Fitrix™

Technical Reference

Edition 0390-120117

Fourth Generation Software Solutions
2814 Spring Road, Suite 300
Atlanta, GA 30339

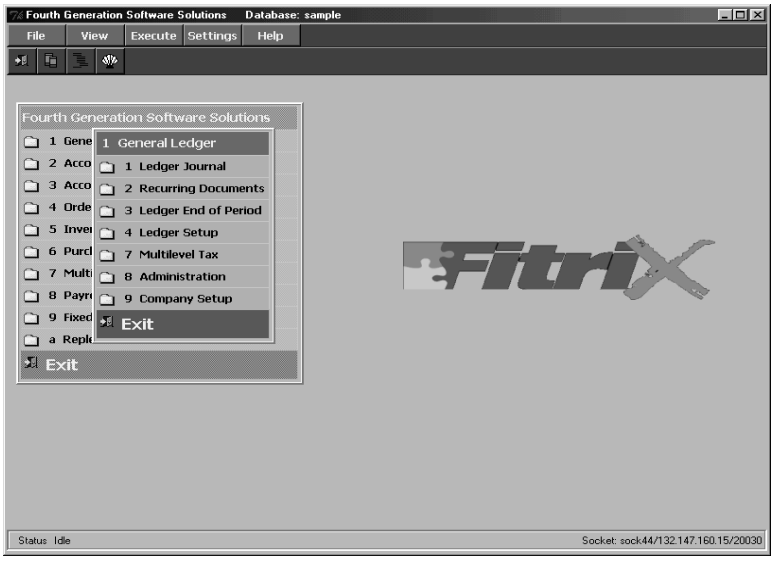
Corporate: (770) 432-7623
Fax: (770) 432-3448
E-mail: info@fitrix.com

Copyright

Copyright (c) 1988-2002 - Fourth Generation Software Solutions Corporation - All rights reserved.

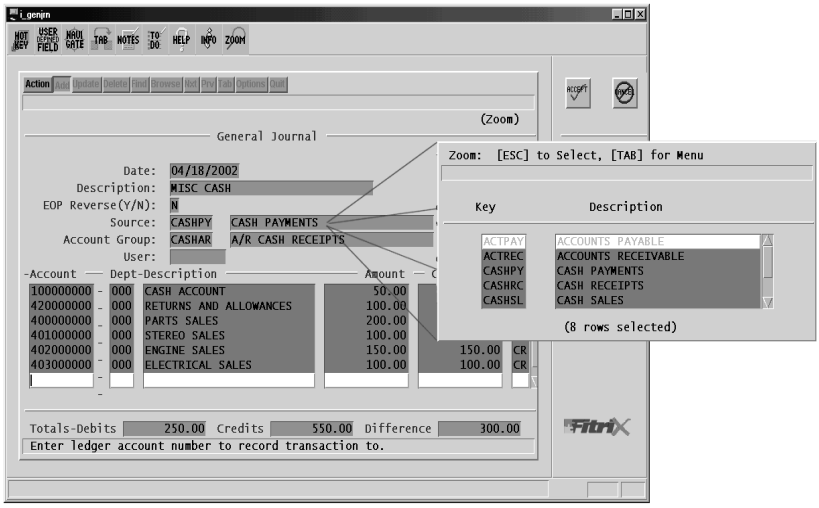
No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system or translated.

Welcome to the Fitrix Technical Reference. This manual is designed to be a focused step-by-step guide. We hope that you find all of this information clear and useful.
Although the pictures in this manual are all of character based screens, please keep in mind that all of our products offer the option of being viewed in a graphic based Windows screen. Examples of graphic based product viewing modes are shown below in Example 1 and Example 2.



Example 1: Menu Graphical Windows Mode

Here is another example:



Example 2: Data Entry Graphical Windows Mode

Displaying our products in graphic mode, as shown in Example 1 and Example 2, is customary for many Fitrix product users.
However, your viewing mode is a user preference. Changing from character based to graphical based is a product specific procedure, so if you wish to view some applications in character mode, and some in graphical mode, that can be done as well.
If you have any questions about how to view your products in graphical mode, please consult your Installation Instructions or contact the Fitrix helpdesk at 1(800)374-6157. You can also contact us by email: support@fitrix.com. Please be prepared to offer your name, your company, telephone number, the product you are using, and your exact question.
We hope you enjoy using our products and look forward to serving you in the future.

Thank You,
Fourth Generation

Table of Contents

Chapter 1: Introduction

Chapter 2: Code Structure

Chapter 3: Interacting with the Database

Chapter 4: Program Design and Modification

Chapter 5: System Administration Issues

Chapter 6: Programming Conventions

1

Introduction

About this Manual

The purpose of this documentation is to give programmers a basic understanding of the design and programming conventions used in building Fitrix Accounting.

This manual covers what we have done to make the Fitrix source code easy to modify and why we have done it. It explains certain aspects of our philosophy of application design that are important to those modifying our applications and constructing compatible applications.

This manual is written for programmers that have some familiarity with building applications in INFORMIX-4GL, the language in which Fitrix Accounting is written. This manual will help you understand the basic flow of the programs well enough to make the most common types of modifications.

This manual does not attempt to teach the Informix language. If you need to understand the working of a particular INFORMIX-4GL command, refer to the INFORMIX-4GL manuals.

This manual also does not attempt to document specific code. source code has the documentation in the code.

The *Fitrix Technical Reference Manual* consists of the following sections:

1. Section 1: Introduction - covers the design of the documentation as well as the underlying philosophy of code design.

2. Section2: Code Structure - offers basic orientation information about the source code. This section covers the ways in which we have broken down the source code and the physical organization of the code on the disk. The application directory structure has been organized to facilitate the modification of code.

3. Section 3: Database Interaction - discusses the data structure of Fitrix Accounting. It looks at the management of the financial database. This section also covers the philosophy of database design, how to interface custom modules to the database design, and what information is stored in the database.

4. Section 4: Program Design and Modification - covers the basic program design used to create the INFORMIX-4GL programs provided with each Fitrix Accounting module. The use of function libraries is discussed in detail as well as different development environments.

5. Section 5: Report Program Structure - contains a style guide for developers working with Fitrix Accounting. The style guide covers conventions and rules developed for creating source code.

6. Section 6: System Administration Issues - contains information about termcaps and setting up printers.

7. Section 7: Programming Conventions - discusses the programming conventions that Fitrix Accounting applications follow. File and program naming conventions are covered as well as rules for formatting input and output.

8. Appendix A: Problems with [CTRL] - [Y] and [CTRL] - [Z] Keys - covers problems that can occur with some keyboards.

9. Appendix B: Troubleshooting Fitrix Business Products - contains a trouble-shooting checklist that can help solve problems encountered when running Fitrix Accounting.

Modifiability by Design

There are two kinds of source code: source code that is a by-product of application development and source code that is developed as a product. The differences are substantial. In the former, the application may have the required features, but the code was not designed to be sold and maintained by anyone other than the developer. In the latter, the code was designed as a product with features beyond the features the application creates. The sum of these features determines how easy the code is to work with, change, and maintain.

Fitrix Accounting source code was designed so that it would be easy for others to modify. Before we put together the source code, we developed a consistent design and a number of conventions for supporting modifiability. To enforce that consistency, a set of "CASE" (Computer Assisted Software Engineering) tools were developed that would automatically generate the basic program for each function we wanted to perform. All of the programs in Fitrix Accounting are developed using these CASE tools so all resulting programs follow exactly the same structure. This concept is called "Modifiability by Design."

What Can Make Tailoring Source Code Difficult

Before a problem can be solved, it must be understood. Those that have modified code know that it has not been as easy as it should be. Why? What modifications are the most difficult and what characteristics make them more difficult?

Before Fitrix Accounting was written, we analyzed the problem of making code easier to modify. We looked at typical accounting transaction-processing applications and we identified common types of modifications.

If these modifications are grouped by type, we see that they form a hierarchy. Some modifications are more difficult than others. Some changes are simplified by using the Fitrix language used to develop Fitrix Accounting. In general, however, this order of difficulty is consistent no matter what particular development environment you work in.

From this list, the reader can learn about the process of tailoring code and the time required for modification. There are common problems inherent in the process of modifying source code. By understanding these underlying problems, it is easier to identify which requested modifications are more expensive because they take more time.

Modification Types by Difficulty

Below, we rank the easiest types of modifications to the most difficult. This is to help us see what needs to be done at the source code level to make the most difficult of these common modifications more practical and less time consuming.

1. Rearrangement of Screens and Reports

This is typically the simplest type of change to any program. It does not involve the addition of any new data or program logic. The change here is local, meaning that it only affects one part of the code. The major code design considerations here revolve around naming and documenting the proper program sections, variables, or data elements involved. Proper design helps when you are trying to identify particular elements within the code.

2. Addition of Calculated Fields to Screens and Reports

The major new code consideration involves data flow, or identifying the proper point within the program at which the calculation should be placed.

3. Change in Local Program Logic or Functionality

Since the change is only in the local program, the only new consideration is the logical structure of the program. This typically involves more of the code than the data flow. In changing program logic, it is necessary to identify the tree structure used and the proper place within the structure to make the change.

4. Addition of New Fields to Single-Reference Databases

By a "single-reference" database, we mean a data storage structure (it may or may not be an actual database product) that is used by only one program. A change to this data structure can affect several parts of the program and can, theoretically, require a change to every line. The purpose of good code design is to allow pro-

grammers to make certain where change is required, and to minimize the number of changes that must be made. The addition of fields, however, is not usually as complicated as the change of an existing field, which is a higher order of modification.

5. Change in Non-Local Program Logic or Functionality

This order requires changes throughout an application, not just to a single program. Behavior of different programs within the application must conform to each other in many respects. A change in particular forms of logic and functionality must be made consistent throughout the application in order to support this conformity. Such issues can involve anything from consistency in user interface, to the correct processing of data. Code design should enable you to identify related parts of different programs and, when possible, allow sharing of sections of code by different programs.

6. Addition of New Fields to Multiple-Reference Databases

Multiple-reference databases are data structures that are referenced in several places or programs within an application. The problems with maintenance here are similar to order four problems, but with the added complexity of having to find affected sections of code in many different programs. It is at this level of change that the importance of a data dictionary comes into play.

7. Change in Field Size/Type in Single-Reference Databases

Changes to existing fields are even more complicated than the addition of new fields. The use of a data dictionary helps, but to a lesser degree than with the previous level of change. The most serious aspect of this problem is identifying all of the variables into which data from the field may be placed during program execution. This involves identifying not only the data flow, but each element used by the affected field in that data flow.

8. Change in Field Size/Type in Multiple-Reference Databases

The order eight problem is complicated by the need to check more lines of code and to interrelate sections of code from different programs.

9. Change of Key Field in Single-Reference Databases

Key fields are the fields used to recall data from the database. Changes to a key field involve not only changes of the seventh order, but additional changes related to data entry and building the key. Most significantly, it involves an actual change in the database itself, requiring minimal rebuilding of the indexes that maintain the

database. The difficulty of rebuilding indexes and restructuring the data in the database depends largely on the development environment in which you are working. In the worst case, it may involve having to write one-time use maintenance programs for unloading and loading all data.

10. Change of Key Field in Multiple-Reference Databases

Level ten problems affect a larger amount of code. Generally, key fields appear most frequently in data entry screens and reports. They are utilized in every access of data. Changes to key fields in databases that are used in many different programs tend to be the most complicated, because they affect many lines of code as well as the database itself.

