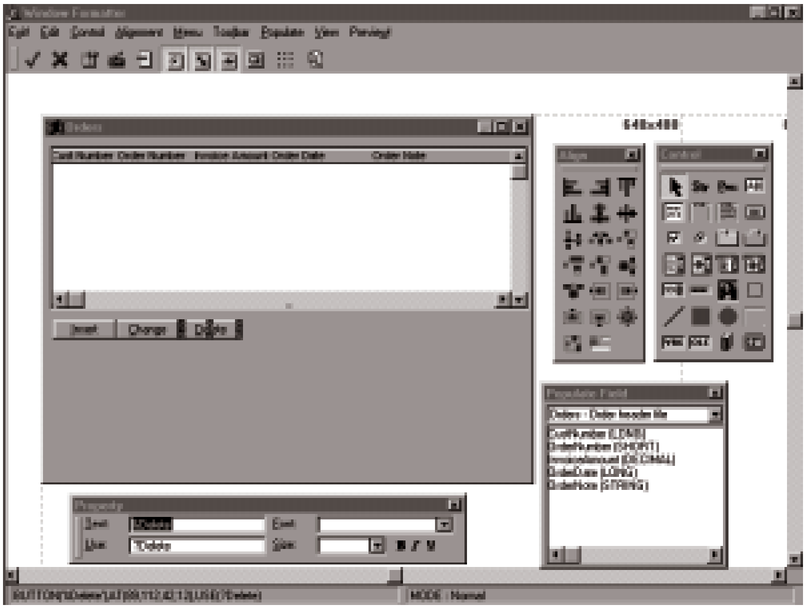
Adding the Browse Update Buttons Template

Next add the standard **Insert**, **Change** and **Delete** buttons for the top browse list box control. Since there are going to be two list boxes on this window, we'll leave these buttons visible for the user. Later we'll add a form procedure for adding or editing an order.

1. **Choose** **Populate ➤ Control Template**, or CLICK on the Control Template tool in the Controls toolbox (last tool icon on the right, bottom row).
2. In the **Select Control Template** dialog, **highlight** the *BrowseUpdateButtons* Control template, then **press** the **Select** button.
3. CLICK below the left edge of the list box.

   The **Insert**, **Change**, and **Delete** buttons all appear together.

   At this point the window should look something like this:



Name the Update Procedure

1. RIGHT-CLICK on the **Delete** button, then **choose** **Actions** from the popup menu.
2. **Type** *UpdateOrder* in the **Update Procedure** box.

   This names the procedure, in the same way that you named the Update procedure for the Customer browse in its **Procedure Properties** dialog.

Naming the Update Procedure for one button in the Control template names it for all three.

3. **Press** the **OK** button.

Placing the Second Browse List Box

Next, place a list box with the contents of the Detail file. This will update automatically when the end user changes the selection in the top list box.

1. **Choose** **Populate ➤ Control Template**, or CLICK on the Control Template tool in the Controls toolbox (last tool icon on the right, bottom row).
2. **Highlight** the *BrowseBox* Control template, then **press** the **Select** button.
3. CLICK directly below the **Insert** button you placed before.

Place the Detail file fields in the List Box Formatter

1. **Highlight** the “ToDo” item below the *second* **File-Browsing List Box** and **press** the **Insert** button.
2. **Highlight** the *Detail* file in the **Insert File** dialog, then **press** the **Select** button.
3. **Press** the **Edit** button.
4. **Highlight** *KeyOrderNumber* in the **Change Access Key** dialog, then **press** the **Select** button.
5. **Highlight** *DTL:OrderNumber* in the **Fields** list, then **press** the **Select** button.
6. **Press** the **New Column** button.
7. **Highlight** *DTL:ProdNumber* in the **Fields** list, then **press** the **Select** button.
8. **Press** the **New Column** button.
9. **Highlight** *DTL:Quantity* in the **Fields** list, then **press** the **Select** button.
10. **Press** the **New Column** button.
11. **Highlight** *DTL:ProdAmount* in the **Fields** list, then **press** the **Select** button.
12. **Increment** the **Data** group’s **Indent** spin box to twelve (12).
13. **Resize** the columns in the sample list box.
14. **Press** the **OK** button to close the List Box Formatter.
15. **Resize** the browse list box control by dragging the handles, making it an appropriate size for display (but leave some space to its right for a button we’re going to place in the bottom right corner of the window).

Set up the Range Limits

1. RIGHT-CLICK on the list box you just placed and **select** Actions from the popup menu.
2. **Press** the ellipsis (...) button for the Range Limit Field.
3. **Highlight** the *DTL:OrderNumber* field in the Components list, then **press** the Select button.
4. **Choose** File Relationship from the Range Limit Type drop down list.
5. **Press** the ellipsis (...) button in the Related File.
6. **Highlight** the *Orders* file in the Select File list, then **press** the Select button.

   These last four steps limit the range of records displayed in the second list box to only those Detail records related to the currently highlighted record in the Orders file's list box.

   This tells the second control template to use the file relationship defined in the data dictionary to synchronize the bottom list to the top.

Format the list box appearance

1. **Select** the Extra tab.
2. **Check** the Vertical and Horizontal boxes.

   This adds horizontal and vertical scroll bars to the list box.
3. **Press** the Font button.

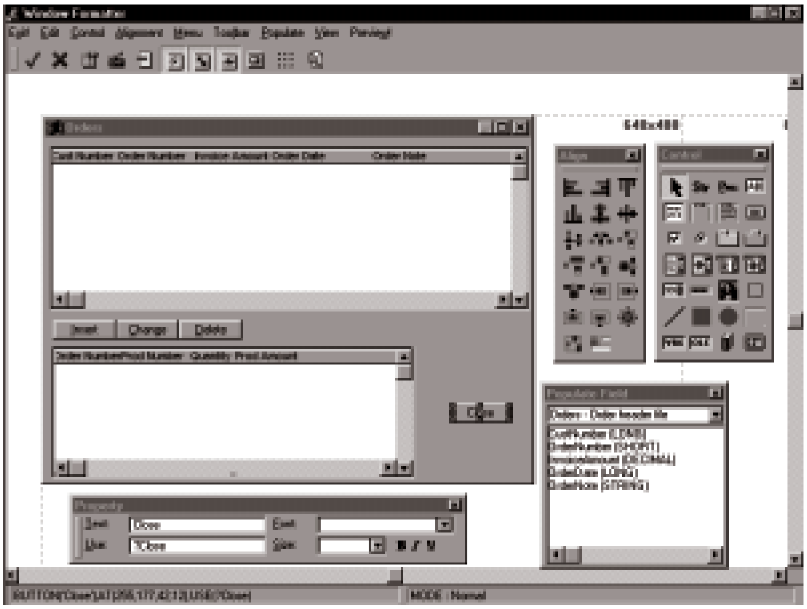
   Although there are no "long" fields in this list box, it will look better if you match the font to the same font used in the top list box.
4. **Choose** a font (your choice), and **set** the size to 8 points.
5. **Press** OK to close the Select Font dialog.
6. **Press** OK to close the List Properties dialog.

Adding the Close Button Control Template

Finally, you can add a **Close** button that closes the window.

1. **Choose** Populate ➤ Control Template, or CLICK on the Control Template tool in the Controls toolbox (last tool icon on the right, bottom row).
2. **Select** the *CloseButton* Control template, then **press** the Select button.
3. CLICK in the lower right corner of the window.

   At this point, your window should look something like this illustration. It may be bigger than the sample area in the Window Formatter, but should not be as big as the desktop:



4. **Choose** Exit! on the Window Formatter menu to close the Window Formatter.

Make the window resizable

1. In the Procedure Properties dialog, **press** the Extensions button.
2. In the Extensions and Control Templates dialog, **press** the Insert button.
3. **Highlight** *WindowResize* in the Select Extension dialog, then **press** the Select button.

   We've used this Extension template several times already, but this time we'll modify its actions instead of simply taking the default behavior.

Specify the resize strategies

1. **Check** the Restrict Minimum Window Size box.

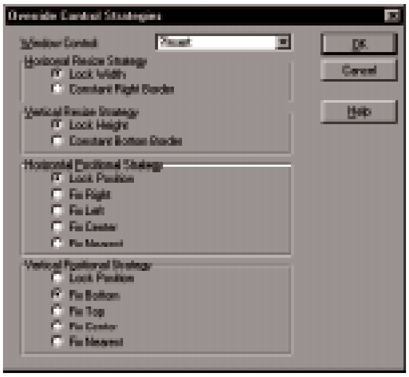
   By checking this box and leaving the Minimum Width and Minimum Height set to zero (0), this template ensures that users cannot make the window any smaller than the designed size of the window.
2. **Press** the Override Control Strategies button.

   The Override Control Strategies dialog appears. This dialog allows you to specify the resize strategy for individual controls.
3. **Press** the Insert button.
4. **Select** *?Insert* from the Window Control drop-down list.

5. **Choose** the *Fix Bottom* radio button in the **Vertical Positional Strategy** set of options.

   This sets the resize strategy for the **Insert** button to keep it a fixed distance from the bottom of the window. Now we'll do the same for the other two update buttons and the Details list box.

6. **Press** the **OK** button.


Override Control Strategies

Window Control: ?Insert

Horizontal Resize Strategy: [✓ Lock Width] Constant Right Border

Vertical Resize Strategy: [✓ Lock Height] Constant Bottom Border

Horizontal Positional Strategy: [✓ Lock Position] Fix Right Fix Left Fix Center Fix Nearest

Vertical Positional Strategy: Lock Position [✓ Fix Bottom] Fix Top Fix Center Fix Nearest

7. **Press** the **Insert** button, then **select** *?Change* from the **Window Control** drop-down list.

8. **Choose** the *Fix Bottom* radio button in the **Vertical Positional Strategy** set of options, then **press** the **OK** button.

9. **Press** the **Insert** button, then **select** *?Delete* from the **Window Control** drop-down list.

10. **Choose** the *Fix Bottom* radio button in the **Vertical Positional Strategy** set of options, then **press** the **OK** button.

11. **Press** the **Insert** button, then **select** *?List:2* from the **Window Control** drop-down list.

12. **Choose** the *Fix Bottom* radio button in the **Vertical Positional Strategy** set of options, then **press** the **OK** button.

13. **Press** the **Insert** button, then **select** *?List* from the **Window Control** drop-down list.

14. **Choose** the *Constant Bottom Border* radio button in the **Vertical Resize Strategy** set of options.

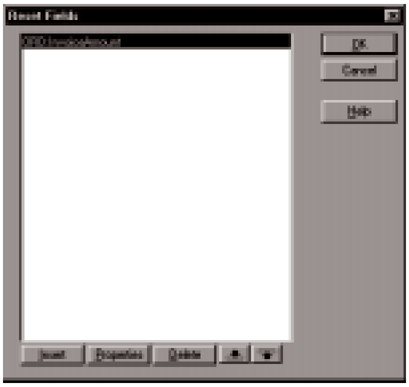
    This sets the resize strategy for the Orders List box to keep the bottom border of the list a fixed distance from the bottom of the window. Therefore, the list will stretch as needed to fill up the space as the window becomes larger.

15. **Press** the **OK** button, then **press** the **OK** button.

Set up a Reset Field

1. In the **Extensions and Control Templates** dialog, **highlight** **Browse on Detail**.
2. **Press** the **Reset Fields** button, then **press** the **Insert** button.
3. **Type** *ORD:InvoiceAmount* in the **Reset Field** field, then **press** the **OK** button.

   This specifies that the Detail file's list box should reset itself whenever the value in the *ORD:InvoiceAmount* field changes. This ensures that any changes you make to an existing order are reflected in this dialog when you return from changing the order.



4. **Press** the **OK** button, then **press** the **OK** button to return to the **Procedure Properties** dialog.

Close the Procedure Properties dialog and Save the Application

1. **Press** the **OK** button in the **Procedure Properties** dialog to close it.
2. **Choose** **File ➤ Save**, or **press** the *Save* button on the tool bar to save your work.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- ♦ You created a new browse procedure, but did it using the *BrowseBox* and *BrowseUpdateButtons* Control Templates instead of the *Browse* Procedure Template.
- ♦ You created a second, range-limited listbox to display related child records.
- ♦ You used the *WindowResize* Extension Template and specified individual control resize strategies.

- ♦ You set a Reset Field on the Detail file’s Browse list so its display is always kept current.

Next we’ll create the UpdateOrder Form procedure to create and maintain the Orders and Detail file records.

11 – ADVANCED TOPICS

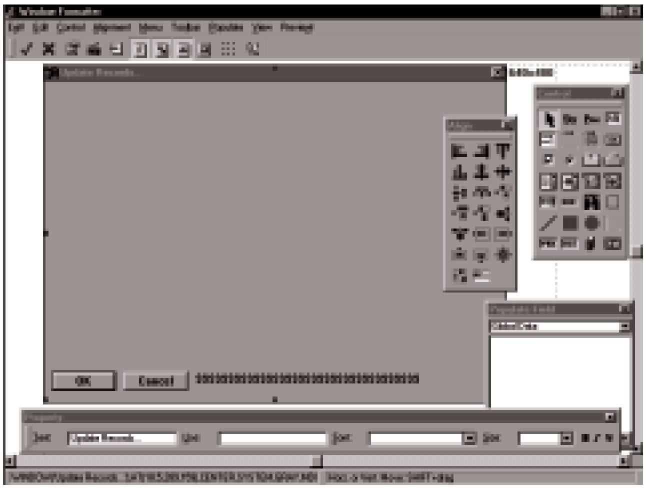
Set Up the UpdateOrder Form

For the Order Update form, we'll place the fields from the Order file on an update form, perform an automatic lookup to the Customer file, add a *BrowseBox* Control template to show and edit the related detail items, calculate each line item detail, then calculate the order total.

Starting Point:
The TUTORIAL.APP file should be open.

Create the Orders file's data entry Form

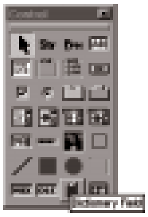
1. **Highlight** *UpdateOrder* in the **Application Tree** dialog, then **press** the **Properties** button.
2. **Highlight** *Form* in the **Select Procedure Type** dialog, **clear** the **Use Procedure Wizard** **check** box, then **press** the **Select** button.
3. **Press** the **Window** button to open the Window Formatter.
4. **Resize** the window taller by DRAGGING its top middle handle (use the Window Formatter's vertical scroll bar to move the viewing area, if necessary).



Place the entry controls for the Orders file

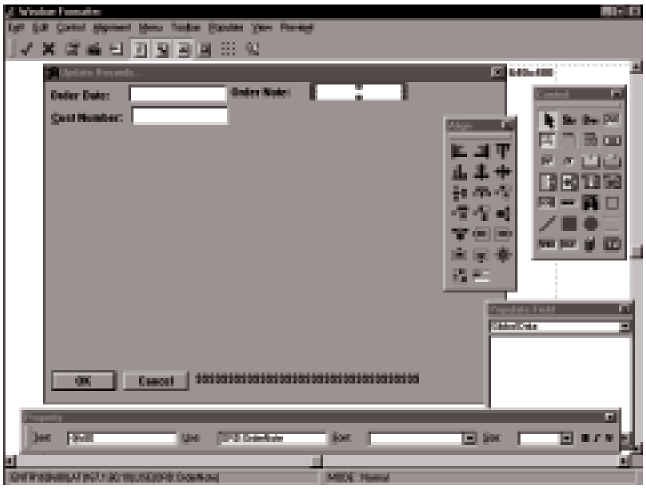
Instead of using the **Populate Field** toolbox to populate the controls, this time we'll use the **Select Field** dialog to populate multiple controls.

1. **Choose** **Populate ➤ Multiple Fields**, or **select** the *Dictionary Field* tool from the **Controls** toolbox.



2. In the **Select Field** dialog, **highlight** the "ToDo" item under the **Update Record on Disk**, then **press** the **Insert** button.
3. **Select** the *Orders* file from the **Insert File** list.
4. **Highlight** *ORD:OrderDate*, then **press** the **Select** button.
5. CLICK near the top left of the window.

   This places both the field and its prompt. The **Select Field** dialog immediately reappears, ready for the next field, making this method of control placement almost as fast as the **Populate Field** toolbox.
6. **Highlight** *ORD:OrderNote*, then **press** the **Select** button.
7. CLICK just to the right of the entry box placed for the date.
8. **Highlight** *ORD:CustNumber*, then **press** the **Select** button.
9. CLICK below the date field's prompt.
10. **Press** the **Cancel** button in the **Select Field** dialog.



Add a lookup procedure call into the customer list

1. **RIGHT-CLICK** on the *ORD:CustNumber* control and **select** **Actions** from the popup menu.

   The standard actions for any entry control allow you to perform data entry validation against a record in another file, either when the control is Selected (just before the user can enter data) or when the control is Accepted (right after the user has entered data).

2. In the **When the Control is Accepted** group box, **press** the ellipsis button (...) for the **Lookup Key** entry box.

3. **Highlight** the *Orders* file in the **Select Key** dialog, then **press** the **Insert** button.

4. **Highlight** the *Customer* file in the **Files** list, and **press** the **Select** button.

   These last two steps add the *Customer* file to the procedure's File Schematic as an automatic lookup from the *Orders* file. This will automatically lookup the related *Customer* file record for you, and the lookup is based on the file relationship set up in the data dictionary.

5. **Highlight** *CUS:KeyCustNumber* in the **Select Key** dialog, then **press** the **Select** button.

   This makes *CUS:KeyCustNumber* the key that will be used to attempt to get a matching valid record from the Customer file for the value the user enters into this control.

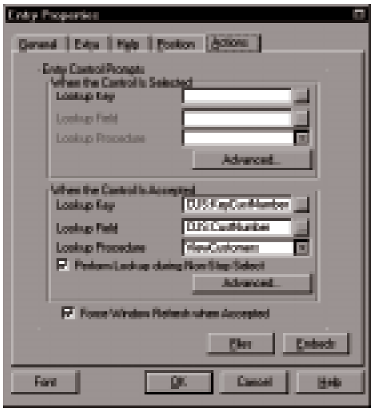
6. **Press** the ellipsis button (...) for the **Lookup Field** entry box.

7. **Highlight** *CUS:CustNumber* from the **Select Component** list, then **press** the **Select** button.

   This makes *CUS:CustNumber* the field that must contain the matching value to the value the user enters into this control.

8. **Choose** the **ViewCustomers** procedure from the **Lookup Procedure** droplist.

   This calls the *ViewCustomers* procedure when the user enters an invalid customer number, so the end user can select from a list of customers.



Entry Control Prompts

When the Control is Selected
Lookup Key
Lookup Field
Lookup Procedure

When the Control is Accepted
Lookup Key: CUS:KeyCustNumber
Lookup Field: CUS:CustNumber
Lookup Procedure: ViewCustomers
☑ Perform Lookup during Non-Stop Select

☑ Force Window Refresh when Accepted

9. **Check** the **Perform Lookup during Non-Stop Select** and **Force Window Refresh when Accepted** boxes to ensure that the data displayed on screen is always valid and current.

10. **Press** the **Embeds** button on the **Actions** tab.

    This displays a list of just the embed points associated with this one control. This is the quickest way to get to a specific control's embed points, and it's the second way you've seen so far to get to an embed point. There is a third method that's still to come.

11. **Highlight** the **Selected** event under **Control Events** then **press** the **Insert** button.

    The *Selected* event occurs just before the control gains input focus. This embed point allows you to do any "setup" specific to this one control.

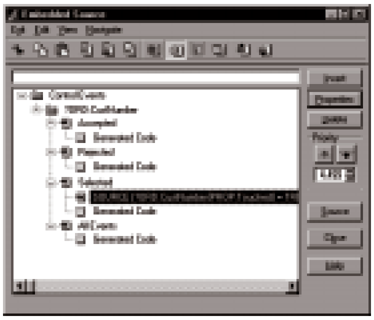
12. **Highlight** **Source** then **press** the **Select** button.

    This calls the Text Editor to allow you to write any Clarion source code you want. Notice that the floating **Populate Field** toolbox is present. Whenever you DOUBLE-CLICK on a field in this toolbox, it places the name of the field (including any prefix) in your code at the insertion point. This not only helps your productivity (less typing), but also ensures you spell them correctly (eliminating mispelled fieldname compiler errors).

13. **Type** the following code:

```
?ORD:CustNumber{PROP:Touched} = TRUE
```

14. **Choose** **Exit!** (and save) to return to the **Embedded Source** dialog.



It is "standard Windows behavior" that, if the user does not enter data into a control and just presses TAB (or CLICKS the mouse) to go on to another control, an *Accepted* event does not happen (this is very different from the way DOS programs work). This allows users to easily TAB through the window's controls without triggering data-entry validation code on each control. However, sometimes you need to override this "Windows standard behavior" to ensure the integrity of your database.

The **?ORD:CustNumber{PROP:Touched} = TRUE** statement uses the Clarion language Property Assignment syntax (see the *Language Reference*). By setting PROP:Touched to TRUE in the *Selected* event for this control, an *Accepted* event is always generated—whether the user has entered data or not. This forces the lookup code generated for you into the *Accepted* event for this control (from the information you entered on the **Actions** tab on the previous page) to execute. This ensures that the user either enters a valid Customer number, or the Customer list pops up to allow the user to select a Customer for the Order.

15. **Press** the **Close** button to return to the **Entry Properties** dialog, then **press** the **OK** button to close it.

Add a “display-only” control

1. **Choose** **Control ➤ String**, or CLICK on the **String** tool in the floating **Controls** toolbox (the icon just right of the “big arrow” on the top row).
2. CLICK to the right of the customer number entry box you placed before.
3. RIGHT-CLICK the string control you just placed, and **select** **Properties** from the popup menu.
4. **Check** the **Variable String** box.

   This specifies the control will display data from a variable, not just a string constant. The **Select Field** dialog automatically appears.
5. **Highlight** the *Customer* file in the **Files** list, then **select** *CUS:Company* from the **Fields** list and **press** the **Select** button.
6. **Press** the **OK** button to close the **String Properties** dialog.

Placing the Detail File’s Control Templates

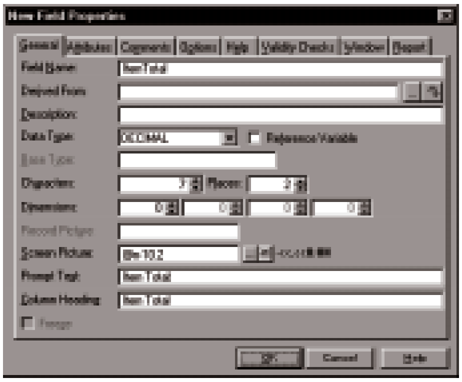
The next key element in this window is a browse list box control, synchronized to the Order Number of this form. This will show all the records in the Detail file related to the currently displayed *Orders* file record.

Add a Detail list

1. **Choose** **Populate ➤ Control Template**, or CLICK on the Control Template tool in the floating **Controls** toolbox (last tool icon on the right, bottom row).
2. **Highlight** *BrowseBox*, then **press** the **Select** button.
3. CLICK below the customer number entry box you placed before.
4. **Highlight** the “ToDo” item below the **File-Browsing List Box** and **press** the **Insert** button.
5. **Select** the *Detail* file from the **Insert File** dialog, then **press** the **Select** button.
6. **Press** the **Edit** button.
7. **Highlight** *KeyOrderNumber* in the **Change Access Key** dialog, then **press** the **Select** button.

8. **Highlight** *DTL:ProdNumber* in the **Fields** list, then **press** the **Select** button.
9. **Select** **Center** from the **Data** group's **Justification** droplist.
10. **Press** the **New Column** button.
11. **Highlight** *DTL:Quantity* in the **Fields** list, then **press** the **Select** button.
12. **Select** **Center** from the **Data** group's **Justification** droplist.
13. **Press** the **New Column** button.
14. **Highlight** *DTL:ProdAmount* in the **Fields** list, then **press** the **Select** button.
15. **Select** **Center** from the **Data** group's **Justification** droplist.
16. **Press** the **New Column** button.
17. **Highlight** *LOCAL DATA UpdateOrder* in the **Files** list, then **press** the **New** button.

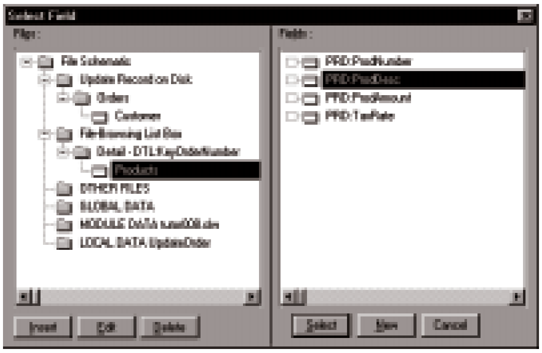
    This **New** button allows you to add a local variable without going all the way back to the **Procedure Properties** window and pressing the **Data** button. We will use this new variable to display the total price for each line item (the quantity multiplied by the unit price).
18. **Type** *ItemTotal* in the **Name** field.
19. **Select** *DECIMAL* from the **Type** dropdown list.
20. **Type** *7* in the **Characters** field.
21. **Type** *2* in the **Places** field then **press** the **OK** button.



Field Name: ItemTotal
Derived From:
Description:
Data Type: DECIMAL
Reference Variable
Base Type:
Characters: 7 Places: 2
Dimensions: 0 0 0 0
Record Picture:
Screen Picture: @n-10.2
Prompt Text: Item Total
Column Heading: Item Total
Freeze

22. **Select** **Center** from the **Data** group's **Justification** droplist.
23. **Press** the **New Column** button.
24. **Highlight** the *Detail* file item below the **File-Browsing List Box** and **press** the **Insert** button.
25. **Select** the *Products* file from the **Insert File** dialog, then **press** the **Select** button.

This adds the *Products* file to the Control template's file schematic as a lookup file. The related record from the *Products* file is automatically retrieved for you so you can display the product description in the list.



26. **Highlight** *PRD:ProdDesc* in the **Fields** list, then **press** the **Select** button.

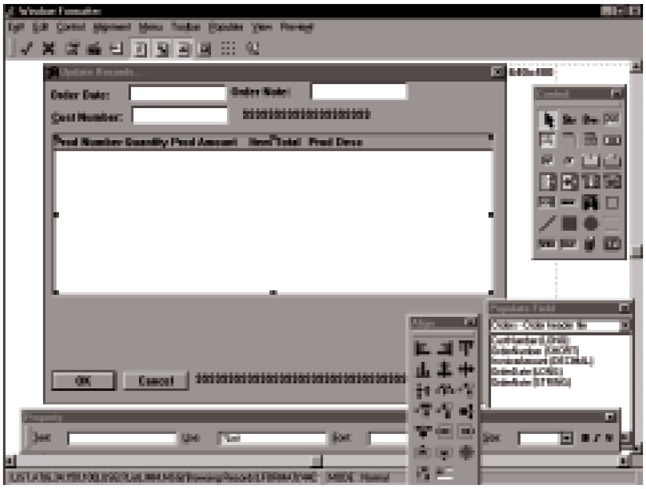
27. **Select** **Left** from the **Data** group's **Justification** droplist.

28. **Resize** the columns and adjust the display formatting as you want (you've done this a couple of times already).

29. **Press** the **OK** button to close the List Box Formatter.

30. **Resize** the list box to display all the fields you populated into it.

