6

Order Entry Abstract and Design

This is the first of the "real world implementation" examples that will comprise the remainder of the book. In this case, the client is looking to do a "proof of concept" project in order to evaluate the viability of Linux and MySQL. The client, Specialty Electrical Supply, Inc. (SESI), has chosen an application that needs to be done: entering orders phoned in by customers. They are interested in the low price point of the Linux operating system and would consider converting to an all-Linux shop if they could be reasonably certain they would not have to hire an expensive server administrator.

The main goal of this chapter is to set out a basic design for a simple GTK+ and MySQL application. The project in question is entering orders for the client, SESI. Although this chapter does not attempt to outline a "formal specification," it does set out to describe the problem in sufficient detail so you can understand what the application should do and why. Its purpose is to give you a "big picture" view of the application so that when the application is built (in Chapter 7, "Construction of the SESI Order Entry Applications"), you will be able to understand how things work together, and you will have a point of reference to understand the coding process. There is an appendix in this book that goes along with this chapter and Chapter 7. It is Appendix A, "Glade-Generated Files from the SESI Order Application."

Currently, SESI uses a system that's based on Excel spreadsheets and Word documents, and one of the managers, who has a strong technical background, manages the LAN as one of his part-time duties. Their customer and product master files are Excel spreadsheets, and they cut and paste orders to Word. They then send the order to the PC's printer for a quick walk down to the warehouse—a very tedious process. The goal is to replace this with a GTK+ windowing interface that can then output each order to a separate text file. Then these text files can be sent to the individual(s) responsible for filling them by FTP, email, or whatever other mechanism may be chosen. Further, the next obvious step will be to create an application that the worker on the shop floor who is filling the order can use to record what was shipped and when and what had to go on backorder. For those of you more on the business end, this will be the "bookings" part of "booking/billing/backlog."

Problem Definition and Design Issues

This section describes the problem in greater detail and works through some design decisions. Because this is a relatively simple problem and should be a straightforward implementation, this chapter does not go into much of the detailed design that a large multi-person project would need in order to ensure success.

Hardware Specifications

A PC will have Linux installed on it; the hard drive will be wiped, and a new Linux distribution will be installed: Red Hat 6.2 with the GNOME desktop (the version current when this book was written, Red Hat 7.x should work fine when it comes out). This is the out-of-the-box default configuration for Red Hat 6.2 (for a total cost of approximately $30 so far). The target machine will probably be a first-generation Pentium box with a 1- or 2-gigabyte hard drive; although you can put Linux on a hard drive as small as 300 megabytes, a 1-gigabyte hard drive is probably the minimum hard drive that should be used. There will be no need for modem/dialout, printing, and so on from this machine; initially, it will exist solely to run this application.

Network Information

The client already has a static IP LAN for file sharing and drive mapping under MS Windows. The existing LAN consists of 15 machines, all running either Windows 95 or Windows 98. There is no central server. No one else will need to access the Linux box except the data entry operator and the LAN administrator.

Existing Data Information

The "customer master file" and "product master file" are currently stored in Microsoft Excel spreadsheets. The data is relatively straightforward; no more information is kept than is needed. The client is confident of the data stored in the spreadsheets; only the data entry operator has touched or altered them in two years. Key values, such as "item number," are unique, addresses are complete and correct, and so on. These files will be output to text files, transferred to the Linux box, and then uploaded to the MySQL database using MySQL utilities (see Listing 6.1).

After that, the database will be considered the source of record for this information, and the spreadsheets will be retired.

The item master file currently has about 600 items in it, and it changes only once or twice a month, if at all. The customer master file currently has about 170 records in it and changes as needed. The user must be able to add a new customer quickly and easily while that customer is on the phone.

Existing Process Specification

The existing process works as described here:

- The data entry clerk receives a phone call (or retrieves the information from the answering machine) with the order information. He or she uses Notepad to enter the following information: customer information, the order, and any special instructions.
- The data entry clerk processing the order then opens three files: the customer and product master files in Excel, and a Word document (template) that will become the order form.
- Using cut and paste, he or she fills out the Word template with the necessary information until the Word document is complete. At that point, it is sent to the printer. The cut-and-paste operation is the single most tedious, time-consuming part of the whole operation because of the differences in format between Excel and Word; for example, you must put in carriage returns (vice tab characters) to separate address lines and so on.
- Several times a day, whatever orders come out of the printer are taken across the building to the warehouse and are given to the individual who will fill the order.

The Desired Process

In the "new improved" process, the data entry clerk will have access to two PCs—one with Windows and the other with Linux. He or she will be able to click an icon for a new order select from a pick list of existing customers, edit their customer information, or enter a new one if needed. The customer information will be filled in and displayed. From there, the data entry clerk can proceed to enter the items ordered by the customer. It is possible to directly type in the item number if it is available, select it from a list of all items, or search on a keyword of the character (not numeric) fields of the item master file.

When the order is complete, the application will write the order to the hard disk in the form of a text file (there will be a command button in the application for this, the data entry clerk will initiate it). The title of the text file will be the name of the customer and the date and time the file was created. Each order will be stored in a separate text file and formatted for quick printout on a printer in the order fulfillment office. Although the current Word document has some formatting, this is not necessary to the work flow and will be done away with.

At the desired time, as desired by either the person creating the order or the people in the order fulfillment section, the completed order text files will be transferred to the PC in the order fulfillment office. The mechanism for this could be any of the following:

- FTP. This method would be either a push operation instigated by the data clerk (i.e., sending the data from the Linux box to the PC in order fulfillment) or a pull operation instigated by the order fulfillment office staff (for example, someone in the order fulfillment office logs in to the Linux box and "pulls" the files over the network). The only potential problem with a "pull" operation is the possibility that a file will be pulled during its write operation.
- Email
- Mapped drives, via Samba
- Other similar mechanism

Whatever mechanism is used, it will be manually initiated by one of the concerned parties. From that point, the order fulfillment office will print each of the text files in turn. Once done, the files will be moved to an archive directory on the local PC. Again, this might be a batch file, or even a manual process. The users in order fulfillment will know how to do this, or they can be taught. From that point forward, the process will proceed as before.

Desired Characteristics of the Software

The following characteristics and other concepts should be included or excluded at the application level:

- The application should allow the user to keep his or her hands on the keyboard as much as possible, avoiding the mouse.
- There is no need for auditing or detailed logging. That is, the application doesn't need to record who changed what data and when. First, the data is not that sensitive, and second, because the data entry clerk is so familiar with the data, he or she can be relied on to understand and solve any problems that come up.
- There is no need for security at the application level beyond that which is standard for a PC (for example, a login). Again, the data is not that sensitive, and after it is processed into orders, it will be copied or transferred from the Linux machine anyway.
- The MySQL database server, client, and application will all be on the same machine.
- There won't be any need to save orders, read or edit existing orders; the process will proceed from start to finish, or else all order information will be lost. This is considered acceptable because an order rarely exceeds 20 items. So, if a customer gets halfway through an order and needs to call back later to finish, there will not be a significant penalty to start over from the beginning. Because of this, each instance of the application can be a different order and can remain open all day if necessary.

User Interface

This section outlines the design layout of the user interface. Going through this step saves a lot of work later on.

The Customer Data Form: frm_main

Figure 6.1 shows the design of the initial screen the user will see when the application opens. This form allows the data entry clerk to select and display customer information or enter the information for a new customer.

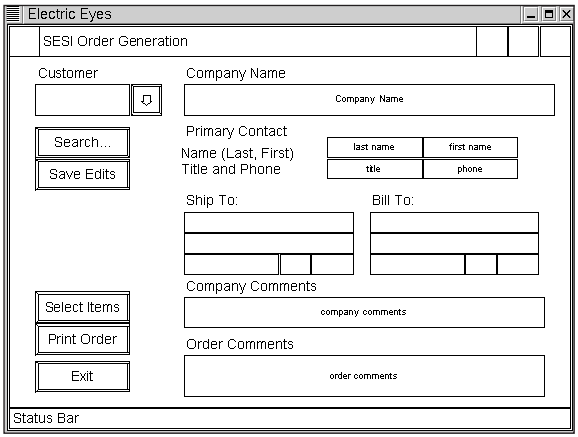

Figure 6.1 frm_main is the initial form the user sees when he starts the application.

Following are some of the features and functions of frm_main:

- The customer number drop-down box (Customer) contains all the customer numbers in the database, plus the word "New" as the first entry in the list. When the user selects New, all the text boxes are cleared, and the software generates a new customer number.
- The Save Edits button is active only if changes have been made to one of the database fields. To avoid disrupting the work flow, the application will not prompt the user if he or she attempts to close the application or change customers while edits are in progress. The Save Edits button provides a visual key to indicate that the data has changed and action must be taken to save the changes.
- There is no menu bar (this keeps the application simple).

The container layout for the form is shown in Figure 6.2. This is part of the GTK+ environment that must be planned for, and it is something that will be new to developers coming from the VB and VC++ environments. For a review of GTK+ and the concept of containers, refer to Chapter 2, specifically the sections titled "VB's 'Form' Reincarnated as GtkWindow Plus GtkFixed or GtkLayout" and "GTK+ Containers as Resizer Controls."

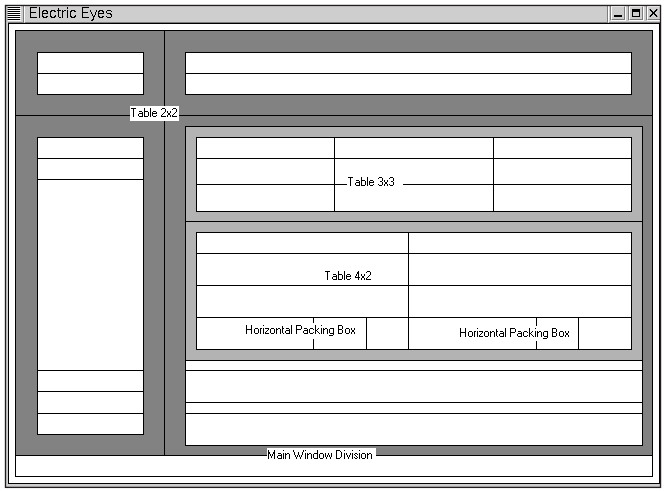

Figure 6.2 frm_main container layout.

As you can see from Figure 6.2, the first widget is a vertical packing box (vbox) with a space count of 2. Please note that in Figure 6.2 the shades of gray add to the visibility; they have nothing to do with the finished product. Initially, the window is a vertical packing box divided into two rows. The widget in the bottom half of that vbox widget is the status bar, and the top is for everything else. Inside the top half of that vbox widget, the first widget to be added is a 2-by-2 table. In the top left and right of that 2-by-2 table are vertical packing boxes, again with space for two widgets each. The bottom left of the 2-by-2 table contains a vertical button box because that vbox will have only command buttons. In the bottom right of the 2-by-2 table is a vertical packing box with space for six widgets. The first of these six widgets holds a 3-by-3 table, the second holds a 4-by-2 table, and the others have spaces for labels and text boxes. This fills out frm_main.

Selecting Items for the Order: frm_items_ordered

The other main form in the application is a window in which the user picks the individual items the caller has ordered and then tallies the totals. This makes up the line items on the invoice (see Figure 6.3).

Some of the functionality of frm_items_ordered is described here:

- The user should be able to add items quickly using the widgets in the upper-left corner. That is, when the user enters the item number and the quantity and clicks Add, the added item shows in the 4 Column CList box. This is intended to be the primary means of order entry because the customer should know the item number she wants to purchase when she calls.

- Alternatively, the user can select the item the customer wants to purchase and then click the Add button between the list boxes, and the item is added to the 4 Column CList box using the quantity from the spin box between the list boxes.
- The Remove button removes only the selected item from the 4 Column CList box.

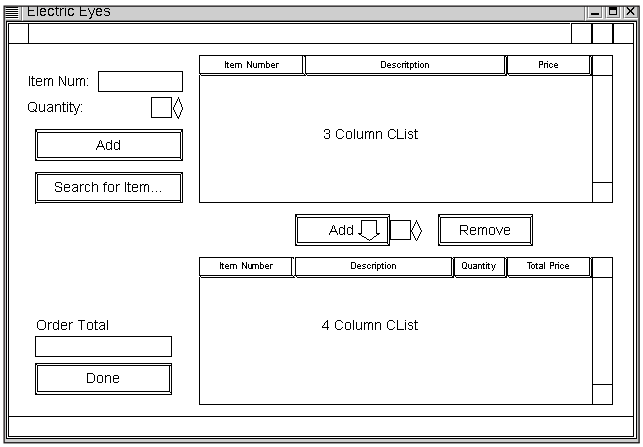

Figure 6.3 The frm_items_ordered layout diagram.

The Search Windows

Each of the windows covered previously has a Search for... button of some type. In frm_main, it is a search for customers. In frm_items_ordered, it is a search through the item master table. These buttons function very similarly: When the user clicks the Search button, a new modal window appears (see Figure 6.4). There, the user enters the search string and clicks the Find button, and the software returns a list of matching records. The search functions as a matching search with a wildcard both before and after the search string; that is, if the user searches for "eat", the list of returned items might include "treat," "eatery," and so on.

The application automatically searches all character columns. When doing a customer search, it attempts to find a match in all the character (non-numeric) fields: name, bill_to_addr1, bill_to_addr2, and so on. This minimizes the complexity presented to the user, and with this implementation (a small data set and all functions performed on the same machine), response time should be more than adequate.

In the Customer search form, the user types a text string into the text box and then clicks Find. The software searches all text fields and returns all records matching any part of the field text. If the desired customer record is found, the user can select it and close the search window, and the selected customer record data automatically appears in the widgets on frm_main.

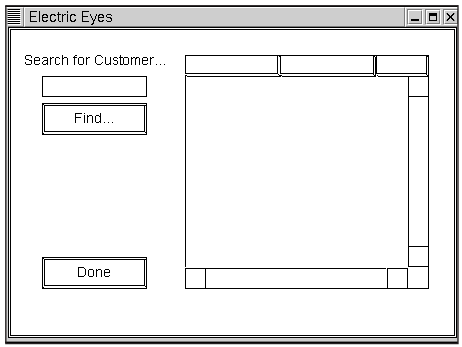

Figure 6.4 The Search for Customer modal window design.

The Item search form works the same as the Customer search form, except that it searches and returns items and fills in the "items ordered" Clist window on frm_items_ordered (refer to Figure 6.3).

In both of the search forms, when the user finds the customer or item they are searching for, they select it and click Done. The software automatically places the data into the correct place on the form. For frm_main, it displays the customer selected by the find operation, and for frm_items_ordered, it selects and displays that item in the 3 Column CList box that displays all the items in the database.

Creating the Database

In this section, you will create and fill the database with the initial data load. You will put all the statements except the `create database` into a file, which you can use to reload the database later if necessary.

Listing 6.1 assumes that a MySQL server is installed and running on the target machine.

First, start the MySQL command line tool by typing this line:

```
% mysql
```

This changes the prompt to `mysql>`.

Then create the database:

```
mysql> CREATE DATABASE sesi;
```

Listing 6.1 creates the SESI database for Chapters 6 and 7. It drops and refreshes the tables with the data being transferred in from the sample text files.

Usage (from the command prompt):

```
%mysql < listing.6.1
```

or

```
%mysql -t < listing.6.1
```

The `-t` option sends the output of the `select` statements to the screen in a table-like format (like you would get from the `mysql>` prompt).

As you can see, C-style comments will work (note the sections that start with `/*` and end with `*/`). However, be careful not to use any semicolons, single quotation marks, or double quotation marks inside the comments! This can cause the parser to think there is something it should be handling, in which case it will issue an error and terminate.

Listing 6.1 **Create and Fill the Database For Specialty Electrical Supply, Inc.**

```
use sesi;

DROP TABLE if exists tbl_items;

CREATE TABLE tbl_items
(
   item           varchar(8)     NOT NULL    PRIMARY KEY,
   description    varchar(50)    NOT NULL,
   price          decimal(4,2)   NOT NULL
);

/* In the following statement and the second LOAD TABLE statement
 * further in the file, note that the fully qualified path name
 * has been put in for the file. This is necessary because MySQL
 * will look in the database directory by default, not in the current
 * local directory or your project directory.

 * Also note that the text files being imported are tab-delimited,
 * and they have the same number of fields as the table has columns.

 * In the following statement, the new_master_item.txt file has been
 * output from another application (probably a spreadsheet) and
 * copied into place so that the LOAD DATA INFILE statement will
 * work correctly. It contains the master list of items that the
 * company sells.
 *
 * Remember: the "/mnt/DOS_hda2..." path name is the one for the
 * test machine that was used to build this application.
 * Substitute the location you use on your machine.
 */

LOAD DATA INFILE "/mnt/DOS_hda2/newriders/book/ch6/new_item_master.txt"
   INTO TABLE tbl_items;


/* Check for the success of the import. It should be approximately
 * 600 records. Compare that to the text file in the statement for the
 * exact count. */

select count(*) from tbl_items;
```

```
DROP TABLE if exists tbl_customers;

CREATE TABLE tbl_customers
(
   num             smallint       NOT NULL   PRIMARY KEY,
   name            varchar(100)   NOT NULL,
   ship_to_addr1   varchar(100),
   ship_to_addr2   varchar(100),
   ship_to_city    varchar(35),
   ship_to_state   char(2),
   ship_to_zip     varchar(10),
   bill_to_addr1   varchar(100),
   bill_to_addr2   varchar(100),
   bill_to_city    varchar(35),
   bill_to_state   char(2),
   bill_to_zip     varchar(10),
   contact_first   varchar(50),
   contact_last    varchar(50),
   phone           varchar(12),
   title           varchar(50),
   comments        varchar(255)
);

/*  In the following LOAD DATA INFILE statement, the cust_mast.txt
 *  file has been exported from elsewhere and put into the location
 *  stated. It contains the customer data. It is the customer master
 *  file.
 */

LOAD DATA INFILE "/mnt/DOS_hda2/newriders/book/ch6/cust_mast.txt"
   INTO TABLE tbl_customers;

/* Again, check the success of the import operation. This time,
 * look for approximately 170 records. The best way to verify
 * that is to know the record count in the text file and to
 * compare it to the result returned by the next statement.
 */

select count(*) from tbl_customers;

/* Now alter tbl_customers to be auto-incremented on the first column.
 * See the Alter Table statement, which is next after this comment
 * block.

 * The following statement explains why you need to ALTER the
 * table that was just created and filled.

 * For the cust_mast.txt file data, the first data field is the
 * customer number, which is the primary key of the table. In the
 * MySQL database, this column should be an auto-incrementing field.
 * However, creating the table with the first field set to auto-
```

continues

Listing 6.1 **Continued**

```
 * increment may cause problems because the customer numbers are not
 * necessarily complete, nor do they start at 1. Therefore, in this
 * case, to avoid problems with the LOAD DATA INFILE statement,
 * you issue an ALTER TABLE statement to change the attributes
 * on the num field so it will auto-increment after the initial data load.

 * Also note that it is not necessary to restate PRIMARY KEY.
 * If you attempt to put it in the ALTER TABLE statement, MySQL
 * will issue a duplicate primary key error message.

 * One final comment: If you end one of these files with a comment,
 * it will produce an error when you attempt to feed it to MySQL.
 * Apparently, it expects a semicolon as the last character
 * in the file.
 */

ALTER TABLE tbl_customers
MODIFY num   SMALLINT
      NOT NULL
      AUTO_INCREMENT;
```

Deploying the Application

Because the application hasn't even been built yet, this might seem like we're jumping the gun. However, the deployment for this application is covered here because it is a rather simple exercise. Actually, for a small user base, this can be a very simple way to get the application out to the user. A word of caution, however: Don't try to use this manner of deployment for more than a handful of users; you'll likely find yourself making a lot of trips to the users' desks to install and upgrade your application. For a user base greater than 3–5, you will need something more scalable, such as an RPM, which is discussed in Chapter 8, "Commission Calculations Abstract and Design." Having said that, I will show you the simplest way to get the application in front of the user.