Three Time-Consuming Tasks Modifying Code

In analyzing the problems that seem to be inherent in modification, certain common types of work emerge. These are the physical tasks that the programmer must do to make a change at any level. Typically, what makes one level of change more difficult than another is that these general tasks grow in complexity as you move up the hierarchy. These tasks are the specific ones that we have tried to simplify in the designing and building of our source.

These tasks are:

1. Finding the Right Parts of the Code to Modify

Before you can make any change, you must identify the affected code. The complexity of this task depends largely on program design. In the worst case, a lack of structure and naming conventions makes it virtually impossible to find the sections of code that need changing. The only way to discover parts of the program affected by the code is to continually test not only the application, but the integrity of the database itself. Even serious problems may not be serious to the user. As you will see, Fitrix Accounting includes a number of techniques used to simplify this task.

2. Making Changes in Multiple Locations.

The more sections of code that must be modified, the more difficult the change. This is what makes the highest order of changes so complicated. Such a change can affect virtually every part of the code. In Fitrix Accounting, everything possible has been done to minimize the number of places in the code that need to be modified to accommodate a change at the application level. If a certain category of change still requires modifications in multiple locations in the code, we document those locations in this manual.

3. Coordination or Concurrence of Functionality Among Various Parts of the Code.

The developer must have a feel for what the code is doing on a functional level of data flow. To accomplish this, the code must be consistent, particularly in the way it handles data throughout the application. Programs must agree in very basic ways in how they handle the data in order for the applications to work. Sometimes identifying this coordination is very difficult because it can take on radically different forms in different parts of the program. As you will see, we have developed a number of resources, such as the "Info" files in the code itself, to help developers with this difficult task.

General Techniques for Creating Modifiable Code

A number of new techniques and concepts have been implemented in the code based on 4GL, SQL, and CASE technologies

1. Extensive Use Of Comments

Commenting your code is critical for all orders of modification. It is the most obvious and useful criteria that can be used in evaluating code. Comments are lines of English in the code that explain how that section of code works. The more extensive the comments, the easier it is for the modifier to understand the section of code. An understanding of the code is absolutely necessary for modification. Comments are especially important as part of the database schemas; that is, the parts of the programs that define the database structure.

Comments can also be used to create options at the code level or show examples of various ways in which the code can be modified to implement various features. Whole subroutines can be "commented out" so that they are not functional until the comment indicator is removed. Many optional features are best provided at this level, since neither the user nor developer have to deal with them until they are required.

Comments are measured as a percentage of source. As stated above, Fitrix Accounting is the most heavily commented code in the industry. This is possible because our CASE development tools have created the basic comments that explain each section of code. We have improved the quality of the automatically generated comments by manually adding text in the source code wherever we have made a large number of manual additions to the generated code. Currently, the ratio of comment lines to lines of code is 3.5 to 10.

2. Modularity

We use the term "modularity" to mean building larger programs from smaller building blocks. Modular design means creating small routines that are put together into code sections which are themselves combined into a complete program. Each of these small routines has a clearly defined function and that function is, to some degree, independent and self-sufficient. Routines are discrete pieces of code like the discrete components that are assembled into any machine. Breaking down sections of code into smaller independent subsections reflects a philosophical decision on design. The same end can be achieved by using a few large, very complex, integrated and specialized parts (the non-modular approach), or by constructing the whole from subsections which are themselves constructed of many smaller, more generalized components (the modular approach).

Modularity has many advantages. Modular code can be broken down into small, easy to understand sections. By isolating individual functions, more comprehensible structures are created. You can find the sections of code that need adjustment easily, and understand its function relative to the whole more clearly. You can also work on one section of code with less concern about changes that affect functions of the program.

The modularity of our programs was built in by our CASE technology. As you will see, each section of our source code is built of small functions. Our internal goal is to keep each function as small as possible. The maximum desired size was set at twenty lines so that you could see each function in its entirety on a single screen.

Because some functions required listing of data elements, this was not always possible. Yet in 95% of cases, the functions meet this basic size requirement. Some functions consist of a single line of code.

3. Routine Libraries

One major advantage in using a modular approach is that you can make repeated use of many of the small routines from which your code is built. By creating smaller independent functions, you build more general purpose code modules. Since they serve a more generalized purpose, these sections of code can be used not once but many times. For instance, the same section of code that formats data for display on the screen can be used in the program that prints that same data in a report. The sections of re-usable code constitute a library shared by other larger sections of code.

The use of routine libraries supports modifiability in two significant ways: First, it reduces the total number of lines of code. Modularity itself tends to produce more lines of code, but if done correctly, this approach increases the re-use of the code dramatically. The result is a major reduction in the number of lines of code.

Second, the extensive use of libraries allows you to make changes that affect many different programs by changing one section of code. The reason that so many higher order modifications are so difficult is that you have to make a similar change in many different programs. If those programs all utilize the same library routine for a given function, changing that one routine automatically affects all programs referencing it. This dramatically reduces the complexity of those higher order changes.

In Fitrix Accounting we use three different libraries of code for our applications. One library contains the general routines used by all data entry and reporting programs. This library is independent of the accounting application and consists of data independent user interface and reporting functions. The second library consists of basic accounting routines that are used by all accounting modules. Finally, there is a third library of routines that is accounting module-specific (accounting "modules" are General Ledger, Accounts Receivable, Accounts Payable, and so on). These routines may be called by the different programs within that specific module.

Later in the documentation we will discuss both the location and content of these various code libraries.

4. Consistent Organization of Source Code

Consistent organization means that programs are broken down in the same ways, and that they use the same basic pattern of data flow and the same basic control structures. It also means that the source code is physically organized in a consistent way. The orderly, hierarchical structure of good modular code is revealed by the breaking down of source code into different directories, files, and sections within files. The consistent organization of the code is reflected in the consistent organization of that directory structure.

How the code is organized is very important, especially for modular programs. Modular design means that there are more levels of organization within the code. If the code is physically well organized, modularity makes finding the proper section for modification easier. However, if physical organization of the code is done poorly, finding the right section can be extremely difficult.

If you are building something with only a few large parts, you do not have to worry about finding them when you need them. If you are building with many small parts, you have to take great care in organizing your storage system to be able to find what you need.

As you will see, once you have learned the code structure for one module and one program in Fitrix Accounting, you have basically learned it for all such programs—it is absolutely consistent. This dramatically reduces the time it takes to learn the code structure, both physical and otherwise, in order to work with it.

Later in the documentation, we deal with the "functional" flow of all Fitrix Accounting programs. As you will see, all data entry programs in Fitrix Accounting work in an almost identical manner.

5. Consistent Naming of All Program Elements

Closely related to consistency in structure is consistency in naming those structures. Directories are tagged to indicate the level and the role they play in overall organizational structure. At a lower level, when they serve identical purposes, they are named in an identical manner. When different code sections serve similar purposes, they also are named in a similar manner. Even at the lowest level, where routines play unique roles, name tags can indicate the type of role they play. Variables that in some way relate to the same data are named accordingly. Variables that belong to the same level of data flow are tagged to emphasize that similarity.

The advantages of standard naming conventions are obvious. Once learned, naming conventions allow the developer to know immediately what function a particular section of code plays. If you understand the rules by which variable names are composed, string searches can be conducted according to the naming conventions to find the specific variables without having to first identify them by following the program data flow.

After we discuss the file organization of our system, we immediately discuss our naming conventions. As you will see, once you learn a few simple patterns, you can understand what you are seeing on the screen almost at a glance.

6. Variables and the Data Dictionary

In a program, a large number of variables play a part in the data flow of the program. They handle information coming from or going to the storage database. Since most of the higher order modifications involve changes to the database structure, these variables play a critical role. Any change to the database must be reflected by corresponding changes in the size and/or type of variables used.

A basic method used to simplify modification is to tie the definition of all variables used for data flow to a central data dictionary. The data dictionary defines the database structure. In each of our programs, the definition of variables takes place in a specific section of code called, "`globals.4gl`", and the definition of all variables that are part of the data flow is done against the database.

If you want to change the size or type of a data element, it is only necessary to change the definition in the data dictionary using Informix SQL or 4GL and then recompile the code. You do not have to change the actual code itself.

7. Localization of Code

All the code (except library code) required to create a specific executable program resides in the same directory. Applications are made of many discrete executable programs. The directory structure that organizes the overall code does not divide separate code sections between different directories if that code is used to make a single executable. The only exception to this is library code, which resides in a common area used by all programs.

The advantage is that when you are working on a specific program, you do not have to move between directories to access the code that generates that program. This simplifies the job of finding and getting to the code that needs changing.

As you will see, utilizing the application itself moves you to the directories in which the source code resides. In other words, the fastest way to move through the organization of the program is to run the program. When you "escape" from the running program to the operating system, you are automatically in the directory that contains the source code for that particular executable. We call this the "automatic location" of code, and it must be built into the application as "localization" is built into the source.

8. Complete Source Availability

Finally, we have provided you with 100% of the source code used in Fitrix Accounting. In other words, there are no pre-compiled components of the application for which you cannot obtain the source code.

Many "source" applications only give you access to certain parts of the application. In other words, you can change one part of an application, because you have the source for it, but there are other parts of the application that are fixed, and cannot be changed. They are provided, even in a source level product, only in an executable form. This is typically done because that section of the application was written in some different language, usually a lower level language such as "C." For practical as well as proprietary reasons those sections of the code are not distributed.

This is not the case with Fitrix Accounting. You have access to all the source other than that for the application runtimes.

The CASE Code Generators

The INFORMIX-4GL environment allows programmers to produce applications more rapidly because it was designed expressly for creating database intensive transaction processing applications. However, there is another level of automation beyond simply using a higher level language. This is called CASE (Computer Assisted Software Engineering). The purpose of CASE technology is to provide an application module that can be used to actually generate the Fitrix language code necessary to produce that application. CASE code generation creates code even more quickly, easily, and efficiently.

Software development must go through three preliminary phases before it reaches the level of practical CASE. These phases constitute the methods under which the rules for creating applications in a Fitrix environment are developed.

The Start-Up Phase

The first such phase is called "Start-Up." This stage is programming from scratch. Programmers create programs and applications from all new code.

During this phase, modularity in design is the key ingredient. Developers must break down each task into subroutines and functions. They must use subroutines and functions to provide menus, do recalculations, and so on. The extensive use of subroutines produces code that is easier to support and allows programmers to graduate to more advanced phases of development.

At this stage, a consistent user interface and rules of application operation need to be established. The user interface is the way the program is viewed and controlled by the user. Developing a consistent, intuitive, and easy to understand interface is a high priority. Establishing these conventions within any package allows you to extend them to future packages. Finally, this first phase is when programming standards and conventions are developed. You need to determine when comments are required, how basic structures, such as if-then-else statements should be constructed, and what format will be used for variables. For example, those beginning with `p_` might be program variables, while those starting with `s_` might be screen variables. The tighter the definitions established during this first phase, the more consistent the products developed, and the easier they will be for future developers to work with and modify.

The Building Block Phase

The "Building Block" stage begins when you write code to perform operations that have been defined in previous applications. At this point, you copy parts of old programs and modify them so that they fit the new project. Existing subroutines and functions are used as building blocks to create new applications.

During this phase, the most important thing is to track which parts of existing code you re-use most frequently. In identifying which functions and subroutines are most useful, you will be able to copy and change them much more quickly. You learn during this phase the importance of keeping the data flow consistent, calling modules the same way, and organizing your code to make the best use of pre-existing modules easier. For example, in developing a report, the first step should always be creating the proper selection criteria; the next step is to get the correct cursor. The following step is to move the cursor data to the actual report generation function. Even when these steps are not completely appropriate to a given application, organizing code consistently allows you to take maximum advantage of existing routines.