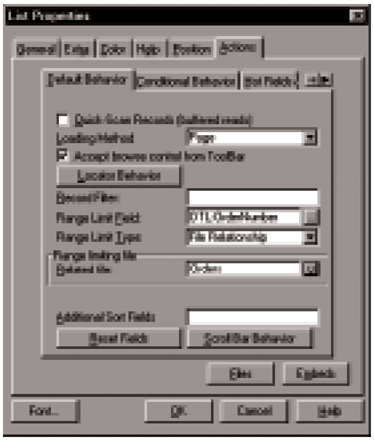
It should now look something like this:



Synchronize the browse list box

We want this list to only display the *Details* file records that are related to the *Customer* file record currenly being edited. Therefore, we need to specify a Range Limit.

1. **RIGHT-CLICK** the list box you just placed, and **select** Actions from the popup menu.
2. **Press** the ellipsis (...) button for the Range Limit Field.
3. **Highlight** the *DTL:OrderNumber* field in the Components list, then **press** the Select button.
4. **Choose** File Relationship from the Range Limit Type drop down list.
5. **Type** *Orders* in the Related File field.


List Properties

General | Extra | Color | Help | Position | Actions

Default Behavior | Conditional Behavior | Hot Fields

☐ Quick-Scan Records (buffered reads)

Loading Method: Page

☑ Accept browse control from Toolbar

Locator Behavior

Record Filter:

Range Limit Field: DTL:OrderNumber

Range Limit Type: File Relationship

Range limiting file

Related file: Orders

Additional Sort Fields:

Reset Fields | Scroll Bar Behavior

Files | Embeds

Font... | OK | Cancel | Help

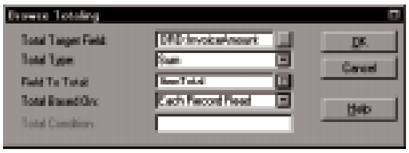
Add an order invoice total calculation

Now we want to calculate the order total and save it in the Orders file.

1. **Press** the small right arrow button just to the right of the Hot Fields tab to scroll the tabs, then **select** the Totaling tab when it appears.
2. **Press** the Insert button.
3. **Press** the ellipsis (...) button for the Total Target Field.
4. **Highlight** the *Orders* file in the Files list, select *ORD:InvoiceAmount* from the Fields list, then **press** the Select button.

   This is the field that will receive the result of the total calculation.
5. **Choose** Sum from the Total Type drop down list.
6. **Press** the ellipsis (...) button for the Field to Total.
7. **Highlight** *LOCAL DATA UpdateOrder* in the Files list, **select** *ItemTotal* from the Fields list, then **press** the Select button.

   This is the field whose contents will be used in the total calculation. So far we've only declared this field and not done anything to put any value into it, but we'll get to that very soon.
8. **Choose** *Each Record Read* from the Total Based On dropdown list.



Total Target Field: DTD:InvoiceAmount
Total Type: Sum
Field To Total: ItemTotal
Total Based On: Each Record Read
Total Condition:

9. **Press** OK to close the **Browse Totaling** dialog.

Change the object name

Now we want to change the name of the browse object to make our code more readable (you'll see why a little later).

1. **Select** the **Classes** tab.
2. **Type** *BrowseDTL* in the **Object Name** field.

Add horizontal and vertical scroll bars

1. **Select** the **Extra** tab.
2. **Check** the **Horizontal** and **Vertical** boxes.
3. **Press** OK to close the **List Properties** dialog.

Add the standard file update buttons

1. **Choose** **Populate ➤ Control Template**, or CLICK on the Control Template tool in the **Controls** toolbox (last tool icon on the right, bottom row).
2. In the **Select Control Template** dialog, **select** the *BrowseUpdateButtons* control template, then **press** the **Select** button.

   The cursor changes to a crosshair and "little book."
3. CLICK below the list box.

   The **Insert**, **Change**, and **Delete** buttons all appear together.
4. RIGHT-CLICK on the **Delete** button, then **choose** **Actions** from the popup menu.
5. **Check** the **Use Edit in place** box.

   Checking this box for one button in the Control template checks it for all three. We will be using the Edit in Place technique to update the *Details* file records instead of an update Form procedure. This will allow us to demonstrate some fairly advanced programming techniques and show just how easy they are to perform within the Application Generator.
6. **Press** the OK button.

Add a "display-only" control for the invoice total

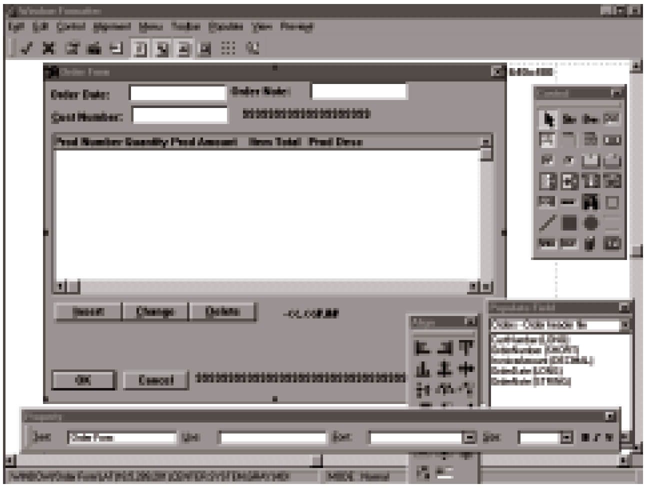
1. **Choose** **Control ä String**, or CLICK on the *String* tool in the **Controls** toolbox.
2. CLICK below the bottom right corner of the list box.

3. RIGHT-CLICK the control you just placed, and **select** **Properties** from the popup menu.
4. **Check** the **Variable String** box.

   This specifies the control will display data from a variable, not just a string constant. The **Select Field** dialog automatically appears.
5. **Highlight** the *Orders* file in the **Files** list, then **select** *ORD:InvoiceAmount* from the **Fields** list and **press** the **Select** button.
6. **Press** the **OK** button.

Change the form window caption and exit the Window Formatter

1. CLICK on the caption bar of the sample window.
2. **Type** *Order Form* in the **Text** field in the floating **Property** toolbox, then **press** TAB.



3. **Choose** **Exit!** to close the Window Formatter.

Making it all Work

There are a couple of things we need to do to make this procedure fully functional—add a Formula, and configure the Edit in Place.

Using the Formula Editor

To make the ItemTotal calculate the correct amount for each Detail record in the browse list box, you need to add a Formula to the procedure. This will also allow the browse totaling to correctly place the invoice total in the ORD:InvoiceAmount field.

1. **Press** the **Formulas** button in the **Procedure Properties** dialog.

   The **Formulas** dialog appears. This dialogs lists all the formulas in the procedure.

2. **Press** the **New** button to add a new formula.

   The **Formula Editor** dialog appears.

3. **Type** *Item Total Formula* in the **Name** field.

4. **Press** the ellipsis (...) button in the **Class** Field.

5. **Highlight** *Format Browse* in the **Template Classes** list, then **press** the **OK** button.

   The **Class** field simply specifies the logical position within the generated source code at which the formula is calculated. The *Format Browse* class tells the *BrowseBox* Control template to perform the calculation each time it formats a record for display in the list box.

6. **Press** the ellipsis (...) button in the **Result** Field.

7. **Highlight** *LOCAL DATA UpdateOrder* in the **Files** list, **select** *ItemTotal* from the **Fields** list, then **press** the **Select** button.

   This names the field that will receive the result of the calculation. This is the field we defined earlier through the List Box Formatter.

8. **Press** the **Data** button in the **Operands** group.

9. **Highlight** the *Detail* file in the **Files** list, **select** *DTL:Quantity* from the **Fields** list, then **press** the **Select** button.

   This places *DTL:Quantity* into the **Statement** field for you. The **Statement** field contains the expression being built. You can type directly into the **Statement** field to build the expression, if you wish.

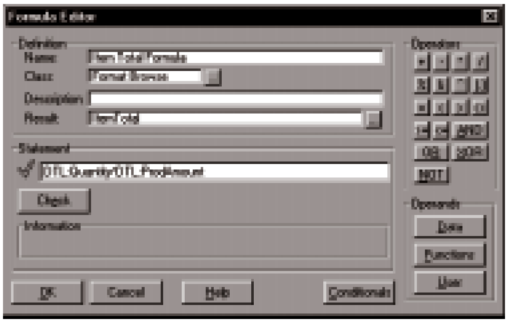
10. **Press** the ***** button in the **Operators** group.

    This is the multiplication operator.

11. **Press** the **Data** button in the **Operands** group.

12. **Highlight** the *Detail* file in the **Files** list, **select** *DTL:ProdAmount* from the **Fields** list, then **press** the **Select** button.
13. **Press** the **Check** button to validate the expression's syntax.

    A green checkmark appears left of the statement, indicating the syntax is correct. If a red X appears, the expression's syntax is incorrect and the highlighted portion of the statement is what you must change.



Name: ItemTotalFormula
Class: Format Browse
Result: ItemTotal
Statement: DTL:Quantity*DTL:ProdAmount

14. **Press** the **OK** button to close the Formula Editor.

    The **Formulas** dialog re-appears.
15. **Press** the **OK** button to close the **Formulas** dialog.

Configuring Edit in Place

Now we need to configure the Edit in Place characteristics. We previously used Edit in Place for the Phones file and simply took all the default behaviors, because that was a fairly simple file. However, now we're editing a line item *Detail* record for an order entry system, which means we need to do some data entry validation beyond simply ensuring the user types in a number that fits the display picture. To do this, we'll need to extend the simple Edit in Place functionality provided by the Application Builder Class (ABC) Library.

Set Column Specific Classes

1. **Press** the **Extensions** button in the **Procedure Properties** dialog.
2. **Highlight** *Update a Record from Browse Box on Detail* then **press** the **Configure Edit in place** button.

   The **Configure Edit in place** dialog apears. The options on this dialog allow you to specify what behavior occurs while the user is editing data and presses ENTER or an arrow key, along with several save options. We'll accept all the defaults.
3. **Press** the **Column Specific** button.
4. **Press** the **Insert** button.

5. **Press** the ellipsis (...) button for the **Field** field.

6. **Highlight** *DTL:ProdNumber*, then **press** the **Select** button.

   The ABC Library contains an object class called EditEntryClass that is the default Edit in Place class. We're going to override some of the methods of the default class for this column so we can modify the default behavior. Adding this field to the **Column Specific** list of fields is what allows us to override the default Edit in Place methods for this one field. We will do this so that we can perform data validation on this field when the user enters data—we want to make sure that they can only enter a number that refers to a valid *Products* file record.

7. **Press** the **OK** button.

8. **Press** the **Insert** button.

9. **Press** the ellipsis (...) button for the **Field** field.

10. **Highlight** *DTL:Quantity*, then **press** the **Select** button.

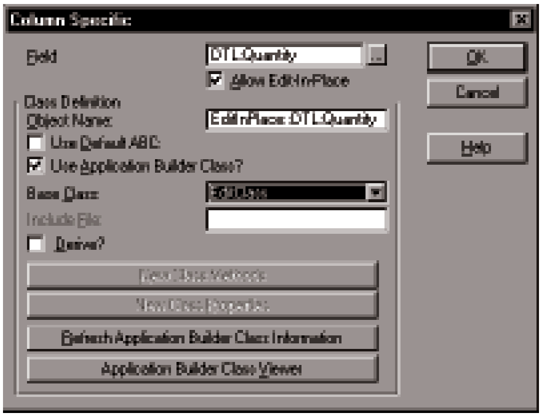
    The ABC Library's EditEntryClass defaults to using an ENTRY control, and for this field we want to use a SPIN control so the user can just spin to the quantity they want to order. Therefore, we need to override some methods for this column too, to have a SPIN instead of an ENTRY control.

15. **Clear** the **Use Default ABC** check box.

    This allows you to specify the exact class from which to derive.

15. **Choose** *EditClass* from the **Base Class** droplist.

    The EditEntryClass does too much that's unnecessary for this, since we're going to override the methods anyway. Therefore, we're going to derive and override from the Base Class all the Edit in place classes were derived from: EditClass.


Column Specific
OK
Cancel
Help
New Class Methods
New Class Properties
Refresh Application Builder Class Information
Application Builder Class Viewer

Field: DTL:Quantity
[✓] Allow Edit-in-Place
Class Definition
Object Name: EditInPlace::DTL:Quantity
[ ] Use Default ABC:
[✓] Use Application Builder Class?
Base Class: EditClass
Include File:
[ ] Derive?

11. **Press** the **OK** button.

12. **Press** the **Insert** button.

13. **Press** the ellipsis (...) button for the **Field** field.

14. **Highlight** *DTL:ProdAmount*, then **press** the **Select** button.

15. **Clear** the **Allow Edit in Place** check box.

    This turns off Edit in Place for this one column of the list box.

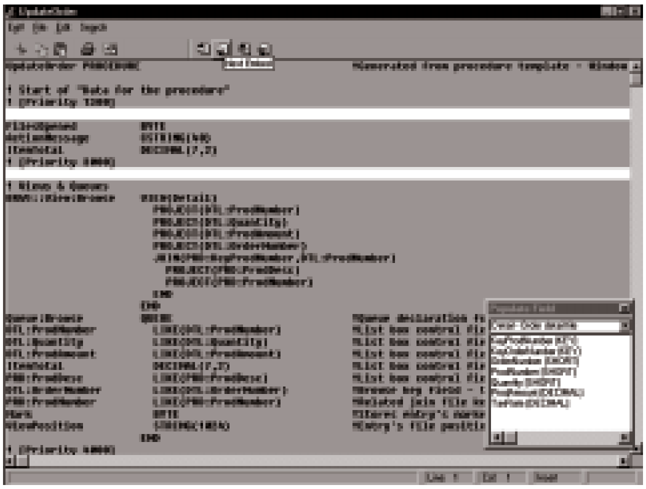
    Only fields in the Primary file for the *BrowseBox* can be edited in place, and the default is that all Primary file fields are editable. In this case, this means all the *Detail* file fields can be edited and the *Products* file fields cannot be edited. However, for this procedure we do NOT want the user to be able to edit the *DTL:ProdAmount* field because we're going to get its value directly from the Products file and we don't want the user to be able to change it. That's why we turned off the **Allow Edit in Place** box.

16. **Press** the **OK** button 5 times to return all the way back to the **Application Tree**.

Using the Embeditor

1. RIGHT-CLICK on *UpdateOrder* and **select** **Source** from the popup menu.

   The **Source** popup menu selection opens the "Embeditor"—the third method of accessing embed points in a procedure. The Embeditor is the same Text Editor you've already used, but opened in a special mode which allows you to not only to edit all the embed points in your procedure, but to edit them within the context of template-generated code. The source code looks like this:



```
UpdateOrder PROCEDURE                                   !Generated from procedure template - Window

! Start of "Data for the procedure"
! [Priority 1300]

FilesOpened          BYTE
ActionMessage        CSTRING(40)
ItemTotal            DECIMAL(7,2)
! [Priority 6000]

! Views & Queues
BRW5::View:Browse    VIEW(Detail)
                       PROJECT(DTL:ProdNumber)
                       PROJECT(DTL:Quantity)
                       PROJECT(DTL:ProdAmount)
                       PROJECT(DTL:OrderNumber)
                       JOIN(PRO:KeyProdNumber,DTL:ProdNumber)
                         PROJECT(PRO:ProdDesc)
                         PROJECT(PRO:ProdNumber)
                       END
                     END
Queue:Browse         QUEUE                              !Queue declaration fo
DTL:ProdNumber         LIKE(DTL:ProdNumber)             !List box control fie
DTL:Quantity           LIKE(DTL:Quantity)               !List box control fie
DTL:ProdAmount         LIKE(DTL:ProdAmount)             !List box control fie
ItemTotal              DECIMAL(7,2)                     !List box control fie
PRO:ProdDesc           LIKE(PRO:ProdDesc)               !List box control fie
DTL:OrderNumber        LIKE(DTL:OrderNumber)            !Browse key field - t
PRO:ProdNumber         LIKE(PRO:ProdNumber)             !Related join file ke
Mark                   BYTE                             !Entry's marke
ViewPosition           STRING(1024)                     !Entry's file positio
                     END
! [Priority 4000]
```

   Notice that most of the code is on a gray background and the points where you can write your code have a white background. There are also identifying comments for each embed point. You can turn these comments on and off as you choose through the **Setup ➤ Application Options** dialog. Once you become familiar with them, you'll probably want to turn them off so you can see more of the actual code.

You'll notice that a message briefly appeared that said "Generating tutorial." The Embeditor displays *all possible* embed points for the procedure within the context of *all* the *possible* code that *may* be generated for the procedure. Notice the distinction here—**Embeditor does not show you the code that will be generated, but all the code which could be generated for you**, if you chose every possible option and placed code into every available embed point. You are not likely to ever do that. Therefore, a lot more generated code shows up in the Embeditor than will actually be in the generated code when you compile your application. After we finish here, we'll go look at the generated code to see the difference.

At the right end of the toolbar are four buttons which are very important to know when working in the Embeditor. These are (from left to right) the **Previous Embed, Next Embed, Previous Filled Embed,** and **Next Filled Embed** buttons (hover your mouse over them and the tooltips appear naming the buttons). They allow you to quickly get from one embed point to another—particularly after you've written code into some of them.

Detecting Changed Orders

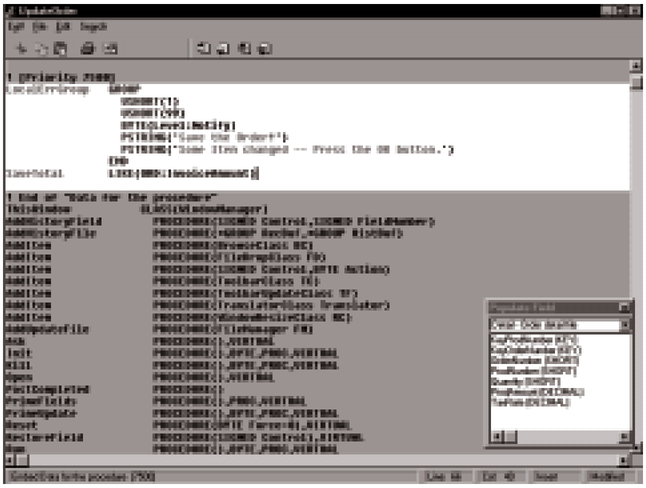
One of the things we want this procedure to do is to detect changes to existing orders and make sure the changes do not result in a data mis-match between the *Orders* and *Detail* files. This system is storing the total dollar amount of an order in the ORD:InvoiceAmount field, so when the user changes a Detail item in an existing Order, we want to make sure the Orders file record is updated, too. There's fairly simple way to do that which will allow us to demonstrate the ABC Library's flexible error handling.

1. **CLICK** on the **Next embed** button (about 5 times) until you get to the embed point immediately preceding the line of code reading `ThisWindow CLASS(WindowManager)` (this should be at **[Priority 7500]**).

   Each embed point potentially has 10,000 priority levels within it. This Embed code Priority level system is designed to allow you to embed your code before or after any generated code—whether that code is generated for you by Clarion's ABC Templates or any third-party templates you choose to use. This makes the embed system completely flexible, allowing you to add your own code at any logical point needed—before or after most any "chunk" of generated code.

2. **Type** the following code:

```
LocalErrGroup GROUP
                USHORT(1)
                USHORT(99)
                BYTE(Level:Notify)
                PSTRING('Save the Order!')
                PSTRING('Some Item changed -- Press the OK button.')
              END
SaveTotal     LIKE(ORD:InvoiceAmount)
```



```
! [Priority 7500]
LocalErrGroup    GROUP
                   USHORT(1)
                   USHORT(99)
                   BYTE(Level:Notify)
                   PSTRING('Save the Order?')
                   PSTRING('Some Item changed -- Press the OK button.')
                 END
SaveTotal        LIKE(ORD:InvoiceAmount)
! End of "Data for the procedure"
ThisWindow           CLASS(WindowManager)
AddHistoryField        PROCEDURE(SIGNED Control,SIGNED FieldNumber)
AddHistoryFile         PROCEDURE(*GROUP RecBuf,*GROUP HistBuf)
AddItem                PROCEDURE(BrowseClass BC)
AddItem                PROCEDURE(FileDropClass FD)
AddItem                PROCEDURE(SIGNED Control,BYTE Action)
AddItem                PROCEDURE(ToolbarClass TC)
AddItem                PROCEDURE(ToolbarUpdateClass TF)
AddItem                PROCEDURE(TranslatorClass Translator)
AddItem                PROCEDURE(WindowResizeClass RC)
AddUpdateFile          PROCEDURE(FileManager FM)
Ask                    PROCEDURE(),VIRTUAL
Init                   PROCEDURE(),BYTE,PROC,VIRTUAL
Kill                   PROCEDURE(),BYTE,PROC,VIRTUAL
Open                   PROCEDURE(),VIRTUAL
PostCompleted          PROCEDURE()
PrimeFields            PROCEDURE(),PROC,VIRTUAL
PrimeUpdate            PROCEDURE(),BYTE,PROC,VIRTUAL
Reset                  PROCEDURE(BYTE Force=0),VIRTUAL
RestoreField           PROCEDURE(SIGNED Control),VIRTUAL
Run                    PROCEDURE(),BYTE,PROC,VIRTUAL
```

Clarion's ABC (Application Builder Class) Templates generate Object Oriented code for you using the ABC Library. The ABC Library contains an error handling class called ErrorClass. This bit of code declares a **LocalErrGroup GROUP** (in exactly the form that the ErrorClass requires—see the *Application Handbook*) containing a "custom" error number and message that we are defining for use by the ErrorClass object in our application. The **SaveTotal** declaration is a local variable which is defined LIKE (always has the same data type) the ORD:InvoiceAmount field. We'll use this variable to hold the starting order total when the user is updating an existing order.

3. **Choose Search ➤ Find** to bring up the **Find** dialog.

4. **Type** *ThisWindow.Init* into the **Find what** field, then **press** the **Find next** button.

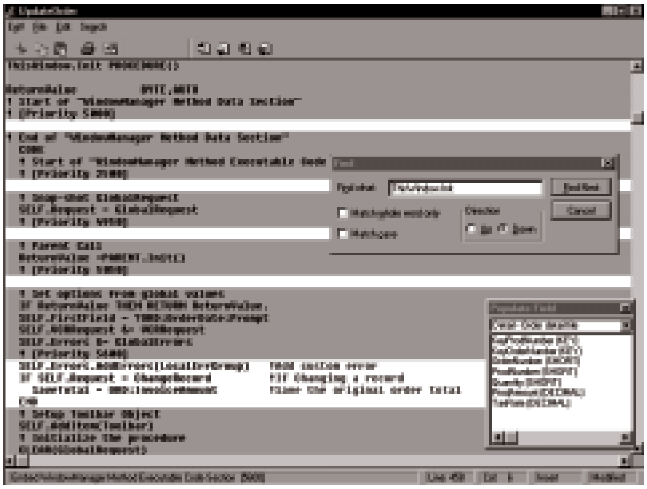
   This takes you directly to the ThisWindow.Init method.

5. In the embed point immediately following the line of code reading `SELF.Errors &= GlobalErrors` (this should be at **[Priority 5600]**), **type** the following code:

```
SELF.Errors.AddErrors(LocalErrGroup)      !Add custom error
IF SELF.Request = ChangeRecord            !If Changing a record
  SaveTotal = ORD:InvoiceAmount           !Save the original order total
END
```

   This code calls the **AddErrors** method of the **GlobalErrors** object to add the **LocalErrGroup** to the list of available errors that the object handles. The **GlobalErrors** object is an instance of the ErrorClass which the ABC Templates declare globally to handle all error conditions in the application. Adding our **LocalErrGroup** enables the **GlobalErrors** object to handle our "custom" error condition. This demonstrates the flexibility of Clarion's ABC Library. The **IF** statement detects when the user is editing an existing order and saves the original order total.

This embed point is in the **ThisWindow.Init PROCEDURE** which performs some necessary initialization tasks. This is a virtual method of the **ThisWindow** object. **ThisWindow** is the object which handles all the window and control handling code.



```
ThisWindow.Init PROCEDURE()

ReturnValue          BYTE,AUTO
! Start of "WindowManager Method Data Section"
! [Priority 5000]

! End of "WindowManager Method Data Section"
  CODE
  ! Start of "WindowManager Method Executable Code
  ! [Priority 2500]

  ! Snap-shot GlobalRequest
  SELF.Request = GlobalRequest
  ! [Priority 4950]

  ! Parent Call
  ReturnValue =PARENT.Init()
  ! [Priority 5000]

  ! Set options from global values
  IF ReturnValue THEN RETURN ReturnValue.
  SELF.FirstField = ?ORD:OrderDate:Prompt
  SELF.VCRRequest &= VCRRequest
  SELF.Errors &= GlobalErrors
  ! [Priority 5600]
  SELF.Errors.AddErrors(LocalErrGroup)      !Add custom error
  IF SELF.Request = ChangeRecord            !If Changing a record
    SaveTotal = ORD:InvoiceAmount           !Save the original order total
  END
  ! Setup Toolbar Object
  SELF.AddItem(Toolbar)
  ! Initialize the procedure
  [illegible](GlobalRequest)
```

You may not have noticed, but the ABC Templates generate exactly one line of executable source code within the **UpdateOrder PROCEDURE** itself (**GlobalResponse = ThisWindow.Run**) so *all* of the functionality of the **UpdateOrder PROCEDURE** actually occurs in object methods—either virtual methods specific to the **UpdateOrder PROCEDURE** itself or standard ABC Library methods. This is true of every ABC Template generated procedure. Generating fully Object Oriented code makes the code generated for you very tight and efficient—only the code that actually needs to be different for an individual procedure is handled differently. Everything else is standard code that exists in only one place and has been tested and debugged to ensure consistent performance.

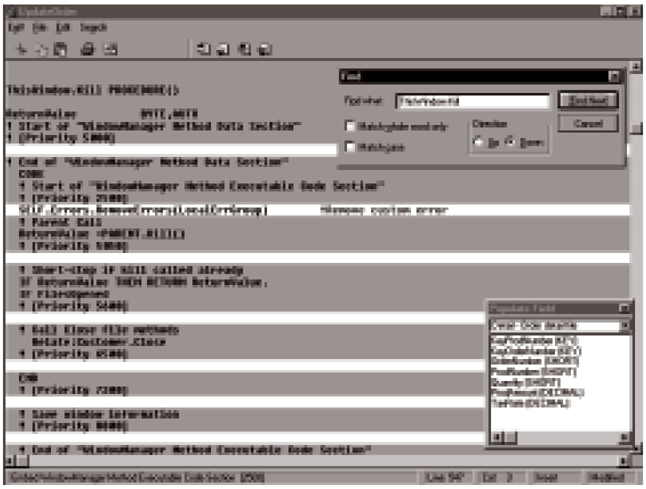
Object Oriented Programming (OOP) in Clarion starts with the CLASS structure. See CLASS in the *Language Reference* for a discussion of OOP syntax. The *Programmer's Guide* contains several articles which discuss OOP in depth, and the *Application Handbook* fully documents Clarion's Application Builder Class (ABC) Library.

6. **Type** *ThisWindow.Kill* into the **Find what** field, then **press** the **Find next** button.