This is actually a very simple operation, primarily because of the small user base (for example, one). When the executable is built and working satisfactorily, the application can be dragged onto the desktop, the icon can be modified a bit, and it should be all set for the user. Remember, however, that in this case, the development and deployment machines are the same. This means the application was (obviously) run during development, and dependency libraries (such as GLib) are installed.

First, find the executable in the directory tree using the File Manager. Grab it with the mouse and drop it onto the desktop, as shown in Figure 6.5.

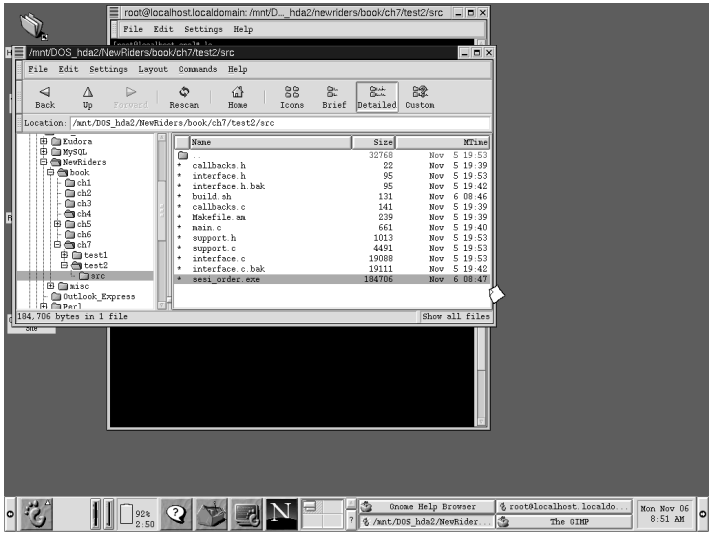

Figure 6.5 Dragging the application to the desktop.

In Figure 6.6, the executable is on the desktop. Notice the icon crossing the boundary from the file list to the desktop. That icon is for sesi_order.exe, which was the name of the application as stated in the compile command. You can double-click on the executable to launch the application. Also, note that drag and drop moves the actual file to the desktop, not just a copy of the file.

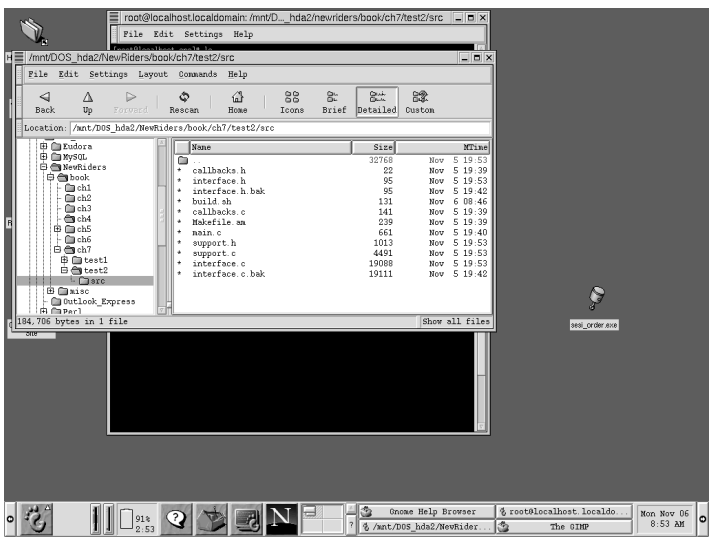

Figure 6.6 After the drag and drop, sesi_order.exe appears on the desktop.

Right-click the sesi_order.exe icon to display the pop-up menu (see Figure 6.7). Select the Properties item, and the Properties dialog box appears.

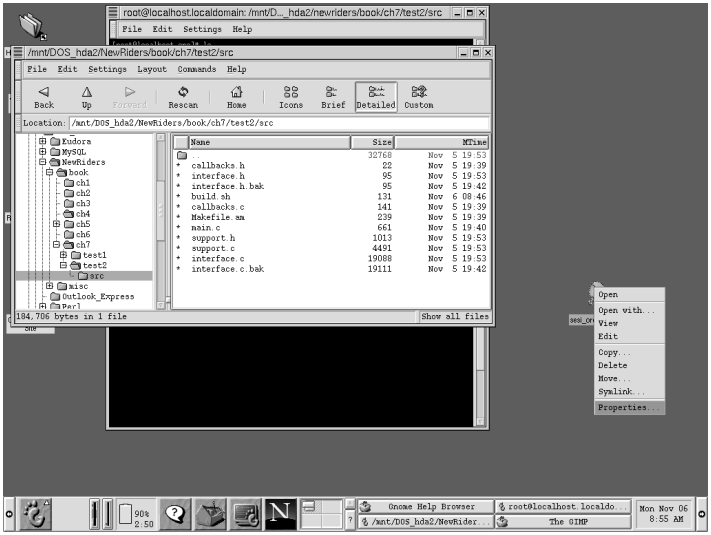

Figure 6.7 Customizing the icon.

In Figure 6.8, notice the entry widget that allows you to set the file name for the icon. In this case, change the name to New Order and leave off the .exe extension.

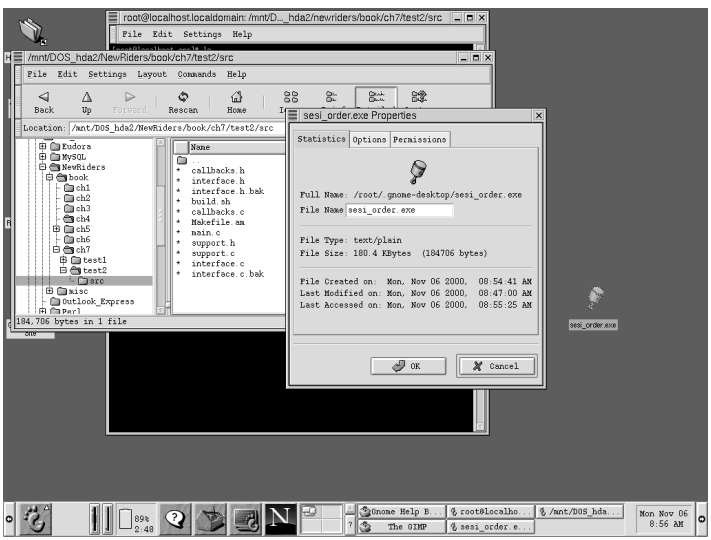

Figure 6.8 Change the file name from sesi_order.exe to New Order.

Next, select the Options tab and then click the icon button. As you can see, the system has selected a piston icon as the default for executables. Clicking the icon brings up the list of stock icons from which to choose. Scroll downward and find an icon for the application (see Figure 6.9).

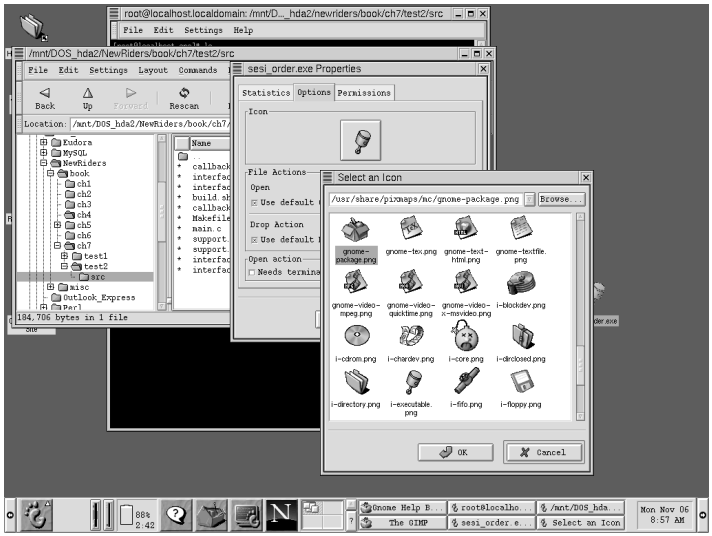

Figure 6.9 A list of stock Gnome icons.

Figure 6.10 shows the final result: an icon on the desktop for creating a new customer order.

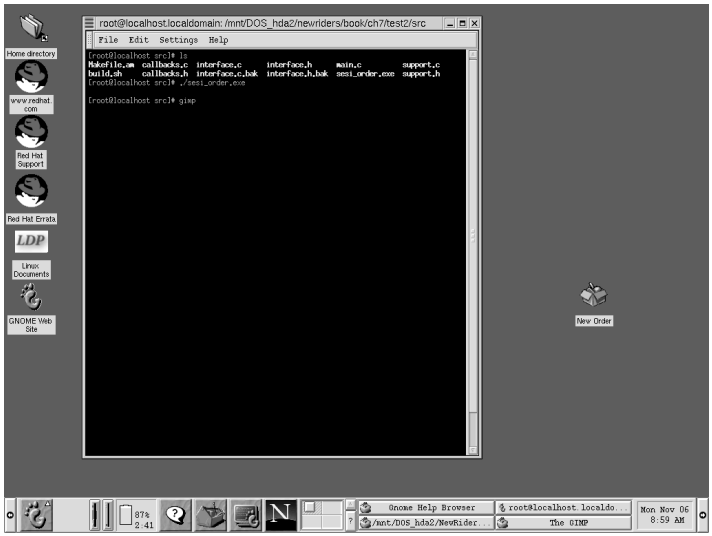

Figure 6.10 The final result: the application on the desktop.

Upgrading the Application

In this case, upgrading the application will be very simple because the client and server are the same machine and only one machine will be running the application. When the application needs to be updated, the developer works at the PC as time permits. Because the "actual" application is on the GNOME desktop (that is, the one the data entry clerk uses on a daily basis), the developer can work with the source code (refer to Chapter 7) and make new compiles down in the project directory without affecting the "production" executable on the user's desktop. When a new version is ready for daily use, the developer moves it to the desktop and deletes the previous version. This also gives some measure of protection in case the user somehow manages to delete the executable on the desktop.

7

Construction of the SESI Order Entry Application

This chapter covers the construction of the Specialty Electrical Supply, Inc. (SESI) Order Entry application as specified and designed in Chapter 6, "Order Entry Abstract and Design." This chapter will move slower than is absolutely necessary and will cover some basic things in detail for readers who are new to this set of tools (MySQL, GTK+, and Glade); for example, certain things will be done step by step here that would be covered in a few sentences or paragraphs in the later sections of the book.

This chapter will proceed along the following lines. First, Glade will be used to construct the user interface. The user interface (UI) will be compiled to make sure that it operates correctly. Next will come the "utility" functions of the application. For example, the function that fills a drop-down combo box with values. These types of functions will be independent of the user interface to the greatest extent possible so that they can be used in other places easily. Another example would be a function that saves the values in the application's text or entry boxes to the database. At the initial project specification, there may be only one place where this function would be used, so it might seem more appropriate to put it in an event callback. However, it may come about at a later release that the "save" functionality is needed in several places. Making the "save" functionality modular from the beginning makes code maintenance that much easier down the road. In the third section, we will connect the two sections to make a functional application.

This chapter will proceed along the path of a "discovery;" that is, I am not going to present the finished final product with all the kinks and trip-ups solved. Instead, I will construct this project as you would. This means I might create or omit something in the UI that will not be discovered until I attempt to integrate the utility functions in the final section. In that case, I have not "gone back" to the UI built with Glade to correct the problem. I will handle it "as discovered, where discovered" to illustrate problems, bugs, and omissions in a realistic manner and hopefully emphasize them in a way that will enable you to learn more about the chosen tools (GTK+ and MySQL).

Finally, remember that the database was constructed in Chapter 6, including the initial fill of data from text files; recall that these text files were extracts from the "previous system" (whatever that may be) and that they have been already imported into the tables in the SESI database (which this application accesses). Appendix A, "Glade-Generated Files from the SESI Order Application," goes along with this chapter and Chapter 6. Appendix A contains the interface.c and sesi.glade files, as generated by Glade.

User Interface Construction with Glade

This section covers the construction of the user interface using Glade. Figures 6.1, 6.2, and 6.3 will be referenced extensively. So you may want to refresh your memory of them or dog-ear their pages.

Starting the frm_main Project

Launch glade from the command line. The UI should be constructed so it can be modified later, knowing that glade will overwrite `interface.c`, append to `callbacks.c`, and not touch `main.c`. Therefore, try not to make changes to files that will be overwritten unless you are willing to document those changes elsewhere so they can be re-created when necessary.

First, set the project options. The name of this application should be set to sesi, making the glade file sesi.glade. Disable Gnome support (tab 1) and gettext support (tab 2). Deselecting these two options will make this project simpler. Gnome support (recall that there is a set of `*_gnome_*` functions built on top of GTK+) is not needed in this or any project in this book, and gettext support is used in the internationalization of strings—again, something that is not needed for these projects because they are targeted at a small number of "in-house" users.

From the palette, select a new window (the top right icon). Name it frm_main in the Properties window and set the title of the window to SESI Customer Order. Then select a vertical packing box and drop it into the newly created window. Set the number of rows to 2 and in the Properties window, set its name to vbox_main. Select a status bar widget from the palette and drop it into the bottom half of the window; change its name to statusbar because this will be the only status bar widget you will use. Then, select a table from the palette and drop it into the top half of vbox_main. Set the number of rows and columns both to 2. Set the table name to table_2_by_2 in the Properties window.

Figure 7.1 shows your progress so far.

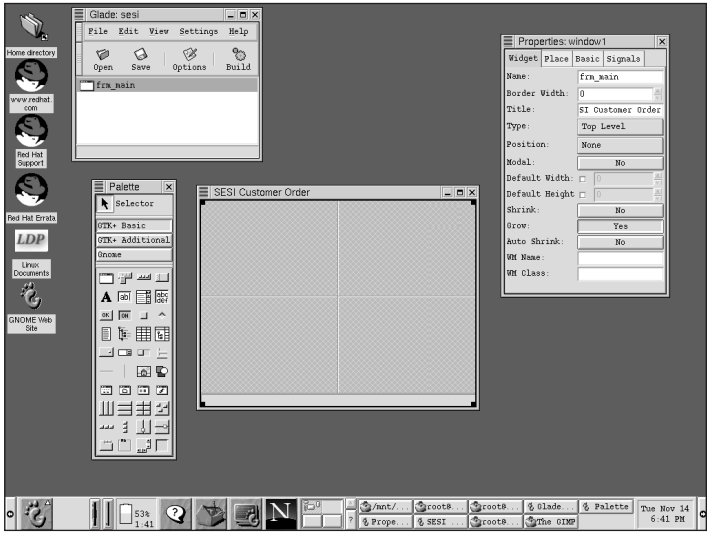

Figure 7.1 The starting structure of frm_main.

Filling Out the Left Side of frm_main

Select the vertical packing box from the palette and drop it into the top left section of table_2_by_2. Set the number of rows to 2. Now repeat this action for the top right section of table_2_by_2. For the names of these vertical packing boxes, enter vbox_customer_number and vbox_customer_name, respectively.

Select a vertical button box widget from the palette and drop it into the bottom left section of table_2_by_2. When prompted, set the number of rows to 5 in accordance with Figure 6.1. Name the vertical button box vbuttonbox since these will be the only command buttons on the form.

Now, within the vertical button box you just created, select the top button. Change its name from button2 to cmd_search and set its label to _Search.... Note the use of the underscore character (the _ in the label); this causes the S in Search to be underlined. Windows users will recognize this as the Alt+S combination, and indeed, there is no reason why your application can't conform to this convention.

With cmd_search still selected, click the Basic tab of the Properties window. Press the Accelerators button (the one marked "Edit...") at the bottom of the window. In modifiers, check the Alt box, type S in the Key text box, and click the … button to the right of the Signal text box. Select the "clicked" signal and click OK. You should now have one Accelerator in CList box at the top of the Accelerators dialog box. Now the key combination of Alt+S should activate the cmd_search clicked event.

Repeat the procedure in the previous paragraph for the rest of the buttons. Change button2's name to cmd_save_edits and its label to read "Save _Edits", indicating that Alt+E should activate the clicked signal. button3 should similarly have its name changed to cmd_select_items with Alt+I as the accelerator, button4 changes to cmd_print_order (Alt+P), and button5 becomes cmd_exit (Alt+X).