The Library Phase

Once you have a good idea which functions are used the most frequently, you can convert those functions into a library routine. These routines are general functions that can be used over and over. They are organized in logical ways which make them easy to find and use.

Using the basic libraries of routines dramatically simplifies the process of creating applications. Once an organization reaches this third phase of code generation, coding becomes a clerical task more than a programming task. All the programmer needs to do is to copy his old program to a new work area, substitute the old variables with the new ones in mid-level and low-level routines, and most of the job is done.

CASE and Code Generation

Using basic CASE techniques, you begin to define the functional flow and data flow of your application programs. In separating functional flow from data flow, you are able to create functions that are data independent and do not rely on global variables. These "data independent" functions receive information from the calling module, manipulate that information, and return information back to the calling modules.

By diagramming the functional flow of these programs, a four-level approach begins to take shape. On the first level is Main. Main is used to begin your program, set initial variables, and call the upper-level routines that run the rest of your program (i.e. a menu function), and finally to end your program. The next level is this set of "upper-level" data independent routines. These functions are called from generic library routines. They don’t rely on global variables or handle data flow. A set of "mid-level" functions acts as a conduit between the data dependent and data independent functions. They prepare the data for manipulation, setting of arrays, fetching of cursors, clearing of variables, and so on. Finally, there is the "low-level" code which is where most of the actual data manipulation takes place.

The four level approach operates like this. Main calls the "menu" or command function like "add", "update", or "find." The upper-level routine fetches the functions needed to do this menu level from the library. The mid-level fetches appropriate data into a cursor. The low-level modifies the data in the cursor and writes data out to the table.

Once this model has been fully developed, you are ready for code generation. Code generation must take into account the rules by which the "mid-level" and "low-level" code is written. Since the programs generated are created according to a highly defined model, very little information, other than the specific data "objects" being manipulated, is needed in order to construct working code.

In using our CASE code generation tools, information is taken from the standard Perform screen, or report image. The data elements are identified and variables for the data flow constructed. Those variables are placed into a "skeleton" of mid-level and low-level routines automatically. When the information needed to generate the code is not part of the Perform screen, the user is prompted to supply it.

We have refined this process for the most common types of application development work—creating data entry environments and reports—to such a degree that we have turned them into commercial products. We offer these generation programs to our customers as CASE tools, Screen and Report.

It is not necessary to purchase these products to modify any of the code we provide, but it will make certain types of modifications much easier. This is especially true if you are adding completely new data entry screens or reports. In Fitrix Accounting, we "spoil" our users by giving them a very sophisticated user interface. If you tried to duplicate this interface manually, even using all of the library routines that are already part of this application, it would take you days to write each new data entry screen. That same effort can be reduced to less than five minutes using our code generator.

All of our basic conventions deal with code that has been written to exacting specifications. The only way to develop such code efficiently is to use these products.

2

Code Structure

Directory and File Organization

The first step towards understanding the design of Fitrix Accounting is to understand the organization of the menus and INFORMIX-4GL programs that make up the total application. This section of the documentation covers how all the system files are organized in the directory hierarchy. This organization may seem initially complicated, however, you will find that for the large number of files involved and the complexity of their relations, such a system is required for efficiency, portability, and simplicity of maintenance.

Installation of Fitrix Accounting

The installation of Fitrix Accounting will produce the directory structure outlined in the following pages. The installation and compile time will vary due to things like hardware variations, user demand of memory, and the order in which modules are installed.

Disk Space Requirements

The need for file storage is growing at a tremendous rate. The amount of space considered adequate five years ago is now not enough. Not only have operating system sizes increased as services have been added, but program sizes have increased as the number of features and the complexity of programs have grown. The introduction of RISC (Reduced Instruction Set Computer) based systems has also multiplied the storage requirements. All of these factors, combined with the increasing size of databases used for normal operation, have added to the requirements for hard disk or rotating memory.

The following paragraphs describe factors involved in space requirements for storing Fitrix Accounting products on a hard disk.

The first factor to consider when determining size requirements is the type of hardware to be used. The main difference between the RISC systems (e.g., the IBM RS/6000, Data General AViiON, Sun SparcStation, and the Motorola 88000 based system) and the CISC (Complex Instruction Set Computer) systems (e.g., those based on the Intel 80386 and 80486 or the Motorola 68020 and 68030) is speed of execution. The RISC machines can execute instructions faster than the older CISC designs. The problem this creates for hard disk storage (rotating memory) is that the number of instructions required to do the same task is generally larger. The practical result is that the average program is twice the size of those on CISC machines. A program of 1 Mb for a 386-based machine requires about 2 Mb for a RISC-based machine. An AIX UNIX operating system for the RS/6000, complete with the development system necessary for maintaining and modifying the Fitrix Accounting packages, may exceed 200 Mb. As a result, a small hard drive for a RISC system is any drive under 400 Mb.

An additional consideration for Fitrix Accounting software is the type of Informix implementation used on the system. The source code takes up a relatively constant amount of space. The chief distinction is whether the 4GL or the RDS version of

Informix (used to create and maintain the Fitrix Accounting software) is used as the primary system. The RDS system is known to be small, fairly quick, and broadly portable, but it is designed to be used for development rather than for operation. The 4GL system creates compiled executable code, which is quicker than the RDS code, in the *.4ge form. The 4GL code is 2 to 5 times the size of the *.4gi RDS code. The average is between 3 and 4 times, depending on the machine on which it is installed. The desirability of maintaining source code on the client's machine or handling maintenance on a separate system, to be installed later on the client's system, should also be considered. "RDS vs. 4GL: Choosing an Informix Environment" on page 4-44

Directory Organization

Working while running the application is fine, but many times you will want to work at the system level. In order to do that, you need to understand the organization of the files that make Fitrix Accounting work.

When you write your first 4GL application, you notice that the application development tool that comes with INFORMIX-4GL places all `.4gl` files in the same directory. This is fine for a "rolodex" program (or maybe even a simple "stores" application), but when the application grows to more than one `.4ge` program, it becomes unmanageable if it is kept under one directory. The number of separate files involved demands useful organization. In Fitrix Accounting, for example, there are hundreds of different files. Putting them all in a single directory would create an intolerable jumble.

Because INFORMIX-4GL doesn’t impose limitations on placement of source files, consistency in file organization must be maintained by the developer. In creating applications where many different people work with the same source code, it is important—perhaps critical—to have a consistent method of file organization. The Fitrix Accounting system of organization has evolved to optimize designing, programming, testing, installing, troubleshooting, and modifying 4GL applications. It is supported by the Menus front-end in a way that allows for easy modification of the application.

When Fitrix Accounting is installed, files are put in only three locations:

1. Start-up shell scripts for running each accounting module and the accounting package as a whole are put into `/usr/bin.` These files all begin with `fg.`

2. Files that contain functions which are application-independent are stored in the "fourgen" library. These are the functions called by the code generator. The path of this library is `$fg/lib` where `$fg` is a system-wide variable containing a directory on the system that contains this library. The default value for the `$fg` variable is `/usr/fourgen`, so the default location of these routines is `/usr/fourgen/lib`.

   The `$fg/lib` directory is further broken down into two subdirectories. The directories `screen.4gs` and `report.4gs` contain the source code for these libraries. The directories `screen.RDS` and `report.RDS` contain the source and object compiles for the Informix Rapid Development System (RDS).

   Typically, unless you are translating to a foreign language, little or no modification is done to the routines in these directories.

3. Files that contain functions specific to an application are stored under the application directory.

The Fitrix Accounting Application Directory

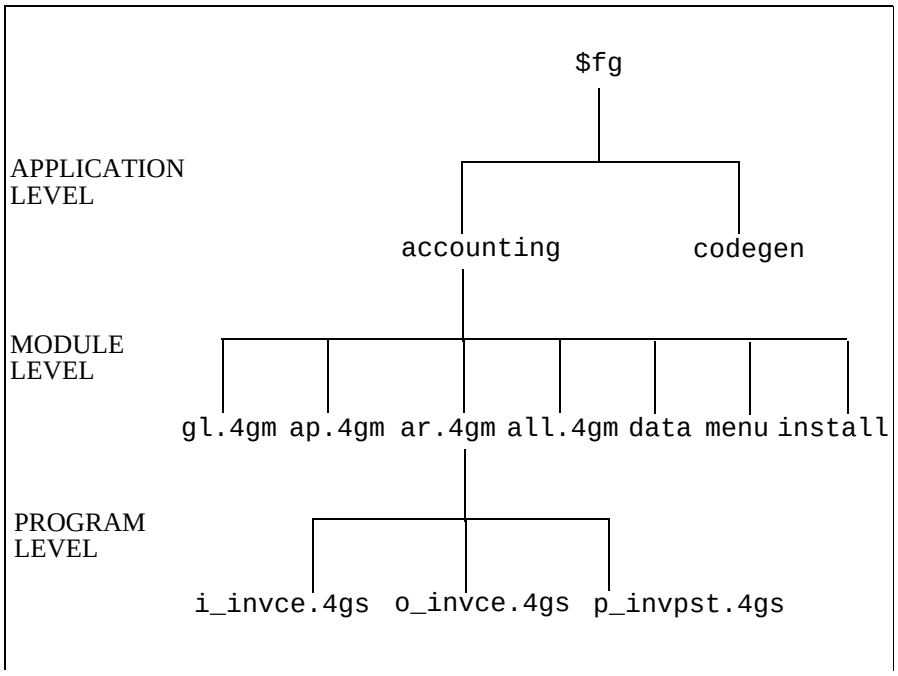
All applications must have a unique name. We use "accounting" as the application name for Fitrix Accounting. This includes all the modules—General Ledger, Accounts Receivable, Accounts Payable, etc. This name identifies the topmost directory in the hierarchy of the directory structure used in the application.

Where you place this "base" accounting directory in your file system doesn't matter. This allows customers to install the application wherever they have the most hard disk space.

In modifying the Fitrix Accounting application, you may want to create a duplicate of the entire accounting system. You can do this simply by installing the Fitrix Accounting system under another directory on your system. This gives you two separate versions of the source code, one that you can modify and one that is unchanged. This procedure allows you to maintain control over changes made to

source code and to identify differences between your modified version and the original at any time. The system elects to run one version or another depending on which directory the various variables are set to "point" to.

Accounting Directory Structure.



Under the "application" directory called "accounting", there are several different types of directories

First are the "module" directories. A module is a free-standing part of the overall application. Though a module may share data files with other modules in the accounting package, it does not require the installation of any other module in order to run. There is one such directory for each module in Fitrix Accounting. In other words, there is a separate directory for the General Ledger, Accounts Receivable, and so on. This directory is named for the module in the format

module_name`.4gm.` We typically use a two letter abbreviation for the module names. Each of these module directories uses a consistent hierarchical structure which is described later.

The next directory within the "accounting" directory is the directory called `all.4gm`. This directory contains programs that are used by all modules. For example, the program for accessing the control file is used by all modules. It is stored in the `all.4gm` directory. The hierarchical structure of this directory is the same as the other `.4gm` directories.

Next, under the accounting directory, we have the directory `data.` This directory contains the database directory or directories (if multiple databases are used). Your $DBPATH variable should be set to point at `.../accounting/data` when using our accounting system with the standard Informix engine. This directory contains the `.dbs` directories used by Informix.