7. In the embed point immediately following the line of code reading `CODE` (this should be **[Priority 2500]**), **type** the following code:

```
SELF.Errors.RemoveErrors(LocalErrGroup)   !Remove custom error
```

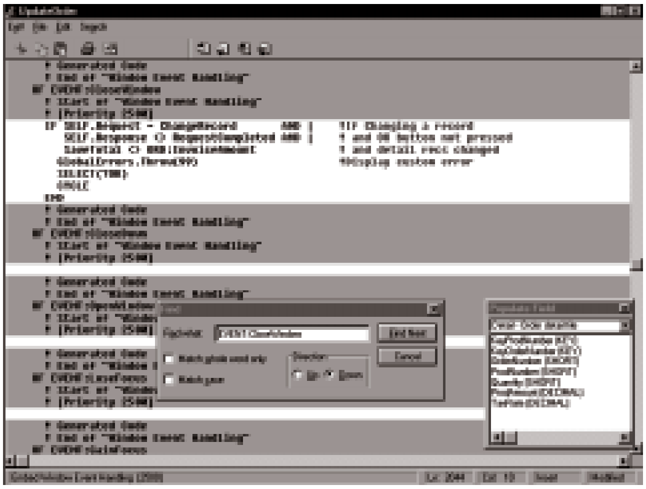
This calls the ABC Library method to remove our "custom" error. The **ThisWindow.Kill** method is a "cleanup" procedure (performs necessary exit tasks) which executes when the user is finished working in the UpdateOrder procedure, so the error is no longer needed at that point.



8. **Type** *EVENT:CloseWindow* into the **Find what** field, then **press** the **Find next** button.

9. In the embed point immediately following the line of code reading `OF EVENT:CloseWindow` (this should be **[Priority 2500]**), **type** the following code:

```
IF  SELF.Request = ChangeRecord       AND | !If Changing a record
    SELF.Response <> RequestCompleted AND | ! and OK button not pressed
    SaveTotal <> ORD:InvoiceAmount          ! and detail recs changed
  GlobalErrors.Throw(99)                    !Display custom error
  SELECT(?OK)                               ! then select the OK button
  CYCLE
END
```



This is the code that will detect any attempt by the user to exit the UpdateOrder procedure without saving the *Orders* file record after they've changed an existing order. Note the vertical bar characters (|) at the end of the first two lines of code. These are absolutely necessary. Vertical bar (|) is the Clarion language line continuation character. This means that the first three lines of this code are a single logical statement which evaluates three separate conditions and will only execute the **GlobalErrors.Throw(99)** statement if all three conditions are true.

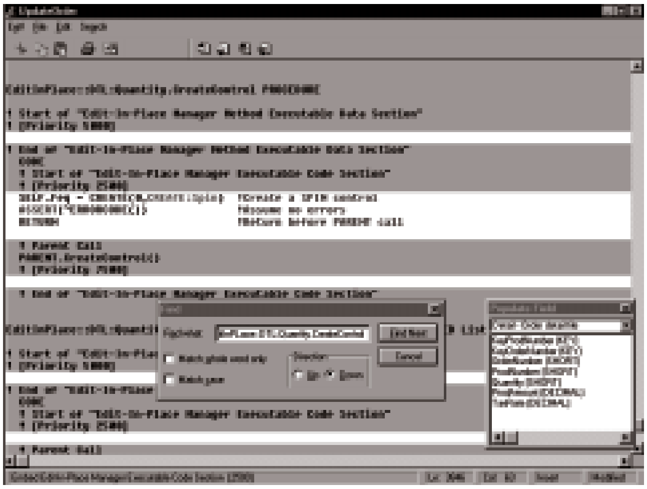
Overriding the Edit in Place Classes

OK, now you've seen an example of how you can *use* the ABC Library in your own embedded source code. Now we'll show you how to *override* a class to provide custom functionality that the ABC Library does not provide. The CLASS declarations for the objects that we named through the **Configure Edit in Place** dialogs are generated for you by the ABC Templates. These CLASSes are both derived from the **EditClass** ABC Library class.

1. **Type** *EditInPlace::DTL:Quantity.CreateControl* into the **Find what** field, then **press** the **Find next** button.

2. In the embed point immediately following the line of code reading `CODE` (this should be **[Priority 2500]**), **type** the following code:

```
SELF.Feq = CREATE(0,CREATE:Spin)      !Create a SPIN control
ASSERT(~ERRORCODE())                  !Assume no errors
RETURN  !Return before PARENT call
```



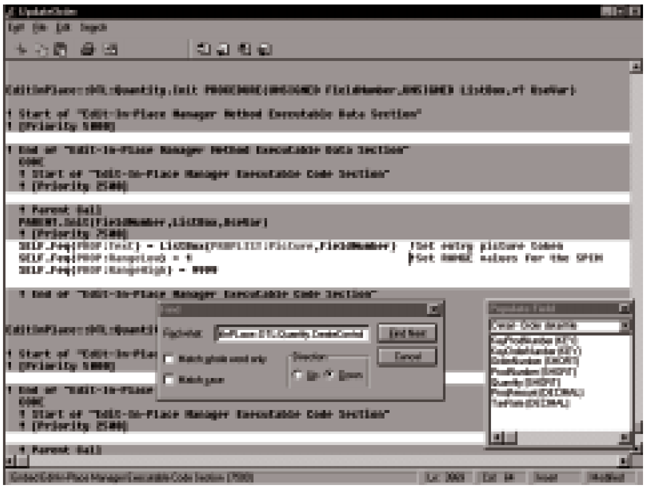
```
EditInPlace::DTL:Quantity.CreateControl PROCEDURE

! Start of "Edit-In-Place Manager Method Executable Data Section"
! [Priority 5000]

! End of "Edit-In-Place Manager Method Executable Data Section"
  CODE
  ! Start of "Edit-In-Place Manager Executable Code Section"
  ! [Priority 2500]
  SELF.Feq = CREATE(0,CREATE:Spin)   !Create a SPIN control
  ASSERT(~ERRORCODE())               !Assume no errors
  RETURN                             !Return before PARENT call

  ! Parent Call
  PARENT.CreateControl()
  ! [Priority 7500]

  ! End of "Edit-In-Place Manager Executable Code Section"

EditInPlace::DTL:Quantit[illegible]
! Start of "Edit-In-Pla[illegible]
! [Priority 5000]

! End of "Edit-In-Place[illegible]
  CODE
  ! Start of "Edit-In-Place Manager Executable Code Section"
  ! [Priority 2500]

  ! Parent Call
```

This code simply creates the SPIN control and assumes there'll be no errors. The RETURN statement is required because we need to ensure the PARENT method call does not take place.

3. **Scroll** to the **EditInPlace::DTL:Quantity.Init** method (this should immediately follow the **EditInPlace::DTL:Quantity.CreateControl** method).

4. In the embed point immediately following the line of code reading `PARENT.Init(FieldNumber,Listbox,UseVar)` (this should be **[Priority 7500]**), **type** the following code:

```
SELF.Feq{PROP:Text} = ListBox{PROPLIST:Picture,FieldNumber}
                                      !Set entry picture token
SELF.Feq{PROP:RangeLow} = 1           !Set RANGE values for the SPIN
SELF.Feq{PROP:RangeHigh} = 9999
```



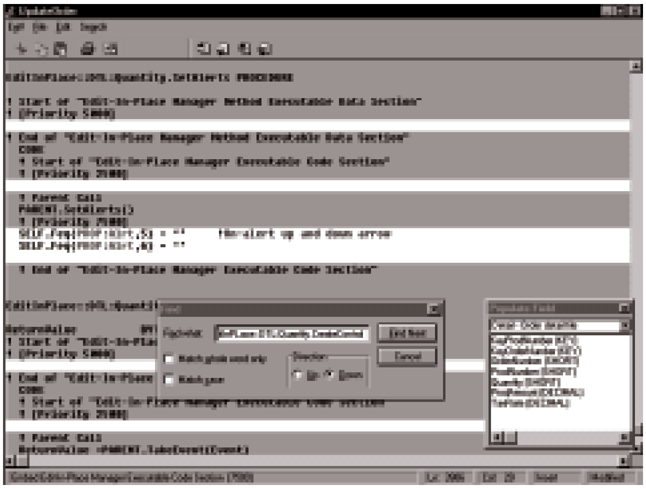
```
EditInPlace::DTL:Quantity.Init PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar)

! Start of "Edit-In-Place Manager Method Executable Data Section"
! [Priority 5000]

! End of "Edit-In-Place Manager Method Executable Data Section"
  CODE
  ! Start of "Edit-In-Place Manager Executable Code Section"
  ! [Priority 2500]

  ! Parent Call
  PARENT.Init(FieldNumber,ListBox,UseVar)
  ! [Priority 7500]
  SELF.Feq{PROP:Text} = ListBox{PROPLIST:Picture,FieldNumber}  !Set entry picture token
  SELF.Feq{PROP:RangeLow} = 1                                  !Set RANGE values for the SPIN
  SELF.Feq{PROP:RangeHigh} = 9999

  ! End of "Edit-In-Place Manager Executable Code Section"

EditInPlace::DTL:Quanti

! Start of "Edit-In-Pla
! [Priority 5000]

! End of "Edit-In-Place
  CODE
  ! Start of "Edit-In-Place Manager Executable Code Section"
  ! [Priority 2500]

  ! Parent Call
```

This code sets the data entry picture token and range of valid data for the SPIN control.

5. **Scroll** to the **EditInPlace::DTL:Quantity.SetAlerts** method (this should immediately follow the **EditInPlace::DTL:Quantity.Kill** method).

6. In the embed point immediately following the line of code reading `PARENT.SetAlerts()` (this should be **[Priority 7500]**), **type** the following code:

```
SELF.Feq{PROP:Alrt,5} = ‘’       !Un-alert up and down arrow
SELF.Feq{PROP:Alrt,6} = ‘’
```



```
EditInPlace::DTL:Quantity.SetAlerts PROCEDURE

! Start of "Edit-In-Place Manager Method Executable Data Section"
! [Priority 5000]

! End of "Edit-In-Place Manager Method Executable Data Section"
  CODE
  ! Start of "Edit-In-Place Manager Executable Code Section"
  ! [Priority 2500]

  ! Parent Call
  PARENT.SetAlerts()
  ! [Priority 7500]
  SELF.Feq{PROP:Alrt,5} = ''      !Un-alert up and down arrow
  SELF.Feq{PROP:Alrt,6} = ''

  ! End of "Edit-In-Place Manager Executable Code Section"

EditInPlace::DTL:Quanti

ReturnValue          BY
! Start of "Edit-In-Pla
! [Priority 5000]

! End of "Edit-In-Place
  CODE
  ! Start of "Edit-In-Place Manager Executable Code Section"
  ! [Priority 2500]

  ! Parent Call
  ReturnValue =PARENT.TakeEvent(Event)
```

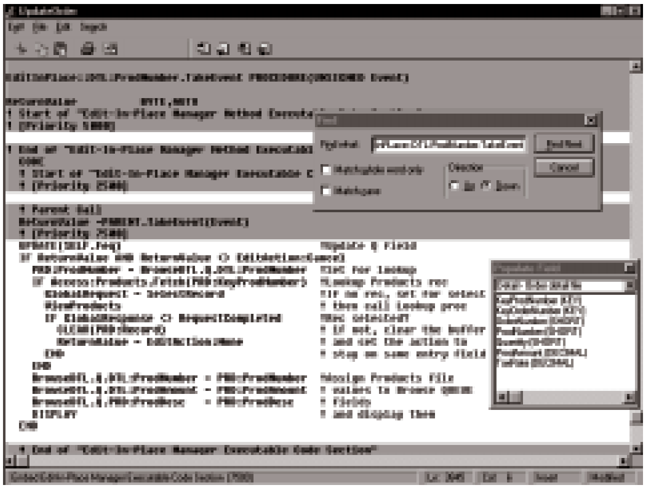
This code “un-alerts” two of the standard keystrokes the EditClass **SetAlerts** method alerts by default (in the PARENT method call). TheEditClass alerts the up and down arrow keys, and the SPIN control uses these keys to operate its spin buttons. That’s why we need to “un-alert” them.

If you wish to see the base class code that we’ve overridden, open the \CLARION5\LIBSRC\ABEIP.CLW file and search for EditClass.

7. **Type** *EditInPlace::DTL:ProdNumber.TakeEvent* into the **Find what** field, then **press** the **Find next** button.

8. In the embed point immediately following the line of code reading `ReturnValue = PARENT.TakeEvent(Event)` (this should be **[Priority 7500]**), **type** the following code:

```
UPDATE(SELF.Feq)                                !Update Q field
IF ReturnValue AND ReturnValue <> EditAction:Cancel
  PRD:ProdNumber = BrowseDTL.Q.DTL:ProdNumber  !Set for lookup
  IF Access:Products.Fetch(PRD:KeyProdNumber)  !Lookup Products rec
    GlobalRequest = SelectRecord                !If no rec, set for select
    ViewProducts                                ! then call Lookup proc
    IF GlobalResponse <> RequestCompleted       !Rec selected?
      CLEAR(PRD:Record)                         ! if not, clear the buffer
      ReturnValue = EditAction:None             ! and set the action to
    END                                         ! stay on same entry field
  END
  BrowseDTL.Q.DTL:ProdNumber = PRD:ProdNumber  !Assign Products file
  BrowseDTL.Q.DTL:ProdAmount = PRD:ProdAmount  ! values to Browse QUEUE
  BrowseDTL.Q.PRD:ProdDesc   = PRD:ProdDesc    ! fields
  DISPLAY                                       ! and display them
END
```



This is the really interesting code. Notice that the first executable statement (generated for you) is a call to the **PARENT.TakeEvent** method. This calls the **EditClass.TakeEvent** method we're overriding so it can do what it usually does (code very similar to the TakeEvent code you already wrote for your DTL:Quantity SPIN control).

All the rest of the code is there to give this derived class extra functionality not present in its parent class. This is the real power of OOP—if you want everything the parent does, plus a little bit more you don't have to duplicate all the code the parent executes, you just call it. In this case, all the extra code is to perform some standard data entry validation tasks. This code will verify whether the user typed in a valid Product Number and if they didn't, it will call the ViewProducts procedure to allow them to choose from the list of products.

Overriding Methods in the Embeditor

There is a very important point to understand about working in the Embeditor when you are overriding methods: *as soon as you type anything into an embed point in an overridable method, you have overridden it*. Even a simple ! comment line makes this happen, because the Application Generator notes that you have written some of your own code, and so generates the proper method prototype into the local CLASS declaration for you. To prevent you from accidentally adding a comment that causes the method override which consequently “breaks” the functionality, the ABC Templates automatically generate PARENT method calls and RETURN statements for you, as appropriate.

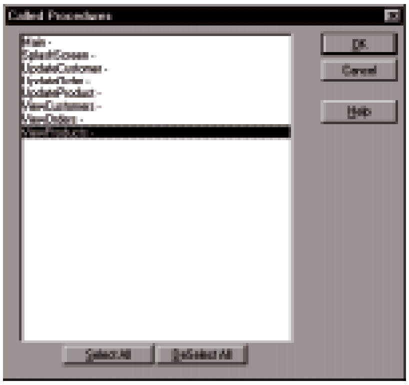
You’ll notice that all of our overridden methods contained generated calls to their PARENT method, and the TakeEvent method also had a generated RETURN statement. Sometimes you want these statements to execute, and sometimes you don’t (usually, you do). For those cases where you do *not* want them to execute, simply write your code in the embed point which comes *before* the PARENT method call and write your own RETURN statement at the end of your code (as we did at the end of the code we wrote for **EditInPlace::DTL:Quantity.CreateControl** method). This means that the generated PARENT method call will never execute. Clarion’s optimizing compiler is smart enough to recognize that these statements can never execute and optimizes them out of compiled object code.

1. **Choose** **Exit!** to close the Embeditor.

Update the Procedure Call Tree

The **EditInPlace::DTL:ProdNumber.TakeEvent** method calls the ViewProducts procedure fom within its code. Since this is just embedded source code, the Application Generator doesn’t know you’ve called this procedure, and needs to be told (if you don’t, you’ll get compiler errors), so it can generate the correct MAP structure for the module containing this procedure.

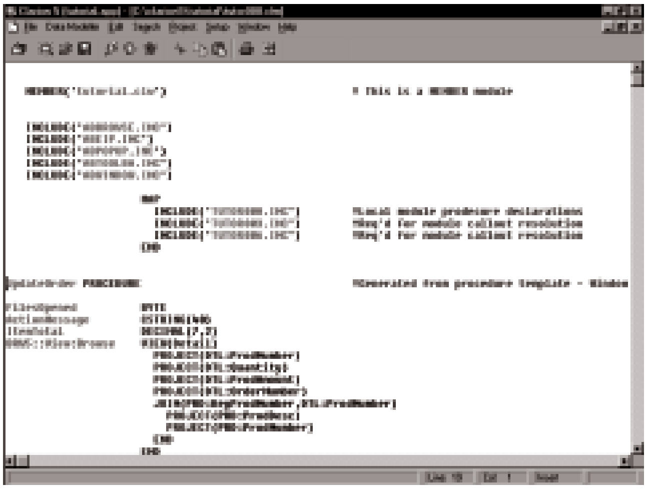
1. **RIGHT-CLICK** on *UpdateOrder* and **select** **Procedures** from the popup menu.
2. **Highlight** *ViewProducts* then **press** the **OK** button.



3. **Choose** **File ➤ Save**, or **press** the *Save* button to save your work.

Generate code

1. **Choose** **Project ➤ Generate** to generate the source code for your application.
2. **RIGHT-CLICK** on *UpdateOrder* and **select** **Module** from the popup menu.



```
  MEMBER('tutorial.clw')                           ! This is a MEMBER module


  INCLUDE('ABBROWSE.INC')
  INCLUDE('ABEIP.INC')
  INCLUDE('ABPOPUP.INC')
  INCLUDE('ABTOOLBA.INC')
  INCLUDE('ABWINDOW.INC')

                   MAP
                     INCLUDE('TUTOR008.INC')       !Local module procedure declarations
                     INCLUDE('TUTOR001.INC')       !Req'd for module callout resolution
                     INCLUDE('TUTOR006.INC')       !Req'd for module callout resolution
                   END


UpdateOrder PROCEDURE                              !Generated from procedure template - Window

FilesOpened        BYTE
ActionMessage      CSTRING(40)
ItemTotal          DECIMAL(7,2)
BRW5::View:Browse  VIEW(Detail)
                     PROJECT(DTL:ProdNumber)
                     PROJECT(DTL:Quantity)
                     PROJECT(DTL:ProdAmount)
                     PROJECT(DTL:OrderNumber)
                     JOIN(PRD:KeyProdNumber,DTL:ProdNumber)
                       PROJECT(PRD:ProdDesc)
                       PROJECT(PRD:ProdNumber)
                     END
                   END
```

The Text Editor appears containing the generated source code for your UpdateOrder procedure. Notice that there is a lot less code here than there was in the Embeditor. All that generated code in the Embeditor was there to provide you with context, and to provide you with embed points with which to override methods, should you need to. However, Clarion's Application Generator and ABC Templates are smart enough to only generate the code you actually need, when you actually need it.

3. **Choose** **Exit!** to close the Text Editor.

Now might be a good time to try out your application. You've got all the data entry portions completed and the only things left to do now are the reports, which we'll get to in the next lesson.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- You created a new Form procedure.
- You created a "Scrolling-Form" metaphor Edit-in-place browsebox to update the Detail file.
- You created a total field to total up the order.
- You created a Formula to total each line item in the order.

- You used the Embeditor to write your embedded source code within the context of template-generated code.
- You used the power of OOP to extend the standard error handling functionality of the ABC Library.
- You used the power of OOP to derive and override the Edit-in-place classes to extend the standard functionality of the ABC Library.
- You generated source code to compare the difference between the code shown in the Embeditor to that which is actually generated.

We’re almost finished with this application. In the next lesson we’ll create the application’s reports .

12 – CREATING REPORTS

A Simple Customer List Report

The last item to cover in this tutorial is adding reports. First, we'll create a simple customer list to introduce you to the Report Formatter. Then we'll create an Invoice Report to demonstrate how you can easily create Relational reports with multi-level group breaks, group totals, and page formatting. Then we'll copy the Invoice Report and limit the copy to print invoice for only one customer at a time.

Starting Point:
The TUTORIAL.APP file should be open.

Updating the Main Menu

First, we need to add menu selections so the user can call the reports, and so the Application Generator will call the appropriate "ToDo" procedures.

Add a menu item

1. RIGHT-CLICK on the *Main* procedure in the **Application Tree** dialog and **choose** **Window** from the popup menu.
2. **Choose** **Menu ➤ Edit Menu** from the Window Formatter's menu.
3. **Highlight** the **P&rint Setup** item in the **Menu Editor** list.
4. **Press** the **Item** button.
5. **Type** *Print &Customer List* in the **Menu Text** field then **press** TAB.

Specify the new item's action

1. **Select** the **Actions** tab.
2. **Choose** *Call a Procedure* from the **When Pressed** drop down list.
3. **Type** *CustReport* in the **Procedure Name** field.
4. **Check** the **Initiate Thread** box.

Add a second menu item

1. **Press** the **Item** button.

2. **Type** *Print &All Invoices* in the **Menu Text** field.
3. **Select** the **Actions** tab.
4. **Choose** *Call a Procedure* from the **When Pressed** drop down list.
5. **Type** *InvoiceReport* in the **Procedure Name** field.
6. **Check** the **Initiate Thread** box.

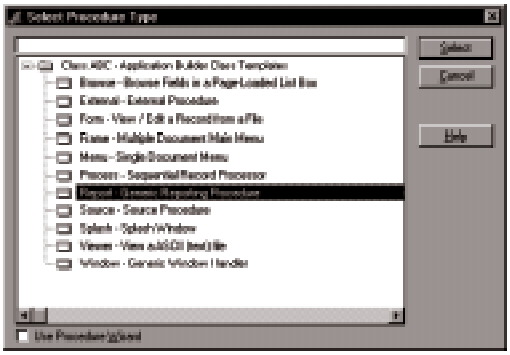
Add a third menu item

1. **Press** the **Item** button.
2. **Type** *Print &One Customer's Invoices* in the **Menu Text** field.
3. **Select** the **Actions** tab.
4. **Choose** *Call a Procedure* from the **When Pressed** drop down list.
5. **Type** *CustInvoiceReport* in the **Procedure Name** field.
6. **Check** the **Initiate Thread** box.
7. **Press** the **Close** button to close the **Menu Editor**.
8. **Choose** **Exit!** to close the Window Formatter (save your work).

Creating the Report

Now you can create the first report, using the Report Formatter.

1. **Highlight** the *CustReport* procedure in the Application Tree.
2. **Press** the **Properties** button.
3. **Highlight** *Report* in the **Select Procedure Type** dialog, **clear** the **Use Procedure Wizard** **check** box, then **press** the **Select** button.


Select Procedure Type

Class ABC - Application Builder Class Templates
Browse - Browse Fields in a Page-Loaded List Box
External - External Procedure
Form - View / Edit a Record from a File
Frame - Multiple Document Main Menu
Menu - Single Document Menu
Process - Sequential Record Processor
Report - Generic Reporting Procedure
Source - Source Procedure
Splash - Splash Window
Viewer - View a ASCII (text) file
Window - Generic Window Handler

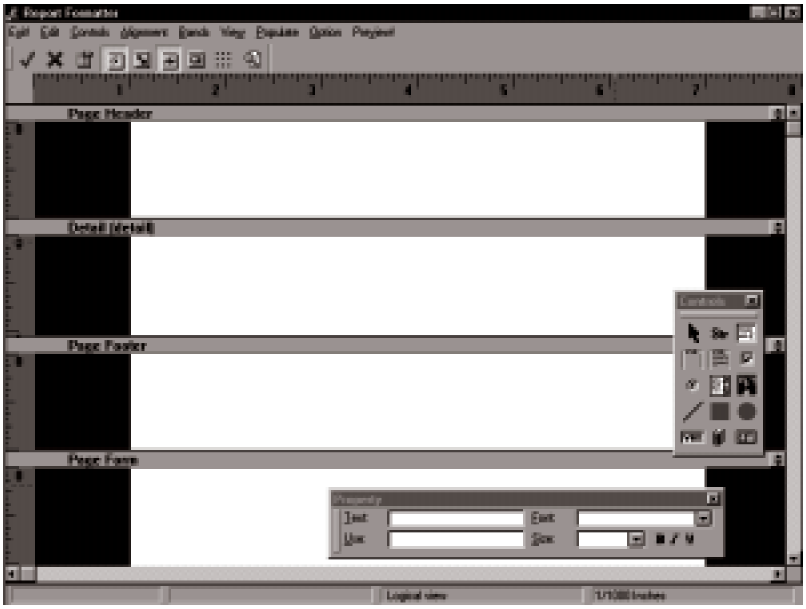
Use Procedure Wizard

4. **Press** the **Report** button in the **Procedure Properties** dialog.

   The Report Formatter appears. Here you can visually edit the report and its controls. The Report Formatter represents the four basic parts of the REPORT data structure by showing the Page Header, Detail, Page

Footer, and Form as four “bands.” Each band is a single entity that prints all together. See the *User’s Guide* chapter on *Using the Report Formatter* for more information on the parts of the report and how the print engine generates them.

For this report, you’ll place page number controls in the header, then place the fields from the Customer file in the Detail band.



Place a string constant

1. **Choose** **Controls ➤ String**, or pick the *String* tool from the **Controls** toolbox.
2. CLICK at the top of the **Page Header** band.
3. **RIGHT-CLICK** on the control, then **choose** **Properties** from the popup menu.

   The **String Properties** dialog appears.
4. **Type** *Page Number:* in the **Text** field, then **press** the **OK** button.

Place a control template to print the Page Number

1. **Choose** **Populate ➤Control Template**, or pick the *Control Template* tool from the **Controls** toolbox.
2. **Highlight** **ReportPageNumber** then **press** the **Select** button.
2. CLICK to the right of the previously placed string.

Populating the Detail

The Detail band prints once for each record in the report. For this procedure, you'll place the fields in a block arrangement, which creates a label report at print time.

1. **Choose** **Populate ➤ Multiple Fields**, or CLICK on the *Dictionary Field* tool in the **Controls** tool box.
2. In the **File Schematic Definition** dialog, **highlight** the "ToDo" folder, then **press** the **Insert** button.
3. **Select** the *Customer* file from the **Insert File** dialog, then **press** the **Select** button.
4. **Press** the **Edit** button.
5. **Highlight** *KeyCustNumber* in the **Change Access Key** dialog, then **press** the **Select** button.
6. **Highlight** *CUS:Company* in the **Fields** list and **press** the **Select** button.
7. CLICK inside the **Detail** band, near its top left corner.
8. **Highlight** *CUS:FirstName* in the **Fields** list and **press** the **Select** button.
9. CLICK inside the **Detail** band, just below the first control.
10. **Highlight** *CUS:LastName* in the **Fields** list and **press** the **Select** button.
11. CLICK inside the **Detail** band, to the right of the control you just placed.
12. **Highlight** *CUS:Address* in the **Fields** list and **press** the **Select** button.
13. CLICK inside the **Detail** band, below the *second* control you placed.

Resize the Detail band

At this point, you probably have very little room left in the Detail band, and need to make it longer.