Next, select a label widget and drop it into the upper-left section of the form, which is also the top of vbox_customer_number. Set the text of the label to Customer Number:. Set the label name to lbl_customer_number and the justification to Left. To finish out the left half of this form, select a combo box and drop it into the open section just below lbl_customer_number. Change the name to cbo_customer_number. At this point, you also need to select the combo-entry widget within cbo_customer_number; you can see the difference by clicking on the combo-entry part of the widget and the drop-down arrow on the right side of the widget. In addition to renaming the entire widget, you also need to change the name of the combo-entry part of the widget to combo-entry_customer_number (refer to Figure 7.6).

Figure 7.2 shows how the form should look at this point.

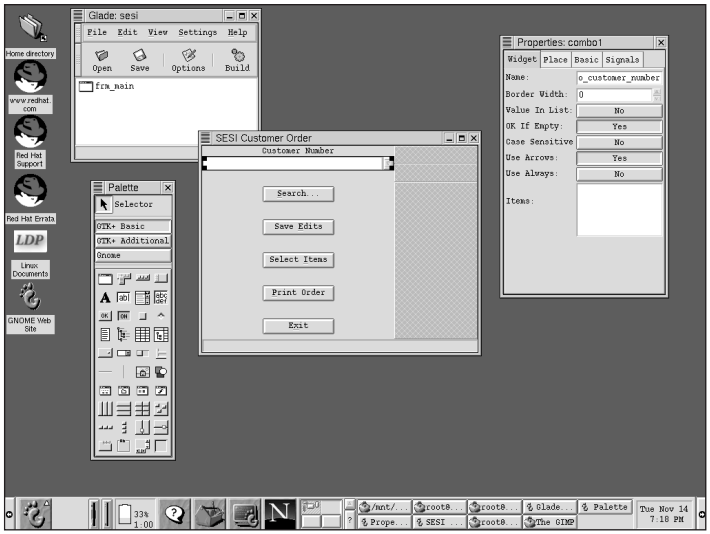

Figure 7.2 frm_main with the left half of the window filled out.

Filling out the Right Side of frm_main

Select another label widget and drop it into the open slot at the top on the right side. Change the name to lbl_customer_name and the label text to Customer Name; make it left justified. Now select a Text Entry widget and drop it into the open space immediately below lbl_customer_name. Name it entry_customer_name.

Select the vertical packing box again and drop it into the lower-right section of table_2_by_2, which at this point should be the only open section. Set the number of rows to 6 and the name to vbox_data. Select the table widget and drop it into the top slot of vbox_data. Set it to 3 rows and 3 columns if that is not the default. Then name it table_3_by_3. Select the table widget one more time, and this time drop it into the second slot of vbox_data, right below table_3_by_3. Set the rows to 4 and the columns to 4, and then name it table_4_by_2.

Select the label widget—twice—and put one each into vbox_data for lbl_order_comments and lbl_customer_comments. Here is your first change from your design of Figures 6.1 and 6.2. In those, Customer Comments were first, and Order Comments were underneath. As it turns out, the Customer Comments probably won't be changed every order, but the Order Comments could be different every time the customer places an order. So reverse the order of the Order and Customer comments labels and text widgets, making them the opposite of their order in Figure 6.2. This will allow the user to tab into the Order Comments—which is much more likely than the user changing or adding Customer Comments. When the label widgets are in place, enter the text widgets and change their names to txt_order_comments and txt_customer_comments. Be sure to set the editable property to Yes.

Figure 7.3 shows your application shaping up nicely.

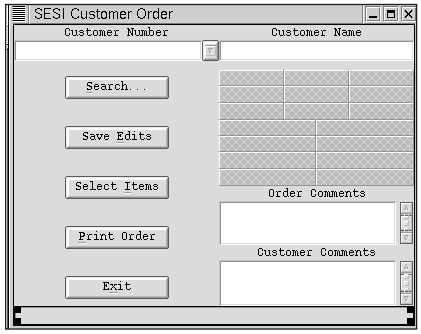

Figure 7.3 The construction of frm_main after you've filled in the right half of the form.

Finishing the frm_main User Interface

Now you'll move on to fill out table_3_by_3 and table_4_by_2. Select the label widget from the palette and drop it into the upper-left box of table_3_by_3. Change its name to lbl_primary_contact and its text to Primary Contact:. Set the justification to Left and under the Place tab of the Properties dialog box, set Col Span to 3. Place lbl_name_last_first in the second row in the leftmost box of table_3_by_3, and place lbl_title_and_phone in the bottom left box.

Next, you insert the text boxes for the contact's first and last name. Select the Text Entry widget from the palette and drop it into the center box of table_3_by_3; name it entry_last. On the Place tab of the Properties window, set the X Expand button to NO, and then go to the Basic tab and select the Width check box (to "true"). Finally, set the value in the Width text box to an even 100. This will make the text box a little narrower; only in the rare case that the person has a very long last name will part of the name be hidden from view. Again, select the Text Entry and repeat the process for entry_first, placing it in table_3_by_3 in the middle row, right box. To finish out table_3_by_3, repeat (again) the two previous steps for entry_title and entry_phone, putting them on the bottom row of table_3_by_3 in the center and far left spaces, respectively.

Fill out labels lbl_ship_to and lbl_bill_to (top row of table_4_by_2) and set their properties as you did for the previous label widgets. Into the middle two rows of table_4_by_2, drop four Text Entry widgets: entry_ship_to_addr1, entry_ship_to_addr2, entry_bill_to_addr1, and entry_bill_to_addr2 (see Figures 6.1 and 6.2 for clarification). Their default values of 158 are acceptable, and their X Expand buttons should be set to No.

The final part of this form's construction is nearly upon you—well, the visual part of it, anyway. Drop a horizontal packing box into each of the remaining bottom two spaces in table_4_by_2, where the City-State-ZIP fields will go for the Ship To and Bill To addresses. The widgets you are creating are entry_ship_to_city, entry_ship_to_st, and entry_ship_to_zip, followed by their equivalent *_bill_to_* widgets. Set the *_city widgets to a width of 100, 35, and 70 for city, state, and zip (for both sets of widgets).

At this point, the visible portion of frm_main is finished; you will add the events shortly. Be sure to hit the Save and Build buttons in Glade. Figure 7.4 shows frm_main as it appears from Glade. You can change the shape of the window (of frm_main) to get some idea of how it will be resized; however, you aren't guaranteed anything until you compile and resize the executable (see the following sidebar, "Compiling frm_main").

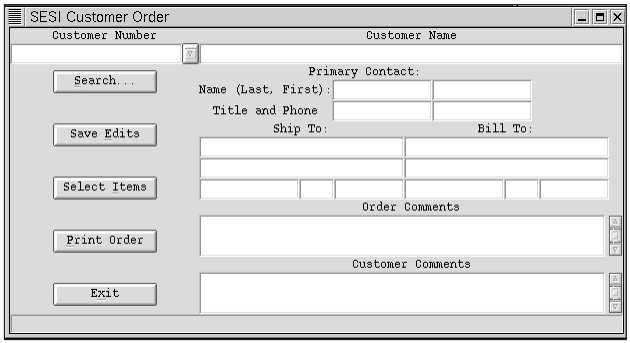

Figure 7.4 Completed frm_main.

Compiling frm_main

At this point, it might be a good idea to compile the application—which consists of only frm_main so far—just to see how it behaves. (For example, you want to see if a text box is resizing as desired.) To do that, you will have to do the following: Comment out the two lines in `main.c` that will cause problems (the lines that reference pixmaps, normally around line 23; these lines reference pixmaps, which won't be used in this application), `cd` to your project directory, `cd` to its src subdirectory, and then send the following from the command line (or put it into a shell script):

```
% gcc -Wall -g *.c `gtk-config --cflags --libs`
% ./a.out
```

After you do this, you will want to delete `main.c` and rebuild it from within Glade. If you don't delete it, Glade won't rebuild it, and because this is an intermediate build, you aren't to the point where you want to keep the changes to `main.c`.

Setting the Events for frm_main

As you might have noticed, you have not set any of the signal/callbacks or what you might be used to calling "events;" nor have you started frm_items_ordered (see Figure 6.3). Next, you will set the signals for frm_main, and later you will repeat all these steps for frm_items_ordered.

At this point—setting the events for frm_main—you are probably better off to err on the side of caution; that is, you should connect more signals than necessary. Because you are going to build most of your utility functions in a modular way, if it turns out that one signal is better than another for a certain function (by having that function already set), you will have to move only a small amount of code. You will not have to open Glade and search for signals.

Setting the Signals

To start with, connect the delete signal for frm_main; this will be your application shutdown code. Select the main window object of frm_main. You can do this by selecting frm_main in the window titled "Glade: sesi." You will know when you have selected the window object because the Properties window will say "Properties: frm_main" in the title bar.

With frm_main selected, click the Signals tab in the Properties window. Click the … button next to the Signal text box to bring up the Select Signal dialog box. Now the question becomes this: "Which signals will you use, or potentially use?" Remember, for an application this small, there is essentially no penalty for overindulging and selecting too many signals. Figure 7.5 shows the Select Signal window.

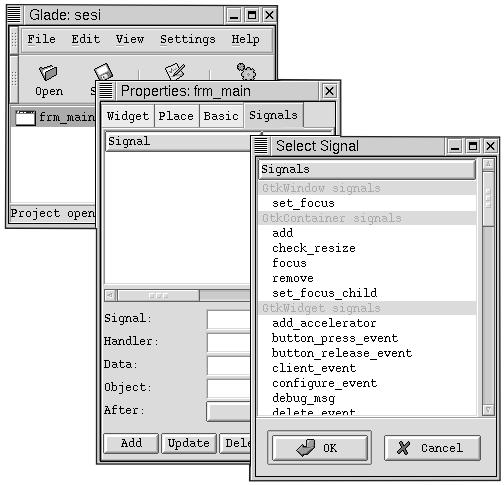

Figure 7.5 The Select Signal dialog box.

Under GtkWindow signals, highlight set_focus and click OK. This returns values to the Signal and Handler text boxes of the Signals tab in the Properties window. The next step is very important: Click the Add button in the lower-left corner or your signal addition will be lost. This should put a `set_focus` signal into the list box at the top of the Signals tab. Repeat that procedure for all signals you think you'll use.

In this case, the following signals have been selected in addition to the `set_focus` event: `button_press_event`, `button_release_event`, `delete_event`, `event`, `hide`, `key_press_event`, `key_release_event`, and the `realize` event. The `delete` event will be the place to call the `gtk_main_quit()` call, and the others just might come in handy at some point in the future. As I said before, if it turns out not to be the case, the price is an empty function.

Testing Event Sequences by Adding `g_print` Statements

At this point, you might want to put a quick `g_print()` function call in all your callbacks, do a quick compile, and check to see that all the functions are firing when you think they are. To do this, you'll need to open `callbacks.c` for your project and add a `g_print()` call, changing the message to indicate which callback function is firing. Follow the compile instructions in the previous sidebar, "Compiling frm_main." If you have downloaded the source code from the companion Web site, you will notice that every callback has either a `g_print()` call or a commented-out `g_print()` call. I have left those in on purpose; in addition to helping with debugging, they make a good demonstration of event sequences.

In fact, doing so confirms a couple of events in GTK+ that you will use to your advantage. When you get to the point that you have the clicked event for `cmd_search` with a statement in it like this

```
g_print("cmd_search clicked...\n");
```

you can confirm that Alt+S does indeed activate the `cmd_search` clicked event and that the keypress event for the window captures all keys. However, this now creates new problems: All keys means *all keys*, including the Tab key, which the user will probably use extensively.

Now select the `cmd_search` button and go to the Signals tab. Again select the … button to view the possible signals for a command button. Select the `clicked` signal under GtkButton Signals, click OK, and then click Add.

Repeat that same action for all the other command buttons on the form. You can look through the list of events for the command button widget, but you won't be using any events other than the `clicked` event. Also, go ahead and add `gtk_main_quit()` to frm_main's delete event callback in `callbacks.c`.

Finally, turn to cbo_customer_number. Recall that when this widget was created, you had to rename both the combo widget and its child combo-entry widget to cbo_customer_number and combo-entry_customer_number, respectively. This distinction will become very important shortly. Figure 7.6 shows the different widgets selected.

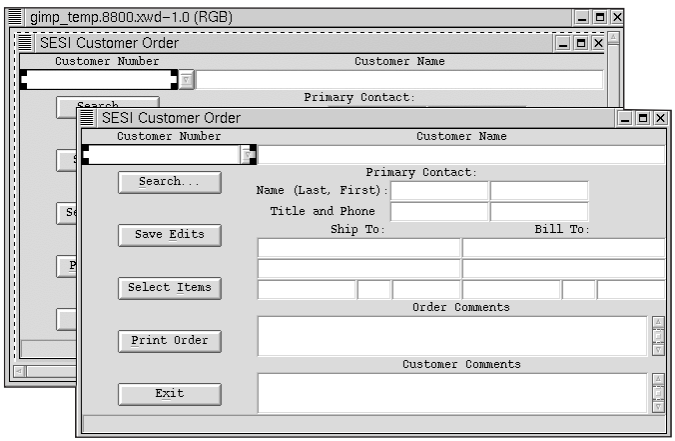

Figure 7.6 The difference between a combo widget and its child combo-entry widget.

With the combo widget selected, select the Properties window and then the Signals tab. Using the … button, bring up the signals for the combo widget. Notice that you have the signals for GtkContainer, GtkWidget, and GtkObject. Looking through the list, none will be used in our application. However, if you go back and select combo-entry_customer_number and then bring up the signals, you will see several that will be needed.

Notice that the Select Signal window brings up the GtkEditable signals. Set the signals for `activate`, `changed`, `delete_text`, and `insert_text`.

Making cmd_save_edits Inactive

Finally, you need to make cmd_save_edits inactive—or in GTK-speak, "insensitive." Select the cmd_save_edits widget, and then open the Property Editor window if it is not already open. Select the Basic tab and set the Sensitive toggle button to "No." This sets the initial state of cmd_save_edits to be grayed-out.

At this point, the UI portion of frm_main using Glade is done. Again, it is probably advisable for you to open `callbacks.c`, enter a `g_print()` statement for each of the callbacks, and then compile and run the application just to see how everything fits together. Otherwise, it is time to move on to frm_items_ordered.

Creating frm_items_ordered

Chapter 6 did not present a container schematic for frm_items_ordered as it did for frm_main. It will be initially divided by a horizontal frame with the buttons, labels, and text widgets on the left, and the CList widgets on the right. The left side will be divided by vertical packing boxes into which the individual widgets will be added, and the right side will be divided by two more frame widgets.

From the Palette, select a window widget. Then, rather than selecting a horizontal packing widget with two columns, select and drop a horizontal pane widget into the window. This effectively allows the same division as a horizontal packing widget, but it allows the user to size the left and right sides. Be sure to set both the Gutter and Handle properties to 10 and deselect the Position check box; this allows the frame to size as needed instead of setting itself all the way to the left as a default.

Next, select a vertical packing box and drop it into the left side of the window; it asks for the number of rows to create. Referencing Figure 6.3, there is a chance to change the design a bit and simplify the UI at the same time. In Figure 6.3, the top leftmost widgets are in the same row of the vertical packing box, the label Item Num:, and its associated editable widget. What if you put each of these widgets into separate rows in your vertical packing box? There would be no need for a horizontal packing box, and it would not affect the vertical dimensions because the CList widgets on the right side would be longer (vertically) than all the widgets on the left side. While it doesn't give any immediate benefits other than simplifying the UI build process a bit, it doesn't appear to cost anything either. So it will be changed. Looking at Figure 6.3 again, you see a total of nine widgets on the left side. All of those will be placed into their own row within the left vertical packing box, which you will call vbox_left.

Select and drop into vbox_left the following items from top to bottom: a label widget, a text-entry widget, another label, a spin button widget, two command buttons, another label, a frame (and then inside the frame, insert label widget). Finally insert into the bottom row a command button. Rename the top label from label1 (or whatever the default name is) to lbl_item_number and set its text to Item Number:; set the justification and other properties as desired. Name the text-entry widget below it entry_item_number and set its max length to 12, which should be more than enough.

Next change the label in Row 3 to lbl_quantity, change the text to Quantity:, and change any other properties as needed. Name the spinbutton widget in row 4 spinbutton_quantity, and set the default Value: to 1 if it is not already set because this will be the most likely value for any given item.

For the next two command buttons down in the vertical packing box, change the names to cmd_add and cmd_search_for_item and set the label text according to Figure 6.3. For cmd_add, put an underscore before the "A" in the label to make it the hotkey and connect the Alt+A key combination to the `clicked` signal by going to the Basic tab in the Properties box and clicking the Accelerators: Edit... button. Check the Alt check box, set the Key to A and the Signal to clicked, and then click Add and Close. Repeat the steps for cmd_search_for_item, making F the hotkey. For both command buttons, set the Border Width property to 5; this improves readability of the UI a bit.

Finally, to fill out vbox_left, set lbl_order_total (the 7th row down) the same way you set the labels in rows 1 and 3. Row 8 now contains a frame widget (without a title), and inside the frame is a label widget. This label widget will be set with the amount of the order total, and because that number cannot be changed except by adding or deleting items, making it a label prevents the user from thinking he can edit it. Set this label to lbl_order_total_numeric and set its default text to 0.00. Also, select the frame surrounding it (probably named frame1, by default) and set its Expand and Fill properties to No.

In row 9, set the properties for cmd_done in the same way you set the command buttons in rows 5 and 6.

Figure 7.7 shows how frm_items_ordered should look after you finish the left side of the horizontal packing box.

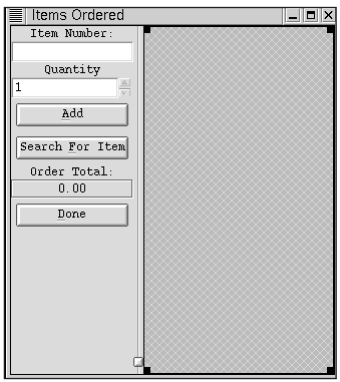

Figure 7.7 frm_items_ordered as it should look after you fill in the left half of the window.