Within the accounting directory, is a directory called all.4gm. In the all.4gm directory is another directory called lib.4gs. The lib.4gs directory holds functions common to all modules. Each .4gm has its own lib.4gs for module specific functions.

The final directory within the accounting directory is the `menu` directory. This is the directory in which all the user menus are stored. Our menus have their own directory structure which is covered in detail in the *Menus User Reference Manual*. That manual deals with the menu system we use to drive Accounting.

To summarize, our application directory `.../accounting` contains the following directories:

Directory	Contents
`all.4gm`	programs used by different modules
`ap.4gm`	programs used by Accounts Payable
`ar.4gm`	programs used by Accounts Receivable
`fa.4gm`	programs used by Fixed Assets
`gl.4gm`	programs used by General Ledger
`ic.4gm`	programs used by Inventory Control
`oe.4gm`	programs used by Order Entry

pu.4gm	programs used by Purchasing
py.4gm	programs used by Payroll
mc.4gm	programs used by Multicurrency
mcr.4gm	programs used by Multicompany Reporting
bin	ancillary programs
data	the database directories
install	files containing installation information
menu	the menus for Accounting

Structure of the .4gm Directories

The ***module***.4gm directories and the all.4gm directory contain the program directories. All source code for a .4ge program is stored in its own directory. These directories are identified by the 4gs extension (the s stands for source).

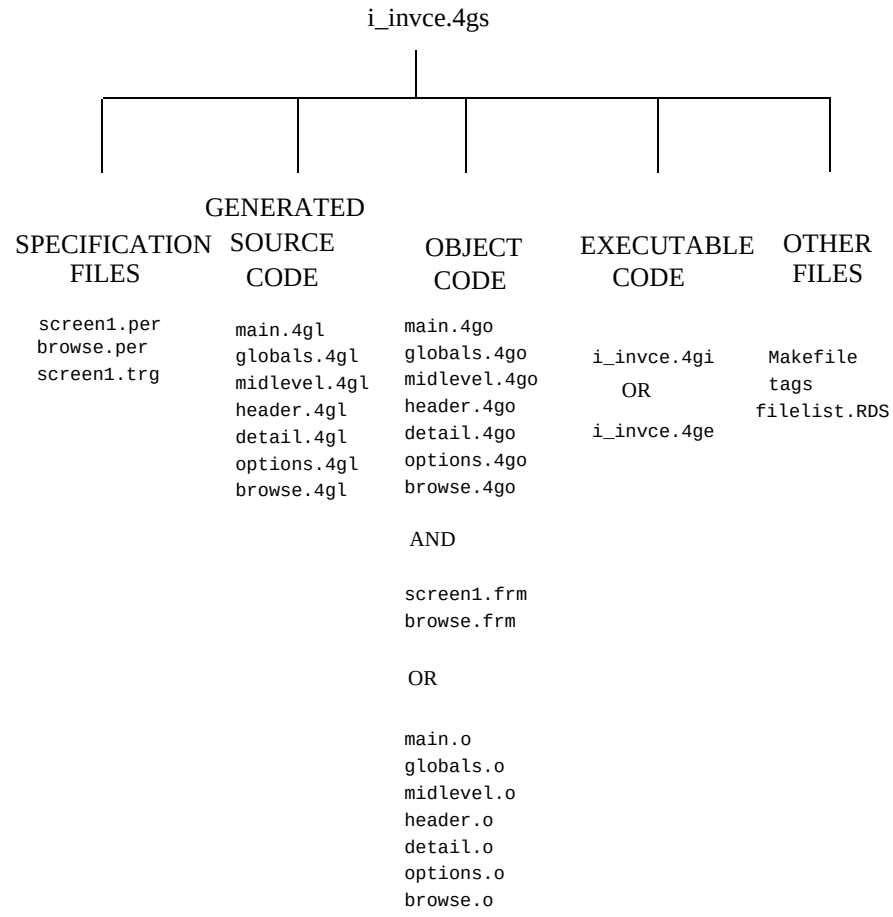
If the program is used for data input (i.e. it contains a screen form), the directory name begins with i_. If it produces "output" such as a report or printed form, it begins with o_. If it produces no output, but posts information from one table to another, it begins with p_. To this prefix we add a six character program identifier that describes the program. In General Ledger, the directory i_chart.4gs contains the source for the program that allows input into the chart of accounts. The directory o_balsht.4gs contains the source for printing a balance sheet. The directory p_yrend.4gs contains the source for the year end posting process.

Here are some examples of program directories in the .../accounting/gl.4gm directory:

Contents of a Program Directory

Under each "program" directory you will find a number of different files.

PROGRAM DIRECTORY FILES



A `Makefile` file gives instructions on how to compile the programs in the directory. An `Info` file provides programmer information about how the program works. Files ending in the extension `.per` are the INFORMIX-4GL forms that is, data entry screens needed for the program. There are also several `.4gl` files that contain the source code for the single `.4ge` program, and the file containing the `.4ge` program itself.

NOTE: If you are using the RDS Informix environment instead of 4GL, the executable file extensions will be `.4gi` instead of `.4ge`. "RDS vs. 4GL: Choosing an Informix Environment" on page 4-44

The `Makefile` is used for compiling the several `.4gl` files into one `.4ge` file. Our use of the UNIX make utility is documented in detail later in this manual.

The `Info` file contains programmers' notes about this program. It includes special information that may not be apparent to someone modifying the program. `Info` might be considered a program specific extension of this manual.

The executable program in this directory has the same name as the directory, but ends in the extension `.4ge`. For example, the path name of the program that allows input in the chart of accounts would be `.../accounting/gl.4gm/i_chart.4gs/i_chart.4ge`. When the menu system needs to run this program, it changes to this directory and then executes this program.

There are two basic sets of 4GL files. As an example of the system files, the following page contains a table of files in the `...accounting/gl.4gm/i_chart.4gs` directory. This directory is for the program used to input chart of accounts data.

All files ending in `.4gl` contain various parts of the source code for the data entry application. All of the files in the table are absolutely standard. For example, there is a `globals.4gl` in every data entry and reporting program directory and it always contains the global variables for that program. Of these files, only the `i_chart.4ge` is a unique name.

Files of the `.../accounting/gl.4gm/i_chart.4gs` program directory:

Table 4: Sample Program File

File	File Content
`Info`	programmer information about program operation
`Makefile`	instructions for "making" the program
`browse.4gl`	INFORMIX-4GL source for browse function
`browse.o`	object code for browse function
`browse.per`	source "browse" screen for viewing a list of documents
`browse.frm`	compiled browse screen
`detail.4gl`	4GL source for the scrolling "detail" region of screen
`detail.o`	object code for the scrolling detail region
`i_chart.4ge`	runtime version of the total program
`globals.4gl`	"source" definition of global variables for the program
`globals.o`	object code for global variables
`header.4gl`	"source" code for the "header" section of document
`header.o`	object code for the header of document
`main.4gl`	source code for the main calling program
`main.o`	object for the main program
`screen1.per`	source for the main data entry screen
`screen1.frm`	compiled data entry screen

The only files that are required to run the application are the `.4ge` files, which contain the executable version of the program, and the compiled screen files, whose names end in `.frm`. All the other files can be considered source code and can be removed from run-time versions of this directory.

Keeping Modified Program Code Separate

Though you can modify the code residing in your `.4gs` directories, we recommend you not do this. It is a good idea to keep those basic sets of program code unmodified.

To simplify the process of making modifications, we have implemented a system through our menu front-end whereby you can easily create a parallel directory to the `.4gs` directory. The purpose of this parallel directory is to allow you to create customized versions of programs and have the menu system call them without changing the menus themselves or modifying the original code.

The parallel directory you create is the `.4gc` directory. The `c` on the end stands for "custom", and the purpose of this directory is to hold the customized 4GL code for your application.

When the menu system goes to find a program, it looks first for the directory with the program name ending in the extension `.4gc`. In other words, it looks first for the modified code. It runs the standard program in the `.4gs` directory only when it does not find a set of modified code.

Managing Parallel Program Directories

When you are modifying source code to create a customized version of a program, we suggest the following procedure:

1. Create a directory with the program name but with the extension `.4gw`. The `w` stands for "working." The menu system will ignore this directory. It gives you a place to work with the code without affecting the program in production.

2. Copy all the source files from the `.4gs` directory to this working directory. This allows you to work with an unmodified version of the source.

3. Make the modifications, testing the executable program locally. Since the menu system ignores this directory, users can be running the unmodified program while you make these changes.

4. When the program is finished and ready to be installed, simply rename the `.4gw` directory to `.4gc`. The next time the users access this program, they will now run the modified version of the code. If problems turn up, you can rename it back to `.4gw` to take it out of use again while you rework it. Users will then automatically be using the old version.

5. If you need to store an old version of customized code, you can do so simply by changing the "extension" to indicate the version of the program that code represents; for example, `program.old` or `program.v12`.

NOTE: Using .4gw directories only applies if you have a version of the program in production (i.e. you are not at the customer site). If you are developing at your office, simply create a .4gc and do all of your work there.

Functions Within Files

Each of the `.4gl` files is further broken down into functions within each file. As we have said before, one of our design goals was to have each function perform a very specific purpose. This means each function is small in terms of lines of code.

"Local" functions, those that are concerned with data flow of the current program, are stored in the local directory.

Data independent functions are stored in a tool or accounting library directory depending upon how the program uses the functions."4GL Library Functions" on page 4-4

Menu Directory Structure

It was mentioned earlier that in the application directory—`$fg/accounting`—there is a directory called "menu." It contains all menus, menu item instruction scripts, and menu item help.

This directory, `$fg/accounting/menu,` is the base directory for the accounting system. As such, its directory path is stored in a special system variable, "mz". The value of this variable, indicated by `$mz`, must be set to "point" at this menu directory in order for the application menus to work. This is normally done by the start-up shell script that runs Fitrix Accounting.

Menus are grouped into "projects" for organizational purposes. A project simply consists of a group of related menus. Menus can support any number of project directories. You can have a project directory for each user on your system, thus providing a specific user with a particular group of menus. You can also have a project directory for each module, thus logically organizing the menus for that module in one directory. This is how Fitrix Accounting is organized.

Menus in one project directory can call menus in other project directories. The UNIX directory system is used to simplify system organization since a given computer may have literally hundreds of menus organized into dozens of projects.

We call these "project" directories because you may want to set up your own menus that are unrelated to specific modules. You may, for example, want to redefine menus by user or class of user. In that case, you could also set up those directories as separate menu projects.

Under the `$mz` directory, `$fg/accounting/menu,` there is a separate directory for each module available on Fitrix Accounting. Each of these directories contains the menus for that specific module. Below is a list of project directories and associated modules.

`apmenu` - Accounts Payable

`armenu` - Accounts Receivable

`famenu` - Fixed Assets

`glmenu` - General Ledger

`icmenu` - Inventory Control

`mainmenu` - Main Start-up Menu

`oemenu` - Order Entry

`pumenu` - Purchasing

`pymenu` - Payroll

Internally, each project is a directory for menus. Within the project directory, each menu is also a unique directory. In other words, each of these project directories contains a bunch of directories, each of which represents a different menu that comes with the package.