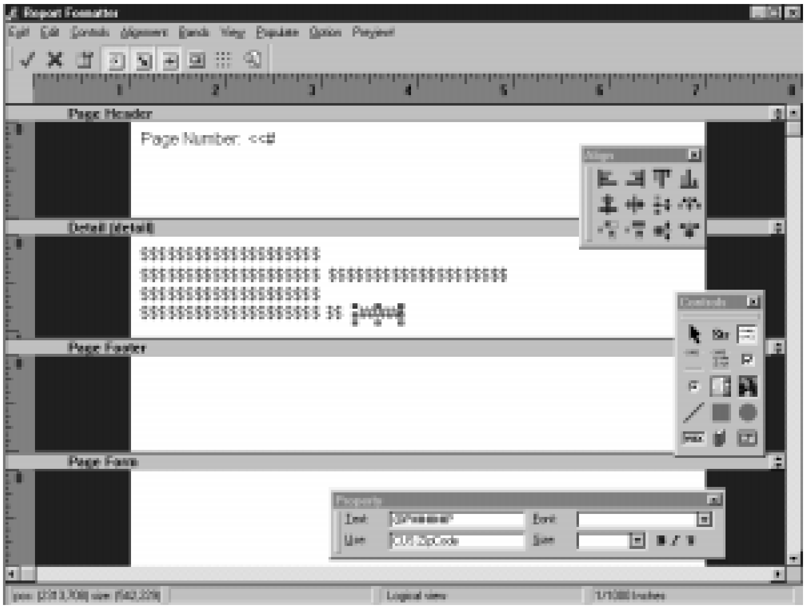
1. **Press** the **Cancel** button to exit multi-populate mode.
2. CLICK inside the **Detail** band, but not on one of the string controls.

   The Detail area's handles appear.
3. **Resize** the **Detail** band by dragging the middle handle on the bottom down—allow for enough room for about two more lines.

Place the rest of the fields

1. **Choose** **Populate ➤ Multiple Fields**, or CLICK on the *Dictionary Field* tool in the **Controls** tool box.
2. **Highlight** *CUS:City* in the **Fields** list, then **press** the **Select** button.
3. CLICK inside the **Detail** band, below the last control you placed.

4. **Highlight** *CUS:State* in the **Fields** list, then **press** the **Select** button.
5. CLICK inside the **Detail** band, to the right of the previously placed control.
6. **Highlight** *CUS:ZipCode* in the **Fields** list, then **press** the **Select** button.
7. CLICK inside the **Detail** band, to the right of the previously placed control.
8. **Press** the **Cancel** button to exit multi-populate mode.



Notice that you have the same set of alignment tools in the Report Formatter that you have already used in the the Window Formatter (choose **Option ➤ Show Alignbox** to bring up the toolbox).

Select a base font for the Report

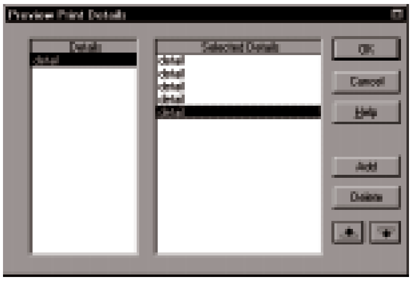
1. **Choose Edit ➤ Report Properties** to set default attributes.
2. **Press** the **Font** button.
3. **Select** a font, style, and size to use as the base font for the report.

   If you don't select a font, it uses the printer's default font.
4. **Press** the **OK** button to close the **Select Font** dialog.
5. **Press** the **OK** button to close the **Report Properties** dialog.

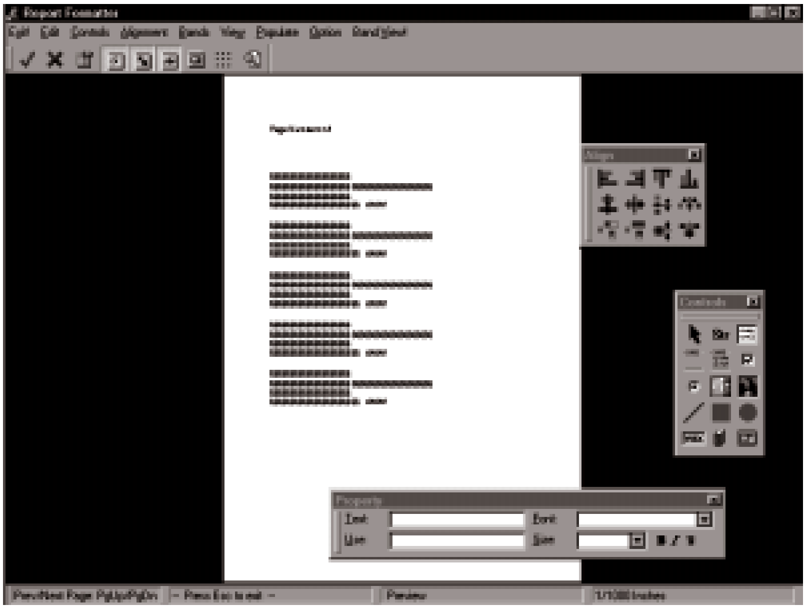
Preview the Report

1. **Choose Preview!** to "visualize" how the printed page will appear.
2. **Highlight** *Detail* in the **Details** list then **press** the **Add** button several times.

This populates the preview with some print bands to view. Because you can have many bands of various types within a single report, you have to select which to see before going to print preview. This way, the Report Formatter knows what to compose on the screen.



3. **Press** the **OK** button.



4. When done “previewing,” **choose** **Band View!**.
5. **Choose** **Exit!** to return to the **Procedure Properties** dialog (save your work).
6. **Press** the **OK** button to close the **Procedure Properties** dialog.
7. **Choose** **File ➤ Save**, or **press** the *Save* button on the tool bar to save your work.

An Invoice Report

Next, we will create one of the most common types of reports. An invoice will make use of most of the files in the data dictionary, demonstrating how to create group breaks and totals. It will also show you how to control pagination based on group breaks.

Creating the Report

1. **Highlight** the *InvoiceReport* procedure.
2. **Press** the **Properties** button.
3. **Highlight** *Report* in the **Select Procedure Type** dialog, **clear** the **Use Procedure Wizard** **check** box, then **press** the **Select** button.

   The **Procedure Properties** dialog appears.

Specify the files for the Report

1. **Press** the **Files** button in the **Procedure Properties** dialog.

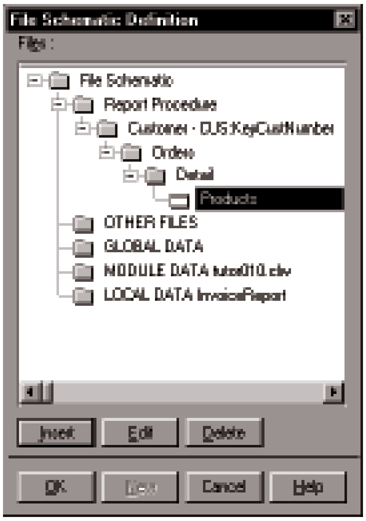
   The **File Schematic Definition** dialog appears.
2. **Highlight** the “ToDo” folder, then **press** the **Insert** button.
3. **Select** the *Customer* file from the **Insert File** dialog, then **press** the **Select** button.
4. **Press** the **Edit** button.
5. **Highlight** *CUS:KeyCustNumber* in the **Change Access Key** dialog, then **press** the **Select** button.

   The report will process all the *Customer* file records in *CustNumber* order.
6. **Highlight** the *Customer* file, then **press** the **Insert** button.
7. **Select** the *Orders* file from the **Insert File** dialog, then **press** the **Select** button.

   It will process all the *Orders* for each *Customer*.
8. **Highlight** the *Orders* file, then **press** the **Insert** button.
9. **Select** the *Detail* file from the **Insert File** dialog, then **press** the **Select** button.

   Each Order will print all the related *Detail* records.
10. **Highlight** the *Detail* file, then **press** the **Insert** button.
11. **Select** the *Products* file from the **Insert File** dialog, then **press** the **Select** button.

Each *Detail* record will lookup the related *Products* file record. At this point, the File Schematic should look like this:



```
File Schematic
  Report Procedure
    Customer - CUS:KeyCustNumber
      Orders
        Detail
          Products
  OTHER FILES
  GLOBAL DATA
  MODULE DATA tutor010.clw
  LOCAL DATA InvoiceReport
```

12. **Press** the **OK** button to return to the **Procedure Properties** dialog.

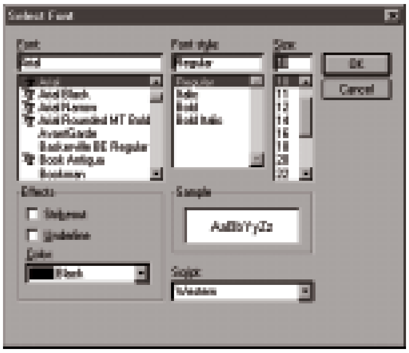
Set the Report defaults

1. **Press** the **Report** button.

2. **Choose Edit ➤ Report Properties** to set the report's default attributes.

3. **Press** the **Font** button.

4. **Select** a font, style, and size to use as the base font for the report.



If you do not select a font for the report, it will print using the printer's default font. You should select a font that you know the user will have (the fonts that ship with Windows are usually safe).

5. **Press** the **OK** button to close the **Select Font** dialog.

6. **Press** the **OK** button to close the **Report Properties** dialog.

Populating the Page Form Band

The Page Form band prints once for each page in the report. Its content is only composed once, when the report is opened. This makes it useful for constant information that will always be on every page of the report.

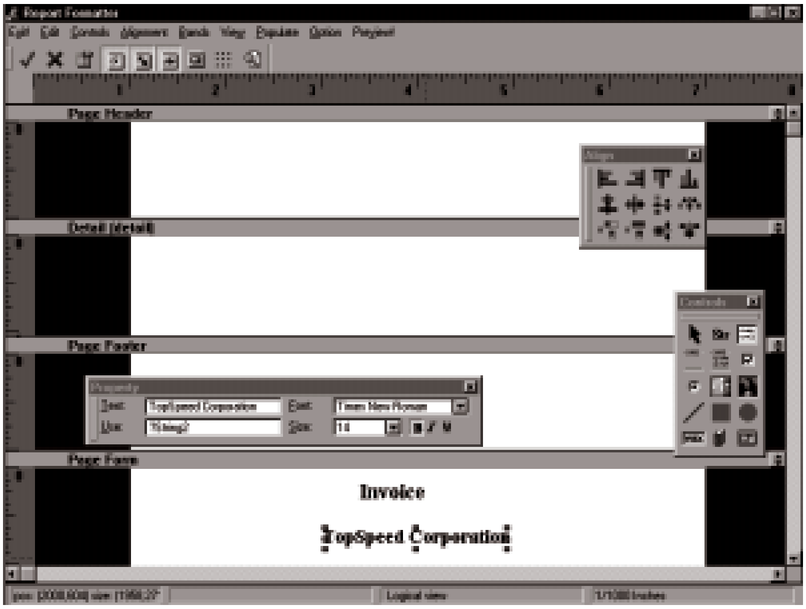
Place a string constant

1. **Choose** **Controls ➤ String**, or pick the *String* tool from the **Controls** toolbox.
2. CLICK at the top middle of the **Page Form** band.
3. DOUBLE-CLICK on the control.

   The **String Properties** dialog appears.
4. **Type** *Invoice* in the **Text** field.
5. **Press** the **Font** button.
6. **Select** a font, style, and size to use for the text (something large and bold would be appropriate for this).
7. **Press** the **OK** button to close the **Select Font** dialog.
8. **Press** the **OK** button to close the **String Properties** dialog.
9. Resize the control so that it's large enough to hold the text, by DRAGGING its handles.

Place the next string constant

1. **Choose** **Controls ➤ String**, or pick the *String* tool from the **Controls** toolbox.
2. CLICK at the top of the **Page Form** band, just below the last string you placed.
3. RIGHT-CLICK on the control, then **choose** **Properties** from the popup menu.
4. **Type** the name of your company in the **Text** field.
5. **Press** the **Font** button and **select** a font, style, and size to use for the text (something just a little smaller than the previous field would be appropriate for this).
6. **Press** the **OK** button to close the **Select Font** dialog.
7. **Press** the **OK** button to close the **String Properties** dialog.
8. **Resize** the control so that it's large enough to hold the text, by DRAGGING its handles.



Text: TopSpeed Corporation
Use: ?String2
Font: Times New Roman
Size: 14

Populating the Detail Band

The Detail band will print every time new information is read from the lowest level “Child” file in the File Schematic. For this Invoice report, the lowest level “Child” file is the *Detail* file (remember that *Products* is a Many to One “lookup” file from the *Detail* file).

1. **Choose** **Populate ➤ Multiple Fields**, or CLICK on the *Dictionary Field* tool in the **Controls** tool box.
2. **Highlight** *Detail* in the **Files** list then **select** *DTL:Quantity* in the **Fields** list and **press** the **Select** button.
3. CLICK inside the **Detail** band, near its top left corner.
4. **Highlight** *DTL:ProdNumber* in the **Fields** list and **press** the **Select** button.
5. CLICK inside the **Detail** band, directly right of the first control.
6. **Highlight** *Products* in the **Files** list then **select** *PRD:ProdDesc* in the **Fields** list and **press** the **Select** button.
7. CLICK inside the **Detail** band, to the right of the control just placed.
8. **Highlight** *Detail* in the **Files** list then **select** *DTL:ProdAmount* in the **Fields** list and **press** the **Select** button.
9. CLICK inside the **Detail** band, to the right of the control just placed.
10. **Highlight** *LOCAL DATA InvoiceReport* in the **Files** list, then **press** the **New** button.

This allows you to add a local variable without going back to the **Procedure Properties** window and pressing the **Data** button. This variable will be used to display the total price for each line item.

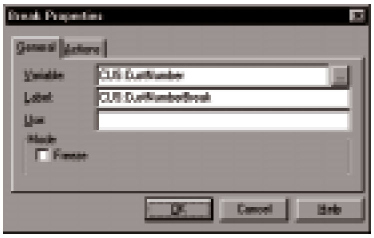
11. **Type** *ItemTotal* in the **Field Name** field.
12. **Select** *DECIMAL* from the **Data Type** dropdown list.
13. **Type** *7* in the **Characters** field.
14. **Type** *2* in the **Places** field then **press** the **OK** button.
15. CLICK inside the **Detail** band, to the right of the last control placed.
16. **Press** the **Cancel** button to exit multi-populate mode.
17. **Move** all the controls to the top of the **Detail** band, aligned horizontally, then **resize** the band so it is just a little taller than the controls.

Adding Group Breaks

We need to print different information on the page for each Invoice. Therefore, we need to create BREAK structures to provide the opportunity to print something every time the *Orders* file information changes and every time the *Customer* file information changes.

1. **Choose Bands ➤ Surrounding Break**, then CLICK on the **Detail** band.

   The **Break Properties** dialog appears.
2. **Press** the ellipsis (...) button for the **Variable** field.
3. **Highlight** *Customers* in the **Files** list then **select** *CUS:CustNumber* in the **Fields** list and **press** the **Select** button.
4. **Type** *CUS:CustNumberBreak* in the **Label** field then **press** the **OK** button.



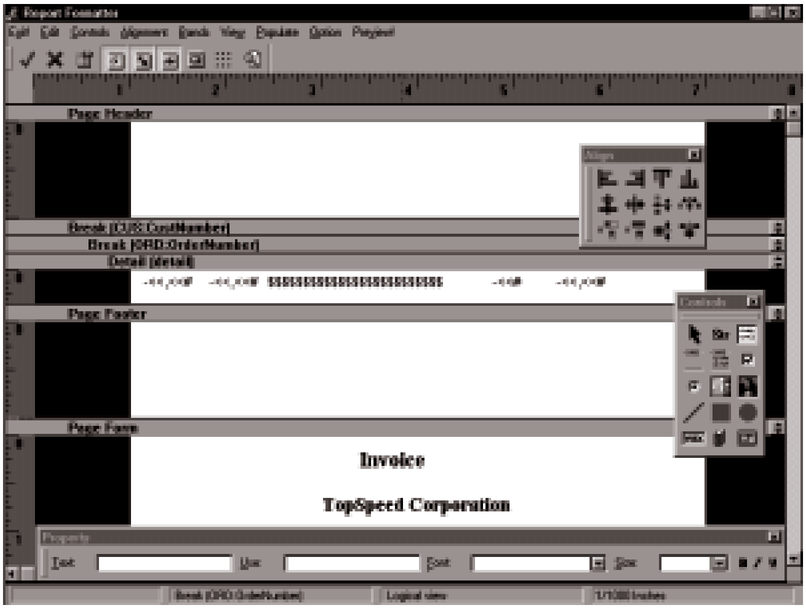
Variable: CUS:CustNumber
Label: CUS:CustNumberBreak

   A **Break (CUS:CustNumber)** band appears above the **Detail** band, which appears indented, meaning it is within the **Break** structure.
5. **Choose Bands ➤ Surrounding Break**, then CLICK on the **Detail** band.

   The **Break Properties** dialog appears.
6. **Press** the ellipsis (...) button for the **Variable** field.

7. **Highlight** *Orders* in the **Files** list then **select** *ORD:OrderNumber* in the **Fields** list and **press** the **Select** button.
8. **Type** *ORD:OrderNumberBreak* in the **Label** field then **press** the **OK** button.

   Now the report design looks something like this:



Create the group Headers and Footers

1. **Choose** **Bands ➤ Group Header**, then CLICK on the **Break (ORD:OrderNumber)** band.

   The **Group Header (ORD:OrderNumber)** band appears above the **Detail** band. This band will print every time the value in the ORD:OrderNumber field changes, at the beginning of each new group of records. We will use this to print the company name, address, along with the invoice number and date.
2. **Choose** **Bands ➤ Group Footer**, then CLICK on the **Break (ORD:OrderNumber)** band.

   The **Group Footer (ORD:OrderNumber)** band appears below the **Detail** band. This band will print every time the value in the ORD:OrderNumber field changes, at the end of each group of records. We will use this to print the invoice total.
3. RIGHT-CLICK on the **Group Footer (ORD:OrderNumber)** band then **choose** **Properties** from the popup menu.

   The **Page/Group Footer Properties** dialog appears.
4. **Check** the **Page after** box.

This causes the print engine to print this band, then initiate Page Overflow. This will compose the **Page Footer** band, issue a form feed to the printer, then compose the **Page Header** band for the next page.

5. **Press** the **OK** button.

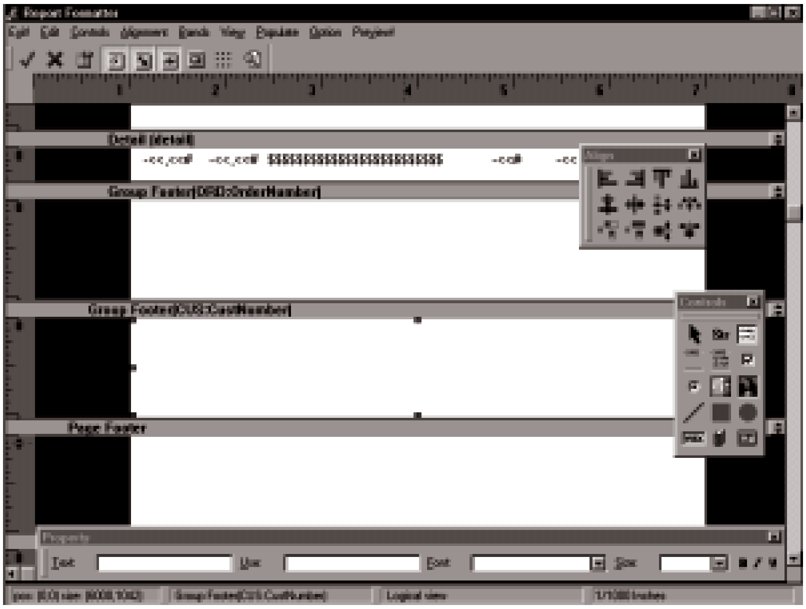
6. **Choose** **Bands ➤ Group Footer**, then CLICK on the **Break (CUS:CustNumber)** band.

   The **Group Footer (CUS:CustNumber)** band appears below the **Group Footer (ORD:OrderNumber)** band. This band will print every time the value in the CUS:CustNumber field changes, at the end of each group of records. We will use this to print invoice summary information for each company.

7. RIGHT-CLICK on the **Group Footer (CUS:CustNumber)** band then **choose** **Properties** from the popup menu.

8. **Check** the **Page after** box.

9. **Press** the **OK** button.


Report Formatter
Edit File Controls Alignment Bands View Populate Option Preview
Detail (detail)
Group Footer(ORD:OrderNumber)
Group Footer(CUS:CustNumber)
Page Footer
Align
Controls
Property
Text Use Font Size
Group Footer(CUS:CustNumber)
Logical view
1/1000 inches

Populating the Group Header Band

Place the *Customer* file fields

1. **Choose** **Populate ➤ Multiple Fields**, or CLICK on the *Dictionary Field* tool in the **Controls** tool box.

2. **Highlight** *Customer* in the **Files** list then **select** *CUS:Company* in the **Fields** list and **press** the **Select** button.

3. CLICK inside the **Group Header (ORD:OrderNumber)** band, near its top left corner.
4. **Highlight** *CUS:FirstName* in the **Fields** list and **press** the **Select** button.
5. CLICK inside the **Group Header (ORD:OrderNumber)** band, just below the first control.
6. **Highlight** *CUS:LastName* in the **Fields** list and **press** the **Select** button.
7. CLICK inside the **Group Header (ORD:OrderNumber)** band, to the right of the control you just placed.
8. **Highlight** *CUS:Address* in the **Fields** list and **press** the **Select** button.
9. CLICK inside the **Group Header (ORD:OrderNumber)** band, below the *second* control you placed.
10. **Highlight** *CUS:City* in the **Fields** list, then **press** the **Select** button.
11. CLICK inside the **Group Header (ORD:OrderNumber)** band, below the last control you placed.
12. **Highlight** *CUS:State* in the **Fields** list, then **press** the **Select** button.
13. CLICK inside the **Group Header (ORD:OrderNumber)** band, to the right of the previously placed control.
14. **Highlight** *CUS:ZipCode* in the **Fields** list, then **press** the **Select** button.
15. CLICK inside the **Group Header (ORD:OrderNumber)** band, to the right of the previously placed control.

Place the *Orders* file fields

1. **Highlight** *Orders* in the **Files** list then **select** *ORD:OrderNumber* in the **Fields** list and **press** the **Select** button.
2. CLICK inside the **Group Header (ORD:OrderNumber)** band, near its top right corner.
3. **Highlight** *ORD:OrderDate* in the **Fields** list, then **press** the **Select** button.
4. CLICK inside the **Group Header (ORD:OrderNumber)** band, below the last control you placed.
5. **Press** the **Cancel** button to close the **Select Field** dialog and exit multi-populate mode.

Place the constant text and column headings

1. **Choose** **Controls ➤ String**, or pick the *String* tool from the **Controls** toolbox.
2. CLICK inside the **Group Header (ORD:OrderNumber)** band, left of the *ORD:OrderNumber* control you placed.
3. **Type** *Order Number:* in the **Text** field of the **Property** toolbox, then **press** TAB.

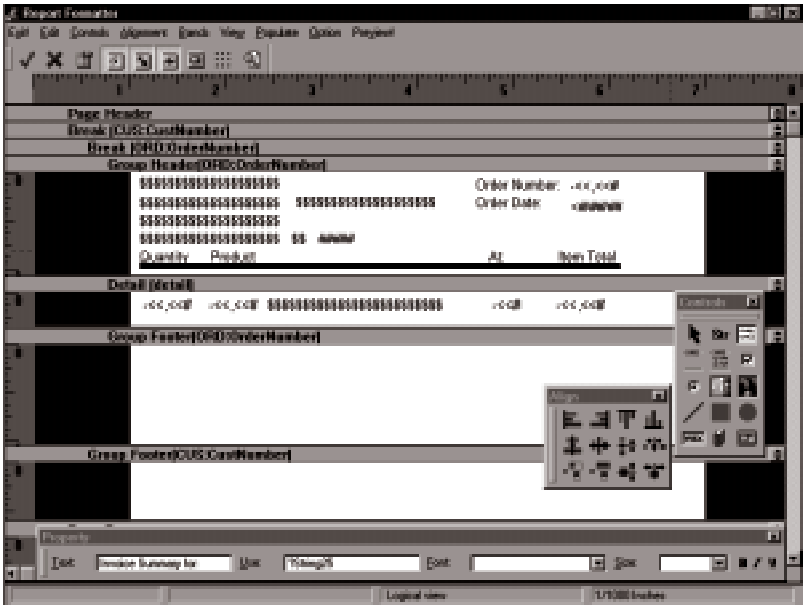
4. **Choose** **Controls ➤ String**, or pick the *String* tool from the **Controls** toolbox.
5. CLICK inside the **Group Header (ORD:OrderNumber)** band, left of the *ORD:OrderDate* control you placed.
6. **Type** *Order Date:* in the **Text** field of the **Property** toolbox, then **press** TAB.
7. **Choose** **Controls ➤ String**, or pick the *String* tool from the **Controls** toolbox.
8. CLICK inside the **Group Header (ORD:OrderNumber)** band, at the left end below the *Customer* file controls you placed.
9. **Type** *Quantity* in the **Text** field of the **Property** toolbox, then **press** TAB.
10. **Choose** **Controls ➤ String**, or pick the *String* tool from the **Controls** toolbox.
11. CLICK inside the **Group Header (ORD:OrderNumber)** band, to the right of the last string you placed.
12. **Type** *Product* in the **Text** field of the **Property** toolbox, then **press** TAB.
13. **Choose** **Controls ➤ String**, or pick the *String* tool from the **Controls** toolbox.
14. CLICK inside the **Group Header (ORD:OrderNumber)** band, to the right of the last string you placed, directly above the *DTL:ProdAmount* control in the **Detail** band.
15. **Type** *At:* in the **Text** field of the **Property** toolbox, then **press** TAB.
16. **Choose** **Controls ➤ String**, or pick the *String* tool from the **Controls** toolbox.
17. CLICK inside the **Group Header (ORD:OrderNumber)** band, to the right of the last string you placed, directly above the *ItemTotal* control in the **Detail** band.
18. **Type** *Item Total* in the **Text** field of the **Property** toolbox, then **press** TAB.

Place a thick line under the column headings

1. **Choose** **Controls ➤ Line**, or pick the *Line* tool from the **Controls** toolbox.
2. CLICK inside the **Group Header (ORD:OrderNumber)** band, under the *Quantity* string you placed.
3. **Resize** the line by DRAGGING its handles until it appears to be a line all across the report under the column headers.
4. RIGHT-CLICK and **choose** **Properties** from the popup menu.
5. **Type** 50 in the **Line Width** field.

   This makes the line much thicker.
6. **Press** the **OK** button to close the **Line Properties** dialog.