Filling Out the Right Side of frm_items_ordered

Now it is time to fill out the right side of the main horizontal packing box. Again, instead of selecting a vertical packing box widget and dropping it onto the right side with the rows set to 3, you will use vertical pane widgets. As before, this gives the user

room for greater customization, and in this case, it helps demonstrate the differences between using packing boxes and panes. Select the vertical pane widget (twice) and place it into the right side of the horizontal pane widget that divides frm_items_ordered. In the top row, select from the palette and drop a CList widget.

For the name of this CList widget, enter clist_items, and then select each of the column header labels in turn. Name them and set their text as shown in Figure 6.3. To avoid confusion, preface the column header label names with "clist_items_"; for example, the leftmost column label should be named clist_items_lbl_item_number. This prevents the possibility of confusing it with lbl_item_number, which is in the upper-left space of this same form. Although such a name is very long, because it's a widget, it will not change after it is created; it will be static for the life of the application, so you only have to set it once. Therefore, the three labels reading across the top of clist_items should be named clist_items_lbl_item_number, clist_items_lbl_description, and clist_items_lbl_price. Returning to clist_items, in the Property Editor under the Basic tab, select the Height and Width check boxes, which forces the CList to be displayed in the correct size. Otherwise the frame object that bounds it on the lower edge will push all the way to the top, effectively hiding clist_items.

Next, place another CList widget into the bottom space on the left side that's created by your two vertical frame widgets. Name it clist_items_ordered, in keeping with your overall naming convention. Change its properties and its column's properties the same way you did for clist_items, but also as depicted in Figure 6.3. That means the first label will be named clist_items_ordered_lbl_item_number, the second clist_items_ordered_lbl_description, and so on. Don't forget to select and set the Height and Width properties on the Basic tab.

Finally, resize both of the CList widgets to show proportionally: clist_items should show five or six lines of available items, whereas clist_items_ordered should show three or four items. Set the Height property of clist_items to 150 and the height of clist_items_selected to an even 100.

The only available space should now be the empty space between the two CList widgets. Into it, drop a horizontal packing box with three columns. I will refer to this as hbox3, the default name on my system; to follow along, you should set yours to the same. For hbox3, set the Border Width to 10, the Size to 3, Homogenous to Yes, and Spacing to 12. On the Place tab, set Shrink to Yes and Resize to No. The rest of the defaults should do fine.

Drop a command button object into the left available spot in hbox3. Set its name to cmd_add_down, and this time for its text, enter Add. At this point, you might be tempted to choose the stock button Down. That option shouldn't be used here because you have disabled Gnome support for this application, and the stock buttons are Gnome functions.

Next, drop another command button into the far right space in hbox3. Name this one cmd_remove, and because none of the stock buttons are what you're after, enter Remove for its label text. Both cmd_remove and cmd_add_down should have their Expand and Fill properties set to No.

Lastly, select a spin button widget and drop it into the center space of hbox3. Name it spinbutton_quantity_down to distinguish it from spinbutton_quantity in the upper-left corner of the form. In the Place tab of spinbutton_quantity_down, the Expand and Fill properties should be set to No.

Figure 7.8 shows frm_items_ordered with the UI filled in.

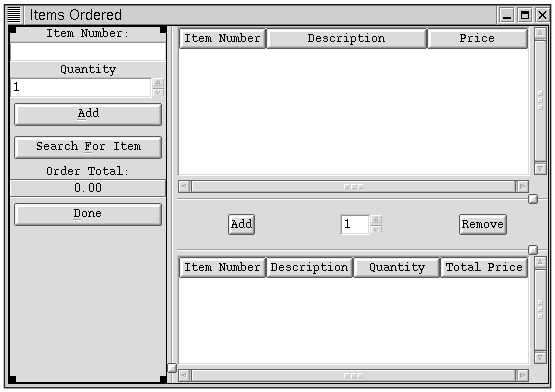

Figure 7.8 frm_itmes_ordered with all widgets added.

Setting the Events for frm_items_ordered

It is time to set the events. Set the same events as with frm_main for frm_items_ordered (at the window level), and set the same for the button widgets. See the Signals tab of frm_main for a list of signals to set. After that, you need to set the signals for the GtkEditable widgets (any text, entry, and in this case, spinbutton controls), as well as the CList widgets.

As you did with the GtkEditable objects on frm_main, set the `activate`, `changed`, `insert text`, and `delete text` signals for entry_item_number on frm_items_ordered. Although you could set the same signals for the spinbutton widgets, that won't be done because no actions are tied to the spinbutton widgets. They will be accessed to find out what integer they show, but nothing more will be done.

For the CList widgets, set the `select_row`, `unselect_row`, `button_press_event`, and `button_release_event` signals. Do so for both CList widgets.

Nearing the end of the UI building process with Glade, the last thing you need to do is build the "find customer" and "find item" search boxes. Figure 7.9 shows both windows built with Glade.

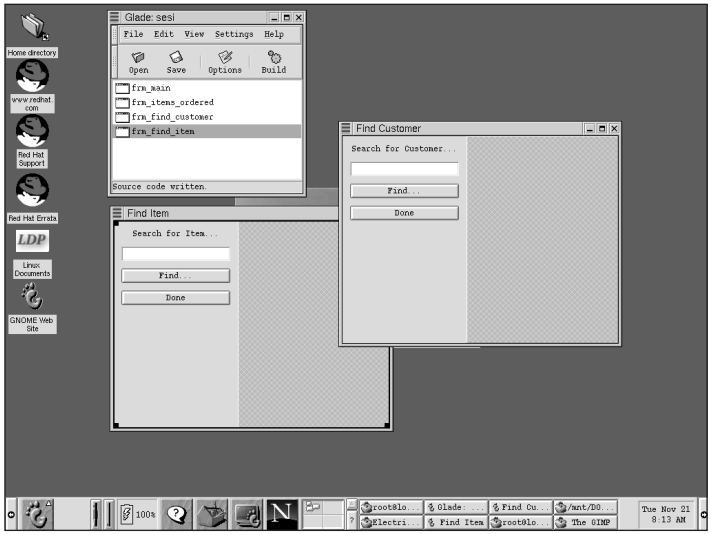

Figure 7.9 frm_find_item and frm_find_customer: Note that each still needs a CList widget on the right half to be complete.

The detailed steps for creating these windows will not be covered here; all the widgets used are defined, and their properties are set along the same lines as were those of the previous widgets. Both are Modal, and their only trapped events are the clicked events of the command buttons and the delete event on the window. frm_find_item should have a CList widget on the right half with three columns, and frm_find_customer should have a CList widget on the right with 17 columns, each corresponding to its related database table. The application will fill in the rows, and if another Find operation occurs, the application will clear and refill the widget.

Utility Functions of the Application

In this section, you will construct the functions that will do the majority of the work for your application. You will construct these functions to make them as independent of any user interface as possible. This not only makes them easier to test during bug-tracking activities ("What is causing the bug: MySQL, the text box, or something else?"), but it forces a clarity of thought on the functions by forcing you to ask "What is this function really trying to do, what minimal inputs are needed, and what minimal outputs are required?" For example, when you're selecting a value (a customer number in this case), the input is the customer number, and for the output you would prefer to have a "record" rather than a number of "fields." So if you can build a function that will query your database and return a single database record, you have a better function than one that returns multiple values.

Creating *sesi_utils.c*

First, create a file called `sesi_utils.c`. This will be the file that contains the utility functions for the application. Refer to Listing 1.1 in Chapter 1, "MySQL for Access and SQL Server Developers and DBAs," if necessary. This section is going to present the file `sesi_utils.c` as a series of listings instead of one large listing so that the individual functions can be covered with a little more detail. It is important that all these utility functions reside in a single file: `sesi_utils.c` (and its companion, `sesi_utils.h`). Also remember that you can get all the files from this book at the book's companion Web site.

First, include the necessary files. Referring to Listing 1.1 (in Chapter 1), `stdio.h` and `mysql.h` were included, but only `mysql.h` needs to be included here. `stdio.h` was for the `printf` functions needed in Listing 1.1, but you will be using GLib functions in your application. `gtk.h` also needs to be declared here so that you have access to its functionality. Therefore, your heading should look like the following:

```
#include <mysql.h>
#include <gtk/gtk.h>
```

connect_to_db() function

The first thing the application must do is connect to the correct MySQL database. Because this application is meant to be opened, used, and closed in short cycles of a few minutes, it will be safe to make this connection to the database a global variable. This is one of those cases in which a global variable makes sense. Otherwise, the developer is faced with one of two choices: to create a non-global variable that gets passed into or out of nearly every function or to initiate a connection to the database for every query and then end that connection after every single operation is completed.

Listing 7.1 is the database connection routine. It has one purpose: to establish a connection to the MySQL database named "sesi."

Listing 7.1 is only one of the functions in the physical file `sesi_utils.c`. It will be much easier to cover each function on an individual basis than to present one long listing and try to cover it as a single topic. At the book's companion Web site, you can get all the code listings.

Listing 7.1 ***connect_to_db()* Function of *sesi_utils.c***

```
void connect_to_db()
{
   /* Listing 7.1
    *
    * This function establishes a connection to database
    * "sesi." It establishes a value for the variable
    * conx, which is a (global) session connection
    * to the database.
    *
    * 0L is equal to NULL, but it is considered more portable.
```

continues

Listing 7.1 **Continued**

```
 * For a Linux-only program, NULL should work fine.
 */

g_print("Establishing db connection...\n");

conx = mysql_init ((MYSQL *)0L);

if (conx == 0L)
   {
     /* What should the software do if this error occurs?
      * Obviously, there is no reason to continue with
      * the application because it is of no use without a
      * MySQL database connection. In a larger application,
      * there would be some logging of the error that occurred
      * and various other pieces of information that would aid
      * in debugging. But for this application, those actions
      * weren't specified. So they will be left out.
      *
      * Therefore, when a fatal error occurs, the software will
      * communicate that to the user, and it will exit.
      */

     g_print("Failure in mysql_init...\n");
     gtk_statusbar_push(GTK_STATUSBAR(lookup_widget(frm_main, "statusbar")),
                  1,
                  "Database initialization failed,"
                  "contact administrator...");

     /* The fatal_msgbox() function is defined next.
      * Basically, it is used whenever an error occurs that
      * makes it useless to continue.
      */

     fatal_msgbox("Database initialization failed.\n"
                  "Contact the system administrator.");
   }

/* In the following call, if the database parameter is misspelled
 * (in this case, "sesi"), it is important to note that the
 * function call will still return OK instead of an error. The reason
 * is that this function primarily connects to a server, not a
 * specific database. If you can't connect to the database
 * for some reason, you won't know until you attempt to query
 * the database for some information.
 */

conx = mysql_real_connect (conx, "localhost", "root", 0L, "sesi", 0, 0L, 0);
if (conx == 0L)
   {
   g_print("Failure in mysql_real_connect...\n");
```

```
     /* The next two function calls are the first examples
      * of error handling in this application. In MySQL, these
      * are the error handling functions. Obviously, these
      * could be very helpful in debugging and error tracing.
      * So the software should attempt to report them.
      */

     g_print("Error Number is %i...\n", mysql_errno(conx));
     g_print("Error Description is %s...\n", mysql_error(conx));
     gtk_statusbar_push(GTK_STATUSBAR(lookup_widget(frm_main, "statusbar")),
                  1,
                  "Database connection failed, "
                  "contact administrator...");

     /* Again, if the mysql_real_connect call returns an error,
      * there is really no point in continuing.
      */

     fatal_msgbox(g_strconcat("The connection to the database could",
                              " not be established.\n\n",
                              "Error number: ",
                              g_strdup_printf("%d", mysql_errno(conx)),
                              "\nError description: \n",
                              g_strdup_printf("%s", mysql_error(conx)),
                              0L)
                 );
   }

g_print("Connected to db...\n");

/* Hit the target database with a simple query as a final
 * confirmation that the connection to the database is open.
 */

if (mysql_query (conx, "select count(*) from tbl_customers") != 0)
   {
            fatal_msgbox(g_strconcat("Unable to connect to ",
                                     "database SESI.\n\n",
                                     "Error Number:  ",
                                     g_strdup_printf("%d",
                                              mysql_errno(conx)),
                                     "\nError Description:  ",
                                     g_strdup_printf("%s",
                                              mysql_error(conx)),
                                     0L)
            );

          g_print("Failure to connect to the correct database...\n");
   }
else g_print("Connected to database \"sesi\"...\n");
```

continues

Listing 7.1 **Continued**

```
    /* Report to the user that everything to this point is okay.  */

    gtk_statusbar_push(GTK_STATUSBAR(lookup_widget(frm_main, "statusbar")), 1,
                       "Connected to database SESI, ready for new order...");

}
```

fatal_msgbox() Function

In Listing 7.1, notice that in three places the `fatal_msgbox()` function is called. That function is shown in Listing 7.2.

Listing 7.2 **The *fatal_msgbox()* Function of *sesi_utils.c***

```
void fatal_msgbox(gchar *msg)
{
   /* Listing 7.2
    *
    * This function is called whenever a fatal but trapable application
    * error has occurred. Message boxes are a very poor way to
    * communciate to the user; however, when an error occurs that should
    * cause the application to terminate, it is nice to give the user
    * some form of feedback that attempts to tell him why and gives
    * him a chance to do something. In this case, basically all the user
    * can do is make a note of the problem and report it to the system
    * administrator.
    * This function uses the gtk_dialog_new() call to create a container
    * message box into which a label and an OK button are added.
    */

   GtkWidget *msgbox;
   GtkWidget *lbl;
   GtkWidget *cmd_ok;

   msgbox = gtk_dialog_new();
   lbl = gtk_label_new(g_strconcat("A fatal error has occurred.\n\n",
                                   msg,"\n\n",
                                   "You may want to make a ",
                                   "note of this information.\n\n",
                                   "The OK button below will terminate ",
                                   "this application",
                                   0L) );

   cmd_ok = gtk_button_new_with_label("Ok");

   gtk_box_pack_start(GTK_BOX(GTK_DIALOG(msgbox)->vbox),
                              lbl, FALSE, FALSE, FALSE);
   gtk_box_pack_start(GTK_BOX(GTK_DIALOG(msgbox)->action_area),
                              cmd_ok, FALSE, FALSE, FALSE);
```

```
    /* Note the following two signal connection calls. The second one
     * should be very familiar to you by now, but notice that rather than
     * setting up a separate callback function in which gtk_main_quit()
     * would be called (as has been done in all examples
     * to this point), gtk_main_quit() is called directly.
     *
     * The first, gtk_signal_connect_object(), is used when the function
     * you are invoking has a single parameter of type GtkObject. Notice
     * that both produce the same effect. The first is needed because
     * there is no guarantee that the user will click the handy "OK"
     * button that has been created; he might hit the kill-window "X"
     * that is in the top right corner of all windows. Without a callback
     * on the delete_event signal, he would return to the application
     * but be unable to do anything.
     */
    gtk_signal_connect_object(GTK_OBJECT(msgbox),
                              "delete_event",
                              GTK_SIGNAL_FUNC(gtk_main_quit),
                              GTK_OBJECT(msgbox) );

    gtk_signal_connect(GTK_OBJECT(cmd_ok),
                       "clicked",
                       GTK_SIGNAL_FUNC(gtk_main_quit),
                       0L);

    gtk_widget_show_all(msgbox);

}
```

connect_to_db() and *fatal_msgbox()* Functions

After you have created the two functions in Listing 7.1 and 7.2, you see that the header area of file `sesi_utils.c` has changed, as shown in Listing 7.3.

Listing 7.3 **The Header of File *sesi_utils.c* After the Functions *connect_to_db()* and *fatal_msgbox()* Are Created**

```
01  #include <mysql.h>
02  #include <gtk/gtk.h>
03
04  #include "support.h"
05
06  GtkWidget *frm_main;
07
08  /* conx is the connection to the database; it is global,
09   * and all functions in the application can use it to
10   * access database "sesi".
11   */
```

continues

Listing 7.3 **Continued**

```
12
13  MYSQL *conx;
14
15  /********** Function Prototypes **************/
16  void connect_to_db();
17  void fatal_msgbox(gchar *msg);
18
19  /********** Utility Functions ****************/
20
```

Lines 4 and 6 are required because the function `connect_to_db()` references the `lookup_widget()` to place text into the status bar widget on frm_main. Line 13 is the global database connection variable. Lines 15 and 19 were added for readability, and line 17 is the function prototype for `fatal_msgbox()`.

Next you'll look at the function that will fill combo box cbo_customer_number with the values for the drop-down box. This is one of the first things the application will have to do.

get_customer_numbers() Function

Listing 7.4 shows the `get_customer_numbers()` function. Notice that it returns a GList object; this is because it is easy to call the `gtk_combo_set_popdown_strings()` function, sending it a GList object as the second parameter.

Listing 7.4 **The *get_conx()* and *get_customer_numbers()* Functions of *sesi_utils.c***

```
void get_conx()
{
   /* This function refreshes the connection to the database as
    * needed. Note its lack of error handling; since that
    * was checked in detail when the application started, this
    * makes the assumption that things will still be OK – normally.
    */

   conx = mysql_init ((MYSQL*)0L);

   /* In the following function call, "root" is the user who was logged
    * in when MySQL was installed via the RPM. You might need to change
    * the user name for your system, depending on how it was installed.
    */

   mysql_real_connect (conx, "localhost", "root", 0L, "sesi", 0, 0L, 0);
}

GList *get_customer_numbers()
{
```

```
    /* This function retrieves the list of customer numbers from
     * the database and fills cbo_customer_numbers on frm_main.
     */

    GList       *list_of_customers = 0L;
    MYSQL_RES   *result_set;
    MYSQL_ROW   row;

    get_conx();

    /* When frm_main opens, it will default to the first item in the
     * drop-down list of the combo box.
     */

    if (mysql_query (conx, "select distinct(num) from tbl_customers") != 0)
       {
                 fatal_msgbox(g_strconcat("Unable to retrieve list",
                                          " of customer numbers.\n\n",
                                          "Error Number:  ",
                                          g_strdup_printf("%d",
                                                  mysql_errno(conx)),
                                          "\nError Description:  ",
                                          g_strdup_printf("%s",
                                                  mysql_error(conx)),
                                          0L)
                 );

               g_print("Failure to retreive list of customers..\n");
       }
    else g_print("Retrieved customer numbers from db...\n");

    /* Now iterate through the rows, get the values for customer
     * number, and add them to cbo_customer_number.
     */

    result_set = mysql_store_result (conx);
    while ((row = mysql_fetch_row (result_set)) != 0L)
       { list_of_customers = g_list_append(list_of_customers, row[0] );
       }

    /* If you put the New customer number at the end of the list, the
     * user can use the up and down arrows to search through the customer
     * records without crossing the "New" value. Adding a new customer
     * will be a rather rare occurrance, and this way the user has to
     * purposefully go to "New" or type it in.
     */

    list_of_customers = g_list_append(list_of_customers, "New");

    return list_of_customers;
}
```

fill_customer_info(), *clear_frm_main()*, and *fill_frm_main()* Functions

Now that the combo box of customer numbers is filled, the user will normally pick one, in which case that customer's information should be entered into the text and entry widgets on frm_main. To perform that function, Listing 7.5 will fill in the customer data from the database; in addition, the functions for clearing the text and entry widgets on frm_main are listed.