As an example of "menu" directories within a "project" directory, here is a list of menu directories in the `apmenu` project:

`aging` - detail menu for printing aging report

`check` - general menu for processing checks

`glact` - detail menu for printing G/L activity

`journ` - general menu for entering transactions

`label` - detail menu for printing vendor labels

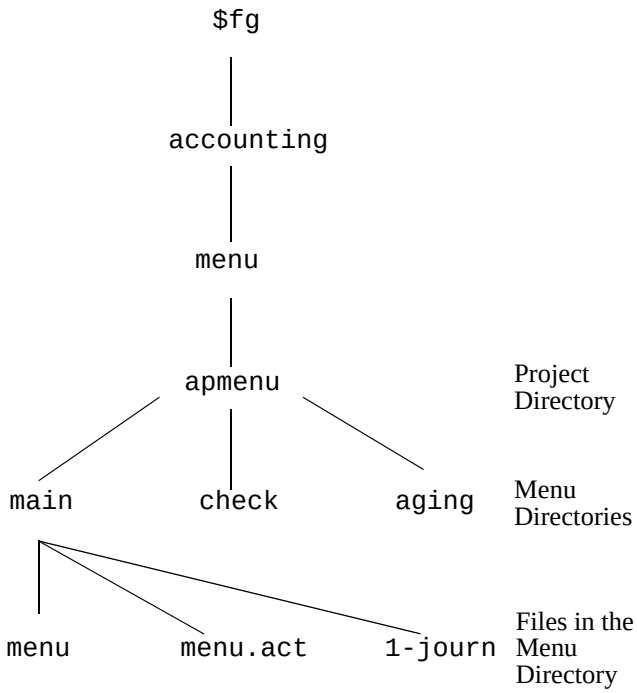
`main` - main menu that calls document menus

`setup` - general menu for setting up the system

`vend` - general menu for accessing vendor info

Diagram of Menu Directories

This diagram provides a visual representation of the file organization of a sample Menus directory:



In the example above, the path starts at `.../accounting/menu.` The project within this directory is `apmenu`. There is a separate project for all other modules, but these are not shown above. There are three subdirectories shown in `apmenu--main, checks,` and `aging.` There are other menus in this directory, but they are not shown. In the `main` directory you see the Image and Action files and an example of one Item Instruction File, `1-journ.` Though there is not enough room to show this, there is an Item Instruction file such as `2-check` for every menu item on the `main` menu. For more information about Menus refer to the *Menus User Reference Manual.*

3

Interacting with the Database

The database is separate from the application that accesses and maintains it. Fitrix Accounting uses SQL statements to extract data from and write data to this table structure, but the tables themselves are accessible from any SQL compatible tool. It is important to know not only what information is stored in the tables, but also how the tables are related.

This section of the manual discusses the organization of the financial database created and maintained by Fitrix Accounting, how to access tables, how to post to these tables, how to merge databases, and database administration issues. Specific flow diagrams illustrating the relationships between accounting activities and the database tables are contained in the module User Reference manuals.

Database Design Criteria

The Fitrix Accounting database structure was designed with several goals:

1. It is easy to understand by both a person using SQL and a developer.
2. It utilizes relational structure to minimize duplicate data.
3. Whenever possible the number of tables used was minimized.
4. It facilitates the automatic integration of standard packages to each other and to customer modules though database design.
5. It uses SQL whenever possible to minimize the maintenance of "summary" information in the database.
6. It provides a complete, on-line audit trail.
7. All audit trail information is easily archivable so that it can be removed from and added to that system as needed.
8. The data structure is designed to minimize the use of storage space.

It should be noted that in this list of priorities, minimization of data storage space comes last, not first. Most accounting systems are not designed this way. Our priority here was to have as much information as possible accessible through the computer and, within that framework, minimize wasted space as much as possible.

Fitrix Accounting Data Flow Overview

Data flows through accounting as follows:

1. Reference and control tables are built.
2. Business transactions are entered into the system.
3. The edit list is printed to facilitate error-checking.
4. The corrected information is posted permanently into the system and open-item tables are updated.
5. Reports are printed based on the recorded entries.

Posting is the process of changing information in one table with information stored in another table. This process generally involves updating or adding rows to a table.

The following discussion is divided into two categories: tables and programs involved prior to posting, and those that are involved during and after posting.

Before-the-Post Tables

Control Tables contain parameters, defaults, and control numbers that determine how a program processes information. For example in the A/R control table (`strcntrc`), the `disc_tax` column indicates whether or not the invoice discount should be applied to the sales tax. `strcntrc` also keeps track of the last A/R document number, which is used for posting.

Reference Tables contain information that is frequently consulted. `strcustr`, for example, is the customer reference table. It contains information about the customer's name, address, phone number, zip code, and so on. That information is seldom changed. Not all of the information in `strcustr`, however, is static. For example the current account balance and last pay date (columns `acct_bal` and `last_pay_date`) are updated as required.

Entry Tables - Header Tables and Detail Tables store the initial entry of business transactions. Each invoice, for example, has one entry in the invoice entry header table (`strinvce`). However, for each invoice, there are one or more related rows of detail information. Thus, if customer Jones orders nine white widgets and six blue widgets, there will be one row in the invoice header table and two rows in the invoice detail table (`strinvcd`).

The following example follows a business transaction through the system. By way of example, Jones and his order of blue and white widgets is used.

Before entering his order, two control tables must be set up: the company control table (`stxcntrc`), and the accounts receivable control table (`strcntrc`)

Several reference tables are also needed: the chart of accounts reference table (`stxchrtr`), the accounting periods reference table (`stxperdr`), a generic information table for codes and descriptions (`stxinfor`), the customer reference table (`strcustr`), the customer ship-to table (`strshipr`), and so on.

With these tables in place, the business transaction, in this case an invoice, can be entered.

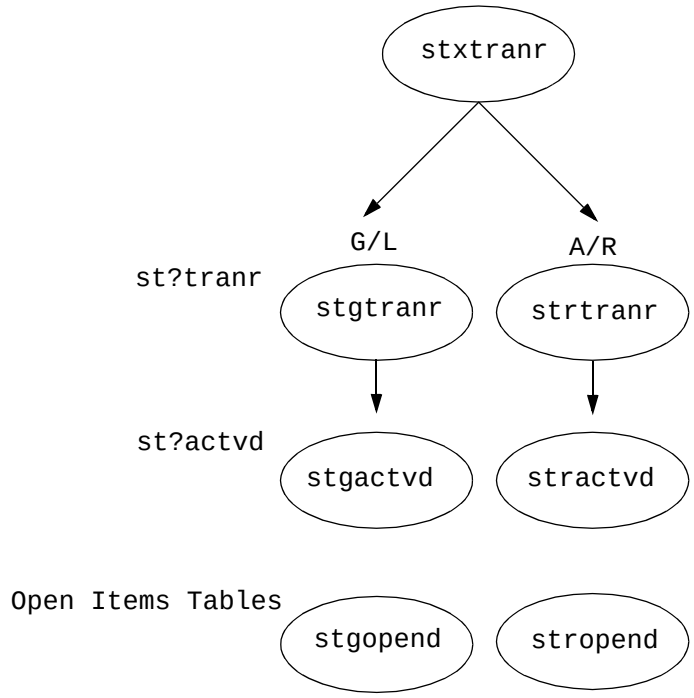
Invoices are entered in the A/R invoice entry header table (`strinvce`). That's where Jones' customer code, his payment terms, his tax account number, and so on are stored. The quantity, number of widgets, description, etc are stored in the A/R invoice entry detail table (`strinvcd`).

Note that the invoice has been entered into the accounting system but not yet posted. Therefore allowing changes to be made the entry to be deleted.

The next step is to print the invoice and post it to A/R. The posted entry cannot now be changed. To undo the effects of an incorrect entry, reversing entries must be made.

Since the after-the-post information is being discussed, the transaction tables need to be examined.

Data Layout After Posting



Posting Tables: At the top or 'root' of the inverted tree is `stxtranr`, the link that ties the various accounting modules together. For every transaction entered into the system there is one and only one entry in `stxtranr`. `stxtranr` contains information that is common to all business transactions. For example, posting date, document number, and the name of the journal that sparked the entry (for example, O/E, A/R, etc.).

The next level down are the header-transaction tables, and below them are the detail-transaction tables. Each module has its own header-transaction and detail-transaction tables because each module has different requirements.

The tree-design was chosen because it reduces data redundancy while at the same time allowing for additional 'legs' or 'branches' to be added without sacrificing the integrity of the original structure.

So what happens when Jones' order for widgets is posted?

In short, the A/R leg of the structure is traversed and then (if installed) the G/L leg. Information common to all transactions is extracted from the original entry document (the invoice) and placed in `stxtranr`. Information common to A/R is extracted and added to `strtranr`, and information specific to that particular A/R document is extracted and added to `stractvd`.

All posted transactions, regardless of the module in which they originate, impact the G/L tables, `stgtranr` and `stgactvd`. After traversing the A/R leg of the tree the G/L leg is traversed. The G/L tables store information necessary to properly update the ledger accounts impacted by a transaction. `stxtranr` has already had a row inserted for the transaction. Header information (general information about the transaction) is added to `stgtranr` and detail information (one row for each account that will eventually be updated by the transaction, the amount, whether it's a debit or credit, etc.) is stored in the `stgactvd` table.

This posting process is similar for all modules. If only one module is installed, then only the leg representing that module and the G/L leg are traversed. If several modules are installed, several may have to be traversed. For example, O/E traverses several legs (O/E, A/R, G/L, and I/C).

There are several things to note:

1. The tree structure represents the data layout *after* posting.
2. After posting, the original entry document is removed. (In O/E the entry documents are retained for a set time, then removed by subsequent postings.)

   NOTE: With the 4.0 release of Accounting, deleting documents during posting will be parameter driven. Documents will not necessarily have to be deleted.
3. The posted data cannot be updated (errors may be 'corrected' by posting additional error-correcting entries).
4. All posting is done using database transactions (i.e., BEGIN WORK, COMMIT WORK) to ensure that the entire posting is successful. If a problem occurs with posting, then all tables are restored to their state prior to the transaction.
5. During posting, rows in open-item tables may be inserted or updated.

Open-Item Tables are in a class of their own. Rows are inserted or updated during posting.

For example, when the invoice above for Jones' widget order was posted it inserted a row in the A/R open items table (`stropend`) to reflect the outstanding balance. The outstanding balance is also stored in the A/R table, but if Jones wants to make payment on his order in monthly installments, the open item entry provides a convenient place to track the balance on the invoice. At any given moment, the open item balance is a reflection of the original amount plus the subsequent debits and credits applied to that invoice. It's faster to query the open items table than it is to retrieve the original amount, and then the subsequent entries that have affected that amount, and then tally the results.

When integrating custom applications recommends that you put information into the entry tables and let the posting routines take it from there, rather than directly calling the posting routines. This makes detecting problems much easier.

Fitrix Accounting Database Table Overview

Information flows in a consistent pattern through each Fitrix Accounting module.

Five types of tables are accessed:

1. reference tables (a control table is a type of reference table)
2. entry tables
3. transaction tables (a posting table)
4. activity tables (a posting table)
5. open item tables

First, information is entered into **reference tables**. Information is added extensively to these tables during the setup phase for a particular module, and later to a lesser extent, after setup is complete and transactions have been entered. For example, before entering transactions in a module, you must set-up the chart of accounts because these accounts are "referenced" when entering transactions. The menu options that access these reference tables are found on the Setup Company Menu and the setup menu for each module.

Reference tables contain information that is accessed by a variety of programs. They are named according to the information they store. For example, `strcustr` is a reference table that stores customer information.

Reference tables store default values. Many data-entry forms provide the end-user with default values in a variety of fields. These default values are retrieved from reference tables. A reference table may be checked during the process of data validation (checking that an end-user entered a legitimate value in a particular field). Lookups are frequently made to reference tables in order to display descriptive information on a form. For example, a customer code may be entered in a field, the program looks up and displays the customer name.