   Your Report design should now look something like this:

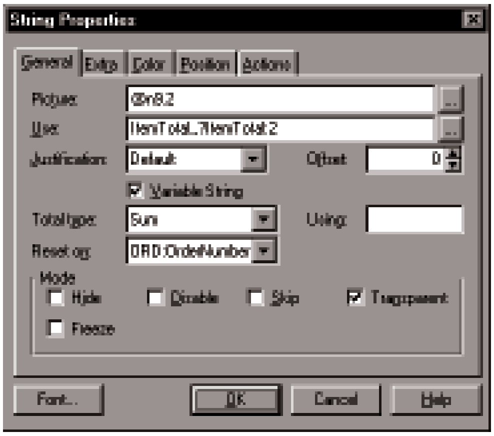


Populating the Invoice Group Footer Band

Place the constant text and total field

1. **Choose** **Controls ➤ String**, or pick the *String* tool from the **Controls** toolbox.
2. CLICK in the middle of the **Group Footer (ORD:OrderNumber)** band.
3. **Type** *Order Total:* in the **Text** field of the **Property** toolbox, then **press** TAB.
4. **Choose** **Controls ➤ String**, or pick the *String* tool from the **Controls** toolbox.
5. CLICK inside the **Group Footer (ORD:OrderNumber)** band, to the right of the string you just placed.
6. RIGHT-CLICK and **choose** **Properties** from the popup menu.
7. **Check** the **Variable String** box.
8. **Press** the ellipsis (...) button for the **Use** field.
9. **Highlight** *LOCAL DATA InvoiceReport* in the **Files** list, **select** *ItemTotal* from the **Fields** field, then **press** the **Select** button.
10. **Type** *@N9.2* in the **Picture** field.
11. **Select** *Sum* from the **Total type** dropdown list.
12. **Select** *ORD:OrderNumberBreak* from the **Reset on** dropdown list.
13. **Press** the **OK** button.

    This will add up all the *ItemTotal* contents for the Invoice and will reset to zero when the value in the *ORD:OrderNumber* field changes.

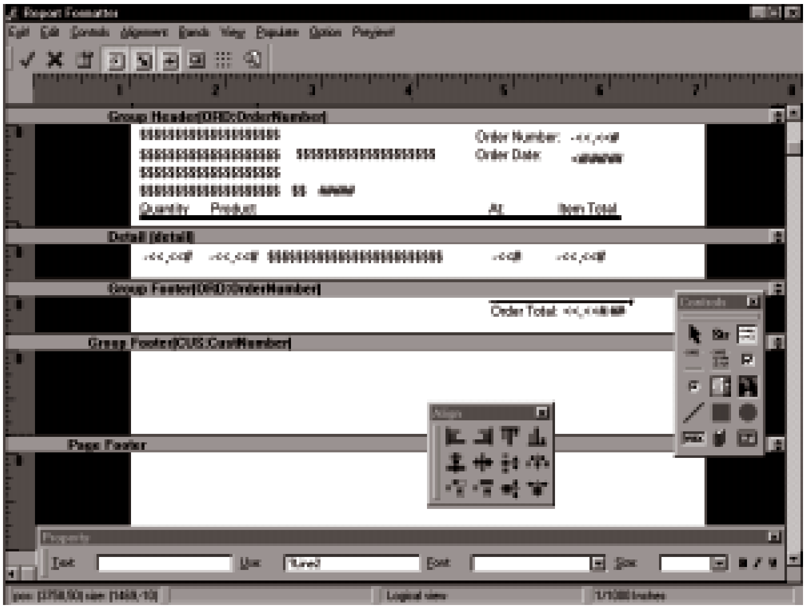


Picture: @n9.2
Use: ItemTotal...?ItemTotal:2
Justification: Default
Offset: 0
☑ Variable String
Total type: Sum
Using:
Reset on: ORD:OrderNumber
Mode
☐ Hide ☐ Disable ☐ Skip ☑ Transparent
☐ Freeze

Place a line above the total

1. **Choose** Controls ➤ Line, or pick the *Line* tool from the Controls toolbox.
2. CLICK inside the Group Footer (ORD:OrderNumber) band, above the controls you just placed.
3. **Resize** the line by DRAGGING its handles until it appears to be above both the controls you just placed.
4. RIGHT-CLICK and **choose** Properties from the popup menu.
5. **Type** 20 in the Line Width field.

   This makes the line just a little bit thicker.
6. **Press** the OK button to close the Line Properties dialog.



pos: [1758,50] size: [1468,10] | Logical view | 1/1000 Inches

Populating the Customer Group Footer Band

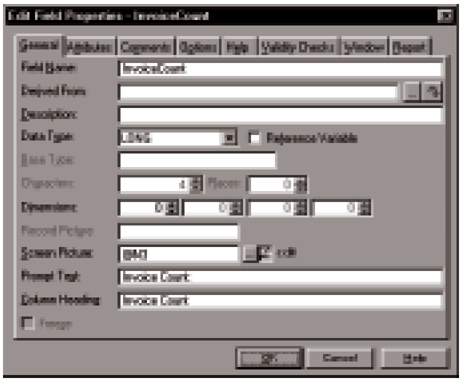
Place the constant text

1. **Choose** **Controls ➤ String**, or pick the *String* tool from the **Controls** toolbox.
2. CLICK in the middle of the **Group Footer (CUS:CustNumber)** band.
3. **Type** *Invoice Summary for:* in the **Text** field of the **PropertyBox** toolbox, then **press** TAB.
4. **Choose** **Controls ➤ String**, or pick the *String* tool from the **Controls** toolbox.
5. CLICK inside the **Group Footer (CUS:CustNumber)** band, below the string you just placed.
6. **Type** *Total Orders:* in the **Text** field of the **PropertyBox** toolbox, then **press** TAB.

Place the total fields

1. **Choose** **Controls ➤ String**, or pick the *String* tool from the **Controls** toolbox.
2. CLICK inside the **Group Footer (CUS:CustNumber)** band, just right of the string you just placed.
3. RIGHT-CLICK and **choose** **Properties** from the popup menu.
4. **Check** the **Variable String** box.
5. **Press** the ellipsis (...) button for the **Use** field.
6. **Highlight** *LOCAL DATA InvoiceReport* in the **Files** list, then **press** the **New** button.
7. **Type** *InvoiceCount* in the **Name** field.

   This is a field that will print the number of invoices printed for an individual company.
8. **Select** *LONG* from the **Data Type** dropdown list.
9. **Type** *@N3* in the **Picture** field, then **press** the **OK** button.



Edit Field Properties - InvoiceCount

Field Name: InvoiceCount
Derived From:
Description:
Data Type: LONG
Reference Variable
Base Type:
Characters: 4 Places: 0
Dimensions: 0 0 0 0
Record Picture:
Screen Picture: @N3
Prompt Text: Invoice Count
Column Heading: Invoice Count
Freeze

10. **Select** *Count* from the **Total type** dropdown list.
11. **Select** *CUS:CustNumberBreak* from the **Reset on** dropdown list.

    This is the same type of total field that we placed in the *ORD:OrderNumber* group footer, but it will only reset when the *CUS:CustNumber* changes.
12. **Select** the **Extra** tab.
13. **Highlight** *ORD:OrderNumberBreak* from the **Tallies** list, then **press** the **OK** button.

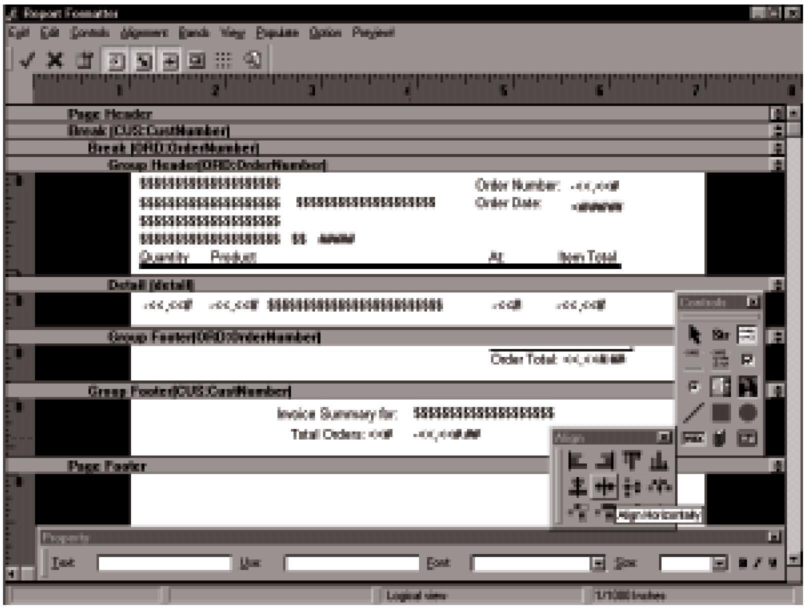
    This total field will count the number of invoices that print for each customer. The Tallies list allows you to select the point(s) at which the total increments. By selecting *ORD:OrderNumberBreak* from the list, the count will only increment when a new invoice begins.
14. **Choose** **Controls ➤ String**, or pick the *String* tool from the **Controls** toolbox.
15. CLICK inside the **Group Footer (CUS:CustNumber)** band, to the right of the string you just placed.
16. RIGHT-CLICK and **choose** **Properties** from the popup menu.
17. **Check** the **Variable String** box.
18. **Press** the ellipsis (...) button for the **Use** field.
19. **Highlight** *LOCAL DATA InvoiceReport* in the **Files** list, then **select** *ItemTotal* in the **Fields** list and **press** the **Select** button.
20. **Select** *Sum* from the **Total type** dropdown list.
21. **Select** *CUS:CustNumberBreak* from the **Reset on** dropdown list.

    This is the same type of total field that we placed in the *ORD:OrderNumber* group footer, but it will only reset when the *CUS:CustNumber* changes.
22. **Press** the OK button.

Place the display field then exit

1. **Choose** **Populate ➤ Multiple Fields**, or CLICK on the *Dictionary Field* tool in the **Controls** tool box.
2. **Highlight** *Customer* in the **Files** list then **select** *CUS:Company* in the **Fields** list and **press** the **Select** button.
3. CLICK inside the **Group Footer (CUS:CustNumber)** band, just right of the *Invoice Summary for:* string you placed.
4. **Press** the **Cancel** button to exit multi-populate mode.

   Your report design is now complete.



5. **Choose** **Exit!** to return to the **Procedure Properties** dialog (be sure to save your report design).

Adding a Formula

To make the *ItemTotal* field contain the correct amount for each *Detail* record in the invoice, you need to add a Formula to the procedure.

1. **Press** the **Formulas** button in the **Procedure Properties** dialog.
2. **Press** the **New** button in the **Formulas** dialog.

   The **Formula Editor** dialog appears.
3. **Type** *Item Total Formula* in the **Name** field.
4. **Press** the ellipsis (...) button for the **Class** field.
5. **Highlight** *Before Print Detail* in the **Template Classes** list, then **press** the **OK** button.

   The *Before Print Detail* class tells the Report template to perform the calculation each time it gets ready to print a Detail.
6. **Press** the ellipsis (...) button for the **Result** field.
7. **Highlight** *LOCAL DATA InvoiceReport* in the **Files** list, **select** *ItemTotal* from the **Fields** list, then **press** the **Select** button.
8. **Press** the **Data** button in the **Operands** group.
9. **Highlight** the *Detail* file in the **Files** list, **select** *DTL:Quantity* from the **Fields** list, then **press** the **Select** button.

This places the *DTL:Quantity* field in the **Statement** field for you. The **Statement** field contains the expression being built, and you can also type directly into it to build the expression, if you wish.

10. **Press** the ***** button in the **Operators** group.
11. **Press** the **Data** button in the **Operands** group.
12. **Highlight** the *Detail* file in the **Files** list, **select** *DTL:ProdAmount* from the **Fields** list, then **press** the **Select** button.
13. **Press** the **Check** button to check the expression's syntax.
14. **Press** the **OK** button to close the Formula Editor.
15. **Press** the **OK** button to close the **Formulas** dialog and return to the **Procedure Properties** dialog.

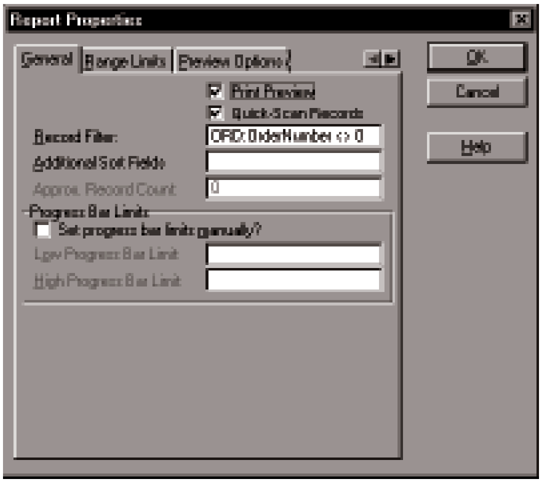
Adding a Record Filter

To make sure the report only prints invoices for the companies that have orders, we'll add a Record Filter.

1. **Press** the **Report Properties** button in the **Procedure Properties** dialog.

   The **Report Properties** dialog appears.

2. **Type** *ORD:OrderNumber <> 0* in the **Record Filter** field.



Print Preview [✓]
Quick-Scan Records [✓]
Record Filter: ORD:OrderNumber <> 0
Additional Sort Fields
Approx. Record Count 0
Progress Bar Limits
Set progress bar limits manually? [ ]
Low Progress Bar Limit
High Progress Bar Limit

This will eliminate all the customers who have not ordered anything. Internally, the Report Template generates a VIEW structure for you (see the *Language Reference*). This VIEW structure, by default, performs an "outer join" on the files you placed in the File Schematic. "Outer join" is a standard term in Relational Database theory—it just means that the VIEW will retrieve all Parent file records, whether there are any related Child file records or not. If it retrieves a Parent record without a Child, the fields in the Child file are all blank or zero, while the Parent file's fields contain valid data. Therefore, this is the condition for which we test.

ORD:OrderNumber <> 0 checks to see if the ORD:OrderNumber field has any value in it other than zero. Since ORD:OrderNumber is the key field in the Orders file that creates the link to the related Customers file record, it must contain a value if there are existing Orders file records for the current Customer. If ORD:OrderNumber does not contain a value other than zero, the current Customers file record is skipped ("filtered out"). This eliminates printing Parent records without related Children (in this case, any Customers without Orders).

3. **Press** the **OK** button.

Change the Progress Window

1. **Press** the **Window** button in the **Procedure Properties** dialog.
2. **Type** *Invoice Progress* in the **Text** field of the **Property** toolobox.
3. **Choose** **Exit!** to return to the **Procedure Properties** dialog.

Exit and Save

1. **Press** the **OK** button in the **Procedure Properties** dialog to close it.
2. **Choose** **File ä Save**, or **press** the *Save* button on the toolbar.

A Range Limited Report

Next, we will limit the range of records that will print.

Creating the Report

1. **Highlight** the *InvoiceReport* procedure.
2. **Choose** Procedure ä Copy....

   The New Procedure dialog appears.
3. **Type** *CustInvoiceReport* in the entry box, then **press** the OK button.

   The copied procedure appears in the application tree, replacing its ToDo.

Modify the new report

1. **Highlight** the *CustInvoiceReport* procedure.
2. RIGHT-CLICK and **choose** Properties from the popup menu.
3. **Press** the Embeds button.
4. **Press** the Contract All button.

   This will make it easier to locate the specific embed point you need.
5. **Locate** the Local Objects folder, then CLICK on its + sign to expand its contents.

   The ABC templates generate object-oriented code. Each procedure instantiates a set of objects which are derived from the ABC Library. The Local Objects folder shows you all the object and methods which you can override for the procedure by simply embedding your own code to enhance the ABC functionality. See the *Easing Into OOP* and *Object Oriented Programming* articles in the *Programmer's Guide* for more on this powerful technique.
6. **Locate** the ThisWindow folder, then CLICK on it to expand it.
7. **Locate** the Init PROCEDURE(),BYTE,VIRTUAL folder, then CLICK on it to expand it.
8. **Locate** the CODE folder, then CLICK on it to expand it.
9. **Highlight** Open Files then **press** the Insert button.

   This embed point is at the beginning of the procedure, before the report has begun to process. It's important that the files for the report already be open because we will call another procedure for the user to select a Customer record. If the files for the report weren't already open, the procedure we call would open the Customer file for itself then close it again and we would lose the data that we want to have for the report. This has to do with multithreading and the Multiple Document Interface (MDI)—see THREAD in the *Language Reference* for more on this.

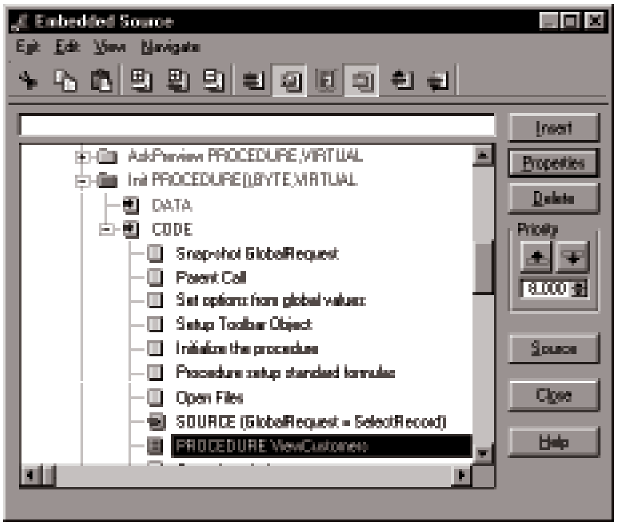
10. **Highlight** *Source* then **press** the **Select** button to call the Text Editor.
11. **Type** in the following code:

```
GlobalRequest = SelectRecord
```

This code sets up a Browse procedure to select a record (it enables the Browse procedure's Select button).

12. **Choose** **Exit!** to return to the **Embedded Source** dialog.
13. **Highlight** the SOURCE you just added then **press** the **Insert** button.
14. **Highlight** *Call a procedure* then **press** the **Select** button.
15. **Select** *ViewCustomers* from the dropdown list then **press** the **OK** button.

This will generate a procedure call to the *ViewCustomers* Browse procedure to allow the user to select which Customer's Invoices to print.



```
AskPreview PROCEDURE,VIRTUAL
Init PROCEDURE(),BYTE,VIRTUAL
  DATA
  CODE
    Snap-shot GlobalRequest
    Parent Call
    Set options from global values
    Setup Toolbar Object
    Initialize the procedure
    Procedure setup standard formulas
    Open Files
    SOURCE (GlobalRequest = SelectRecord)
    PROCEDURE ViewCustomers
```

Notice that there are now two entries displayed under the embed point. At each embed point you can place as many items as you want, mixing Code Templates with your own SOURCE or PROCEDURE Calls. You can also move the separate items around within the embed point using the arrow buttons, changing their logical execution order (the first displayed is the first to execute). Note well that moving them will change the assigned **Priority** option setting for the moved item if you attempt to move a higher priority item in front of another with a lower priority setting.

16. **Press** the **Close** button to return to the **Procedure Properties** dialog.

Set the Range Limit

1. **Press** the **Report Properties** button.

   The **Report Properties** dialog appears. This dialog allows you to set either Record Filters or Range Limits (along with Hot Fields and Detail Filters—see the *User's Guide* for more on these).

   Record Filters and Range limits are very similar. A Record Filter is a conditional expression to filter out unwanted records from the report, while a Range Limit limits the records printed to only those matching specific key field values. They can both be used to create reports on a subset of your data files, but a Range Limit requires a key and a Record Filter doesn't. This makes a Record Filter completely flexible while a Range Limit is very fast. You can use both capabilities if you want to limit the range then filter out unneeded records from that range.

2. **Select** the **Range Limits** tab.
3. **Press** the ellipsis (...) button for the **Range Limit Field**.
4. **Highlight** *CUS:CustNumber* then **press** the **Select** button.
5. **Leave** *Current Value* as the **Range Limit Type** then **press** the **OK** button.

   Current Value indicates that whatever value is in the field at the time the report begins is the value on which to limit the report. Since the user will choose a Customer record from the *ViewCustomer* procedure, the correct value will be in the *CUS:CustNumber* field when the report begins.

Exit and Save

1. **Press** the **OK** button in the **Procedure Properties** dialog to close it.
2. **Choose** **File ä Save**, or **press** the *Save* button on the toolbar to save your work.

A Single Invoice Report

Next, we will print a single invoice from the Browse list of orders.

Creating the Report

1. **Highlight** the *CustInvoiceReport* procedure.
2. **Choose** **Procedure** ➤ **Copy...**.

   The **New Procedure** dialog appears.
3. **Type** *SingleInvoiceReport* in the entry box, then **press** the **OK** button.
4. **Press** the **Same** button in the **Procedure name clash** dialog.

   The copied procedure appears unattached at the bottom of the application tree. We'll "connect the lines" after we finish with the report.

Delete the embed code

1. **Highlight** the *SingleInvoiceReport* procedure.
2. RIGHT-CLICK and **choose** **Properties** from the popup menu.
3. **Press** the **Embeds** button.
4. **Press** the **NextFilled** button (the button at the far right end of the toolbar).
5. **Press** the **Delete** button.
6. **Answer** *Yes* to the **Are you sure?** question.
7. **Press** the **Delete** button again and **answer** *Yes* to the **Are you sure?** question.
8. **Press** the **Close** button.

Change the File Schematic

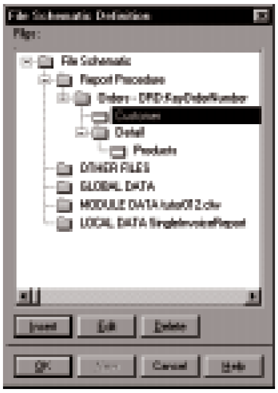
First, we need to change the order of the files in the File Schematic. We'll end up with all the same files, but instead of the *Customer* file as the Primary file (first file in the File Schematic), we need the *Orders* file to be the Primary file for the procedure so we can easily limit the range to a single invoice.

1. **Press** the **Files** button.
2. **Highlight** the *Customer* file then **press** the **Delete** button.

   This causes all the files to disappear.
3. **Highlight** the "ToDo" folder, then **press** the **Insert** button.
4. **Select** the *Orders* file from the **Insert File** dialog, then **press** the **Select** button.
5. **Press** the **Edit** button.

6. **Highlight** *KeyOrderNumber* in the **Change Access Key** dialog, then **press** the **Select** button.
7. **Highlight** the *Orders* file, then **press** the **Insert** button.
8. **Select** the *Detail* file from the **Insert File** dialog, then **press** the **Select** button.
9. **Highlight** the *Detail* file, then **press** the **Insert** button.
10. **Select** the *Products* file from the **Insert File** dialog, then **press** the **Select** button.
11. **Highlight** the *Orders* file again, then **press** the **Insert** button.
12. **Select** the *Customer* file from the **Insert File** dialog, then **press** the **Select** button.

    We've selected all the same files, but now the Primary file is the *Orders* file and the related *Customer* file record will be looked up. This is important, because we need to limit this report to a single invoice and that would be much more difficult to do if the *Customer* file were the Primary. At this point, the File Schematic should look like this:



13. **Press** the **OK** button.

Set the Range Limit

1. **Press** the **Report Properties** button.
2. **Select** the **Range Limits** tab.
3. **Press** the ellipsis (...) button for the **Range Limit Field**.
4. **Highlight** *ORD:OrderNumber* then **press** the **Select** button.
5. **Leave** *Current Value* as the **Range Limit Type** then **press** the **OK** button.

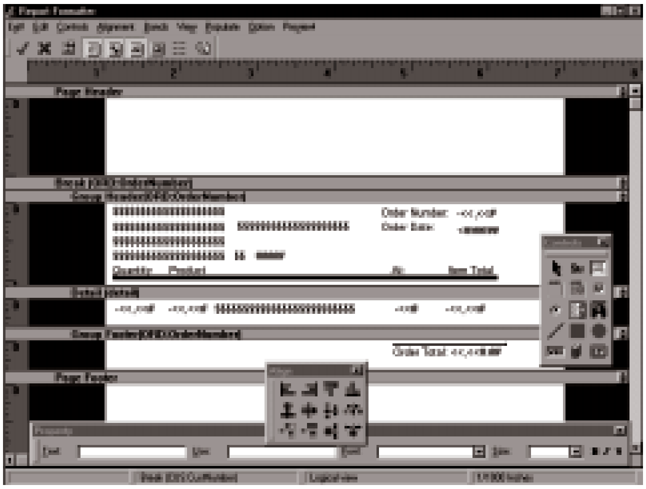
    Current Value indicates that whatever value is in the field at the time the report begins is the value on which to limit the report. Since the user will run this report from the *ViewOrders* procedure, the correct value will be in the *ORD:OrderNumber* field when the report begins.

Modify the new report

Now we need to change the report itself, to only print a single invoice.

1. **Press** the **Report** button.
2. **RIGHT-CLICK** on the **Break(CUS:CustNumber)** band and **choose Delete** from the popup menu.

   This removes not only the Group Break, but also the Group Footer that was associated with it, leaving you a report design that looks like this:



3. **Choose Exit!** to return to the **Procedure Properties** dialog.

Exit and Save

1. **Press** the **OK** button in the **Procedure Properties** dialog to close it.
2. **Choose File ➤ Save**, or **press** the *Save* button on the toolbar to save your work.

Connect the Lines

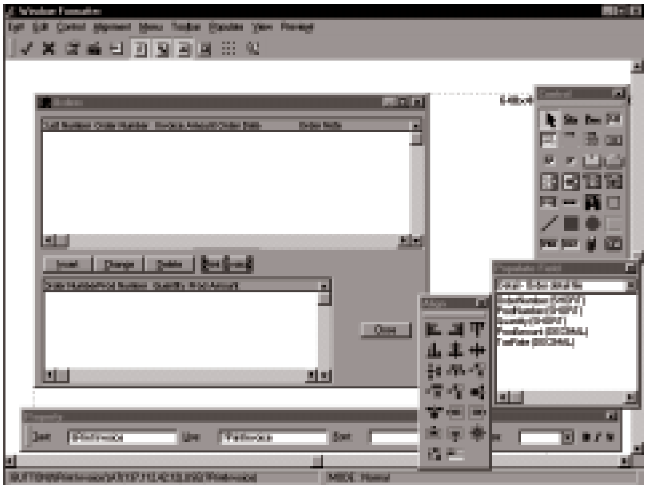
1. **Highlight** the *ViewOrders* procedure.
2. **RIGHT-CLICK** and **choose Window** from the popup menu.
3. **Choose Populate ➤ Control Template** (or **CLICK** on the *Control Template* tool in the **Controls** toolbox (in the the bottom right corner).
4. **Highlight BrowsePrintButton** then **press** the **Select** button.
5. **Highlight Browse on Orders** then **press** the **Select** button.
6. **CLICK** to the right of the **Delete** button to place the new button control.
7. **RIGHT-CLICK** on the new button and **choose Properties** from the popup menu.
8. **Type** *&Print Invoice* into the **Text** field.

9. **Type** *?PrintInvoice* into the **Use** field.
10. **Choose** the **Actions** tab.
11. **Select** **SingleInvoiceReport** from the **Print Button** droplist.

    This Control Template is sepcifically designed to run a range-limited report based onthe currently highlighted record in the list box we selected (Browse onOrders). The *Orders* file record buffer will contain the correct value to allow the *Current Value* Range Limit on the SingleInvoiceReport to work. It also automatically adds this button's action to the popup menu for the browse.

12. **Press** the **OK** button.

    Your screen design should look something like this:



13. **Choose** **Exit!** to return to the **Application Tree**.
14. **Choose** **File ➤ Save**, or **press** the *Save* button on the toolbar to save your work.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- ♦ You added several menu items to your main menu.
- ♦ You created a simple Customer List report.
- ♦ You created a relational report to print all Invoices.
- ♦ You range-limited a report to print Invoices for a single customer.
- ♦ You range-limited a report to print a single Invoice from the current record highlighted in a Browse list.