Listing 7.5 **The *fill_customer_info()*, *clear_frm_main()*, and *fill_frm_main()* Functions**

```
void fill_customer_info()
{

   /* This function will retrieve a customer record from
    * the database and fill in the text boxes on frm_main by
    * calling fill_frm_main().
    */

   GtkCombo    *cbo;
   MYSQL_RES   *result_set;
   MYSQL_ROW   row;

   /* First, connect to the database, and then get the customer
    * number from the entry (child) widget of cbo_customer_number.
    */

   get_conx();

   cbo = GTK_COMBO(lookup_widget(frm_main, "cbo_customer_number"));
   g_print(gtk_entry_get_text(GTK_ENTRY(cbo->entry)));
   g_print("\n");

   /* Check to see if a new customer is being added rather than
    * querying for an existing customer. If so, clear frm_main
    * and call create_new_customer().
    */

   if (g_strcasecmp(gtk_entry_get_text (GTK_ENTRY(cbo->entry)), "New") == 0)
       {
            g_print("Creating new customer record...\n");
            clear_frm_main();

            create_new_customer();

            /* Exit this routine after setting up for a
             * new customer record. */

            return;
       }
```

```
/* What if cbo_customer_number is blank? Clear the form, because
 * there is no customer information associated with a "blank"
 * customer number.
 */

if (g_strcasecmp(gtk_entry_get_text (GTK_ENTRY(cbo->entry)), "") == 0)
    {
         g_print("No Data:  Blank Customer Record...\n");
         clear_frm_main();

         return;
    }

/* mysql_query() returns a result only indicating whether or not the
 * query sent was legal or not. In this case, sending a query
 * with a customer number that does not exist is legal, it just
 * returns zero rows.
 */

if (mysql_query (conx,
                 g_strconcat("select * ",
                             "from tbl_customers where num = ",
                             gtk_entry_get_text (GTK_ENTRY(cbo->entry)),
                             0L
                             )
                ) != 0
   )
  {
     g_print("mysql_query failure in fill_customer_info()...\n");
     gtk_statusbar_push(GTK_STATUSBAR(lookup_widget(frm_main,
                                                    "statusbar")), 1,
                   "Error retrieving customer data (illegal query)."
                   " Contact administrator.");
  }
else
  {
     g_print("Fetching customer data...\n");
     gtk_statusbar_push(GTK_STATUSBAR(lookup_widget(frm_main,
                                                     "statusbar")), 1,
                                        "Fetching Customer data...");

     result_set = mysql_store_result (conx);

     if (mysql_num_rows (result_set) == 0)
     {
        g_print("Invalid Customer Number...\n");
        gtk_statusbar_push(GTK_STATUSBAR(lookup_widget(frm_main,
                                                       "statusbar")), 1,
                                          "Invalid Customer Number...");

        /* Clear frm_main to avoid any confusion. */
        clear_frm_main();
```

continues

Listing 7.5 **Continued**

```
        }
        else
        {
          /* There could also be a check here to make sure not more than
           * one row returned, which would indicate that something is
           * wrong tbl_customers.num should be the table key.
           *
           * However, because that key has been specified
           * in the database structure (and to keep this routine a bit
           * simpler), that check will be skipped.
           */

          row = mysql_fetch_row (result_set);
          g_print("Preparing for widget fill...\n");

          /* Fill frm_main with customer data. The parameter being sent
           * is the row of data from the sesi database.
           */

          fill_frm_main(row);
          gtk_statusbar_push(GTK_STATUSBAR(lookup_widget(frm_main,
                             "statusbar")), 1,
                             "Customer data filled in...");
        }

     }
   return;

}

void clear_frm_main()
{

    /* This function clears all the text boxes on frm_main that
     * display customer information.
     */

    g_print("Clearing customer information...\n");

    gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                                        "entry_customer_name")), "");

    gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                                          "entry_ship_to_addr1")), "" );
    gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                                          "entry_ship_to_addr2")), "");
    gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                                          "entry_ship_to_city")), "");
    gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                                          "entry_ship_to_st")), "");
```

```
    gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                                            "entry_ship_to_zip")), "");

    gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                                            "entry_bill_to_addr1")), "");
    gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                                            "entry_bill_to_addr2")), "");
    gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                                            "entry_bill_to_city")), "");
    gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                                            "entry_bill_to_st")), "");
    gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                                            "entry_bill_to_zip")), "");

    gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main, "entry_first")), "");
    gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main, "entry_last")), "");
    gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main, "entry_title")), "");
    gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main, "entry_phone")), "");

    /* Delete the text from the customer comments
     * text box. The order comments box should not
     * change no matter which customer is displayed
     * because the order comments go with the current
     * instantiation of this application.
     */

    gtk_text_set_point(GTK_TEXT(lookup_widget(frm_main,
                                    "txt_customer_comments")), 0);
    g_print("current insertion point is %i\n...",
            gtk_text_get_point(GTK_TEXT(lookup_widget(frm_main,
                                     "txt_customer_comments"))));

    g_print("delete returned %i\n",
              gtk_text_forward_delete (GTK_TEXT(lookup_widget(frm_main,
                                                 "txt_customer_comments")),
                          gtk_text_get_length(GTK_TEXT(lookup_widget(frm_main,
                                                 "txt_customer_comments")))
             ));
}

void fill_frm_main(MYSQL_ROW in_row)
{
    /* This function will fill in the text boxes on frm_main
     * that display the customer information.
     *
     * Clear the form so that the fill operation starts from a known
     * state.
     */

    clear_frm_main();
```

continues

Listing 7.5 **Continued**

```
/* in_row is the parameter to this function. It contains one
 * row from tbl_customers, and that information is what is
 * to be displayed.
 */

gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                   "entry_customer_name")), in_row[1]);

if (in_row[2] != 0L ) {
   gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                      "entry_ship_to_addr1")), in_row[2]);
}
if (in_row[3] != 0L ) {
   gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                      "entry_ship_to_addr2")), in_row[3]);
}
if (in_row[4] != 0L ) {
   gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                      "entry_ship_to_city")), in_row[4]);
}
if (in_row[5] != 0L ) {
   gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                      "entry_ship_to_st")), in_row[5]);
}
if (in_row[6] != 0L ) {
   gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                      "entry_ship_to_zip")), in_row[6]);
}

if (in_row[7] != 0L ) {
   gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                      "entry_bill_to_addr1")), in_row[7]);
}
if (in_row[8] != 0L ) {
   gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                      "entry_bill_to_addr2")), in_row[8]);
}
if (in_row[9] != 0L ) {
   gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                      "entry_bill_to_city")), in_row[9]);
}
if (in_row[10] != 0L ) {
   gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                      "entry_bill_to_st")), in_row[10]);
}
if (in_row[11] != 0L ) {
   gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                      "entry_bill_to_zip")), in_row[11]);
}
```

```
    g_print("Filling contact information...\n");
    if (in_row[12] != 0L ) {
        gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                           "entry_first")), in_row[12]);
    }

    if (in_row[13] != 0L ) {
        gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                           "entry_last")), in_row[13]);
    }

    if (in_row[14] != 0L ) {
        gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                           "entry_phone")), in_row[14]);
    }

    if (in_row[15] != 0L ) {
        gtk_entry_set_text(GTK_ENTRY(lookup_widget(frm_main,
                           "entry_title")), in_row[15]);
    }

    if (in_row[16] != 0L ) {
        gtk_text_insert(GTK_TEXT(lookup_widget(frm_main,
                                 "txt_customer_comments")),
                        0L, 0L, 0L,
                        in_row[16], -1
                       );
    }

    /* Because the data retrieved has not been edited,
     * set the need_to_save_edits flag to false and "gray-out"
     * cmd_save_edits to give the user a visual cue that the
     * data has not changed since it was pulled from the database.
     */

    need_to_save_edits = FALSE;
    gtk_widget_set_sensitive(lookup_widget(frm_main,
                                           "cmd_save_edits"), FALSE);
}
```

So far, this chapter has not addressed the code for entering a new customer. That code is shown in Listing 7.14.

fill_items_ordered() Function

When the user finds the desired customer, he will want to start entering items ordered. So that code is next. First, the "speed entry" method (which uses the widgets at the top left of frm_items_ordered) will be covered. Then, the slower method will be described using clist_items and cmd_add_down, which is meant to be a more mouse-intensive way of working. However, before any of that, the form has to fill in all the items and be prepared for use. Listing 7.6 does precisely that.

Listing 7.6 *fill_items_ordered()* **Function from** *sesi_utils.c*

```
void fill_items_ordered()
{
   MYSQL_RES   *result_set;
   MYSQL_ROW   db_row;
   gchar       *clist_row[3] = {"", "", ""};

   /* This function does the initial fill of widgets in
    * frm_items_ordered, as needed. It is anticipated that it will
    * normally only be called from the realize or show events of
    * frm_items_ordered because the list of available items in the
    * database is not expected to change while the application is in
    * use.
    * The database access code will operate in a manner similar to other
    * functions previously listed.
    */

   get_conx();
   if (mysql_query (conx, "select * from tbl_items") != 0)
      {
          /* If this query is unable to return a list of items,
           * what should be done? Theoretically, if the user
           * knows the item number, the application should
           * be able to query the database for the desired information
           * (price). However, it is more likely that something
           * has gone wrong with the connection to the database and
           * any read operation against the database will fail. Still,
           * it should be given the benefit of the doubt…
           */

        g_print("Failure to retreive list of items...\n");

          /* …instead of a fatal_msgbox() call. If something
           * has really gone wrong, the calls to the database to
           * get the price of the item should produce the
           * fatal_msgbox() call.
           */
      }
   else g_print("Retrieved customer numbers from db...\n");

   /* Now iterate through the rows, get the values for customer number,
    * and add them to cbo_customer_number.
    */

   result_set = mysql_store_result (conx);
   while ((db_row = mysql_fetch_row (result_set)) != 0L)
      {  clist_row[0] = db_row[0];
         clist_row[1] = db_row[1];
         clist_row[2] = db_row[2];
```

```
        gtk_clist_append(GTK_CLIST(lookup_widget(frm_items_ordered,
                                                 "clist_items")),
                         clist_row);
    }

}
```

speed_add() Function

Next, the normal course of events is that the user will use the widgets in the upper-left corner of frm_items_ordered to type in item numbers, tab to the Add button (cmd_add), and then repeat this cycle for the entire order. That is, when the customer is on the phone, they know what items they want and either the customer has those order numbers ready, or the person using the software knows the order numbers—at least the item numbers for the most common items. This will be called "speed add" and it is covered in Listing 7.7.

Listing 7.7 **Function *speed_add()* for Quickly Adding Items to the Order When the Item Number Is Known**

```
void speed_add()
{
   MYSQL_RES   *result_set;
   MYSQL_ROW   db_row;
   gchar       *sql;
   gchar       *clist_row[4] = {"", "", "", ""};
   gint        int_quantity;
   gchar       *str_quantity;
   gdouble     dbl_total_price;
   gchar       *str_total_price;
   gchar       *str_total_price_formatted;
   gchar       *str_order_total;
   gchar       *str_order_total_formatted;

   /* This function will be called whenever the user clicks on cmd_add,
    * which is in the upper-left corner of frm_items_ordered. This is
    * the "speedy" method of entering items, as opposed to the "slow"
    * method in function slow_add (the next function after this one).

    * First, get the item number and get that item's price from the
    * database.*/

   get_conx();
   sql = g_strconcat("select * from tbl_items where item = '",
                     gtk_editable_get_chars(
                           GTK_EDITABLE(lookup_widget(frm_items_ordered,
                                                      "entry_item_number"
                               )) , 0, -1), "'", 0L);
```

continues

Listing 7.7 **Continued**

```
g_print("sql is %s\n", sql);

if (mysql_query (conx, sql) != 0)
   {
     g_print("Failure to retreive item information from mysql_query...\n");
   }
else
   {
       result_set = mysql_store_result (conx);

       /* If the program gets to this point, it has issued a correct
        * SQL statement against the database; however, the item
        * number could be a non-existent item number. As with
        * fill_customer_info(), the software needs to check that the
        * number of rows is greater than 0.*/

        if (mysql_num_rows (result_set) == 0)
        {
            g_print("Invalid Item Number...\n");
            gtk_statusbar_push(GTK_STATUSBAR(lookup_widget(frm_main,
                                  "statusbar")), 1,
                              "Invalid Item Number...");

            /* If the user gets to this point, they should see that
             * no line has been added to the lower CList box. This is
             * an assumption that you would not want to make on a
             * project any larger than this one, but the way
             * this project is defined makes it an acceptable tradeoff.
             */

         }
         else
         {
             g_print("Retreived item information...\n");

             /* Now that the item number has been verified,
              * get quantity, do the math,
              * and add to clist_items_ordered.
              */

             db_row = mysql_fetch_row (result_set);

             /* The next two calls demonstrate how to get the same
              * information from a spinbutton as two different
              * data types - int and gchar*.*/

             int_quantity = gtk_spin_button_get_value_as_int(GTK_SPIN_BUTTON(
                                 lookup_widget(frm_items_ordered,
                                 "spinbutton_quantity")));
```

```
                str_quantity = gtk_editable_get_chars(
                                GTK_EDITABLE(
                                  lookup_widget(frm_items_ordered,
                                  "spinbutton_quantity")
                                ), 0, -1);

                dbl_total_price =  int_quantity * atof(db_row[2]);
                g_print("dbl_total_price is %f\n", dbl_total_price);

                str_total_price = g_strdup_printf("%f", dbl_total_price);

                clist_row[0] = db_row[0];
                clist_row[1] = db_row[1];
                clist_row[2] = str_quantity;

                /* Next, format the output by finding the decimal and then
                 * including the next three characters for output as in
                 * ".xx".*/

                str_total_price_formatted = g_strndup(str_total_price,
                                            strcspn(str_total_price, ".") + 3);

                clist_row[3] = str_total_price_formatted;

                gtk_clist_append(GTK_CLIST
                             (lookup_widget(frm_items_ordered,
                                            "clist_items_ordered")),
                                clist_row);

                /* Finally, recalculate the total and fill in that
                 * label. It is easier to keep a running tally than to
                 * try to access the contents of the CList widget.
                 */

                dbl_order_total = dbl_order_total + dbl_total_price;

                str_order_total = g_strdup_printf("%f", dbl_order_total);
                str_order_total_formatted = g_strndup(str_order_total,
                                            strcspn(str_order_total, ".") +3);

                gtk_label_set_text(GTK_LABEL(lookup_widget
                                               (frm_items_ordered,
                                                "lbl_order_total_numeric")),
                                   str_order_total_formatted
                                  );

            }
        }
    }
```

slow_add() Function

Listing 7.8 is the code for the user that is going to click his way through the order. Here, the user will select an item from clist_items, click cmd_add_down (the Add button between the CList widgets), and repeat that process.

Listing 7.8 **Function *slow_add()* for Adding Items to the Order Using the Mouse**

```
void slow_add()
{

   /* This function is the more mouse-intensive way to add
    * ordered items to the list, which tends to make it
    * a slower way to add items. It is called when the
    * user clicks the "Add" button between the two
    * list boxes on frm_items_ordered.
    */

   GtkCList    *clist_target;
   gint        row_target = -1;

   gchar       *cell_item_number;
   gchar       *cell_item_description;
   gchar       *cell_item_price;

   gchar       *clist_row[4] = {"", "", "", ""};
   gint        int_quantity;
   gchar       *str_quantity;
   gdouble     dbl_total_price;
   gchar       *str_total_price;
   gchar       *str_total_price_formatted;
   gchar       *str_order_total;
   gchar       *str_order_total_formatted;

   clist_target = GTK_CLIST(lookup_widget(frm_items_ordered, "clist_items"));

   /* The following call gets the row that is selected, not focused.
    * The 0 is for the "0th" item in the list of selected rows.
    */

   row_target = (gint)g_list_nth_data( (clist_target)->selection, 0 );

   g_print("Row to move down is %i...\n", row_target);

   if (row_target == -1)
      {
         gtk_statusbar_push(GTK_STATUSBAR(lookup_widget(frm_main,
                                                        "statusbar")), 1,
                            "No Item Selected...");

         g_print("No Item selected...\n");

         return;
      }
```

```
/* The next three calls get the information about the
 * item selected in CList_items, the list of available
 * items that a customer can order. They retrieve the
 * item number, the description, and the price for
 * use later in the function.
 */

gtk_clist_get_text(GTK_CLIST(lookup_widget(frm_items_ordered,
                  "clist_items")),
                  row_target, 0, &cell_item_number);
gtk_clist_get_text(GTK_CLIST(lookup_widget(frm_items_ordered,
                  "clist_items")),
                  row_target, 1, &cell_item_description);
gtk_clist_get_text(GTK_CLIST(lookup_widget(frm_items_ordered,
                  "clist_items")),
                  row_target, 2, &cell_item_price);


/* Spinbutton1 is the spinbutton next to cmd_add_down, between the
 * two CList boxes. Forgot to change the name on that one… :-|
 */

int_quantity = gtk_spin_button_get_value_as_int(GTK_SPIN_BUTTON(
                                lookup_widget(frm_items_ordered,
                                "spinbutton1")));
str_quantity = gtk_editable_get_chars(
                               GTK_EDITABLE(
                                lookup_widget(frm_items_ordered,
                                "spinbutton1")
                               ), 0, -1);

 /* Compute the price by multiplying quantity with price,
  * then prepare the CList_row[] array by setting the
  * values that will be added to the CList widget of
  * items the customer has ordered, clist_items_ordered.
  */

 dbl_total_price =  int_quantity * atof(cell_item_price);
 g_print("dbl_total_price is %f\n", dbl_total_price);

 str_total_price = g_strdup_printf("%f", dbl_total_price);

 clist_row[0] = cell_item_number;
 clist_row[1] = cell_item_description;
 clist_row[2] = str_quantity;

 str_total_price_formatted = g_strndup(str_total_price,
                                       strcspn(str_total_price, ".") + 3);

 /* The previous function call set formatted the price correctly
  * for display; the next sets the last field in the array to
```

continues

Listing 7.8 **Continued**

```
     * that formatted price. Immediately after that, the clist_row[]
     * array is added to clist_items_ordered.
     */

    clist_row[3] = str_total_price_formatted;

    gtk_clist_append(GTK_CLIST
                     (lookup_widget(frm_items_ordered,
                                    "clist_items_ordered")),
                    clist_row);

    /* Recalculate the running total. */

    dbl_order_total = dbl_order_total + dbl_total_price;
    str_order_total = g_strdup_printf("%f", dbl_order_total);
    str_order_total_formatted = g_strndup(str_order_total,
                                          strcspn(str_order_total, ".") +3);

    gtk_label_set_text(GTK_LABEL(lookup_widget
                                    (frm_items_ordered,
                                     "lbl_order_total_numeric")),
                       str_order_total_formatted
                       );

}
```

remove_ordered_item() Function

Next follows the code to remove a line item that has been added. Listing 7.9 covers removing an item from clist_items_ordered, which also subtracts the appropriate amount from the order total.