A control table is a particular type of reference table. There is a company-wide control table, `stxcntrc`, accessed by the `Update Company Information` menu option. In addition, each module has a control table, accessed by the `Update` <*`module`*> `Defaults` option. Control tables contain a single row and store basic information about the company or a specific module.

Other examples of reference tables are the tables that store customer information, vendor information, and inventory information.

Next, transactions are entered and stored in **entry tables**. Transactions constitute a business's daily activity. Examples of transactions are a purchase from a vendor, sale to a customer, receipt of cash, and receipt of shipped merchandise.

Entry tables are named according to the type of transaction they store. For example, the name `cash` is used for tables recording cash receipts or cash payments and the name `invc` is used for the tables containing vendor or customer invoices. Entry tables are maintained using an "update" option such as `Update Cash Receipts`.

When documents are posted, rows are inserted or updated in transaction and activity tables, and in some cases, open item and reference tables.

Typically the documents (stored in entry tables) are deleted (if they are not recurring or incomplete). Usually the original document contains much more information about a business transaction than you want to save on the system. Therefore, the information from these documents is condensed and stored in a new, more terse, and directly useful form.

The first type of table updated during posting is the **transaction table**. Each module has a transaction table. There is also a system-wide transaction table (`stxtranr`).

Each row in a transaction table is uniquely identified by the original journal and document number of a posted document. Transaction tables are part of a header detail relationship with activity tables: the transaction table is the header table.

Transaction tables are updated by the posting process.

The first transaction table that is always updated during posting is the module transaction table (e.g. `stptranr`, `strtranr`, etc). This table contains header information (information that pertains to the entire transaction) unique to a particular module. For example, the transaction table for documents affecting the general ledger contains the account period and account year associated with the transaction.

The second transaction table that is always updated during posting is the system-wide transaction table shared by all modules (`stxtranr`). This table contains header information that is shared by all types of documents, including the date of posting and its posting number.

Posting in all modules results in the insertion of a row into the G/L transaction table (`stgtranr`). This table exists even if the General Ledger module is not installed. It stores information that pertains to postings that will affect the chart of accounts.

In addition, rows may be inserted into the transaction tables of other modules at the time of posting.

Transaction table names always contain the letters `tran` (which stands for "transaction").

The second type of table updated during posting is the **activity table**. Activity tables are part of a header/detail relationship with transaction tables. The activity table is the detail table.

At the time of posting, rows are inserted into activity tables.

Rows are always inserted into the module's activity table. In the case of the General Ledger, this is the stgactvd table. Non G/L modules also update the G/L activity table. This is the case regardless of whether General Ledger is installed. In other words, the G/L transaction (`stgtranr`) and activity tables (`stgactvd`) are provided even if General Ledger is not installed. It stores the ledger account, department, and credit or debit amount that is to be posted to the chart of accounts.

Activity table names always contain the letters `actv` (which stands for activity).

The third and final type of table in which rows are created during posting is the **open item table**. A/R and A/P are the only modules with open item tables. This table is used to track information which will be updated at some future time. In the Accounts Receivable module, for example, posted customer invoices update the open item table. Later cash receipts may be applied to existing open items.

Rows in open item tables may be inserted or updated by a transaction. This is in contrast to transaction and activity tables where posting always results in rows being inserted.

Entering Information Into Reference and Entry Tables

Reference tables are header-only tables. Entry tables typically comprise a header table and a detail table. In the case of entry tables, the header portion of a document consists of general information about the document. With a customer invoice, for example, the header comprises a customer code, description of the invoice, and so forth. The detail section of the document consists of the individual detail lines that make up the document. Again, in the case of a customer invoice, this would be the individual lines that represent products ordered by the customer.

If a transaction is divided into a header and detail section, these sections are stored in different tables. There is only one header "row" (or "record") for each transaction, but there may be any number (though the usual limit is 100) of detail rows per transaction.

The header and detail sections of a transaction typically represent a one-to-many relationship (though there could also be a one-to-one relationship). One row in the header database table may be associated with any number of rows in the detail table. Header/detail relationships are maintained by all entry, listing, and posting programs.

For entry tables, the header/detail relationship is straightforward. What you see on the screen is generally reflected in the database table structure. The top of the screen records information stored in the entry header table. This table is identified by the `e` ending its full name (e.g., `strinvce`). There is only one row in this table for each transaction.

The lower portion of a data-entry form—the scrolling "detail" area—is stored in the entry detail table identified by the `d` ending its name (e.g., `strinvcd`). There can be any number of rows in this table per transaction. Both tables are keyed by transaction number so that they always appear together when displayed on a data-entry form or when listed on a report.

Entering Information Into Transaction and Activity Tables

After a transaction document is posted, the table relationship becomes more complex because rows may be inserted into several tables for each transaction.

The first reason that the relationship is more complex is that the transaction header is divided into at least two different tables. The "system" transaction table, `stxtranr`, contains the information held in common by all Fitrix Accounting transactions: document type, number, post date, and so on. In turn, each module has its own transaction table. For example, the transaction table for the general ledger is `stgtranr`; the transaction table for the accounts receivable is `strtranr`. A transaction may create entries in additional transaction tables, as well. Most posted transactions in modules have a potential impact on the general ledger chart of accounts balance; therefore at the time of posting, entries are created in the general ledger transaction table.

The detail section of the transaction is similarly divided among several different activity tables. Each module has its own activity table. Different modules can post to each other's activity. For example, most transactions post to the general ledger activity table `stgactvd`, and both A/R transactions and O/E transactions post into the account receivable activity table, `stractvd`.

The entry detail table and module activity table relationship is not necessarily a one-to-one relationship. For example, one detail line of an order entry invoice is stored as one row in an entry table, but when posted, can generate two rows in the general ledger activity table (representing a credit to one account and a debit to another).

A row may also be inserted in a module activity table that stores a value that was not originally stored in the entry table detail table. For example, in the ledger activity table again, there is frequently a "balancing account" entry that represents certain totals from the transaction. These totals are not in the detail table, but a summary of the information in them.

Each module has a unique audit trail. The header information, stored both in the system-wide transaction table and the module transaction table, consists of general information about this activity. Together, the activity tables and transaction tables can be seen as the header and detail sections of a complex system-wide transaction recording a business transaction. This transaction may be stored in multiple header

and detail tables. Instead of one-to-many, the header/detail relationship may be seen as one (system transaction table) to one (module transaction table) to another one (optional module transaction table) to many (module activity table) to others (another module activity table).

As you can see, it can be a mistake to think of the system as hierarchical in the sense of many third generation programs. Though the entry transactions have a fairly traditional one-to-many structure, information is stored in a truly "relational" structure once it is posted to the system, becoming part of the financial database.

Entering Information Into Open Item Tables

The open item tables store summary information about transactions. They are utilized for fast access to information. Not all modules contain open item tables. Though such tables can be joined to related transaction and activity tables, each row in an open item table represents a complete transaction.

Open item tables store a transaction's original and current balance. The Accounts Receivable module contains an open item table. In the case of Accounts Receivable, the original balance preserved in this table was an original invoice total. A total is technically not part of a header or detail section of a transaction. Like the header information, there is only one total for the transaction, but this total is the sum of the detail rows. So the total is related to both the header and the detail sections of the transaction in different ways.

As more activity is posted to the system, the transaction balance stored in the open item table changes. The current balance is a reflection both of the first transaction that created the open item and all subsequent transactions that have had an effect on that balance.

Viewing Database Table Descriptions

In order to provide you with the most current database information available, we have developed a method by which you can quickly view this information on-line, rather than listing it here in this documentation. Database table descriptions are contained in a subdirectory under each module directory. The `$fg/accounting/<module>.4ge/Doc/maps` file contains descriptions of each table and column in the accounting database. These descriptions are much more detailed than past descriptions and are much easier to obtain.

The `maps` file also contains special markers (e.g. .iX strcntrc). These markers are used with a special tagging feature which allows you to view the descriptions of a table from any location simply by typing `tag -t <table name>`. Typing this command will invoke the vi editor on the `maps` file and the table you typed will be displayed on your screen. The tags feature also allows you to view function descriptions in the same way. For more information on the tags feature see "The Tag Utility" on page 4-18.

Using Database Transactions for Posting

In Fitrix Accounting, many programs post data from one (or more) table to another (or multiple) tables. In other words, the posting process reads data stored in one or more tables and subsequently inserts rows and/or updates rows in other tables.

Fitrix Accounting posting programs are named with a "`p_`" prefix. The posting programs take advantage of "database transactions." The use of database transactions ensure data integrity.

Using transactions guarantees that everything between your `begin work` and `commit work` statements is either done 100%, or is not done at all (this is important if a problem occurs).

Transactions provide the ability to "rollback work" (automatically un-do the changes made to the database since the last `begin work` command).

They give you the ability to "rollforward database." That is the ability to take a backup of your database, and apply ALL changes made to it since that backup to bring it up to date.

When running the Informix standard engine, you must make sure to clean out the logfile fairly often. If you are using the OnLine engine, the logfile is handled automatically.

Posting from Custom Applications to Fitrix Accounting General Ledger

The discussion of the financial database cannot be complete until you understand how to get your own data into that database from your custom applications.

Modifiability is the cornerstone of all applications. The ability to tailor software to meet your end-users' needs is what makes you successful in the marketplace. We assume when you buy the package that you are going to be writing completely cus-

tom entry front-ends that capture business transactions unique to your business. It is also assumed that many of these transactions should generate accounting information that you want managed by Fitrix Accounting.

It may seem that you need to know a lot about the database structure in order to capture and manage this accounting information. However, this process has been made easy for you. You are provided with certain functions that can be used from within your custom application to post data into the financial database on a regular basis. These functions are the same ones that we use within Fitrix Accounting to put data in the accounting system.

To interface a custom module with the General Ledger you must call the `gl_post` function in your custom module. Refer to the function `gl_post` in `$fg/accounting/all.4gm/lib.4gs/gl_post.4gl`

Your program must pass the proper arguments to `gl_post`:

```
function gl_post(post_or_check,   orig_journal,   doc_no,
  post_no,  post_date,  doc_date,  ref_code,  doc_desc,
  inv_chk_no,     acct_no,     department,     amount,
  debit_credit)
```

You can examine any of the Fitrix Accounting modules to see how they interact with the `gl_post` function.

The `gl_post()`function posts to the G/L activity tables: `stxtranr`, `stgtranr`, and `stgactvd`

Data Posting Functions

There are functions for posting to the "activity" tables of each module. There is also a function for posting to transaction tables. All of these functions are in the accounting function library under `$fg/accounting/all.4gm/lib.4gs`. The posting functions for each module are named with the two letter identifier for that module (`gl, ar, ap, oe,` etc.) followed by `_post`. For several modules, there is a related function called *`module`*`_last`, which is called at the end of the transaction to check and validate it. The transaction posting routine is called `trx_post`.

Take a closer look at one of these routines. Suppose you now own our General Ledger package, but you've already written a data entry package for your client. You've chosen Informix as your 4GL because it's easy to modify, and now it's time to merge your data into G/L. A nice attribute of library functions is that once they are written, you don't have to know all of the details about how they work when you need to use them. All you need to know is how to call them.

You don't even need to know how the G/L tables are structured. Just pass the data to the posting routines that have been written for you. If you want to know specifically how they work, just print them out. You will find that over 25% of the text consists of comments, including different ways to use the functions.