Now we'll look at where to go next.

What's Next?

Congratulations, you made it to the end of the **Application Generator** tutorial! Welcome to the growing community of Clarion developers!

While this tutorial application is by no means a "shrink-wrap" program, it has demonstrated the normal process of using the Application Generator and all its associated tools to create an application that actually performs some reasonably sophisticated tasks. Along the way, you have used most of Clarion's high-level tool set, and seen just how much work can be done *for you* without writing source code. You have also seen how just a *little* embedded source can add extra functionality to the template-generated code, and how you can easily override the default ABC Library classes.

A Short Resource Tour

You have many resources at your disposal to help you with your Clarion programming. Here is a short tour of two of the more important ones which you have right at your fingertips:

1. **Choose** **Help ➤ Contents**.

   This is the Contents page for Clarion's extensive on-line Help system.

2. **Press** the **How do I ... ?** button.

   This opens Clarion's on-line Help file and takes you to a section of commonly asked questions and their answers. This list of topics is the first place you should look whenever you ask yourself any question about Clarion programming that starts with "How do I ... ?" These topics answer many of the most common questions that newcomers to Clarion have, so quite often, you'll find the answer is here.

3. **Press** the **Back** button, then **press** the **Guide to Examples** button.

   This topic provides jumps to the discussions of all the example programs which come with Clarion. Here you'll find the various tips, tricks, and techniques which the examples demonstrate so you can adapt them for use in your own programs.

4. **Press** the **Back** button, then **press** the **Late Breaking News** button.

   This topic always gives you the latest, up-to-the-minute information about the most current release of Clarion you have installed. You should always go through this section any time you get a major upgrade or interim release. There are generally a few last-minute details which you will find are only documented in this section. That makes it well worth the reading.

5. **Close** on-line Help then **task-switch** to the operating system's file manager utility (that's Explorer, in Windows 95, 98, or NT).

6. **Put** your Clarion CD in your CD-ROM drive, then **navigate** to the \DOCS subdirectory.

7. DOUBLE-CLICK on the *C5-UG.PDF* file (you must have installed Acrobat Reader from the Clarion CD to read this file).

   This brings up the *User's Guide* in Acrobat Reader. This is the entire book, on-line and available from your CD (or hard drive, if you chose to copy the .PDF files there). This means that, for the weight of a CD you can have all of Clarion's documentation available to you anywhere (and Acrobat has full text search). Notice that on the left side you have a set of Table of Contents jumps available to you. You also can go directly to any individual page you choose (and we'll show you how, right now).

9. **Press** the "end of document" VCR control (it looks like a right-pointing triangle with a vertical line on its right side), then **press** the "previous page" VCR control (it looks like a left-pointing triangle) a couple of times.

   You're now looking at the *Index* portion of the *User's Guide*.

10. **Select** the "zoom in" tool (a magnifying glass with a + sign in it), then CLICK anywhere on the page (if the cursor changes to a "pointing finger" then you'll jump directly to the item it points at instead of zooming in).

    You can zoom in to any text you choose for your reading ease using the zoom tool. Note the page number of any item you wish.

11. CTRL+CLICK to zoom back out to the previous view.

12. **Press** CTRL+5 to bring up the **Go To Page** dialog, enter the page number of the item you noted, then **press** the **OK** button.

    Now you see how easy it is to quickly get to any place you need to.

13. **Close** the file and **exit** Acrobat Reader.

Where to next?

So where should you go from here to learn more? The best places to go next (besides creating an application of your own design) are:

- The *Application Handbook* explains Clarion's ABC Templates and ABC Library—all the tools they provide for you in the Application Generator.
- The *Clarion Language Tutorial* in the next chapter is the best place to go next to start learning the Clarion language.
- TopSpeed offers educational seminars at various locations. Call Customer Service at (800) 354-5444 or (954) 785-4555 to enroll.
- Join (or form) a local Clarion User's Group and participate in joint study projects with other Clarion developers.
- Participate in TopSpeed's forum on CompuServe (GO TOPSPEED) or the *comp.lang.clarion* Internet Newsgroup to network with other Clarion programmers from all around the world (**Strongly recommended!**).

Good luck and keep going—the programming power that Clarion puts at your fingertips just keeps on growing as you learn more!

13 - CLARION LANGUAGE TUTORIAL

Clarion—the Programming Language

The foundation of the Clarion application development environment is the Clarion programming language. Clarion is a 4th Generation Language (4GL) that is both business-oriented and general-purpose. It is business-oriented in that it contains data structures and commands that are highly optimized for data file maintenance and business programming needs. It is also general-purpose because it is fully compiled (not interpreted) and has a command set that is functionally comparable to other 3GL langauges (such as C/C++, Modula-2, Pascal, etc.). This distinction is more fully discussed in the *Foreword—Origins of the Clarion Language* in the *Language Reference*.

By now, you should have completed all the Application Generator tutorials in the preceding chapters. The purpose of this language tutorial is to introduce you to the fundamental aspects of the Clarion language—particularly as it relates to business programming using the Windows event-driven paradigm. Clarion language keywords are in ALL CAPS and this tutorial concentrates on explaining the specific use of each keyword and its interaction with other language elements only in the specific context within which it is used. You should always refer to the *Language Reference* for a more complete explanation of each individual keyword and its capabilities.

When you complete this brief tutorial, you will be familiar with:

- The basic structure of a Clarion procedural program.
- The most common event-handling code structure.
- How to compile and link hand-coded programs.

Event-driven Programming

Windows programs are event-driven. The user causes an event by CLICKING the mouse on a screen control or pressing a key. Every such user action causes Windows to send a message (an event) to the program which owns the window telling it what the user has done.

Once Windows has sent the message signaling an event to the program, the program has the opportunity to handle the event in the appropriate manner. This basically means the Windows programming paradigm is exactly opposite from the DOS programming paradigm—the operating system (Windows) tells the program what the user has done, instead of the program telling the operating system what to do.

This is the most important concept in Windows programming—that the user is in control (or should be) at all times. Therefore, Windows programs are reactive rather than proactive; they always deal with what the user has done instead of directing the user as to what to do next. The most common example of this is the data entry dialog. In most DOS programs, the user must follow one path from field to field to enter data. They must always enter data in one field before they can go on to the next (and they usually can only go on to a specific "next" entry field). This makes data validation code simple—it simply executes immediately after the user has left the field.

In Windows programs, the user may use a mouse or an accelerator key to move from control to control, at any given time, in no particular order, skipping some controls entirely. Therefore, data validation code should be called twice to ensure that it executes at least once: once when the user leaves the entry control after entering data, and again when the user presses OK to leave the dialog. If it isn't executed on the OK button, required data could be omitted. This makes Windows programs reactive rather than proactive.

Hello Windows

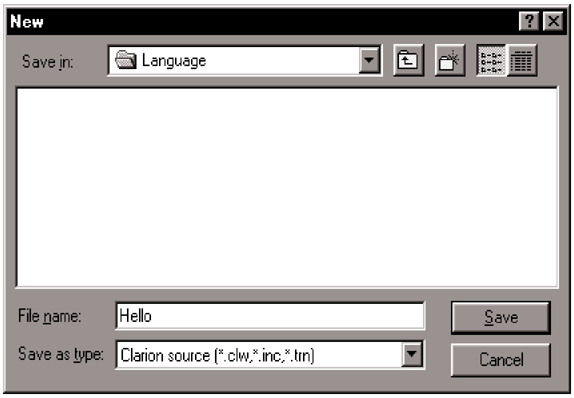
Traditionally, all programming language tutorials begin by creating a "Hello World" type of program—and so does this one.

Starting Point:
The Clarion environment should be open, and the TUTORIAL.APP should be closed.

Create the Source file

1. Using the appropriate tool in your operating system (File Manager, Explorer, etc.), **create** a new directory called *Language* under the **Clarion5** subdirectory.
2. **Return** to Clarion.
3. **Choose** **File ➤ New ➤ Source.**

   The **New** dialog appears. It's a standard windows *Open File* dialog, allowing you to change the directory and type in the filename.
4. **Select** the \CLARION5\LANGUAGE directory.
5. **Type** *Hello* in the **File Name** field.



6. **Press** the **Save** button.

   This creates an empty *HELLO.CLW* (.CLW is the standard extension for Clarion source code files) and places you in the Text Editor.

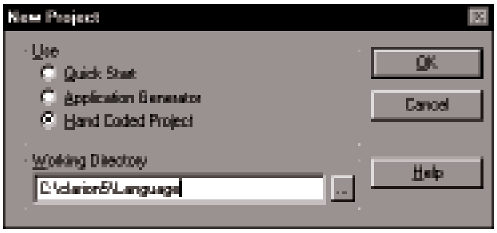
Create the Project file

1. **Choose** **Project ➤ New.**

   The **New Project** dialog appears. When completely hand-coding a Clarion program, you also need a Project file (*HELLO.PRJ*) to control the compile and link process.

2. **Choose** the **Hand Coded Project** radio button.

3. **Type** *C:\CLARION5\LANGUAGE* in the **Working Directory** box (or use the ellipsis button to select the directory from the **Change Working Directory** dialog).



   When hand-coding a program, the directory containing the Project file (.PRJ) becomes the working directory for the project. When you select the project, Clarion automatically makes the directory containing the Project file the working directory.

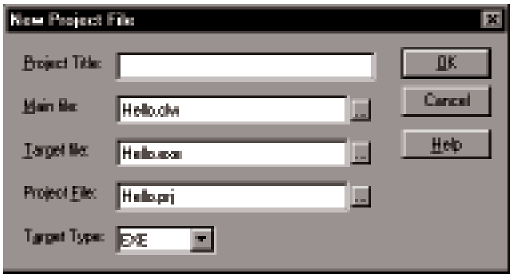
4. **Press** the **OK** button.

   The **New Project File** dialog appears.

5. **Type** *Hello Windows* in the **Project Title** box, then press TAB.

6. **Type** *HELLO.CLW* in the **Main File** box, then press TAB.

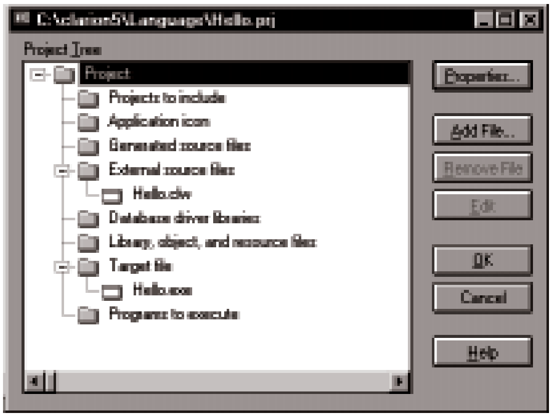
Once you've named the main source module for the project, the rest of the controls are automatically filled out for you with default values.



Project Title:
Main file: Hello.clw
Target file: Hello.exe
Project File: Hello.prj
Target Type: EXE

7. **Press** the **OK** button.

The **Project Editor** dialog appears. This controls the source modules the compiler includes in the project and the libraries the linker links in to create the executable program. When you hand-code a complete application instead of using the Application Generator, you must update the Project yourself to include all source code modules, file drivers, icons, and external libraries your program uses. See the *Using the Project System* chapter in the *User's Guide* for complete information on how to maintain your Project.



Project
Projects to include
Application icon
Generated source files
External source files
Hello.clw
Database driver libraries
Library, object, and resource files
Target file
Hello.exe
Programs to execute

8. **Press** the **OK** button.

The *HELLO.CLW* file in the Text Editor now has focus.

Notice that the environment's title bar text now reads: *Clarion 5 (HELLO.PRJ) - (C:\CLARION5\LANGUAGE\HELLO.PRJ)*. This always tells you the current project and working directory. It is important to be aware of this when hand-coding, since simply opening a source file in the Text Editor does not change the current Project for the compiler.

Write the Program

1. **Type** in the following code:

```
  PROGRAM
  MAP
  END
MyWin   WINDOW('Hello Windows'),SYSTEM
        END
  CODE
  OPEN(MyWin)
  ACCEPT
  END
```

This code begins with the **PROGRAM** statement. This must be the first non-comment statement in every Clarion program. Notice that the keyword PROGRAM is indented in relation to the word **MyWin**. In Clarion, the only statements that begin in column one (1) of the source code file are those with a statement label. A label must begin in column one (1), by definition. The PROGRAM statement begins the Global data declaration section.

Next there is an empty **MAP** structure. The **END** statement is a required terminator of the MAP data structure. A MAP structure contains prototypes which define parameter data types, return data types, and various other options that tell the compiler how to deal with your procedure calls (this is all covered later in this tutorial). A MAP structure is required when you break up your program's code into PROCEDUREs. We haven't done that yet, but we still need it because there is an OPEN statement in the executable code.

When the compiler processes a MAP structure, it automatically includes the prototypes in the \CLARION5\LIBSRC\BUILTINS.CLW file. This file contains prototypes for almost all of the Clarion language built-in procedures (including the OPEN statement). If the empty MAP structure were not in this code, the compiler would generate an error on the OPEN statement.

MyWin is the label of the **WINDOW** data structure (the "M" must be in column one). In Clarion, windows are declared as data structures, and not dynamically built by executable code statements as in some other languages. This is one of the aspects of Clarion that makes it a 4GL. Although Clarion can dynamically build dialogs at runtime, it is unnecessary to do so. By using a data structure, the compiler creates the Windows resource for each dialog, enabling better performance at runtime.

The **('Hello Windows')** parameter on the **WINDOW** statement defines the title bar text for the window. The **SYSTEM** attribute adds a standard Windows system menu to the window. The **END** statement is a required terminator of the WINDOW data structure. In Clarion, all complex structures (both data and executable code) must terminate with an END or a period (.). This means the following code is functionally equivalent to the previous code:

```
  PROGRAM
  MAP.
MyWin   WINDOW('Hello Windows'),SYSTEM.
  CODE
  OPEN(MyWin)
  ACCEPT.
```

Although functionally equivalent, this code would become much harder to read as soon as anything is added into the MAP, WINDOW, or ACCEPT structures. By convention, we use the END statement to terminate multi-line complex statements, placing the END in the same column as the keyword it is terminating while indenting everything within the structure. We only use the period to terminate single-line structures, such as IF statements with single THEN clauses. This convention makes the code easier to read, and any missing structure terminators much easier to find.

The **CODE** statement is required to identify the beginning of the executable code section. Data (memory variables, data files, window structures, report structures, etc.) are declared in a data section (preceding the CODE statement), and executable statements may only follow a CODE statement.

Since this program does not contain any PROCEDUREs (we'll get to them in the next chapter), it only has a Global Data section followed by three lines of executable code. Variables declared in the Global Data section are visible and available for use anywhere in a program.

The **OPEN(MyWin)** statement opens the window, but does not display it. The window will only appear on screen when a DISPLAY or ACCEPT statement executes. This feature allows you to dynamically change the properties of the window, or any control on the window, before it appears on screen.

ACCEPT is the event processor. Most of the messages (events) from Windows are automatically handled internally for you by ACCEPT. These are the common events handled by the runtime library (screen re-draws, etc.). Only those events that actually may require program action are passed on by ACCEPT to your Clarion code. This makes your programming job easier by allowing you to concentrate on the high-level aspects of your program.

The ACCEPT statement has a terminating **END** statement, which means it is a complex code structure. ACCEPT is a looping structure, "passing through" all the events that the Clarion programmer might want to handle (none, in this program—we'll get back to this shortly), then looping back to handle the next event.

An ACCEPT loop is required for each window opened in a Clarion program. An open window "attaches" itself to the next ACCEPT loop it encounters in the code to be its event processor.

For this program, ACCEPT internally handles everything the system menu (placed on the window by the SYSTEM attribute) does. Therefore, when the user uses the system menu to close the window, ACCEPT automatically passes control to any statement immediately following its terminating END statement. Since there is no other explicit Clarion language statement to execute, the program ends. When any Clarion program reaches the end of the executable code, an implicit RETURN executes, which, in this case, returns the user to the operating system.

2. CLICK on the `Run` button.

   The program compiles and links, then executes. The window's title bar displays the "Hello Windows" message, and you must close the window with the system menu.

Hello Windows with Controls

The program you just created is the smallest program it is possible to create in Clarion. Now we'll expand on it a bit to demonstrate adding some controls to the window and handling the events generated by those controls.

Change the Source code

1. **Edit** the code to read:

```
    PROGRAM
    MAP
    END
  MyWin   WINDOW('Hello Windows'),AT(,,100,100),SYSTEM        !Changed
            STRING('Hello Windows'),AT(26,23),USE(?String1)   !Added
            BUTTON('OK'),AT(34,60),USE(?Ok),DEFAULT           !Added
          END
    CODE
    OPEN(MyWin)
    ACCEPT
      IF ACCEPTED() = ?Ok THEN BREAK.                         !Added
    END
```

NOTE: The Window Formatter is available to you in the Text Editor, just as it is in the Application Generator. To call the Window Formatter, place the insertion point anywhere within the WINDOW structure then press CTRL+F. The only restrictions are that the Control Template and Dictionary Field tools are unavailable (they are specific to the Application Generator).

The change is the addition of the **STRING** and **BUTTON** controls to the WINDOW structure. The STRING places constant text in the window, and the BUTTON adds a command buttton.

The only other addition is the **IF ACCEPTED() = ?Ok THEN BREAK.** statement. This statement detects when the user has pressed the OK button and BREAKs out of the ACCEPT loop, ending the program. The ACCEPTED procedure returns the field number of the control for which EVENT:Accepted was just generated (EVENT:Accepted is an EQUATE contained in the \CLARION5\LIBSRC\EQUATES.CLW file, which the compiler automatically includes in every program).

?Ok is the Field Equate Label of the BUTTON control, defined by the control's USE attribute (see *Field Equate Labels* in the *Language Reference*). The compiler automatically equates ?Ok to the field number it assigns the control (using Field Equate Labels helps make the code more readable).

When the ACCEPTED procedure returns a value equal to the compiler-assigned field number for the OK button, the BREAK statement executes and terminates the ACCEPT loop.

2. CLICK on the `Run` button.

   The program compiles and links, then executes. The window's title bar still displays the "Hello Windows" message, and now, so does the constant text in the middle of the window. You can close the window either with the system menu, or the OK button.

Common form Source code

There are other ways to write the code in the ACCEPT loop to accomplish the same thing. We'll go straight to the most common way, because this is more similar to the style of code that the Application Generator generates for you from the Clarion ABC Templates.

1. **Edit** the code to read:

```
  PROGRAM
  MAP
  END
MyWin   WINDOW('Hello Windows'),AT(,,100,100),SYSTEM
          STRING('Hello Windows'),AT(26,23),USE(?String1)
          BUTTON('OK'),AT(34,60),USE(?Ok),DEFAULT
        END
  CODE
  OPEN(MyWin)
  ACCEPT
    CASE FIELD()                !Added
    OF ?Ok                      !Added
      CASE EVENT()              !Added
      OF EVENT:Accepted         !Added
        BREAK                   !Added
      END                       !Added
    END                         !Added
  END
```

In this code you have one CASE structure nested within another. A CASE structure looks for an exact match between the expression immediately following the keyword CASE and another expression immediately following an OF clause (although these only show one OF clause each, a CASE structure may have as many as necessary).

The **CASE FIELD()** structure determines to which control the current event applies. When the FIELD procedure returns a value equal to the field number of the OK button (the ?Ok Field Equate Label) it then executes the CASE EVENT() structure.

The **CASE EVENT()** structure determines which event was generated. When the EVENT procedure returns a value equal to EVENT:Accepted (an EQUATE contained in the \CLARION5\LIBSRC\EQUATES.CLW file) it then executes the BREAK statement.

Nesting CASE EVENT() within CASE FIELD() allows you to put all the code associated with a single control in one place. You could just as easily nest a CASE FIELD() structure within a CASE EVENT() structure, reversing the code, but this would scatter the code for a single control to multiple places,.

2. CLICK on the **Run** button.

   Again, you can close the window either with the system menu, or the **OK** buttton, just as with the previous code, but now the code is structured in a common style.

Hello Windows with Event Handling

There are two types of events passed on to the program by ACCEPT: **Field-specific** and **Field-independent** events.

A **Field-specific** event occurs when the user does anything that may require the program to perform a specific action related to a single control. For example, when the user presses TAB after entering data in a control, the field-specific EVENT:Accepted generates for that control.

A **Field-independent** event does not relate to any one control but may require some program action (for example, to close a window, quit the program, or change execution threads).

Nesting two CASE structures as we just discussed is the most common method of handling field-specific events. The most common method of handling field-independent events is a non-nested CASE EVENT() structure, usually placed immediately before the CASE FIELD() structure.

Change the Source code

1. **Edit** the code to read:

```
  PROGRAM
  MAP
  END
MyWin   WINDOW('Hello Windows'),AT(,,100,100),SYSTEM
          STRING('Hello Windows'),AT(26,23),USE(?String1)
          BUTTON('OK'),AT(34,60),USE(?Ok),DEFAULT
        END
  CODE
  OPEN(MyWin)
  ACCEPT
    CASE EVENT()                          !Added
    OF EVENT:OpenWindow                   !Added
      MESSAGE('Opened Window')            !Added
    OF EVENT:GainFocus                    !Added
      MESSAGE('Gained Focus')             !Added
    END                                   !Added
    CASE FIELD()
    OF ?Ok
```

```
        CASE EVENT()
        OF EVENT:Accepted
          BREAK
        END
      END
    END
```

The new **CASE EVENT()** structure handles two field-independent events: EVENT:OpenWindow and EVENT:GainFocus. The **MESSAGE** procedure used in this code is just to visually display to you at runtime that the event was triggered. Instead of the MESSAGE procedure, you would add here any code that your program needs to execute when these events are triggered by the user.

This demonstrates the basic logic flow and code structure for procedural window procedures—an ACCEPT loop containing a CASE EVENT() structure to handle all the field-independent events, followed by a CASE FIELD() structure with nested CASE EVENT() structures to handle all field-specific events.

2. CLICK on the **Run** button.

   Notice that EVENT:GainFocus and EVENT:OpenWindow both generate when the window first displays (in that order). EVENT:GainFocus will re-generate when you ALT+TAB to another application then ALT+TAB back to *Hello Windows*.

Adding a PROCEDURE

In *Hello Windows* we have an example of a very simple program. Most modern business programs are not that simple—they require the use of Structured Programming techniques. This means you break up your program into functional sections that each perform a single logical task. In the Clarion language these functional sections are called PROCEDUREs.

First, we'll add a PROCEDURE to the *Hello Windows* program.

Change the Source code

1. **Edit** the code to read:

```
  PROGRAM
  MAP
Hello PROCEDURE                                         !Added
  END
  CODE                                                  !Added
  Hello                                                 !Added

Hello   PROCEDURE                                       !Added
MyWin   WINDOW('Hello Windows'),AT(,,100,100),SYSTEM
          STRING('Hello Windows'),AT(26,23),USE(?String1)
          BUTTON('OK'),AT(34,60),USE(?Ok),DEFAULT
        END
```

```
    CODE
    OPEN(MyWin)
    ACCEPT
      CASE EVENT()
      OF EVENT:OpenWindow
        MESSAGE('Opened Window')
      OF EVENT:GainFocus
        MESSAGE('Gained Focus')
      END
      CASE FIELD()
      OF ?Ok
        CASE EVENT()
        OF EVENT:Accepted
          BREAK
        END
      END
    END
```

The only changes are at the beginning of the program. Inside the **MAP** structure we now see the **Hello PROCEDURE** statement which prototypes the *Hello* procedure. A prototype is the declaration of the procedure for the compiler, telling the compiler what to expect when your code calls the procedure. This prototype indicates that the procedure takes no parameters and does not return a value. All PROCEDUREs in your program must be prototyped in a MAP structure. See *PROCEDURE Prototypes* in the *Language Reference* for more on prototypes.

The keyword **CODE** immediately following the MAP structure terminates the Global data section and marks the beginning of the Global executable code section, which only contains the **Hello** statement—a call to execute the Hello procedure. A PROCEDURE which does not return a value is always called as a single executable statement in executable code.

The second **Hello PROCEDURE** statement terminates the Global executable code section and marks the beginning of the code definition of the *Hello* procedure.

A PROCEDURE contains a data declaration section just as a PROGRAM does, and so, also requires the keyword **CODE** to define the boundary between data declarations and executable code. This is why the rest of the code did not change from the previous example. This is the Local data declaration section for the PROCEDURE.

The biggest difference between Global and Local data declarations is the scope of the declared data. Any data item declared in a Local data section is visible only within the PROCEDURE which declares it, while any data item declared in the Global data section is visible everywhere in the program. See *Data Declarations and Memory Allocation* in the *Language Reference* for a full discussion of all the differences.

2. **CLICK** on the **Run** button.

   The program executes exactly as it did before. The only difference is that the *Hello* PROCEDURE can now be called from anywhere within the program—even from within another PROCEDURE. This means that, even though the program may execute the procedure many times, the code for it exists just once.

Adding a PROCEDURE

A Clarion PROCEDURE which does not directly RETURN a value can only be called as a separate executable statement—it cannot be used in an expression, parameter list, or an assignment statement. A PROCEDURE which *does* directly RETURN a value must always contain a RETURN statement and may be used in expressions, parameter lists, and assignment statements. You can call a PROCEDURE which *does* directly RETURN a value as a separate statement if you do not want the value the PROCEDURE returns, but doing this generates compiler warnings (unless the PROCEDURE's prototype has the PROC attribute).

Structurally, both types of PROCEDURE are equivalent—they both have Local data sections, followed by the executable code section that begins with the keyword CODE.

Change the Source code

1. **Edit** the MAP structure to read:

```
  MAP
Hello       PROCEDURE
EventString PROCEDURE(LONG PassedEvent),STRING              !Added
  END
```

This adds the prototype for the **EventString PROCEDURE**. EventString receives a **LONG** parameter called **PassedEvent** that may not be omitted, and returns a **STRING**. The data types of all parameters passed to a PROCEDURE are specified inside the parentheses following PROCEDURE, each separated by a comma (if there are multiple parameters being passed). The data type of the return value of a PROCEDURE is specified following the closing parenthesis of the parameter list. Both types of PROCEDUREs may receive parameters, see the *Prototype Parameter Lists* section in the *Language Reference* for a more complete discussion of parameter passing.

2. **Add** the EventString PROCEDURE definition to the end of the file:

```
EventString   PROCEDURE(LONG PassedEvent)
ReturnString  STRING(255)
  CODE
  CASE PassedEvent
  OF EVENT:OpenWindow
    ReturnString = 'Opened Window'
  OF EVENT:GainFocus
    ReturnString = 'Gained Focus'
  ELSE
    ReturnString = 'Unknown Event: ' & PassedEvent
  END
  RETURN(ReturnString)
```

The **EventString** label (remember, it must be in column one) on the **PROCEDURE** statement names this procedure, while the parameter list attached to the PROCEDURE keyword names the **LONG** parameter **PassedEvent**. There must always be an equal number of parameter names listed on the PROCEDURE statement as there are parameter data types listed in the prototype for that PROCEDURE.

ReturnString is a local variable declared as a 255 character **STRING** field, on the stack. The **CODE** statement terminates the procedure's Local data section. The **CASE PassedEvent** structure should look familiar, because it is the same as a CASE EVENT() structure, but its CASE condition is the PassedEvent instead of the EVENT() procedure. This CASE structure simply assigns the appropriate value to the ReturnString variable for each event that is passed to the procedure.