Listing 7.9 **Function *remove_ordered_item()* from *sesi_utils.c***

```
void remove_ordered_item()
{

   /* This function removes a line from clist_items_ordered,
    * most likely because the order entry clerk made a mistake
    * or the customer changed his mind. In either case,
    * the item needs to be removed and the order total price
    * must be recalculated.
    */

   GtkCList  *clist_target;
   gint      row_target = -1;
   gchar     *cell_line_item_price;
```

```
    clist_target = GTK_CLIST(lookup_widget(frm_items_ordered,
                                           "clist_items_ordered"));

    /* The following call gets the row that is selected, not focused.
     * The 0 is for the "0th" item in the list of selected rows.
     */

    row_target = (gint)g_list_nth_data( (clist_target)->selection, 0 );

    g_print("Row to delete is %i...\n", row_target);

    if (row_target == -1)
       {
          gtk_statusbar_push(GTK_STATUSBAR(lookup_widget(frm_main,
                             "statusbar")), 1,
                             "Select an item to remove first...");

          g_print("No Item selected from clist_items_ordered...\n");

          return;
       }

    /* …else continue with the remove operation… */

    /* First, get the amount of this line item so that it can be
     * subtracted from the total before the line is deleted.
     */

    gtk_clist_get_text(GTK_CLIST(lookup_widget(frm_items_ordered,
                       "clist_items_ordered")),
                       row_target, 3, &cell_line_item_price);

    dbl_order_total = dbl_order_total - atof(cell_line_item_price);

    gtk_label_set_text(GTK_LABEL(lookup_widget
                                    (frm_items_ordered,
                                     "lbl_order_total_numeric")),
                         g_strdup_printf("%f", dbl_order_total)
                        );

    /* Finally, remove the line item that is selected. */

    gtk_clist_remove(clist_target, row_target);
}
```

select_item() Function

The `select_item()` function is a utility function that is used in several places. It takes a single input parameter, searches clist_items for the input parameter, and then selects and shows the selected row in clist_items. The code is in Listing 7.10.

Listing 7.10 **Function *select_item()* from *sesi_utils.c*, Which Selects and Shows the Line in *clist_items* That Corresponds to Its Parameter**

```
void select_item(gchar *target_item_num)
{
   /* This function is required for frm_items_ordered to
    * work – the find item widget will search the database for an
    * item number and then call this function to select that item in
    * clist_items. When given as an input for an item number (the
    * only parameter to this function), this function iterates
    * through all lines in clist_items and finds the one that
    * matches the input parameter.
    */

   GtkCList    *target_clist;
   gint        number_of_rows;
   gint        counter;
   gchar       *clist_item_number;

   g_print("Target item number is:  %s...\n", target_item_num);

   target_clist = GTK_CLIST(lookup_widget(frm_items_ordered, "clist_items"));

   /* First, find out how many rows are in the CList widget.
    */

   number_of_rows = ((target_clist)->rows);

   g_print("number_of_rows is:  %i", number_of_rows);

   /* Iterate through all the rows searching for the target row. */

   for(counter = 0; counter < number_of_rows; counter++)
    {
         gtk_clist_get_text(GTK_CLIST(lookup_widget(
                                       frm_items_ordered,
                                       "clist_items")),
                     counter, 0, &clist_item_number);

         if (g_strcasecmp(clist_item_number, target_item_num) == 0)
            {
               g_print("Found target line %i in clist_items..\n", counter);
               break;
            }
         else
            {
               /* continue searching... */
            }

    }
```

```
    /* When you have found the desired line in clist_items, select and move it
     * into view in the window.
     */

    gtk_clist_select_row(target_clist, counter, 0);
    gtk_clist_moveto(target_clist, counter, 0, 0, 0);

}
```

enter_found_items() Function

If the user needs to search for an item, he will open up frm_find_item. There, the function `enter_found_items()` (shown in in Listing 7.11) will enable him to search the database for matches on the desired character string (the single parameter). With a list of matching items, `enter_found_items()` (see Listing 7.11) will populate the CList widget in frm_find_item. The user will then pick from that list a single row to return to frm_items_ordered.

Listing 7.11 **Function *enter_found_items()* from *sesi_utils.c*; It Searches the Database for Matching Items and Populates the CList Widget in *frm_find_item***

```
void enter_found_items(gchar *str)
{

    /* This function will search the tbl_items table for possible
     * matches to the user's text (which is "str", the parameter passed
     * into this function). It will display those found records
     * in clist_found_items.
     *
     * First, connect to the database and get a result set of
     * the possible rows.
     *
     * Remember that the search will be on a string (the
     * input parameter), against
     * all possible columns that could match a string,
     * that is, all non-numeric fields.
     */

    MYSQL_RES   *result_set;
    MYSQL_ROW   db_row;
    gchar       *sql;
    gchar       *clist_row[3] = {"", "", ""};
    gint        counter;
    gint        number_of_rows;

    g_print("starting create_find_item_clist...\n");

    get_conx();
```

continues

Listing 7.11 **Continued**

```
/* In MySQL, the percent character is the wildcard for all
 * possible matches.
 */

sql = g_strconcat("select * from tbl_items where item like '%",
                  str,
                  "%' or description like '%",
                  str,
                  "%'",
                  0L);

g_print("sql is :  %s\n", sql);

if (mysql_query (conx, sql) != 0)
   {
     g_print("Failure to retreive find item data from mysql_query...\n");
   }
else
   {

       /* Retrieve the results and clear clist_found_items. */

       result_set = mysql_store_result (conx);
       db_row = mysql_fetch_row (result_set);

       /* Clear the CList widget of all items. */
       gtk_clist_clear(GTK_CLIST(lookup_widget(frm_find_item,
                     "clist_found_items")));

       number_of_rows = mysql_num_rows(result_set);
       g_print("number_of_rows is:  %i", number_of_rows);

       /* Iterate through the result set and add each row to
        * clist_found_items.
        */

       for (counter = 0; counter < number_of_rows; counter++)
         {
               clist_row[0] = db_row[0];
               clist_row[1] = db_row[1];
               clist_row[2] = db_row[2];

               gtk_clist_append(GTK_CLIST(lookup_widget
                                (frm_find_item,
                                "clist_found_items")),
                                clist_row);

               /* Fetch the next row. */
```

```
                db_row = mysql_fetch_row (result_set);
            }
        }

    g_print("exiting create_find_item_clist...\n");

}
```

select_customer() and *enter_found_customers()* Functions

Function `enter_found_customers()` (see Listing 7.12) performs a similar function in frm_find_customer. Function `select_customer()` (also Listing 7.12) sets cbo_customer_number to a specified customer and refreshes the data displayed in frm_main. Both are called from the callback initiated by the Done button on frm_find_customer.

Listing 7.12 **Functions *select_customer()* and *enter_found_customers()* from *sesi_utils.c***

```
void select_customer(gchar *target_customer_num)
{

    /* Set cbo_customer_number to target_customer_num, the
     * parameter passed in, and then call fill_customer_info.
     */

    GtkCombo *cbo;

    cbo = GTK_COMBO(lookup_widget(frm_main, "cbo_customer_number"));
    gtk_entry_set_text(GTK_ENTRY(cbo->entry), target_customer_num);

    /* Use the existing customer information fill routine. */

    fill_customer_info();
}

void enter_found_customers(gchar *str)
{

    /* This function searches for matches to str, the parameter
     * passed in, in tbl_customers, and then enters those
     * records to clist_found_customer.
     *
     * First, connect to the database and get a result_set of
     * the possible rows.
     *
     * Remember that the search will be on a string, against
     * all possible columns that could match a string.
     */

    MYSQL_RES   *result_set;
```

continues

Listing 7.12 **Continued**

```
MYSQL_ROW   db_row;
gchar       *sql;
gchar       *clist_row[17] = {"","","","","",
                              "","","","","",
                              "","","","","",
                              "",""};
gint        counter;
gint        number_of_rows;

g_print("starting enter_found_customer...\n");

get_conx();

/* In MySQL, the percent character is the wildcard for all
 * possible matches.
 */

sql = g_strconcat("select * from tbl_customers where name like '%",
                  str,
                  "%' or ship_to_addr1 like '%",
                  str,
                  "%' or ship_to_addr2 like '%",
                  str,
                  "%' or ship_to_city like '%",
                  str,
                  "%' or ship_to_state like '%",
                  str,
                  "%' or ship_to_zip like '%",
                  str,
                  "%' or bill_to_addr1 like '%",
                  str,
                  "%' or bill_to_addr2 like '%",
                  str,
                  "%' or bill_to_city like '%",
                  str,
                  "%' or bill_to_state like '%",
                  str,
                  "%' or bill_to_zip like '%",
                  str,
                  "%' or contact_first like '%",
                  str,
                  "%' or contact_last like '%",
                  str,
                  "%' or phone like '%",
                  str,
                  "%' or title like '%",
                  str,
                  "%' or comments like '%",
                  str,
                  "%'",
                  0L);
```

```
g_print("sql is :  %s\n", sql);

if (mysql_query (conx, sql) != 0)
   {
     g_print("Failure to retreive find item data from mysql_query...\n");
   }
else
   {

       /* The query succeeded, so store the result
        * and prepare the CList widget to display the
        * results.
        */

       result_set = mysql_store_result (conx);
       db_row = mysql_fetch_row (result_set);

       /* Clear the CList widget of all items. */
       gtk_clist_clear(GTK_CLIST(lookup_widget(frm_find_customer,
                                               "clist_found_customer")));

       number_of_rows = mysql_num_rows(result_set);
       g_print("number_of_rows is:  %i", number_of_rows);

       /* Fill the array, which will in turn fill
        * clist_found_customer.
        */

       for (counter = 0; counter < number_of_rows; counter++)
         {
              clist_row[0] = db_row[0];
              clist_row[1] = db_row[1];
              clist_row[2] = db_row[2];
              clist_row[3] = db_row[3];
              clist_row[4] = db_row[4];
              clist_row[5] = db_row[5];
              clist_row[6] = db_row[6];
              clist_row[7] = db_row[7];
              clist_row[8] = db_row[8];
              clist_row[9] = db_row[9];
              clist_row[10] = db_row[10];
              clist_row[11] = db_row[11];
              clist_row[12] = db_row[12];
              clist_row[13] = db_row[13];
              clist_row[14] = db_row[14];
              clist_row[15] = db_row[15];
              clist_row[16] = db_row[16];
              clist_row[17] = db_row[17];

              /* Finally, append the row to clist_found_customer. */
```

continues

Listing 7.12 **Continued**

```
                gtk_clist_append(GTK_CLIST(lookup_widget
                                (frm_find_customer,
                                 "clist_found_customer")),
                                  clist_row);

                /* Fetch the next row. */

                db_row = mysql_fetch_row (result_set);
            }
        }
}
```

write_order() Function

Function `write_order()` produces the final result of this application. Its purpose is to create a filename and then write the order to that disk file. In this case (lacking a better option), it writes to the current directory. It is rather long but fairly straightforward; see Listing 7.13

Listing 7.13 **Function *write_order()* from *sesi_utils.c*. At the End of the Listing are Functions *right_pad()* and *left_pad()*,Which are Only Used by *write_order()*.**

```
void write_order()
{
   /* This function computes an appropriate filename, gathers
    * data from the various forms in the application, and writes
    * an order to disk. It uses the current directory by default.
    */

   gchar      *str_now;
   time_t     now;
   GtkCList   *target_clist;
   gint       number_of_line_items;
   GtkCombo   *cbo;
   gchar      *cust_name_num;
   gchar      *file_name;
   FILE       *fp;
   gchar      *str_ship_to_csz, *str_bill_to_csz;
   gint       counter;
   gchar      *cell_item_number, *cell_description, *cell_quantity, *cell_price;
   gchar      *str_order_total, *str_order_total_formatted;

   /* First, some basic error checking.
    * Has a customer been selected?
    */

   cbo = GTK_COMBO(lookup_widget(frm_main, "cbo_customer_number"));
```

```
if (g_strcasecmp(gtk_entry_get_text (GTK_ENTRY(cbo->entry)), "New") == 0)
    {
         g_print("New customer record, not valid for writing an order...\n");
         gtk_statusbar_push(GTK_STATUSBAR(lookup_widget(frm_main,
                             "statusbar")), 1,
                             "New is not a valid customer number "
                             "for order generation...");
         return;
    }

if (g_strcasecmp(gtk_entry_get_text (GTK_ENTRY(cbo->entry)), "") == 0)
    {
         g_print("No customer record, not valid for writing an order...\n");
         gtk_statusbar_push(GTK_STATUSBAR(lookup_widget(frm_main,
                             "statusbar")), 1,
                             "Customer Number can not be blank...");
         return;
    }

/* Have items been ordered? */

target_clist = GTK_CLIST(lookup_widget(frm_items_ordered,
                         "clist_items_ordered"));
number_of_line_items = ((target_clist)->rows);

if (number_of_line_items == 0)
  {
     gtk_statusbar_push(GTK_STATUSBAR(lookup_widget(frm_main,
                        "statusbar")), 1,
                        "No items have been selected for this invoice...");

     return;
  }

/* When the error checking is done, it is time to generate a
 * filename for this order.
 *
 * First come the customer name and number.
 */

cust_name_num = g_strconcat(gtk_entry_get_text(
                              GTK_ENTRY(lookup_widget(frm_main,
                                        "entry_customer_name"))),
                             ".",
                             gtk_entry_get_text(GTK_ENTRY(cbo->entry)),
                             ".",
                             0L
                             );

g_print("cust_name_num is: %s\n", cust_name_num);
```

continues

Listing 7.13 **Continued**

```
/* Next are the date and time of the order.
 *
 * The g_strstrip() call is necessary because the ctime() call
 * returns a CR/LF character at the end of the string. g_strstrip()
 * removes all non-printable characters from the start and end
 * of the string that it is given as its parameter.
 */

time(&now);
str_now = g_strstrip((gchar *) ctime(&now));

g_print("ctime returns:  %s\n", str_now);

/* Now you'll put them all together to get the filename of the order.
 * Note that the spaces and colons will be replaced by dots.
 */

file_name = g_strconcat(cust_name_num, str_now, ".txt", 0L);
g_print("file_name is %s\n", file_name);

/* The g_strdelimit() function replaces all occurrences, any
 * member of the second parameter with the first. In this
 * case, any space, colon, OR comma will be replaced with
 * a period.
 *
 * Note that the second parameter is surrounded by double
 * quotes, and the third parameter is surrounded by single quotes.
 *
 * The idea here is to create a filename that is sufficiently
 * descriptive and will not cause problems on the largest
 * number of operating systems. This application was
 * built with the idea that the output file produced by this
 * function would be transferred to another machine – most
 * likely an FTP transfer to a Win32 machine.
 */

file_name = g_strdelimit(file_name, " :,", '.');

g_print("file_name is now %s\n", file_name);

/* Now to open a file handle for writing. */

if ((fp = fopen (file_name, "w")) == 0L)
   {
       /* Could not write to the file for some reason. */
       gtk_statusbar_push(GTK_STATUSBAR(lookup_widget(frm_main,
                          "statusbar")), 1,
                          "Unable to write to file, "
                          "contact system administrator...");
   }
else
   {
```

```
/* File handle is open for write operation. */
gtk_statusbar_push(GTK_STATUSBAR(lookup_widget(frm_main,
                  "statusbar")), 1,
                  "File is open for write operation...");

/* This file will be fixed width, and to be on the safe side,
 * it will be assumed that the "page" is 70 columns wide instead
 * of the normal 80.
 */

/* First, write the header information: date, time, customer name
 * and number, and so on.
 */

fprintf(fp, "Specialty Electrical Supply, Inc."); fprintf(fp, "\n");
fprintf(fp, "Shipping Order Form");               fprintf(fp, "\n");
fprintf(fp, "================================="); fprintf(fp, "\n");
                                                  fprintf(fp, "\n");

fprintf(fp, str_now);             fprintf(fp, "\n");
fprintf(fp, cust_name_num);       fprintf(fp, "\n");
                                  fprintf(fp, "\n");

/* Write the addresses to the file in a side-by-side format. */

fprintf(fp, right_pad("Ship to Address", " ", 35));
fprintf(fp, "Bill to Address"); fprintf(fp, "\n");

fprintf(fp, right_pad(gtk_entry_get_text(GTK_ENTRY(
                        lookup_widget(frm_main,
                        "entry_ship_to_addr1"))), " ", 35));
fprintf(fp, gtk_entry_get_text(GTK_ENTRY(lookup_widget(frm_main,
                        "entry_bill_to_addr1"))));
fprintf(fp, "\n");

fprintf(fp, right_pad(gtk_entry_get_text(GTK_ENTRY(
                        lookup_widget(frm_main,
                        "entry_ship_to_addr2"))), " ", 35));
fprintf(fp, gtk_entry_get_text(GTK_ENTRY(lookup_widget(frm_main,
                        "entry_bill_to_addr2"))));
fprintf(fp, "\n");

/* Gather the city, state, and ZIP info into one string, and then pad it.
 */

str_ship_to_csz = g_strconcat
                  (
                      gtk_entry_get_text(GTK_ENTRY(
                          lookup_widget(frm_main,
                          "entry_ship_to_city"))), " ",
                      gtk_entry_get_text(GTK_ENTRY(
```