The modules follow the philosophy that data is entered into "entry" tables, then posted into activity tables. Before posting, an edit list is produced that allows users to identify errors. They can then go back and correct those errors with the data entry interface before posting. The program that produces this edit list does all data validation done during posting, so if the edit list produces an error, the user can go back and change the data in the entry tables.

All posting library routines are written to be used in the edit list phase as well as the posting phase. If you indicate that you are not posting, the routine will complete the data validation without actually posting to the activity tables. The routines used for posting to G/L are `gl_post` and `gl_last`. The `gl_post` routine is called once for each G/L entry for the transaction, and the `gl_last` routine is called at the end of the transaction to validate data and report any errors that may have occurred.

The prerequisite for posting to G/L is that the total of the credit amounts that are applied to accounts must equal the total of all debits for any one transaction. This guarantees that your G/L is always in balance. Numbers are always posted as positive. They may be entered, for example, as negative debits to an account, but the `gl_post` routine will actually post that as a positive credit to the account.

Posting Function Description Section

Each posting routine contains extensive comments describing its operation. Here is an example of the description section of the `gl_post` posting routine:

```
##########################################################################
function gl_post(post_or_check, orig_journal, doc_no,
                 post_no, post_date, doc_date, ref_code, doc_desc,
                 inv_chk_no, acct_no, department, amount, debit_credit)
# returns true/false based on pass/fail of the routine
##########################################################################
# This routine takes the G/L transaction data as arguments,
# and either posts to G/L or checks for an ok posting.
# It is designed to be run in "CHECK" mode during the edit list phase,
# and in "POST" mode during the posting phase.
# It modifies elements in the global record post_gl defined in
# gl_glob.4gl in this directory.
# All routines that post into the 4gen general ledger module should
# do so via this routine.
# This line is called once for each G/L transaction
#
# Data elements passed:
#    post_or_check char(5),  "POST" or "CHECK"  tells gl_post to check
#                            the posting (for edit lists), or post it.
#    orig_journal char(2),   original journal (AR, OE, etc...)
#    doc_no integer,         document number (sequential and congruent
#                            for each document within a journal)
#    post_no integer,        posting number (post only)
#    post_date date,         posting date (post only)
#    doc_date date,          document entry date
#    ref_code char(6),       customer code
#    doc_desc char(30),      document description
#    inv_chk_no char(12),    invoice or check number
#    acct_no integer,        account number
#    department char(3),     sub-account (department) code
#    amount like stgactvd.amount,   always a positive amount
#    debit_credit char(1)    debit or credit the account by amount
#
# status and description variables returned in the post_gl record:
# 0 - Successful
# 1 - fg_error("lib_all","gl_post",1) - Cannot Read G/L Control Table.
#     - return false
# 3 - fg_error("lib_all","gl_post",3) - Document Number out of Sequence
#     - (warning condition only) - continue
# 4 - fg_error("lib_all","gl_post",4) - Duplicate Document Number Exists
#     - (error condition) - return false
# 5 - fg_error("lib_all","gl_post",5) - Cannot Insert a New Document
#     - (sql insert command failed) - return false
# 6 fg_error("lib_all","gl_post",6)Account Not Listed in Chart of Accounts
#     -  return false
#
```

```
# What this process does:
#Check to see if GL is installed.  If not, then all statuses are P(posted)
# If last doc no isn't this doc no (first in a group)
#     if last doc no isnt this doc no - 1, then set warning #4
#     call trx_post - this (possibly) posts a new document into
#         stxtranr (top level)
#         trx_post returns:
#         0 = if "CHECK" or if "POST" and posted ok
#         1 = insert failed (post only)
#         2 = duplicate found
#     if "CHECK", check for duplicate stgtranr
#     if "POST", insert into stgtranr
# if (first in group)
# if "CHECK", then return
# if "POST" then:
#     insert the row into stgactvd
#
# This process is designed to be run in conjunction with the gl_last()
# library function that checks to make sure all credits match all debits
# and returns false if they don't.
#
```

Note the following about this function:

It can be called during the edit list phase of posting by passing `"CHECK"` in the `post_or_check` argument.

It relies on a `post_gl` record defined in your globals file (the layout of that record is in the `gl_glob.4gl` file in the library).

Because it must do the lookup to validate the ledger accounts, it returns the name of the account in the `acct_desc` element of the `post_gl` record so you don't have to perform that lookup to show the account on your edit/posting report.

It returns either true or false depending on data validity, and if it returns false, it sets the description variable in the `post_gl` record to the text of the failure. You may also call the standard error routine if the function fails. All errors are known and are described to the user along with possible solutions.

If the function returns false and you are in the posting phase, then you must issue a `rollback work` command to guarantee that none of the postings for this transaction have been committed to the database.

Sample Program Using the Posting Function

Again, the `gl_post` function is called for each credit/debit to a G/L account. Once all lines of a transaction have been run through the `gl_post` function, you then call `gl_last` (passing no arguments), and it returns either true or false depending on whether or not the entire document is correct. Like for `gl_post,` you may use the standard error routines for `gl_last`. Below is an example of how you can call these functions and how you would trap errors:

```
if gl_post(
  check_post, "GL", new_doc_no, post_no,today,
  curs.doc_date, curs.doc_src, curs.doc_desc,
  curs.acct_no, curs.department,curs.amount,
  curs.debit_credit
) = false
then
  if check_post = "POST"
  then
    rollback work
    call fg_error("lib_all","gl_post",post_gl.status)
    call end_report(post_gl.description)
  end if
end if
```

The preceding code would be run for each line of the document. Then, you would call the `gl_last` function at the end of the document:

```
if gl_last() = false
then
  if check_post = "POST"
  then
    rollback work
    call fg_error("lib_all","gl_last", post_gl.status)
  end if
  call end_report(post_gl.description)
else
  commit work
end if
```

Writing posting routines from your custom front-ends to any of the Fitrix Accounting modules is just this easy. You don’t even need to know the layout of the activity tables.

If there are any problems with postings, the user will be told what went wrong and given a chance to correct the problem. Using database transactions insures 100% integrity of all activity tables.

All other posting routines in the Fitrix Accounting library can be used in a similarly straightforward manner to get data from your front-end into the accounting system.

SQL Query Optimization

In more recent releases, work has been done to speed up certain functions of the program. Though this optimization may not have been performed in all parts of the product, you should be aware of the methodology involved. Many of these issues relate directly to the Informix Rapid Development System as opposed to the regular compiled versions of INFORMIX-4GL code.

How RDSQL Constructs a Query Plan

The RDSQL optimizer (a part of sqlexec or sqlturbo) selects a processing strategy for each RDSQL statement. This strategy, or query plan, determines the order in which RDSQL searches tables (the table-selection order) and which indices are needed to process the query.

The optimizer establishes an efficient table-selection order by applying a set of criteria to the tables involved in the query. These table-selection criteria are derived from general principles of query optimization. For example, an indexed search retrieves an arbitrary value faster than a non-indexed (row-by-row) search, and processing speed increases when the number of rows the query processor must examine decreases. If one criterion cannot order the tables, the optimizer applies another, and so on, through the last criterion. If this criterion cannot order the tables, the selection order becomes the order of the tables in the `FROM` clause.

Note: The terms "filter" and "join" are integral to a discussion of query optimization. They are defined in the following paragraphs if you are not familiar with them.

A "filter" is a condition expressed in a `WHERE` clause that removes undesired rows from a single table.

A "join" temporarily links two or more tables so that they can be queried as a single table. You can join tables when they contain columns that store comparable data, such as the `customer_num` columns in the customer and orders tables of the stores database.

The optimizer applies the following criteria to establish the table-selection order. The first criterion is applied to the `FROM` clause; subsequent criteria are applied to the entire `WHERE` clause.

Criterion 1: Outer Joins

The optimizer gives priority to the dominant table in an outer join, i.e. selects this table to be processed first, without applying subsequent criteria. By processing the dominant table first, RDSQL can preserve all of its rows.

When a query outer joins the result of a nested outer join to a third table, the optimizer gives priority to the outermost dominant table, followed by the innermost dominant table. ("outermost" and "innermost" refer to the level of nesting in the FROM list.)

Examples:

```
SELECT ...
FROM x, OUTER y
WHERE x.a = y.a
```

(table order is `x,` then `y`)

```
SELECT ...
FROM x, OUTER (z, OUTER y)
WHERE x.a = z.a AND
      x.b = y.b
```

(table order is `x,` then `z,` then `y`)

In the preceding example, `x` is the outermost table, and `z` is dominant over `y.`

Another example:

```
SELECT...
FROM x, y, OUTER z
WHERE x.a = y.a AND
      y.b = z.b
```

(table order is: `x,` then `y,` then `z`, or `y,` then `x,` then `z`)

In the preceding example, the optimizer gives both table `y` and `x` priority over `z,` the subservient table. Since `x` and `y` are at the same level in the `FROM` list, the optimizer must apply additional criteria to order the tables. With similar exceptions, the first criterion is normally sufficient to order the tables (the optimizer creates a temporary index as discussed later in this section). Thus the optimizer uses indices, as

well as other criteria, to order the query joins. Having determined an efficient order for RDSQL to perform the joins, the optimizer applies the same or similar criteria to order the tables.

If only one of the columns involved in a join is indexed, the optimizer gives priority to the table with the non-indexed join column. This allows RDSQL to use the indexed column to perform the join.

Example: (assumes `x.a` is indexed, but `y.a` is not)

```
SELECT...
FROM x,y
WHERE x.a = y.a
```

(table order is `y,` then `x`)

If both columns involved in the join are indexed, but only one is uniquely indexed, the optimizer gives priority to the table with the non-uniquely indexed join column. This allows RDSQL to use the uniquely indexed column to perform the join.

Example: (assumes `x.a` is indexed [unique] and `y.a` is indexed [duplicates])

```
SELECT...
FROM x,y
WHERE x.a = y.a
```

(table order is `y,` then `x`)

If both tables have indexed join columns, and both are either uniquely or non-uniquely indexed, the optimizer gives priority to the table with the columns containing the index with the fewest parts. This allows RDSQL to use the column containing the index with the most parts to perform the join.

If none of the columns involved in any join are indexed, the optimizer creates and then indexes a temporary table that is a copy of the largest table, the one with the most rows. RDSQL drops the temporary table when the query is finished.

Criterion 2: Filter Columns

If the optimizer cannot make a decision based on join conditions, it next examines filter columns.

If only one of the tables involved in a join has a filter column, the optimizer gives priority to that table. This allows RDSQL, in theory, to process the table with the fewest rows first.

Example:

```
SELECT...
FROM x,y
WHERE x.a = y.a AND
       y.a = 10        (table y filter)
```

(table order is `y,` then `x`)

If both tables have filters, but only the filter column of one table is indexed, the optimizer gives priority to that table.

Example: (assumes only `x.b` is indexed, but `y.c` is not)

```
SELECT...
FROM x,y
WHERE x.a = y.a AND
      x.b = 60 AND y.c = 20
```

(table order is `x,` then `y`)

If both tables have indexed filter columns, but only one has a unique index, the optimizer gives priority to that table.

Example: (assumes `x.b` is indexed [duplicate] and `y.c` is indexed [unique])

```
SELECT...
FROM x,y
WHERE x.a = y.a AND
      x.b = 10 AND y.c = 20
```

(table order is `y,` then `x`)

If both tables have indexed filter columns, and both are either uniquely or non-uniquely indexed, the optimizer gives priority to the table with the columns used in the index with the most parts.