The interesting code here is the **RETURN(ReturnString)** statement. A PROCEDURE without a return value does not require an explicit RETURN, since it always executes an implicit RETURN when there is no more code to execute. However, a PROCEDURE prototyped to return a value always contains an explicit RETURN statement which specifies the value to return. In this case, the RETURN statement returns whichever value was assigned to the ReturnString in the CASE structure.

3. **Edit** the Hello procedure's CASE EVENT() structure to read:

```
CASE EVENT()
OF EVENT:OpenWindow
  MESSAGE(EventString(EVENT:OpenWindow))            !Changed
OF EVENT:GainFocus
  MESSAGE(EventString(EVENT:GainFocus))             !Changed
END
```

This changes the **MESSAGE** procedures to display the returned value from the **EventString** procedure. The event number is passed to EventString as the event EQUATE to make the code more readable.

4. **CLICK** on the **Run** button.

   The program still executes exactly as it did before.

Moving Into the Real World—Adding a Menu

Hello Windows is a nice little demonstration program, but it really doesn't show you much to do with real-world business programming. Therefore, we're now going to expand it to include some real-world functionality, starting with a menu.

Change the Source code

1. **Edit** the beginning of the file to read:

```
  PROGRAM
  MAP
Main        PROCEDURE                               !Added
Hello       PROCEDURE
EventString PROCEDURE(LONG PassedEvent),STRING
  END
  CODE
  Main                                              !Changed
```

This adds the **Main PROCEDURE** prototype to the MAP structure and replaces the call to Hello with a call to the **Main** procedure.

2. **Add** the Main PROCEDURE definition to the end of the file:

```
Main      PROCEDURE
AppFrame  APPLICATION('Hello Windows'),AT(,,280,200),SYSTEM,RESIZE,MAX
            MENUBAR
              MENU('&File'),USE(?File)
                ITEM('&Browse Phones'),USE(?FileBrowsePhones)
                ITEM,SEPARATOR
                ITEM('E&xit'),USE(?FileExit),STD(STD:Close)
              END
              ITEM('&About!'),USE(?About)
            END
          END
  CODE
  OPEN(AppFrame)
  ACCEPT
    CASE ACCEPTED()
    OF ?About
      Hello
    END
  END
```

The **Main PROCEDURE** accepts no parameters. It contains the **AppFrame APPLICATION** structure. An APPLICATION structure is the key to creating Windows Multiple Document Interface (MDI) programs. An MDI application can contain multiple execution threads This is the MDI parent application frame that is required to create an MDI application.

The **MENUBAR** structure defines the menu items available to the user. The **MENU('&File')** structure creates the standard File menu that you see in most Windows programs. The ampersand (&) preceding the “F” specifies that the “F” is the menu’s accelerator key, and will be underlined at runtime by the operating system.

The **ITEM('&Browse Phones')** creates a menu item the we will use to call a procedure (we’ll get to that shortly). The **ITEM,SEPARATOR** statement creates a dividing line in the menu following the *Browse Phones* selection.

The **ITEM('E&xit')** creates a menu item to exit the procedure (and the program, since this procedure is the only procedure called from the Global executable code). The **STD(STD:Close)** attribute specifies the standard window close action to break out of the ACCEPT loop. This is why you don’t see any executable code associated with this menu item in the ACCEPT loop—the Clarion runtime library takes care of it for you automatically.

The **ITEM('&About!')** statement creates a menu item on the action bar, right next to the File menu. The trailing exclamation point (!) is a programming convention to give end-users a visual clue that this item executes an action and does not drop down a menu, despite the fact that it is on the action bar.

The **ACCEPT** loop contains only a **CASE ACCEPTED()** structure. The ACCEPTED procedure returns the field number of the control with focus when EVENT:Accepted is generated. We can use the ACCEPTED procedure here instead of the FIELD and EVENT procedures because menu items only generate EVENT:Accepted. The **OF ?About** clause simply calls the **Hello** procedure.

2. **CLICK** on the **Run** button.

   The program executes and only the **About!** and **File ➤ Exit** items actually do anything. Notice, though, that **File ➤ Exit** does terminate the program, despite the fact that we wrote no code to perform that action.

Really Moving Into the Real World—Adding a Browse and Form

Having a menu is nice, but now it's time to do some real-world business programming. Now we're going to add a data file and the procedures to maintain it.

Change the Global code

1. **Edit** the beginning of the file to read:

```
  PROGRAM
  INCLUDE('Keycodes.CLW')                           !Added
  INCLUDE('Errors.CLW')                             !Added
  MAP
Main           PROCEDURE
BrowsePhones   PROCEDURE                            !Added
UpdatePhones   PROCEDURE(LONG Action),LONG          !Added
Hello          PROCEDURE
EventString    PROCEDURE(LONG PassedEvent),STRING
  END
Phones         FILE,DRIVER('TopSpeed'),CREATE       !Added
NameKey          KEY(Name),DUP,NOCASE               !Added
Rec              RECORD                             !Added
Name               STRING(20)                       !Added
Number             STRING(20)                       !Added
                 END                                !Added
               END                                  !Added
InsertRecord     EQUATE(1)                          !Added
ChangeRecord     EQUATE(2)                          !Added
DeleteRecord     EQUATE(3)                          !Added
ActionComplete   EQUATE(1)                          !Added
ActionAborted    EQUATE(2)                          !Added
    CODE
    Main
```

The two new **INCLUDE** statements add the standard EQUATEs for keycodes and error numbers. Using the EQUATEs instead of the numbers makes your code more readable, and therefore, more maintainable.

The MAP structure has acquired two more procedure prototypes: the **BrowsePhones PROCEDURE** and **UpdatePhones PROCEDURE(LONG Action),LONG**. The BrowsePhones procedure will display a list of the records in the file and UpdatePhones will update individual records.

In the interest of coding simplicity (you'll see why shortly), the BrowsePhones procedure will simply display all records in the file in a LIST control. This is not exactly the same as a Browse procedure in template generated code (which is page-loaded in order to handle very large files), but will serve a similar purpose in this program.

Also in the interest of coding simplicity, UpdatePhones is a PROCEDURE which takes a parameter. The LONG parameter indicates what file action to take: ADD, PUT, or DELETE. The LONG return value indicates to the calling procedure whether the user completed or aborted the action. Again, this is not the same as a Form procedure in template generated code (which is a PROCEDURE using Global variables to signal file action and completion status), but will serve the same purpose in this simple program.

The **Phones FILE** declaration creates a simple data file using the TopSpeed file driver. There are two data fields: **Name** and **Number** which are both declared as **STRING(20)**. Declaring the Number field as a STRING(20) allows it to contain phone numbers from any country in the world (more about that to come).

The five **EQUATE** statements define constant values that make the code more readable. **InsertRecord**, **ChangeRecord**, and **DeleteRecord** all define file actions to pass as the parameter to the UpdatePhones procedure. The **ActionComplete** and **ActionAborted** EQUATEs define the two possible return values from the UpdatePhones procedure.

2. **Edit** the *Main* procedure's CASE ACCEPTED() code to read:

```
CASE ACCEPTED()
OF ?FileBrowsePhones                               !Added
  START(BrowsePhones,25000)                        !Added
OF ?About
  Hello
END
```

The **START(BrowsePhones,25000)** statement executes when the user chooses the Browse Phones menu selection. The START procedure creates a new execution thread for the BrowsePhones procedure and the second parameter (25000) specifies the size (in bytes) of the new execution thread's stack. You must use the START procedure when you are calling a procedure containing an MDI child window from the APPLICATION frame, because each MDI child window must be on a separate execution thread from the APPLICATION frame. If you do not START the MDI child, you get the

"Unable to open MDI window on APPLICATION's thread" runtime error message when you try to call the BrowsePhones procedure and the program immediately terminates.

Once you have started a new thread, an MDI child may simply call another MDI child on its same thread without starting a new thread. This means that, although BrowsePhones and UpdatePhones both contain MDI windows, only BrowsePhones must START a new thread, because it is called from the application frame. BrowsePhones will simply call UpdatePhones without starting a new thread.

Add the BrowsePhones PROCEDURE

1. **Add** the data section of the BrowsePhones PROCEDURE definition to the end of the file:

```
BrowsePhones  PROCEDURE
PhonesQue     QUEUE
Name            STRING(20)
Number          STRING(20)
Position        STRING(512)
              END
window  WINDOW('Browse Phones'),AT(,,185,156),SYSTEM,GRAY,RESIZE,MDI
          LIST,AT(6,8,173,100),ALRT(MouseLeft2),USE(?List) |
              ,FORMAT('84L|M~Name~80L~Number~'),FROM(PhonesQue),VSCROLL
          BUTTON('&Insert'),AT(20,117),KEY(InsertKey),USE(?Insert)
          BUTTON('&Change'),AT(76,117,35,14),KEY(EnterKey),USE(?Change)
          BUTTON('&Delete'),AT(131,117,35,14),KEY(DeleteKey),USE(?Delete)
          BUTTON('Close'),AT(76,137,35,14),KEY(EscKey),USE(?Close)
        END
```

The **PhonesQue QUEUE** structure defines the data structure that will contain all the records from the the Phones FILE to display in the LIST control. A Clarion QUEUE is similar to a data file because it has a data buffer and allows an indeterminate number of entries, but it only exists in memory at runtime. A QUEUE could also be likened to a dynamically allocated array in other programming languages. See the *Memory Queues* chapter in the *Language Reference* for more information on QUEUE structures.

The PhonesQue QUEUE contains three fields. The **Name** and **Number** fields duplicate the fields in the Phones FILE structure and will display in the two columns defined in the LIST control's FORMAT attribute. The **Position** field will contain the return value from the POSITION procedure for each record in the Phones FILE. Saving each record's Position will allow us to immediately re-get the record from the data file before calling UpdatePhones to change or delete a record.

The **window WINDOW** structure contains one LIST control and four BUTTON controls. The **LIST** control is the key to this procedure. The **ALRT(MouseLeft2)** on the LIST alerts DOUBLE-CLICK so the ACCEPT statement will pass an EVENT:AlertKey to our Clarion code. This will let us write code to bring up the UpdatePhones procedure to change the record the user DOUBLE-CLICKS on.

The vertical bar (**|**) at the end of the LIST statement is the Clarion line continuation character, meaning that the LIST control continues on the next line with the **FORMAT('84L|M~Name~80L~Number~')** attribute (you can put it all on one line if you want). The parameter to the FORMAT attribute defines the appearance of the LIST control at runtime.

It is best to let the List Box Formatter tool in the Window Formatter write format strings for you, since they can become very complex very quickly. This format string defines two columns. The first column is 84 dialog units wide (84), left justified(L), has a right border (|) that is resizable (M) and “Name” is its heading (~Name~). The second column is 80 dialog units wide (80), left justified(L), and “Number” is its heading (~Number~).

FROM(PhonesQue) on the LIST specifies the source QUEUE the LIST control will display, and **VSCROLL** adds the vertical scroll bar. The LIST will display the Name and Number fields of all entries in the PhonesQue while ignoring the Position field because the FORMAT attribute only specified two columns.

The LIST automatically handles users scrolling through the list without any coding on our part. The LIST does this because there is no IMM (immediate) attribute present. If there were an IMM attribute, we would have to write code to handle scrolling records (as the page-loaded Browse procedure template does).

The **BUTTON('&Insert')** statement defines a command button the user will press to add a new record. The **KEY(InsertKey)** attribute specifies that the button is automatically pressed for the user when they press INSERT on the keyboard. Notice that the other three BUTTON controls all have similar KEY atributes. This means you don’t have to write any special code to handle keyboard access to the program versus mouse access.

2. **Add** the beginning of the executable code section of the BrowsePhones PROCEDURE definition to the end of the file:

```
CODE
DO OpenFile
DO FillQue
OPEN(window)
ACCEPT
  CASE EVENT()
  OF EVENT:OpenWindow
    DO QueRecordsCheck
  OF EVENT:AlertKey
    IF KEYCODE() = MouseLeft2
      POST(EVENT:Accepted,?Change)
    END
  END
END
```

The beginning of the **CODE** section starts with the **DO OpenFile** statement. DO executes a ROUTINE, in this case, the OpenFile ROUTINE, and when the code in the ROUTINE is done executing, control returns to the next line of code following the DO statement that called the ROUTINE.

The code defining a ROUTINE must be at the bottom of the procedure, following all the main logic, because the ROUTINE statement itself terminates the executable code section of a PROCEDURE.

There are two reasons to use a ROUTINE within a PROCEDURE: to write one set of code statements that need to execute at multiple logical points within the procedure, or to make the logic more readable by substituting a single DO statement for a set of code statements that perform a single logical task.

In this case, the DO OpenFile statement serves the second purpose by moving the code that opens or creates the data file out of the main procedure logic. Next, the **DO FillQue** statement executes the code that fills the PhonesQue QUEUE structure with all the records from the data file. These two DO statements make it very easy to follow the logic flow ofthe procedure.

The **CASE EVENT()** structure's **OF EVENT:OpenWindow** clause executes **DO QueRecordsCheck** to call a ROUTINE that checks to see if there are any records in PhonesQue.

Next, the **OF EVENT:AlertKey** clause contains the **IF KEYCODE() = MouseLeft2** structure to check for DOUBLE-CLICK on the LIST. Since the ALRT(MouseLeft2) attribute only appears on the LIST control, we know that the **POST(EVENT:Accepted,?Change)** statement will only execute when the user DOUBLE-CLICKS on the LIST.

The POST statement tells ACCEPT to post the event in its first parameter (EVENT:Accepted) to the control in its second parameter (?Change). The effect of the POST(EVENT:Accepted,?Change) statement is to cause the EVENT:Accepted code for the Change button to execute, just as if the user had pressed the button with the mouse or keyboard.

This illustrates a very good coding practice: write specific code once and call it from many places. This is the structured programming concept that gave us PROCEDUREs and ROUTINEs. Even though the EVENT:Accepted code for the change button is not sectioned off separately in a PROCEDURE or ROUTINE, using the POST statement this way means that the one section of code is all you need to maintain—if the desired logic changes, you'll only have to change it in one place.

3. **Add** the rest of the executable code section of the BrowsePhones PROCEDURE definition to the end of the file:

```
      CASE FIELD()
      OF ?Close
        CASE EVENT()
        OF EVENT:Accepted
          POST(EVENT:CloseWindow)
        END
      OF ?Insert
        CASE EVENT()
        OF EVENT:Accepted
          IF UpdatePhones(InsertRecord) = ActionComplete
            DO AssignToQue
            ADD(PhonesQue)
            IF ERRORCODE() THEN STOP(ERROR()).
            SORT(PhonesQue,PhonesQue.Name)
            ENABLE(?Change,?Delete)
          END
          GET(PhonesQue,PhonesQue.Name)
          SELECT(?List,POINTER(PhonesQue))
        END
      OF ?Change
        CASE EVENT()
        OF EVENT:Accepted
          DO GetRecord
          IF UpdatePhones(ChangeRecord) = ActionComplete
            DO AssignToQue
            PUT(PhonesQue)
            IF ERRORCODE() THEN STOP(ERROR()).
            SORT(PhonesQue,PhonesQue.Name)
          END
          GET(PhonesQue,PhonesQue.Name)
          SELECT(?List,POINTER(PhonesQue))
        END
      OF ?Delete
        CASE EVENT()
        OF EVENT:Accepted
          DO GetRecord
          IF UpdatePhones(DeleteRecord) = ActionComplete
            DELETE(PhonesQue)
            IF ERRORCODE() THEN STOP(ERROR()).
            DO QueRecordsCheck
            SORT(PhonesQue,PhonesQue.Name)
          END
          SELECT(?List)
        END
      END
    END
    FREE(PhonesQue)
    CLOSE(Phones)
```

Following the CASE EVENT() structure is the **CASE FIELD()** structure. Notice that each OF clause contains its own **CASE EVENT()** structure, and each of these only contains an **OF EVENT:Accepted** clause. Because of this, we could have replaced the CASE FIELD() structure with a CASE ACCEPTED() structure and eliminated the nested CASEs. This would actually have given us slightly better performance—too slight to actually notice on-screen, though. The reason we didn’t is consistency; you will more often have occasion to trap more field-specific events than just EVENT:Accepted, and when you do, this nested CASE structure code is the logic to use, so it’s a good habit to make now. It also demonstrates the kind of code structure that is generated for you by Clarion’s templates in the Application Generator.

The **OF ?Close** clause executes the **POST(EVENT:CloseWindow)** when the user presses the Close button. Since there’s no second parameter to the POST statement naming a control, the event posts to the WINDOW (and should be a field-independent event). EVENT:CloseWindow causes the ACCEPT loop to terminate and execution control drops to the first statement following the ACCEPT’s terminating END statement. In this case, control drops to the **FREE(PhonesQue)** statement, which frees all the memory used by the QUEUE entries (effectively closing the QUEUE). The **CLOSE(Phones)** statement then closes the data file. Since there are no other statements to execute following CLOSE(Phones) the procedure executes an implicit RETURN and goes back to the *Main* procedure (where it was called from).

The **OF ?Insert** clause contains the **IF UpdatePhones(InsertRecord) = ActionComplete** structure. This calls the UpdatePhones PROCEDURE, passing it the InsertRecord constant value that we defined in the Global data section, and then checks for the ActionComplete return value.

The **DO AssignToQue** statement executes only when the user actually adds a record. AssignToQue is a ROUTINE that assigns data from the Phones FILE’s record buffer to the PhonesQue QUEUE’s data buffer. Then the **ADD(PhonesQue)** statement adds a new entry to PhonesQue. The **IF ERRORCODE() THEN STOP(ERROR()).** statement is a standard error check that you should execute after any FILE or QUEUE action that could possibly return an error (another good habit to form).

The **SORT(PhonesQue,PhonesQue.Name)** statement sorts the PhonesQue QUEUE entries alphabetically by the Name field. Since there is no PRE attribute on the PhonesQue QUEUE structure, you must reference its fields using Clarion’s Field Qualification syntax by prepending the name of the structure containing the field (PhonesQue) to the name of the field (Name), connecting them with a period (PhonesQue.Name). See *Field Qualification* in the *Language Reference* for more information.

The **ENABLE(?Change,?Delete)** statement makes sure the Change and delete buttons are active (if this was the first entry in the QUEUE, these buttons were dimmed out by the QueRecordsCheck ROUTINE). The **GET(PhonesQue,PhonesQue.Name)** statement re-gets the new record from the sorted QUEUE, then **SELECT(?List,POINTER(PhonesQue))** puts the user back on the LIST control with the new record highlighted.

The code in the **OF ?Change** clause is almost identical to the code in the OF ?Insert clause. There is an added **DO GetRecord** statement that calls a ROUTINE to put the highlighted PhonesQue entry’s related file record into the Phones file record buffer. The only other difference is the **PUT(PhonesQue)** statement that puts the user’s changes back in the PhonesQue.

The code in the **OF ?Delete** clause is almost identical to the code in the OF ?Change clause. The difference is the **DELETE(PhonesQue)** statement that removes the entry from the PhonesQue and the call to **DO QueRecordsCheck** to see if the user just deleted the last record.

4. **Add** the ROUTINEs called in the BrowsePhones PROCEDURE definition to the end of the file:

```
AssignToQue ROUTINE
  PhonesQue.Name = Phones.Rec.Name
  PhonesQue.Number = Phones.Rec.Number
  PhonesQue.Position = POSITION(Phones)

QueRecordsCheck ROUTINE
  IF NOT RECORDS(PhonesQue)
    DISABLE(?Change,?Delete)
    SELECT(?Insert)
  ELSE
    SELECT(?List,1)
  END

GetRecord ROUTINE
  GET(PhonesQue,CHOICE(?List))
  IF ERRORCODE() THEN STOP(ERROR()).
  REGET(Phones,PhonesQue.Position)
  IF ERRORCODE() THEN STOP(ERROR()).

OpenFile  ROUTINE
  OPEN(Phones,42h)
  CASE ERRORCODE()
  OF NoError
  OROF IsOpenErr
    EXIT
  OF NoFileErr
    CREATE(Phones)
    IF ERRORCODE() THEN STOP(ERROR()).
    OPEN(Phones,42h)
    IF ERRORCODE() THEN STOP(ERROR()).
  ELSE
    STOP(ERROR())
    RETURN
  END

FillQue ROUTINE
  SET(Phones.NameKey)
  LOOP
    NEXT(Phones)
    IF ERRORCODE() THEN BREAK.
    DO AssignToQue
    ADD(PhonesQue)
    IF ERRORCODE() THEN STOP(ERROR()).
  END
```

As you can see, there are five ROUTINEs for this procedure. Notice that, although these ROUTINEs look similar to a PROCEDURE, they do not contain CODE statements. This is because a ROUTINE shares the procedure's Local data and does not *usually* have a data declaration section of its own (a ROUTINE *can* have its own data section—see ROUTINE in the *Language Reference* for a full discussion of this issue).

The **AssignToQue ROUTINE** performs three assignment statements. The **PhonesQue.Name = Phones.Rec.Name** statement copies the data in the Name field of the Phones FILE record buffer and places it in the Name field of the PhonesQue QUEUE data buffer. Since there is no PRE attribute on the Phones FILE structure, you must also reference its fields using Clarion's Field Qualification syntax by stringing together the FILE name (Phones), the RECORD name (Rec), and the name of the field (Name), connecting them all with a period (Phones.Rec.Name). See *Field Qualification* in the *Language Reference* for more information.

The **PhonesQue.Number = Phones.Rec.Number** statement assigns the data in the Phones file's Number field to the PhonesQue's Number field. The **PhonesQue.Position = POSITION(Phones)** statement assigns the return value of the POSITION procedure to the PhonesQue.Position field. This value lets us retrieve from disk the one specific record that is currently in the Phones file's record buffer. The POSITION procedure does this for every Clarion file driver, and is therefore the recommended method of specific record retrieval across multiple file systems.

The **QueRecordsCheck ROUTINE** checks to see if there are any records in the PhonesQue. The **IF NOT RECORDS(PhonesQue)** structure uses the logical NOT operator against the return value from the RECORDS procedure. If RECORDS(PhonesQue) returns zero, then the NOT makes the condition true and the code following the IF executes (zero is always false and the NOT makes it true). If RECORDS(PhonesQue) returns anything other than zero, the code following the ELSE will execute (any non-zero number is always true and the NOT makes it false). Therefore, if there are no records in the PhonesQue, **DISABLE(?Change,?Delete)** executes to dim out the Change and Delete buttons, then the **SELECT(?Insert)** statement places the user on the Insert button (the next logical action). If there are records in the PhonesQue, then the **SELECT(?List)** statement places the user on the LIST.

The **GetRecord ROUTINE** synchronizes the Phones file's record buffer and PhonesQue's data buffer with the currently highlighted entry in the LIST. The **GET(PhonesQue,CHOICE(?List))** statement uses the CHOICE procedure to "point to" the currently highlighted entry in the LIST and GET the related QUEUE entry into the PhonesQue's data buffer (of course, always checking for unexpected errors). Then the **REGET(Phones,PhonesQue.Position)** statement uses the saved record position information to retrieve the Phones FILE record into the record buffer.

The **OpenFile ROUTINE** either opens or creates the Phones FILE. The **OPEN(Phones,42h)** statement attempts to open the Phones file for shared access. The second parameter (42h) is a hexadecimal number (signaled by the trailing “h”). Clarion supports the Decimal, Hexadecimal, Binary, and Octal number systems. This number represents Read/Write, Deny None access (fully shared) to the file (see the OPEN statement in the **Language Reference** for more on file access modes). We’re requesting shared access because this is an MDI program and the user could call multiple copies of this procedure in the same program. However, this program does not do all the concurrency checking required for a real multi-user application. See *Multi-User Considerations* in the *Programmer’s Guide* for more on the concurrency checking issue.

The **CASE ERRORCODE()** structure checks for any error on the OPEN. The **OF NoError OROF IsOpenErr** clause (now you can see why we included ERRORS.CLW file) executes the **EXIT** statement to immediately return from the ROUTINE. It is very important not to confuse EXIT with RETURN, since RETURN immediately terminates the PROCEDURE, while EXIT only terminates the ROUTINE. RETURN is valid to use within a ROUTINE, just be sure you want to terminate the PROCEDURE and not simply terminate the ROUTINE.

The **OF NoFileErr** clause detects that there is no data file to open.The **CREATE(Phones)** statement will then create an new empty data file for us. You must be sure that, if you intend to use the CREATE statement that the FILE declaration also contains the CREATE attribute, otherwise the CREATE statement will not be able to create the file for you. The CREATE statement does not open the file for processing, so that explains the second **OPEN(Phones,42h)** statement. The code in the **ELSE** clause executes if any error other than these occur. The **STOP(ERROR())** statement displays the ERROR procedure’s message to the user in a system modal window allowing the user the opportunity to abort the program (returning to Windows) or ignore the error. The **RETURN** statement then terminates the procedure if the user chooses to ignore the error.

The **FillQue ROUTINE** fills the PhonesQue QUEUE with all the records in the Phones file. The **SET(Phones.NameKey)** statement sets up the processing order and starting point for the Phones file. The Phone.NameKey parameter makes the processing order alphabetic based on the Name field. The absence of a second parameter to the SET statement makes the starting point the beginning (or end) of the file. The **LOOP** structure has no condition, which means you must place a BREAK statement somewhere in the LOOP or else get an infinite loop. The **NEXT(Phones)** statement retrieves the next record from the Phones data file, then the **IF ERRORCODE() THEN BREAK.** statement ensures that we will BREAK out of the LOOP when all the records have been read. The **DO AssignToQue** statement calls the AssignToQue ROUTINE that we’ve already discussed, and **ADD(PhonesQue)** adds the new record to the QUEUE.

Add the UpdatePhones PROCEDURE

1. **Add** the data section of the UpdatePhones PROCEDURE definition to the end of the file:

```
UpdatePhones  PROCEDURE(LONG Action)
ReturnValue   LONG,AUTO
window  WINDOW('Update Phone'),AT(,,185,92),SYSTEM,GRAY,RESIZE,MDI,MASK
          PROMPT('N&ame:'),AT(14,14),USE(?Prompt1)
          ENTRY(@s20),AT(68,13),USE(Phones.Rec.Name),REQ
          PROMPT('N&umber:'),AT(14,43),USE(?Prompt2)
          ENTRY(@s20),AT(68,42),USE(Phones.Rec.Number)
          BUTTON('OK'),AT(45,74),USE(?Ok),REQ,DEFAULT
          BUTTON('Cancel'),AT(109,74,32,14),USE(?Cancel)
        END
```

The **UpdatePhones PROCEDURE(LONG Action)** statement defines a PROCEDURE that receives a single LONG data typed parameter that will be called “Action” within the PROCEDURE (no matter what variable or constant is passed in).

ReturnValue LONG,AUTO declares a LONG variable that remains uninitialized by the compiler (due to the AUTO attribute). By default, memory variables in Clarion are initialized to all blanks or zero (depending on their data type). Specifying AUTO saves a bit of memory, but the caveat is that you must be sure you are going to assign a value to the uninitialized variable before you ever check its contents, otherwise you could create an intermittently occurring bug that would be really difficult to track down.