continues

Listing 7.13 **Continued**

```
                            lookup_widget(frm_main,
                            "entry_ship_to_st"))), "  ",
                        gtk_entry_get_text(GTK_ENTRY(
                            lookup_widget(frm_main,
                            "entry_ship_to_zip"))),
                        0L
                    );

    str_bill_to_csz = g_strconcat
                    (
                        gtk_entry_get_text(GTK_ENTRY(
                            lookup_widget(frm_main,
                            "entry_bill_to_city"))), " ",
                        gtk_entry_get_text(GTK_ENTRY(
                            lookup_widget(frm_main,
                            "entry_bill_to_st"))), "  ",
                        gtk_entry_get_text(GTK_ENTRY(
                            lookup_widget(frm_main,
                            "entry_bill_to_zip"))),
                        0L
                    );

    fprintf(fp, right_pad(str_ship_to_csz, " ", 35));
    fprintf(fp, str_bill_to_csz);
    fprintf(fp, "\n");
    fprintf(fp, "\n");

    fprintf(fp, "Order Detail Information\n");
    fprintf(fp, "========================\n");
    fprintf(fp, "\n");

    fprintf(fp, right_pad("Item Num", "-", 12));
    fprintf(fp, right_pad("Item Description", "-", 37));
    fprintf(fp, left_pad("Quantity", "-", 8));
    fprintf(fp, left_pad("Price", "-", 13));
    fprintf(fp, "\n");

    /* Iterate through clist_items_ordered and write the
     * order information for each line.
     */

    for (counter = 0; counter < number_of_line_items; counter++)
      {
          gtk_clist_get_text(target_clist, counter, 0, &cell_item_number);
          gtk_clist_get_text(target_clist, counter, 1, &cell_description);
          gtk_clist_get_text(target_clist, counter, 2, &cell_quantity);
          gtk_clist_get_text(target_clist, counter, 3, &cell_price);
```

```
                fprintf(fp, right_pad(cell_item_number, " ", 12));
                fprintf(fp, right_pad(cell_description, " ", 40));
                fprintf(fp, left_pad(cell_quantity, " ", 5));
                fprintf(fp, left_pad(cell_price, " ", 13));

                fprintf(fp, "\n");

                str_order_total = g_strdup_printf("%f", dbl_order_total);
                str_order_total_formatted = g_strndup(str_order_total,
                                          strcspn(str_order_total, ".") +3);

            }
          fprintf(fp, left_pad("==============", " ", 70)); fprintf(fp, "\n");
          fprintf(fp, left_pad(str_order_total_formatted, " ", 70));

          fprintf(fp, "\n");

          fprintf(fp, "Order Comments\n");
          fprintf(fp, "==============\n");

          fprintf(fp, gtk_editable_get_chars(GTK_EDITABLE(lookup_widget(frm_main,
                    "txt_order_comments")), 0, -1));

          fprintf(fp, "\n");

          fclose(fp);

          gtk_statusbar_push(GTK_STATUSBAR(lookup_widget(frm_main,
                             "statusbar")), 1,
                             "Order file has been created.  "
                             "Push exit to close...");
       }

}

gchar *right_pad(gchar *in_str, gchar *pad_char, gint final_length)
{

   /* This function pads characters to the right of in_str, to
    * a length of final_string.
    */

   while (strlen(in_str) < final_length)
     {
        in_str = g_strconcat(in_str, pad_char, 0L);
     }

   return in_str;
}
```

continues

Listing 7.13 **Continued**

```
gchar *left_pad(gchar *in_str, gchar *pad_char, gint final_length)
{
   /* This function pads characters to the left of in_str, to
    * a length of final_string.
    */



   while (strlen(in_str) < final_length)
     {
        in_str = g_strconcat(pad_char, in_str, 0L);
     }

   return in_str;
}
```

update_database() function

The `update_database()` function writes changes to the database. It takes the "don't force it, just get a bigger hammer" approach: It overwrites all available fields in the record based on the table key ("num," the customer number). See Listing 7.14.

Listing 7.14 **Function *update_database()*, Which Writes Updates to *tbl_customers***

```
void update_database()
{
   /* This routine will update the sesi database when changes to a
    * customer record have been made.
    */

   gchar    *sql;
   GtkCombo *cbo;

   /* Update tbl_customers; don't try to figure out which text
    * box was edited, just update all fields.
    */

   get_conx();

   cbo = GTK_COMBO(lookup_widget(frm_main, "cbo_customer_number"));

   sql = g_strconcat("update tbl_customers set ",
                     "name = '",
gtk_entry_get_text(GTK_ENTRY(lookup_widget(frm_main,
                                    "entry_customer_name"))), "', ",
                     "ship_to_addr1 = '", gtk_entry_get_text(GTK_ENTRY(
                                          lookup_widget(frm_main,
                                          "entry_ship_to_addr1"))), "', ",
```

```
                    "ship_to_addr2 = '", gtk_entry_get_text(GTK_ENTRY(
                                        lookup_widget(frm_main,
                                        "entry_ship_to_addr2"))), "', ",
                    "ship_to_city = '", gtk_entry_get_text(GTK_ENTRY(
                                        lookup_widget(frm_main,
                                        "entry_ship_to_city"))), "', ",
                    "ship_to_state = '", gtk_entry_get_text(GTK_ENTRY(
                                        lookup_widget(frm_main,
                                        "entry_ship_to_st"))), "', ",
                    "ship_to_zip = '", gtk_entry_get_text(GTK_ENTRY(
                                        lookup_widget(frm_main,
                                        "entry_ship_to_zip"))), "', ",
                    "bill_to_addr1 = '", gtk_entry_get_text(GTK_ENTRY(
                                        lookup_widget(frm_main,
                                        "entry_bill_to_addr1"))), "', ",
                    "bill_to_addr2 = '", gtk_entry_get_text(GTK_ENTRY(
                                        lookup_widget(frm_main,
                                        "entry_bill_to_addr2"))), "', ",
                    "bill_to_city = '", gtk_entry_get_text(GTK_ENTRY(
                                        lookup_widget(frm_main,
                                        "entry_bill_to_city"))), "', ",
                    "bill_to_state = '", gtk_entry_get_text(GTK_ENTRY(
                                        lookup_widget(frm_main,
                                        "entry_bill_to_st"))), "', ",
                    "bill_to_zip = '", gtk_entry_get_text(GTK_ENTRY(
                                        lookup_widget(frm_main,
                                        "entry_bill_to_zip"))), "', ",
                    "contact_first = '", gtk_entry_get_text(GTK_ENTRY(
                                        lookup_widget(frm_main,
                                        "entry_first"))), "', ",
                    "contact_last = '", gtk_entry_get_text(GTK_ENTRY(
                                        lookup_widget(frm_main,
                                        "entry_last"))), "', ",
                    "phone = '", gtk_entry_get_text(GTK_ENTRY(
                                        lookup_widget(frm_main,
                                        "entry_phone"))), "', ",
                    "title = '", gtk_entry_get_text(GTK_ENTRY(
                                        lookup_widget(frm_main,
                                        "entry_title"))), "', ",
                    "comments = '", gtk_editable_get_chars(GTK_EDITABLE(
                                        lookup_widget(frm_main,
                                        "txt_customer_comments")),
                                        0, -1), "' ",
                    "where num = ", gtk_entry_get_text(GTK_ENTRY(cbo->entry)),
                    0L
                   );

/* Finally, check for the success or failure of the update statment. */

if (mysql_query (conx, sql) != 0)
   {
```

continues

Listing 7.14 **Continued**

```
        g_print("Failure to update customer record,"
                  " contact administrator...\n");
          gtk_statusbar_push(GTK_STATUSBAR(lookup_widget(frm_main,
                                                         "statusbar")), 1,
                  "Failure to update customer record, contact administrator.");
      }
   else
      {
       g_print("Customer record has been updated...\n");
         gtk_statusbar_push(GTK_STATUSBAR(lookup_widget(frm_main,
                           "statusbar")), 1,
                           "Customer record has been updated...");

      }

   g_print("sql is %s\n", sql);

}
```

create_new_customer() Function

At the end of `sesi_utils.c` is the `create_new_customer()` function. The little piece of magic it does is to insert a new record into the sesi database, inserting into the "name" field. Recall that only the "num" and "name" fields are set to NOT NULL. By inserting a new record with only a "name," the autonumber feature of tbl_customers sets the "num" field of the newly created record to max(num) +1. `create_new_customer()` (shown in Listing 7.15) takes advantage of this behavior by creating the record and then selecting the maximum number from the database, which should be the newly created record (if that operation succeeded). It then sets frm_main to show the newly created record. Note that the user can either select "New" from cbo_customer_number or type it in. Either way, cbo_customer_number changes to reflect the new customer number.

Listing 7.15 **The *create_new_customer()* Code from *sesi_utils.c***

```
void create_new_customer()
{
   /* This routine creates a new customer. It does this by inserting a
    * record where only the "num" and "name" fields are entered. The
    * autonumbering feature of MySQL automatically creates a new
    * customer number in the "num" field. The routine then returns the
    * maximum "num" from tbl_customers, which has to be the customer
    * just created.
    * With this customer number, frm_main is set to show the newly
    * created customer, which will have blank text and entry boxes.
    * The user then needs to enter the customer information, such as
```

```
 * phone, address, and so on.
 */

gchar       *sql;
MYSQL_RES   *result_set;
MYSQL_ROW   db_row;
get_conx();

sql = "insert into tbl_customers (name) values ('new customer')";

/* Send the query against the database. */

if (mysql_query (conx, sql) != 0)
   {
     g_print("Failure to create new customer record...\n");
       return;
   }
else
   {
       g_print("New customer record created...\n");
       gtk_statusbar_push(GTK_STATUSBAR(lookup_widget(frm_main,
                          "statusbar")), 1,
                          "New customer record has been created...");

   }

/* Refresh the combo box of customer numbers. */

gtk_combo_set_popdown_strings(GTK_COMBO(lookup_widget(frm_main,
                                        "cbo_customer_number")),
                              get_customer_numbers());

/* Get the max(num) from tbl_customers, which should be the record
 * just created.
 */

sql = "select max(num) from tbl_customers";

if (mysql_query (conx, sql) != 0)
   {
     g_print("Failure to retrieve newly created customer record...\n");
       return;
   }
else
   {
       g_print("New customer record created...\n");

       result_set = mysql_store_result (conx);
       db_row = mysql_fetch_row (result_set);
```

continues

Listing 7.15 **Continued**

```
            /* Next set cbo_customer_number to the just-retrieved
             * customer number, which then displays all the fields
             * for that customer. In this case, those text and
             * entry boxes will be blank, all except for "name."
             */

            select_customer(db_row[0]);

            gtk_statusbar_push(GTK_STATUSBAR(lookup_widget(frm_main,
                               "statusbar")), 1,
                               "New customer record has been "
                               "created and retrieved, ready for edit...");

        }
}
```

Finishing *sesi_utils.c*

To finish up with `sesi_utils.c`, you'll start where you began. Listing 7.16 is the header section of `sesi_utils.c` after the code is finished.

Listing 7.16 **Final Header Configuration for *sesi_utils.c***

```
#include <mysql.h>
#include <gtk/gtk.h>

/* stdlib.h is needed for atof function.
 * string.h is needed for the string search functions used
 *          when formatting numbers for output.
 * time.h   is needed for the write-to-file operation.
 * stdio.h  is also needed for the write-to-file operation.
 */

#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <stdio.h>

#include "support.h"

/********** Global Variables *****************/

GtkWidget *frm_main;
GtkWidget *frm_items_ordered;
GtkWidget *frm_find_item;
GtkWidget *frm_find_customer;
```

```
/* conx is the connection to the database; it is global,
 * and all functions in the application can use it to
 * access database "sesi".
 */

MYSQL      *conx;

/* dbl_order_total is a global variable that
 * can be calculated and accessed without
 * converting to and from gchar/gfloat/gdouble
 * and so on.
 */

gdouble    dbl_order_total = 0;

/* The variable need_to_save_edits tells whether
 * changes have been made to any of the text
 * widgets that display database fields from
 * tbl_customers. Because it needs to be checked
 * from many different places and at different times,
 * making it global saves time and effort.
 */

gboolean   need_to_save_edits;

/********** Function Prototypes **************/

void    connect_to_db();
void    fatal_msgbox(gchar *msg);
void    get_conx();
GList   *get_customer_numbers();
void    fill_customer_info();
void    clear_frm_main();
void    fill_frm_main(MYSQL_ROW in_row);
void    speed_add();
void    slow_add();
void    remove_ordered_item();
void    select_item(gchar *target_item_num);
void    enter_found_items(gchar *str);
void    select_customer(gchar *target_customer_num);
void    enter_found_customers(gchar *str);
void    write_order();
gchar   *right_pad(gchar *in_str, gchar *pad_char, gint final_length);
gchar   *left_pad(gchar *in_str, gchar *pad_char, gint final_length);
void    create_new_customer();
```

Connecting the Interface to the Utility Functions

The previous two sections concentrated on building the user interface and constructing the functions to do the majority of the work. Now put the two together to produce a final product.

callbacks.c

Listing 7.17 lists all the callbacks from `callbacks.c` that have any code in them other than what Glade produces. (If you download `callbacks.c` for this project from the companion Web site, you will see more functions than are in Listing 7.17.)

Listing 7.17 **Selected Functions from *callbacks.c***

```
#ifdef HAVE_CONFIG_H
#  include <config.h>
#endif

#include <gtk/gtk.h>
#include <mysql.h>

#include "callbacks.h"
#include "interface.h"
#include "support.h"
#include "sesi_utils.h"

GtkWidget *frm_main;
GtkWidget *frm_items_ordered;
GtkWidget *frm_find_item;
GtkWidget *frm_find_customer;

MYSQL     *conx;
gboolean  need_to_save_edits;

gboolean
on_frm_main_delete_event               (GtkWidget       *widget,
                                        GdkEvent        *event,
                                        gpointer         user_data)
{
   g_print("on_frm_main_delete_event...\n");
   gtk_main_quit();

   /* The "return FALSE;" call below interrupts the
    * delete event. Change it to true to "not delete"
    * the form. In this case, terminating the application is
    * the desired behavior. So it is left as FALSE (the
    * default by Glade).
    */
```

```
   return FALSE;
}


void
on_frm_main_realize                    (GtkWidget       *widget,
                                        gpointer         user_data)
{
   g_print("on_frm_main_realize event...\n");
   connect_to_db();

}


void
on_cmd_search_clicked                  (GtkButton       *button,
                                        gpointer         user_data)
{
   g_print("on_cmd_search_clicked event...\n");
   gtk_widget_show_all (frm_find_customer);

}


void
on_cmd_save_edits_clicked              (GtkButton       *button,
                                        gpointer         user_data)
{
   g_print("on_cmd_save_edits_clicked event...\n");
   update_database();

   need_to_save_edits = FALSE;
   gtk_widget_set_sensitive(lookup_widget(frm_main, "cmd_save_edits"), FALSE);
}


void
on_cmd_select_items_clicked            (GtkButton       *button,
                                        gpointer         user_data)
{
   g_print("on_cmd_select_items_clicked event...\n");
   gtk_widget_show_all(frm_items_ordered);

}


void
on_cmd_print_order_clicked             (GtkButton       *button,
                                        gpointer         user_data)
{
```

continues

Listing 7.17 **Continued**

```
   g_print("on_cmd_print_order_clicked event...\n");
   write_order();

}


void
on_cmd_exit_clicked                    (GtkButton       *button,
                                        gpointer         user_data)
{
   g_print("on_cmd_exit_clicked event...\n");
   gtk_widget_destroy(frm_main);
   gtk_main_quit();
}


void
on_combo_entry_customer_number_changed (GtkEditable     *editable,
                                        gpointer         user_data)
{
   g_print("on_combo_entry_customer_number_changed event...\n");
   fill_customer_info();

}


gboolean
on_frm_items_ordered_delete_event      (GtkWidget       *widget,
                                        GdkEvent        *event,
                                        gpointer         user_data)
{
   g_print("on_frm_items_ordered_delete_event...\n");

   /* If the user clicks the "X" (close window) button in the top
    * right of the window, GTK+ will proceed to delete the window.
    * This is not the desired behavior; it is preferred that the
    * user click the "Done" button, but since that cannot be
    * guaranteed, the software should still react correctly regardless
    * of which way the user attempts to close the window. Therefore,
    * the "return TRUE" call below halts the delete event for the
    * window. The next time the user opens the form, it will be
    * in the same state as it was when it was closed, which is
    * acceptable.
    *
    * Instead, hide the form first, then return the TRUE ("halt").
    */

   gtk_widget_hide(frm_items_ordered);
   return TRUE;
}
```

```
void
on_frm_items_ordered_realize           (GtkWidget       *widget,
                                        gpointer         user_data)
{
   g_print("on_frm_items_ordered_realize event...\n");
   fill_items_ordered();

}


void
on_frm_items_ordered_show              (GtkWidget       *widget,
                                        gpointer         user_data)
{
   g_print("on_frm_items_ordered_show event...\n");

   /* Here is a bit of an afterthought – the numeric
    * columns in the CList widgets should be right
    * justified. This can, of course, be set from Glade;
    * however, that portion of this project is considered
    * stable, and it shouldn't be messed with "after the
    * fact." Therefore, the justification can be set here,
    * and incorporated into the Glade project file for
    * the next version.
    */

    gtk_clist_set_column_justification(GTK_CLIST(
                                    lookup_widget(frm_items_ordered,
                                    "clist_items")),
                                     2, GTK_JUSTIFY_RIGHT);
    gtk_clist_set_column_justification(GTK_CLIST(
                                     lookup_widget(frm_items_ordered,
                                     "clist_items_ordered")),
                                     2, GTK_JUSTIFY_RIGHT);
    gtk_clist_set_column_justification(GTK_CLIST(
                                     lookup_widget(frm_items_ordered,
                                     "clist_items_ordered")),
                                     3, GTK_JUSTIFY_RIGHT);
}


void
on_cmd_Add_clicked                     (GtkButton       *button,
                                        gpointer         user_data)
{
   g_print("on_cmd_Add_clicked event...\n");
   speed_add();

}
```