The **WINDOW** structure has the **MASK** attribute, which means that the user’s data entry patterns are checked as the data is input, instead of the default Standard Windows Behavior (SWB) of “free-form” data entry.

The two **PROMPT** and **ENTRY** controls combine to provide the user’s data entry controls. The two **BUTTON** controls will allow the user to complete or abort the current file action.

The PROMPT controls define the screen prompt text and the accelerator key to navigate to the ENTRY control following the PROMPT, The accelerator keys are formed using ALT plus the letter following the ampersand. For example, ('N&ame:') indicates ALT+A will give input focus to the ENTRY(@s20),AT(68,13),USE(Phones.Rec.Name) control.

The **USE** attributes of the ENTRY controls name the data fields that automatically receive the user’s input at runtime. The runtime library ensures that the current value in the variable named in the USE attribute displays when the control gains input focus. When the user enters data in the ENTRY control then moves on to another control, the runtime library ensures that the variable named in the USE attribute receives the current value displayed in the control.

The **REQ** attribute on the first ENTRY control means that the user cannot leave it blank, while the **REQ** attribute on the OK button checks to make sure that the user entered data into all the ENTRY controls with the REQ attribute. This required fields check is only done when the button with the REQ attribute is pressed, because the user may not have even gone to the the ENTRY with the REQ attribute.

2. **Add** the main logic of the UpdatePhones PROCEDURE definition to the end of the file:

```
CODE
OPEN(window)
DO SetupScreen
ACCEPT
  CASE FIELD()
  OF ?Phones:Rec:Number
    CASE EVENT()
    OF EVENT:Selected
      DO SetupInsert
    END
  OF ?Ok
    CASE EVENT()
    OF EVENT:Accepted
      EXECUTE Action
        ADD(Phones)
        PUT(Phones)
        DELETE(Phones)
      END
      IF ERRORCODE() THEN STOP(ERROR()).
      ReturnValue = ActionComplete
      POST(EVENT:CloseWindow)
    END

  OF ?Cancel
    CASE EVENT()
    OF EVENT:Accepted
      ReturnValue = ActionAborted
      POST(EVENT:CloseWindow)
    END
  END
END
RETURN(ReturnValue)
```

The **DO SetupScreen** statement calls the SetupScreen ROUTINE to perform some window initialization code. Notice that it follows the **OPEN(Window)** statement. When you are going to dynamically alter the window in a procedure, it must be opened first.

The **OF ?Phones:Rec:Number** clause in the **CASE FIELD()** structure demonstrates two important points. The first is the Field Equate Label, itself. The **USE(Phones.Rec.Number)** attribute contains periods in the field name and periods are not valid in Clarion labels. Therefore, to construct a Field Equate Label for Phones.Rec.Number, the compiler substitutes colons for the periods (because colons are valid in a Clarion label).

The second important point is the **OF EVENT:Selected** clause in the **CASE EVENT()** structure. EVENT:Selected generates when the control gains input focus but before the user gets to input data. The **DO SetupInsert** statement executes to offer the user an option then setup the display and data entry format of the ENTRY control.

The **OF EVENT:Accepted** code in **OF ?Ok** is the code that actually writes the record to disk. The **EXECUTE Action** structure executes exactly one of the **ADD(Phones)**, **PUT(Phones)**, or **DELETE(Phones)** statements.

An EXECUTE structure is similar to the IF and CASE structures in that it conditionally executes code based on the evaluation of a condition. The EXECUTE condition must evaluate to an integer in the range of 1 to *n* (where *n* is the number of code statements within the EXECUTE structure), then it executes the single ordinal line of code within the structure that corresponds to the value of the condition.

In this code, EXECUTE looks at the Action parameter then executes ADD(Phones) if the value of Action is one (1), PUT(Phones) if Action is two (2), or DELETE(Phones) if Action is three (3).

Generally, when you evaluate which Clarion code structure to use for an instance of conditional code execution (IF/ELSIF, CASE, or EXECUTE) the IF/ELSIF structure is the most flexible and least efficient, CASE is less flexible but much more efficient than IF/ELSIF, and EXECUTE is not flexible but highly efficient. Therefore, when the condition to evaluate can resolve to an integer in the range of 1 to *n*, use EXECUTE, otherwise try to use CASE. If CASE it is not flexible enough, then resort to IF/ELSIF.

The **IF ERRORCODE() THEN STOP(ERROR**()). statement will check for an error, no matter which statement EXECUTE performed. The **ReturnValue = ActionComplete** statement sets up the return to the calling procedure, signalling that the user completed the file action, then the **POST(EVENT:CloseWindow)** terminates the ACCEPT loop, dropping control to the **RETURN(ReturnValue)** statement.

The **OF ?Cancel** code does almost the same thing, without writing anything to disk. The **ReturnValue = ActionAborted** assignment statement sets up the return to the calling procedure, signaling that the user aborted the file action, then the **POST(EVENT:CloseWindow)** terminates the ACCEPT loop, dropping control to the **RETURN(ReturnValue)** statement.

3. **Add** the ROUTINEs called in the UpdatePhones PROCEDURE definition to the end of the file:

```
SetupScreen  ROUTINE
  CASE Action
  OF InsertRecord
    CLEAR(Phones.Rec)
    TARGET{PROP:Text} = 'Adding New Number'
  OF ChangeRecord
    TARGET{PROP:Text} = 'Updating ' & CLIP(Phones.Rec.Name) |
                          & '''s Phone Number'
    IF Phones:Rec:Number[1] <> '+'
        ?Phones:Rec:Number{PROP:Text} = '@P###-###-####P'
    END
  OF DeleteRecord
    TARGET{PROP:Text} = 'Deleting ' & CLIP(Phones.Rec.Name) |
                        & '''s Phone Number'
    DISABLE(FIRSTFIELD(),LASTFIELD())
    ENABLE(?Ok,?Cancel)
  END
```

```
SetupInsert  ROUTINE
  IF Action = InsertRecord
    CASE MESSAGE('International?','Format',ICON:Question, |
                                BUTTON:Yes+BUTTON:No,BUTTON:No,1)
    OF BUTTON:Yes
      TARGET{PROP:Text} = 'Adding New International Number'
      ?Phones:Rec:Number{PROP:Text} = '@S20'
      Phones:Rec:Number[1] = '+'
      DISPLAY
      SELECT(?,2)
    OF BUTTON:No
      TARGET{PROP:Text} = 'Adding New Domestic Number'
      ?Phones:Rec:Number{PROP:Text} = '@P###-###-####P'
    END
  END
```

The **SetupScreen ROUTINE** starts by evaluating **CASE Action**. When the user is adding a record (**OF InsertRecord**) the **CLEAR(Phones.Rec)** statement clears out the record buffer by setting all the fields to blank or zero. The **TARGET{PROP:Text} = 'Adding New Number'** statement uses Clarion's runtime property syntax to dynamically change the title bar text for the window to "Adding New Number." Clarion's property syntax allows you to dynamically change any property (attribute) of an APPLICATION, WINDOW, or REPORT structure in executable code. See *Appendix C - Property Assignments* in the *Language Reference* for more on properties.

TARGET is a built-in variable that always "points to" the currently open WINDOW structure. The curly braces ({}) delimit the property itself, and PROP:Text is an EQUATE (contained in PROPERTY.CLW, automatically included by the compiler like EQUATES.CLW) that identifies the parameter to the data element (in this case, the WINDOW statement).

The **OF ChangeRecord** code **TARGET{PROP:Text} = 'Updating ' & CLIP(Phones.Rec.Name) & '''s Phone Number'** does the same thing, but changes the title bar text to read "Updating *Someone*'s Phone Number." The ampersand (&) is the Clarion string concatenation operator and the CLIP(Phones.Rec.Name) procedure removes trailing spaces from the name. The **IF Phones:Rec:Number[1] <> '+'** structure checks for a plus sign in the first character of the Number string field. The plus sign is used here as a signal that the number is an international number.

Notice that Phones:Rec:Number[1] is addressing the first byte of the field as if it were an array. But you'll recall that there was no DIM attribute on the declaration (the DIM attribute declares an array). All STRING, CSTRING, and PSTRING data types in Clarion are also implicitly an array of STRING(1),DIM(SIZE(StringField)). This means you can directly refer to any single character in any string, whether it was declared as an array or not.

If the number is not international, the **?Phones:Rec:Number{PROP:Text} = '@P###-###-####P'** statement uses the same type of property syntax to change the control's entry picture token. Notice that PROP:Text is used to do this, just as it was used previously to change the window's title bar text. The reason is that PROP:Text refers to whatever is the parameter of the control.

Therefore, on a WINDOW('title text') it refers to the title text, and on an ENTRY(@S20) it refers to the picture token (@S20).

The **OF DeleteRecord** code is similar to the ChangeRecord code. The **DISABLE(FIRSTFIELD(),LASTFIELD**()) statement uses the FIRSTFIELD() and LASTFIELD() procedures to dim out all the controls on the window, then **ENABLE(?Ok,?Cancel)** un-dims just the OK and Cancel buttons.

The **SetupInsert ROUTINE** executes just before the user gets to the Number ENTRY control. The IF **Action = InsertRecord** checks Action and only executes the **CASE MESSAGE** structure when the user is adding a new record. The MESSAGE procedure can be used to create simple Yes/No choices for users. In this case, the user is asked whether the new number is International.

The **OF BUTTON:Yes** code executes when the user has pressed the Yes button on the MESSAGE dialog. The **TARGET{PROP:Text} = 'Adding New International Number'** statement changes the window's title bar text, then **?Phones:Rec:Number{PROP:Text} = '@S20'** changes the ENTRY control's picture token. The **Phones:Rec:Number[1] = '+'** statement places a plus sign in the first character position, then **DISPLAY** displays it and **SELECT(?,2)** places the user's insertion point at the second position in the current control.

The **OF BUTTON:No** code is similar, changing the window's title bar text and the control's entry picture token.

Update the Project file

Since we added a FILE structure to the program, we have to add its file driver to the Project so it can be linked into the program. If you don't, you'll get an error message something like "link error: TOPSPEED is unresolved in file hello.obj."

1. **Choose** **Project ➤ Edit...**

   The **Project** dialog appears.

2. **Highlight** **Database Driver Libraries** then **press** the **Add File...** button.
3. **Highlight** **TOPSPEED** then press the **OK** button.
4. **Press** the **OK** button to close the **Project** dialog.
5. CLICK on the **Run** button.

   The program executes.

ABC Template Generated OOP Code

When you examine the source code generated for you by the Application Generator, you'll see some fundamental differences from the code we just wrote. The reason for that is the Application Builder Class (ABC) templates generate code which extensively uses the ABC Library—a set of Object Oriented Programming (OOP) classes.

The code we just finished writing is Procedural code, not OOP code. Now we want to take a quick look at the generated OOP code to show you how what you just learned relates to the code you'll see generated for you. This will be just a quick look to highlight the major differences.

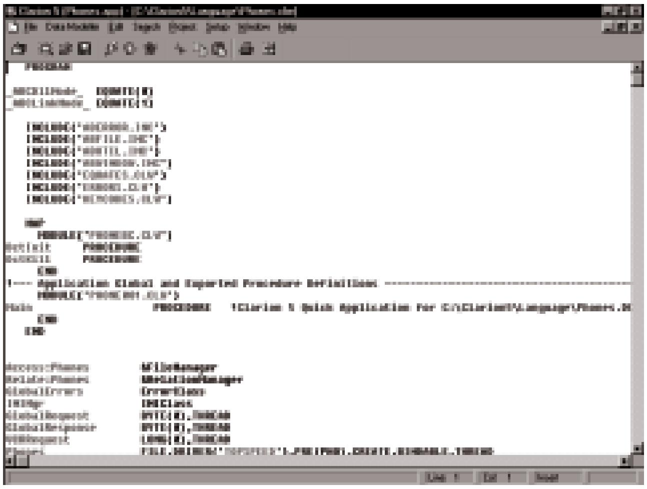
Quick Start an application

1. **Choose** **File ➤ New ➤ Application.**
2. **Type** *Phones* in the **File Name** field and check the **Quick Start** box, then press the **Save** button.
3. **Name** the file *Phones*, define *Name* and *Number* fields both with @S20 pictures, and make the *Name* field a *Duplicate* key, then press the **OK** button.
4. **Choose** **Project ➤ Generate** to generate source code.

Look at the Program Source

Now let's examine the source code that was generated for you.

1. **Choose** the **Module** tab and highlight *Phones.clw*.
2. RIGHT-CLICK and **choose** **Module** from the popup menu.



```
  PROGRAM

_ABCDllMode_  EQUATE(0)
_ABCLinkMode_ EQUATE(1)

  INCLUDE('ABERROR.INC')
  INCLUDE('ABFILE.INC')
  INCLUDE('ABUTIL.INC')
  INCLUDE('ABWINDOW.INC')
  INCLUDE('EQUATES.CLW')
  INCLUDE('ERRORS.CLW')
  INCLUDE('KEYCODES.CLW')

  MAP
    MODULE('PHONEBC.CLW')
DctInit     PROCEDURE
DctKill     PROCEDURE
    END
!--- Application Global and Exported Procedure Definitions --------------------------------
    MODULE('PHONE001.CLW')
Main                  PROCEDURE   !Clarion 5 Quick Application for C:\Clarion5\Language\Phones.
    END
  END


Access:Phones        &FileManager
Relate:Phones        &RelationManager
GlobalErrors         ErrorClass
INIMgr               INIClass
GlobalRequest        BYTE(0),THREAD
GlobalResponse       BYTE(0),THREAD
VCRRequest           LONG(0),THREAD
Phones               FILE,DRIVER('TOPSPEED'),PRE(PHO),CREATE,BINDABLE,THREAD
```

The PROGRAM Module

This is the **PROGRAM** module (the code should look similar to this, but may not be *exactly* the same). The basic structure of a Clarion OOP program is the same as the procedural. The first two statements are **EQUATE** statements which define constant values that the ABC Library requires. Following those are several **INCLUDE** statements. The INCLUDE statement tells the compiler to place the text in the named file into the program at the exact spot the INCLUDE statement.

```
        PROGRAM
  _ABCDllMode_  EQUATE(0)
  _ABCLinkMode_ EQUATE(1)
        INCLUDE('ABERROR.INC')
        INCLUDE('ABFILE.INC')
        INCLUDE('ABUTIL.INC')
        INCLUDE('ABWINDOW.INC')
        INCLUDE('EQUATES.CLW')
        INCLUDE('ERRORS.CLW')
        INCLUDE('KEYCODES.CLW')
```

The first four **INCLUDE** files (all starting with “AB” and ending with “.INC”) contain CLASS definitions for some of the ABC Library classes. The next three INCLUDE files (all ending with “.CLW”) contain a number of standard EQUATE statements used by the ABC Template generated code and ABC Library classes.

```
        MAP
          MODULE('PHONEBC.CLW')
  DctInit     PROCEDURE
  DctKill     PROCEDURE
          END
  !--- Application Global and Exported Procedure Definitions -----------
          MODULE('PHONE001.CLW')
  Main        PROCEDURE   !Clarion 5 Quick Application
          END
        END
```

The **MAP** structure contains two **MODULE** structures. The first declares two procedures **DctInit** and **DctKill** that are defined in the **PHONEBC.CLW** file. These two procedures are generated for you to properly initialize (and un-initialize) your data files for use by the ABC Library. The second **MODULE** structure simply names the application’s first procedure to call (in this case, *Main*).

The next two lines of code are your first OOP statements:

```
  Access:Phones       &FileManager
  Relate:Phones       &RelationManager
```

The **Access:Phones** statement declares a reference to a **FileManager** object, while the **Relate:Phones** statement declares a reference to a **RelationManager** object. These two references are initialized for you by the DctInit procedure, and un-initialized for you by the DctKill procedure. These are very important statements, because they define the manner in which you will adress the data file in your OOP code.

The next two lines of code declare a **GlobalErrors** object and an **INIMgr** object.

```
GlobalErrors    ErrorClass
INIMgr          INIClass
```

These objects handle all errors and your program's .INI file (if any), respectively. These objects are used extensively by the other ABC Library classes, so must be present (as you will shortly see).

```
GlobalRequest   BYTE(0),THREAD
GlobalResponse  BYTE(0),THREAD
VCRRequest      LONG(0),THREAD
Phones          FILE,DRIVER('TOPSPEED'),PRE(PHO),CREATE,BINDABLE,THREAD
KeyName           KEY(PHO:Name),DUP,NOCASE
Record            RECORD,PRE()
Name                STRING(20)
Number              STRING(20)
                  END
                END
```

Following that are three Global variable declarations which the ABC Templates use to communicate between procedures, followed by the **Phones FILE** declaration. Notice that the global variables all have the **THREAD** attribute. THREAD is required since the ABC Templates generate an MDI application by default, which makes it necessary to have separate copies of global variables for each active thread (which is what the THREAD attribute does).

The global **CODE** section only has six lines of code:

```
CODE
GlobalErrors.Init
INIMgr.Init('Phones.INI')
DctInit
Main
DctKill
GlobalErrors.Kill
```

The first two statements call **Init** methods (in OOP parlance, a procedure in a class is called a "method"—see *CLASS* in the *Language Reference* and the two OOP articles in the *Programmer's Guide* for more on OOP in general). These are the constructor methods for the **GlobalErrors** and **INIMgr** objects. You'll notice that the **INIMgr.Init** method takes a parameter. In the ABC Library, all object constructor methods are explicitly called and are named **Init**. There are several reasons for this. The Clarion language *does* support automatic object constructors (and destructors) and you are perfectly welcome to use them in any classes you write. However, automatic constructors cannot receive parameters, and many of the ABC Library Init methods must receive parameters. Therefore, for consistency, all ABC object constructor methods are explicitly called and named **Init**. This has the added benefit of enhanced code readability, since you can explicitly see that a constructor is executing, whereas with autiomatic constructors you'd have to look at the CLASS declaration to see if there is one to execute or not.

The **DctInit** procedure call initializes the **Access:Phones** and **Relate:Phones** reference variables so the template generated code (and any embed code that you write) can refer to the data file methods using **Access:Phones.Methodname** or **Relate:Phones.Methodname** syntax. This gives you a consistent way to reference any file in an ABC Template generated program—each FILE will have corresponding Access: andRelate: objects.

The call to the **Main** procedure begins execution of the rest of your program for your user. Once the user returns from the Main procedure, **DctKill** and **GlobalErrors.Kill** perform some necessary cleanup operations before the return to the operating system.

The UpdatePhones Module

Now let’s examine the source code that was generated for you for one of your procedures. We’ll look at the *UpdatePhones* procedure as a representative, since all ABC Template generated procedures will basically follow the same form (again, your code should look similar to this, but may not be *exactly* the same).

1. **Choose** **File ➤ Close.**
2. **Highlight** *UpdatePhones*, then RIGHT-CLICK and **choose** **Module** from the popup menu.

The first thing to notice is the **MEMBER** statement on the first line. This is a required statement telling the compiler which PROGRAM module this source file “belongs” to. It also marks the bginning of a Module Data Section—an area of source code where you can make data declarations which are visible to any procedure in the same source module, but not outside that module (see *Data Declaration and Memory Allocation* in the *Language Reference*).

```
MEMBER('Phones.clw')        ! This is a MEMBER module
INCLUDE('ABRESIZE.INC')
INCLUDE('ABTOOLBA.INC')
INCLUDE('ABWINDOW.INC')
MAP
  INCLUDE('PHONE004.INC')   !Local module prodecure declarations
END
```

The three **INCLUDE** files contain CLASS definitions for some of the ABC Library classes. Notice that the list of INCLUDE files here is different than the list at the global level. You only need to INCLUDE the class definitions that the compiler needs to know about to compile this single source code module. That’s why the list of INCLUDE files will likely be a bit different from module to module.

Notice the **MAP** structure. By default, the ABC Templates generate “local MAPs” for you containing **INCLUDE** statements to bring in the prototypes of the procedures defined in the module and any procedures called from the module. This allows for more efficient compilation, because you'll only get a global re-compile of your code when you actually change some global data item, and not just by adding a new procedure to your application. In this case, there are no other procedures called form this module.

The **PROCEDURE** statement begins the **UpdatePhones** procedure (which also terminates the Module Data Section).

```
UpdatePhones PROCEDURE      !Generated from procedure template - Window

CurrentTab            STRING(80)
FilesOpened           BYTE
ActionMessage         CSTRING(40)
History::PHO:Record   LIKE(PHO:RECORD),STATIC
```

Following the **PROCEDURE** statement are four declaration statements. The first two are common to most ABC Template generated procedures. They provide local flags used internally by the template generated code. The **ActionMessage** and **History::PHO:Record** declarations are specific to a Form procedure. They declares a user message and a "save area" for use by the Field History Key ("ditto" key) functionality provided on the toolbar.

After the WINDOW structure comes the following object declarations:

```
ThisWindow            CLASS(WindowManager)
Ask                     PROCEDURE(),VIRTUAL
Init                    PROCEDURE(),BYTE,PROC,VIRTUAL
Kill                    PROCEDURE(),BYTE,PROC,VIRTUAL
                      END
Toolbar               ToolbarClass
ToolBarForm           ToolbarUpdateClass
Resizer               WindowResizeClass
```

The last three are simple object declarations which create the local objects which will enable the user to use the toolbar and resize the window at runtime. The interesting code here is the **ThisWindow CLASS** declaration. This **CLASS** structure declares an object derived from the **WindowManager** class in which the **Ask**, **Init**, and **Kill** methods of the parent class (WindowManager) are overridden locally. These are all **VIRTUAL** methods, which means that all the methods inherited from the WindowManager class will be able to call the overridden methods. This is a very important concept in OOP (see *CLASS* in the *Language Reference* and the two OOP articles in the *Programmer's Guide* for more on this).

Following that comes all of the executable code in your procedure:

```
  CODE
  GlobalResponse = ThisWindow.Run()
```

That's right—one single, solitary statement! The call to **ThisWindow.Run** is the only executable code in your entire procedure! So, you ask, "Where's all the code that provides all the funtionality I can obviously see happening when I run the program?" The answer is, "In the ABC Library!" or, at least most of it is! The good news is that all the standard code to operate any procedure is built in to the ABC Library, which makes your application's “footprint” very small, since all your procedures share the same set of common code which has been extensively debugged (and so, is not likely to introduce any bugs into your programs).

All the functionality that must be explicit to this one single procedure is generated for you in the overridden methods. In this procedure’s case, there are only three methods that needed to be overridden. Depending on the functionality you request in the procedure, the ABC Templates will override different methods, as needed. You also have embed points available in every method it is possible to override, so you can easily “force” the templates to override any method for which you need slightly different functionality by simply adding your own code into those embed points (using the Embeditor in the Application Generator).

OK, so let’s look at the overridden methods for this procedure.

```
ThisWindow.Ask PROCEDURE
  CODE
  CASE SELF.Request
  OF InsertRecord
    ActionMessage = 'Adding a Phones Record'
  OF ChangeRecord
    ActionMessage = 'Changing a Phones Record'
  END
  QuickWindow{Prop:Text} = ActionMessage
  PARENT.Ask
```

The really interesting line of code in the **ThisWindow.Ask PROCEDURE** is last. The last statement, **PARENT.Ask**, calls the parent method that this method has overridden to execute its standard functionality. The **PARENT** keyword is very powerful, because it allows an overridden method in a derived class to call upon the method it replaces to “do its thing” allowing the overridden method to incrementally extend the parent method’s functionality.

```
ThisWindow.Init PROCEDURE()
  CODE
  SELF.Request = GlobalRequest
  IF PARENT.Init() THEN RETURN Level:Notify.
  SELF.FirstField = ?PHO:Name:Prompt
  SELF.VCRRequest &= VCRRequest
  SELF.Errors &= GlobalErrors
  CLEAR(GlobalRequest)
  CLEAR(GlobalResponse)
  SELF.AddItem(ToolBar)
  SELF.AddUpdateFile(Access:Phones)
  SELF.HistoryKey = 734
  SELF.AddHistoryFile(PHO:Record,History::PHO:Record)
  SELF.AddHistoryField(?PHO:Name,1)
  SELF.AddHistoryField(?PHO:Number,2)
  SELF.AddItem(?Cancel,RequestCancelled)
  Relate:Phones.Open
  FilesOpened = True
  SELF.Primary &= Relate:Phones
  SELF.OkControl = ?OK
  IF SELF.PrimeUpdate() THEN RETURN Level:Notify.
```

```
    OPEN(QuickWindow)
    SELF.Opened=True
    Resizer.Init(AppStrategy:Surface,Resize:SetMinSize)
    SELF.AddItem(Resizer)
    Resizer.AutoTransparent=True
    ToolBarForm.HelpButton=?Help
    SELF.AddItem(ToolbarForm)
    SELF.SetAlerts()
    RETURN Level:Benign
```

There are several interesting lines of code in the **ThisWindow.Init PROCEDURE**. This is the **ThisWindow** object's constructor method, so all the code in it performs the initialization tasks specifically required by the **UpdatePhones** procedure.

The first statement, **SELF.Request = GlobalRequest**, retrieves the global variable's value and places it in the **SELF.Request** property. **SELF** is another powerful Clarion OOP keyword, which always means "the current object" or "me." SELF is the object prefix which allows class methods to be written generically to refer to whichever object instance of a class is currently executing.

The second statement calls the **PARENT.Init()** method (the parent method's code to perform all its standard functions) before the rest of the procedure-specific initialization code executes. Following that are a number of statements which initialize various necessary properties. The **Relate:Phones.Open** statement opens the Phones data file for processing, and if there were any related child files needed for Referential Integrity processing in this procedure, iyt would also open them (there aren't, in this case).

```
  ThisWindow.Kill PROCEDURE()
    CODE
    IF PARENT.Kill() THEN RETURN Level:Notify.
    IF FilesOpened
      Relate:Phones.Close
    END
```

In addition to calling the **PARENT.Kill()** method to perform all the standard closedown functionality (like closing the window), **ThisWindow.Kill** closes all the files opened in the procedure, then sets the **GlobalResponse** variable.

3. **Choose File ➤ Close.**

Where to Go From Here

This tutorial has been just a brief introduction to the Clarion programming language and the ABC Template generated code. There is much more to the Clarion language and the ABC Library than has been covered here, so there's a lot more to learn. So where do you go from here?

- The articles in the *Programmer's Guide.* These essays cover various aspects of programming in the Clarion language. Although they do not take a tutorial approach, they do provide in depth information on the specific areas they cover, and there are several articles which deal specifically with OOP in Clarion.
- The *Application Handbook* fully documents the ABC Library and ABC Templates. This is your prime resource for how to get the most out of Clarion's Application Generator technology.
- The *Language Reference* is the "Bible" of the Clarion language. Reading the entire manual is always a good idea.
- Examine and dissect source code generated for you by the Application Generator. After doing this tutorial, the basic structure of the code should look familiar, even if the specific logic does not.
- The *User's Guide* contains tutorials on using the Debuggers. This will allow you to step through your Clarion code as it executes to see exactly what effect each statement has on the operation of your program.
- Take one of the educational seminars that TopSpeed Corp. offers. Call Customer Service at (800) 354-5444 or (954) 785-4555 to enroll.
- Join (or form) a local Clarion User's Group and participate in joint study projects.
- Participate in TopSpeed's forum on CompuServe (GO TOPSPEED) or the Internet Newsgroup (*comp.lang.clarion*) to network with other Clarion programmers all around the world (**Strongly recommended!**).

INDEX

NOTES