continues

Listing 7.17 **Continued**

```
void
on_cmd_search_for_item_clicked         (GtkButton       *button,
                                        gpointer         user_data)
{
   g_print("on_cmd_search_for_item_clicked event...\n");
   gtk_widget_show_all (frm_find_item);
}


void
on_cmd_done_clicked                    (GtkButton       *button,
                                        gpointer         user_data)
{
   g_print("on_cmd_done_clicked event...\n");
   gtk_widget_hide (frm_items_ordered);
}


void
on_cmd_add_down_clicked                (GtkButton       *button,
                                        gpointer         user_data)
{
   g_print("on_cmd_add_down_clicked event...\n");
   slow_add();

}


void
on_cmd_remove_clicked                  (GtkButton       *button,
                                        gpointer         user_data)
{
   g_print("on_cmd_remove_clicked event...\n");
   remove_ordered_item();

}


void
on_clist_items_select_row              (GtkCList        *clist,
                                        gint             row,
                                        gint             column,
                                        GdkEvent        *event,
                                        gpointer         user_data)
{
   g_print("on_clist_items_select_row event...\n");
   g_print("row is %i...\n", row);

}
```

```
void
on_clist_items_ordered_select_row      (GtkCList        *clist,
                                        gint             row,
                                        gint             column,
                                        GdkEvent        *event,
                                        gpointer         user_data)
{
   g_print("on_clist_items_ordered_select_row event...\n");
   g_print("Row to remove is %i\n", row);
}


void
on_frm_main_show                       (GtkWidget       *widget,
                                        gpointer         user_data)
{
   g_print("on_frm_main_show event...\n");
   gtk_combo_set_popdown_strings(GTK_COMBO
                     (lookup_widget(frm_main, "cbo_customer_number")),
                     get_customer_numbers()
                     );

}


gboolean
on_frm_find_customer_delete_event      (GtkWidget       *widget,
                                        GdkEvent        *event,
                                        gpointer         user_data)
{
   g_print("on_frm_find_customer_delete_event...\n");

   /* Returning true halts the delete event. */

   gtk_widget_hide(frm_find_customer);
   return TRUE;
}


void
on_cmd_find_customer_clicked           (GtkButton       *button,
                                        gpointer         user_data)
{
   g_print("on_cmd_find_customer_clicked event...\n");
   enter_found_customers((gchar *) gtk_editable_get_chars(GTK_EDITABLE(
                     lookup_widget(frm_find_customer, "entry_find_customer")),
                     0, -1));
}
```

continues

Listing 7.17 **Continued**

```
void
on_cmd_find_customer_done_clicked      (GtkButton       *button,
                                        gpointer         user_data)
{
   gchar      *target_customer;
   GtkCList   *clist_target;
   gint       row_target;

   g_print("on_cmd_find_customer_done_clicked event...\n");

   /* Get the customer number of the selected row in
    * clist_found_customer, and send it to select_customer().
    */

   clist_target = GTK_CLIST(lookup_widget(frm_find_customer,
                            "clist_found_customer"));
   row_target = (gint) g_list_nth_data ( (clist_target)->selection, 0);
   gtk_clist_get_text(clist_target,
                      row_target, 0, &target_customer);

   g_print("Target customer is:  %s\n", target_customer);

   select_customer(target_customer);

   /* Hide the form. */

   gtk_widget_hide (frm_find_customer);
}


gboolean
on_frm_find_item_delete_event          (GtkWidget       *widget,
                                        GdkEvent        *event,
                                        gpointer         user_data)
{
   g_print("on_frm_find_item_delete_event...\n");

   /* Returning true halts the delete event. */

   gtk_widget_hide(frm_find_item);
   return TRUE;
}


void
on_cmd_find_item_clicked               (GtkButton       *button,
                                        gpointer         user_data)
{
   g_print("on_cmd_find_item_clicked event...\n");
```

```
   enter_found_items((gchar *) gtk_editable_get_chars(GTK_EDITABLE(
                     lookup_widget(frm_find_item, "entry_find_item")),
                     0, -1));
}


void
on_cmd_find_item_done_clicked      (GtkButton       *button,
                                    gpointer         user_data)
{
   gchar      *target_item;
   GtkCList   *clist_target;
   gint       row_target;

   g_print("on_cmd_find_item_done_clicked event...\n");

   /* Get the item number of the selected row in clist_items_found,
    * and send it to select_item().
    */

   clist_target = GTK_CLIST(lookup_widget(frm_find_item, "clist_found_items"));
   row_target = (gint) g_list_nth_data ( (clist_target)->selection, 0);
   gtk_clist_get_text(clist_target,
                     row_target, 0, &target_item);

   select_item(target_item);

   /* Hide the form. */

   gtk_widget_hide (frm_find_item);
}


void
on_entry_customer_name_changed         (GtkEditable     *editable,
                                        gpointer         user_data)
{
   g_print("on_entry_customer_name_changed event...\n");
   gtk_widget_set_sensitive(lookup_widget(frm_main, "cmd_save_edits"), TRUE);
   need_to_save_edits = TRUE;
}


void
on_entry_last_changed                  (GtkEditable     *editable,
                                        gpointer         user_data)
{
   g_print("on_entry_last_changed event...\n");
   gtk_widget_set_sensitive(lookup_widget(frm_main, "cmd_save_edits"), TRUE);
   need_to_save_edits = TRUE;
}
```

continues

Listing 7.17 **Continued**

```
void
on_entry_first_changed                 (GtkEditable     *editable,
                                        gpointer         user_data)
{
   g_print("on_entry_first_changed event...\n");
   gtk_widget_set_sensitive(lookup_widget(frm_main, "cmd_save_edits"), TRUE);
   need_to_save_edits = TRUE;
}


void
on_entry_title_changed                 (GtkEditable     *editable,
                                        gpointer         user_data)
{
   g_print("on_entry_title_changed event...\n");
   gtk_widget_set_sensitive(lookup_widget(frm_main, "cmd_save_edits"), TRUE);
   need_to_save_edits = TRUE;
}


void
on_entry_phone_changed                 (GtkEditable     *editable,
                                        gpointer         user_data)
{
   g_print("on_entry_phone_changed event...\n");
   gtk_widget_set_sensitive(lookup_widget(frm_main, "cmd_save_edits"), TRUE);
   need_to_save_edits = TRUE;
}


void
on_entry_ship_to_addr1_changed         (GtkEditable     *editable,
                                        gpointer         user_data)
{
   g_print("on_entry_ship_to_addr1_changed event...\n");
   gtk_widget_set_sensitive(lookup_widget(frm_main, "cmd_save_edits"), TRUE);
   need_to_save_edits = TRUE;
}


void
on_entry_ship_to_addr2_changed         (GtkEditable     *editable,
                                        gpointer         user_data)
{
   g_print("on_entry_ship_to_addr2_changed event...\n");
   gtk_widget_set_sensitive(lookup_widget(frm_main, "cmd_save_edits"), TRUE);
   need_to_save_edits = TRUE;
}
```

```
void
on_entry_bill_to_addr2_changed         (GtkEditable     *editable,
                                        gpointer         user_data)
{
   g_print("on_entry_bill_to_addr2_changed event...\n");
   gtk_widget_set_sensitive(lookup_widget(frm_main, "cmd_save_edits"), TRUE);
   need_to_save_edits = TRUE;
}


void
on_entry_ship_to_city_changed          (GtkEditable     *editable,
                                        gpointer         user_data)
{
   g_print("on_entry_ship_to_city_changed event...\n");
   gtk_widget_set_sensitive(lookup_widget(frm_main, "cmd_save_edits"), TRUE);
   need_to_save_edits = TRUE;
}


void
on_entry_ship_to_st_changed            (GtkEditable     *editable,
                                        gpointer         user_data)
{
   g_print("on_entry_ship_to_st_changed event...\n");
   gtk_widget_set_sensitive(lookup_widget(frm_main, "cmd_save_edits"), TRUE);
   need_to_save_edits = TRUE;
}


void
on_entry_ship_to_zip_changed           (GtkEditable     *editable,
                                        gpointer         user_data)
{
   g_print("on_entry_ship_to_zip_changed event...\n");
   gtk_widget_set_sensitive(lookup_widget(frm_main, "cmd_save_edits"), TRUE);
   need_to_save_edits = TRUE;
}


void
on_entry_bill_to_city_changed          (GtkEditable     *editable,
                                        gpointer         user_data)
{
   g_print("on_entry_bill_to_city_changed event...\n");
   gtk_widget_set_sensitive(lookup_widget(frm_main, "cmd_save_edits"), TRUE);
   need_to_save_edits = TRUE;
}
```

continues

Listing 7.17 **Continued**

```
void
on_entry_bill_to_st_changed            (GtkEditable     *editable,
                                        gpointer         user_data)
{
   g_print("on_entry_bill_to_st_changed event...\n");
   gtk_widget_set_sensitive(lookup_widget(frm_main, "cmd_save_edits"), TRUE);
   need_to_save_edits = TRUE;
}


void
on_entry_bill_to_zip_changed           (GtkEditable     *editable,
                                        gpointer         user_data)
{
   g_print("on_entry_bill_to_zip_changed event...\n");
   gtk_widget_set_sensitive(lookup_widget(frm_main, "cmd_save_edits"), TRUE);
   need_to_save_edits = TRUE;
}


void
on_txt_customer_comments_changed       (GtkEditable     *editable,
                                        gpointer         user_data)
{
   g_print("on_txt_customer_comments_changed event...\n");
   gtk_widget_set_sensitive(lookup_widget(frm_main, "cmd_save_edits"), TRUE);
   need_to_save_edits = TRUE;
}
```

main.c

In closing, Listing 7.18 is `main.c` from this application. Once Glade has written this file, it will not overwrite it. That means you can change it, but if you need Glade to create a new one, you will have to delete or rename the existing one.

Listing 7.18 ***main.c* For the SESI Order Application**

```
/*
 * Initial main.c file generated by Glade. Edit as required.
 * Glade will not overwrite this file.
 */

#ifdef HAVE_CONFIG_H
#  include <config.h>
#endif

#include <gtk/gtk.h>
#include <mysql.h>
```

```
#include "interface.h"
#include "support.h"

GtkWidget *frm_main;
GtkWidget *frm_items_ordered;
GtkWidget *frm_find_item;
GtkWidget *frm_find_customer;

MYSQL *conx;

int
main (int argc, char *argv[])
{
  gtk_set_locale ();
  gtk_init (&argc, &argv);

  /*
   * The following code was added by Glade to create one of each
   * component (except popup menus), just so that you see something
   * after building the project. Delete any components you
   * don't want shown initially.
   */
  frm_main = create_frm_main ();
  gtk_widget_show (frm_main);

  frm_items_ordered = create_frm_items_ordered();

  frm_find_item = create_frm_find_item();

  frm_find_customer = create_frm_find_customer();

  gtk_main ();
  return 0;
}
```

Notice that all four of the windows are created, but only frm_main is shown at this point. The others are shown and hidden as needed.

Compiling the Program

Listing 7.19 is the file used to compile this program. If you get it from the companion Web site, it will be called build.sh. To compile, at the command line send

```
% ./build.sh
```

Listing 7.19 File *build.sh* contents: The Commands Used to Compile the SESI Order Application

```
01  clear
02
03  gcc -Wall -g -o sesi_order.exe callbacks.c interface.c \
04              support.c   main.c  sesi_utils.c \
05      `gtk-config --cflags --libs` \
06      -I/usr/include/mysql \
07      -L/usr/lib/mysql -lmysqlclient -lm -lz
```

Line 1 simply clears the screen; this makes it more readable if any compile errors occur. Lines 3 and 4 are the compile commands and the target files; the backslash character (\) is the line-continuation character.

Line 5 sets the GTK+ flags and libraries. (Don't forget that those are back-tick marks, not single quotation marks.) Line 6 "includes" the MySQL library, whereas line 7 "links" the MySQL, math, and zlib libraries.

Project Post-Mortem

Every project that was worth doing should have a post-project debrief. What went wrong? What could have been done better? While the ideas are still fresh, you have to ask yourself what things should go into the next version.

The application probably could have made all changes be automatically saved (to the customer record, frm_main) and the cmd_save_edits button could have been removed altogether. The times when edits won't be saved will be very rare. So such a change probably wouldn't impact usefulness and would save more keystrokes than it created. Instead, simply run the update database procedure every time frm_main is moved off the current record.

Regarding the changes made to frm_items_ordered, simple is best. If a set of widgets can be put into a vertical packing box as opposed to a vertical packing box with a number of child horizontal packing boxes, that is simpler and better to implement, assuming it doesn't affect usability.

Check the resize action of the window widgets early in the development cycle—as soon as possible after all the child widgets have been filled in—to see that the form is resizing correctly. As an example, see Figure 7.10. Notice the wasted space when this form is resized. Even though this form will rarely be used (due to the way it is used and its short life cycle), it would have been nice for it to look correct even when maximized.

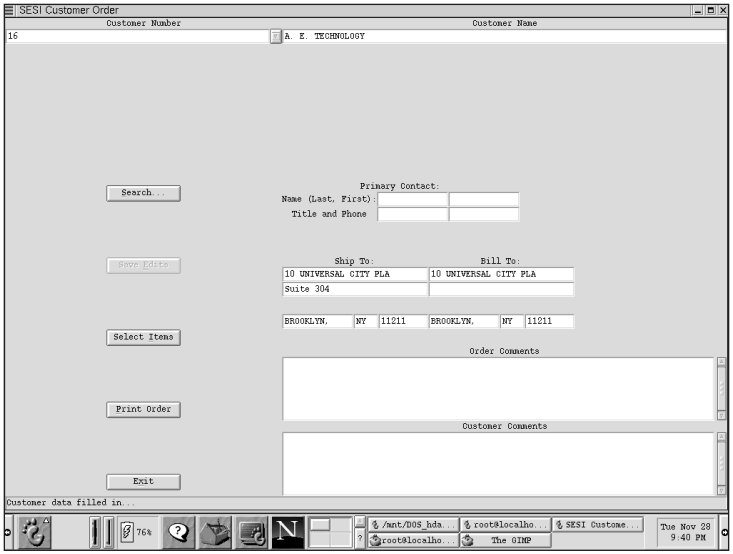

Figure 7.10 frm_main maximized; clearly this is not the desired result!

Figure 7.10 shows frm_main in its maximized state. Fortunately, there is no really good reason for the user to do this while using the application! Clearly, more time could have been taken to make sure that the resize action was a bit more, um, palatable. Compare Figure 7.10, however, with Figure 7.11.

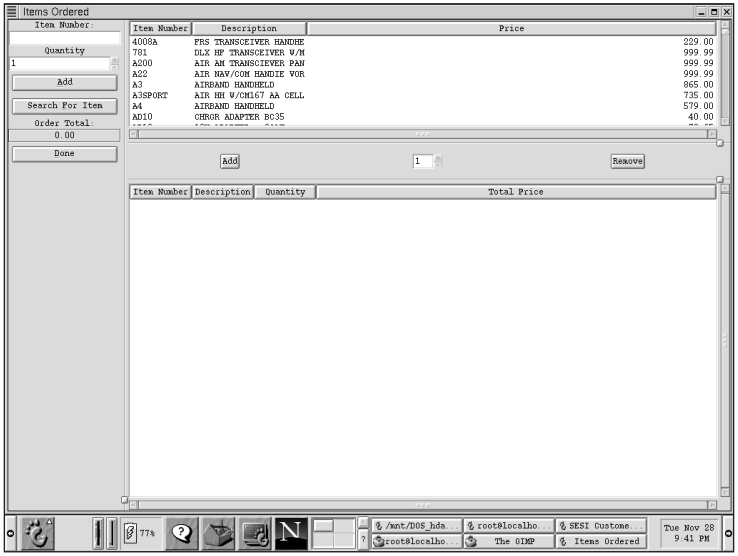

Figure 7.11 frm_items_ordered maximized; now that's more like it!

Figure 7.11 shows frm_items_ordered maximized, and although it was never intended to *be* maximized, it doesn't look too bad. The primary difference between them is that frm_main used all vertical and horizontal packing boxes, while frm_items_ordered used the horizontal and vertical paned widgets.

The add-quantity box between the CList boxes in frm_items_ordered probably could have been done away with. In fact, so could the other quantity spinbutton in the upper-left corner. Instead, the user could have just entered the same item twice to clist_items_ordered, and the result would have been the same. This only makes sense if it is far more normal to order only one of something; if quantities of two or more occur (for example, 40% of the time or more), then perhaps the way it was done is best. This is something that would need study in the actual environment in which it was being used.

A double-click event on the rows of the clist_widgets would have been useful; for example, to double-click clist_items and to have it run the same code as `on_cmd_add_down_clicked()` would have been an easy and functional addition using already existing code. Unfortunately, there is no double-click event in the GTK+ event model.

The desired functionality of being able to do nearly everything—at least the common things, anyway—all from the keyboard was achieved. Someone who takes time to learn the keystrokes and the various key shortcuts (like Shift-Tab to go back one widget in Tab order) will be able to enter orders extremely quickly.