Example: (assumes both `x.b` and `y.c` are indexed [duplicates], but `y.c` and `y.d` are used in a composite index):

```
SELECT...
FROM x,y
WHERE x.a = y.a AND
      x.b = 50 AND y.c = 20 AND y.d = 40
```

(table order is `y,` then `x`)

Criterion 3: Table Size

If none of the preceding criteria are sufficient to order the tables, the optimizer gives priority to the table with the fewest rows. It knows this only by issuing the `UPDATE STATISTICS` statement. This is also true in those rare instances when the optimizer cannot apply other criteria because of query structure or complexity.

When tables have the same number of rows, table order is irrelevant, and the order becomes the order in the `FROM` clause. Note: Because table size affects optimization, you should systematically update the table information in the system catalog so that RDSQL has current information. Use the `UPDATE STATISTICS` statement as described in the reference manual.

Improving Performance Through Indexing

In general, indexing is a trade-off with time gained during some operations and time lost during others. Every modification to an indexed column involves updating the corresponding `.idx` file. Thus, in a very large database, an index that helps performance during complex queries and sorting operations hurts performance during operations that add, change, or delete data. The more rows you modify at a time, the greater the impact. During random modifications, such as deleting a few rows from an indexed table, the effect is negligible.

You can address this problem by taking advantage of the inherent flexibility of a C-ISAM-based relational database. You need not commit yourself to a rigid indexing strategy. Instead, your applications can create the indices you need and drop them when they get in the way. You might, for example, create indices in anticipation of batch report writing during the night and drop them in the morning before there are huge data-entry needs. You must carefully evaluate the performance trade-offs based on first-hand knowledge of your database application.

Because it takes time to create an index on a table already containing data, you should create indices only for queries you or your applications use regularly. For spur-of-the-moment queries (the kind you do only once), you may be able to dispense with indices, relying entirely on auto-indexing to optimize processing.

You should not create indices on tables with fewer than 200 rows. The speed you gain does not offset the time required to open and search the index file.

In general, you should not index columns that can contain only a few possible values (yes/no responses, marital status, zip codes in a small city, and so on). This kind of data produces distorted indices, defeating the optimizing strategy of RDSQL.

It may occasionally make sense to index a column with many duplicate values, provided your anticipated queries will regularly fetch other values. For example, a state column that holds primarily one specific value (185 `WA`'s versus 18 `OR`'s, 22 `CA`'s, 12 `AK`'s, and so on) might benefit from an index if most queries fetch values other than `WA`. If, on the other hand, most queries fetch the dominant value (`WA`), an index hurts performance. This is a borderline case, and you must fine-tune your indexing strategy accordingly.

If you anticipate many queries that place conditions on a single column, you should index that column. If you anticipate queries that place conditions on several columns as a unit, you should put a composite index on all of the columns (assuming previous recommendations do not apply).

For queries similar to:

```
SELECT * FROM items
  WHERE order_num  1015
```

you should put an index on the `order_num` column.

For queries similar to:

```
SELECT * FROM items
  WHERE stock_num = 4 AND
    manu_code = "HRO"
```

you should create a composite index on both the `stock_num` and `manu_code` columns.

The order of the columns in a composite index can be significant. Assume that, in addition to the preceding query, you anticipate queries such as the following:

```
SELECT * FROM items
  WHERE stock_num = 5 AND
    manu_code = "HRO"
```

or

```
SELECT * FROM items
  WHERE stock_num  5
```

In the preceding case, `stock_num` must precede `manu_code` in the composite index. Otherwise the index cannot be used for optimizing the second query.

If you expect many queries to join a single column in one table with a single column in another table, you should at least index the column in the largest table (i.e. the table with the most rows).

For queries similar to

```
SELECT * FROM items, stock
  WHERE items.stock_num = stock.stock_num
```

you should index the `stock_num` column in the items table because that table has many more rows than the stock table.

For queries similar to

```
SELECT * FROM items, stock
   WHERE items.stock_num=stock.stock_num
   AND items.manu_code = stock.manu_code
```

you should make a composite index on the `stock_num` and manu_code columns in the items table.

The speed with which the `dbload` utility loads data into a database is greatly affected by the presence of indices. You may want to drop indices before running `dbload` and restore them when the program is finished.

For information on indices by table, refer to the section titled "Database Table Contents."

As previously demonstrated, join columns require indices for efficient processing. Version 2.10 of INFORMIX-SQL, INFORMIX-ESQL/C, and INFORMIX-ESQL/COBOL, as well as Version 1.10 of INFORMIX-4GL incorporate auto-indexing to optimize the processing of non-indexed queries. If your product features auto-indexing, the optimizer applies its own indexing strategy when you do not index at least one join column. To do so, the optimizer creates and then indexes a temporary table that is a copy of the largest table (the one with the most rows). RDSQL drops the temporary table when the query is finished.

Although it is transparent, auto-indexing still involves creating, indexing, and later, dropping a temporary table. It is therefore not appropriate for queries you perform often, such as those within an application. For such queries, a temporary table that is repeatedly created and dropped is extremely inefficient. Consequently, you should add indices according to the guidelines in this section and rely on auto-indexing only for one-time queries.

Merging Databases with `dbmerge`

Fitrix Accounting products are designed to be modified, and are frequently upgraded. Therefore, the task of merging a new version of a database with an established, possibly modified version is encountered regularly. The goal is to preserve all changes made to the established database while incorporating the features of the new version.

Our organization has automated the procedures for installing upgrades to our products. The code generator builds a 4GL program, `dbmerge`, that automatically applies the rules required to install a new version of a product without jeopardizing

the integrity of the installed database. That is, `dbmerge` checks to see if the installed database currently on their system is the same as the new database it knows about. The following rules are observed by the program:

- For each table, a test is made in the installed database for an existing table. If the table does not already exist, it is created.
- For each column in a table, a test is made in the installed database for an existing column by a given name. If it does not exist, the column is added after the last known field matched.
- A test is made to determine whether the new and installed database columns have the same basic data type. If they do not, an error message is generated for the column in question, and a resolution to the conflict will have to be applied manually.
- If the column is of type char, the longest column length is used. If the column is integer or smallint, the column type is made integer. If the column type is float or smallfloat, the type is made float. If the column type is decimal, the column remains decimal with the highest level of precision and length maintained.
- No column names are ever changed. No columns are ever dropped.
- Views are always rebuilt.
- Indices are only rebuilt if they have changed.

What Does `dbmerge` Do?

- it builds and modifies tables
- it unloads and loads tables

Unload files (<table>.unl) are created by capturing data in the existing database. There is one unload file created for each table. There is no intermediate script created to do this unload; `dbmerge` takes care of this directly. It relies on Informix's `dbload` program to load the data, but it doesn't rely on anything in order to create the `.unl` files.

The `dbmerge` Process:

1. The process begins by providing `dbmerge` with a picture of all of a product's tables (for example, `dbmerge` must have an exact description of each column in each table in A/R).
2. `dbmerge` merges an existing database with the new release database or creates a new database if there were no existing database.
3. Finally, `dbmerge` intelligently loads the data stored in the unload files into the modified or newly created tables.

How Does `dbmerge` Merge Databases?

`Dbmerge` will add tables, but not delete existing tables. It will add columns, but not delete columns. It will enlarge columns but not narrow.

The following chart provides an example of `dbmerge`'s logic:

Existing Database Column	New Release Column	Action Taken
char 30	char 30	no action taken
char 20	char 30	widen existing column to 30 char dec-
imal 10	decimal 12	widen existing col to 12
decimal 12,2	decimal 12,3	alter table to 12,3 (or 10.2) (or 9.3) (or 10.3)

Recompiling the `dbmerge` program

Typing `make` or `make.rds` in the accounting directory does not cause each `dbmerge` program to be remade. In order to recompile the `dbmerge` program you must type `make` or `make.rds` in the directory in which `dbmerge` resides.

For example, to recompile the dbmerge program for A/P change directories to `$fg/accounting/data/ap` and type `make` or `make.rds`.

When is dbmerge Used?

dbmerge is designed to be an installation-level program. The first step in installing a new version of our software is to tar off the files from media to hard disk. These files consist of unload files and programs (not tables). The next step is to invoke fg.install which calls dbmerge.

The pcdtablr Table and dbmerge

- This table contains information that defines each table in a product.
- It contains the following columns:

```
prodid          ar, ap, gl, etc.
line_no         1-n (for ordering the tables within a product)
tabname         table name
```

Example .unl file for pcdtablr:

```
ar|1|pcdtablr|
ar|2|strcntrc|
ar|3|strcustr|
ar|4|stropend|
```

dbmerge syntax:

dbmerge.4ge [-d database] [-dbs dbspace] [-s | -i | -I |-l directory | -u directory] [-log logpath]

-d database specifies the database to use (default: standard)

-dbs dbspace specifies the dbspace to build the database in.

-s tells dbmerge to update the schemas

-i tells dbmerge to (possibly drop and) build all defined indexes

-I tells dbmerge to drop and build all defined indexes

-l tells dbmerge to load data from .unl files located in the specified directory into database tables.

-u tells dbmerge to unload data from database tables into.unl files located in the specified directory. (NOTE: the unload files are named (tablename).unl where tablename is the name of the table that the unloaded data represents)

-log tells dbmerge to create the database with logging. Logpath is the fullpathname of the directory to put the logfile. (NOTE: the logpath name cannot be longer then 64 characters and is usually the current directory ('pwd').

NOTE: with the "-l" and the "-u" flags you need to specify a directory in which to find existing unload files or the directory in which to store unload files.

Merging Data

Where data tables already exist, new data may not be indiscriminately loaded over or into the new data. These are subjective decisions which will vary between local installations. "Trigger" files (`triggers.4gl`) control how `dbmerge` handles these situations. These files need to be examined and modified by the person supervising the installation.

The `dbmerge` program also loads data into, or unloads data from, the data tables. The invocation rules for the program can be viewed by typing the following command from a UNIX prompt:

```
fglgo $fg/accounting/data/all/dbmerge.4gi
```

Moving the Database to a New Location

If it becomes necessary to move the database to a new location in the file system, you must convert all direct UNIX pathnames of Informix tables (example: `$fg/accounting/data/standard.dbs/stxchrt128`) to relative path names (`stxchrt128`). If this conversion does not take place, an error similar to the following appears:

`Cannot open database table %s.`

A utility called `rmdirpath` is enclosed with base files to perform this tedious chore automatically.

The syntax is:

`$fg/accounting/bin/rmdirpath -d` *<database_name>*

NOTE: `rmdirpath` does not work when running Informix 4.0 unless you are logged in as Informix.

Exporting Activity Data

There is an option in many Fitrix Accounting modules that deletes obsolete records from data files. Deleted activity will have already been posted, and will be older than a user-specified date or period. Prior to purging such dated records, you may want to export the data to a file which can then be backed up.

The records that are deleted come from particular activity files (tables). In each case the `p_delete` program carries out the deletion of obsolete activity. The files or tables affected by the `p_delete` program will vary from module to module, and not all modules use `p_delete`.

This section of the documentation presents a sample SQL script for unloading the obsolete data prior to deletion. In addition, a listing of modules and tables affected by the `p_delete` program appears.

This sample SQL unload script backs off the activity data for the `stgtranr` table into its unload file, `stgtranr.unl`:

```
unload to "stgtranr.unl"
   select stxtranr.doc_date, stgtranr.*
   from stxtranr, stgtranr
   where stxtranr.orig_journal = stgtranr.orig_journal
     and stxtranr.doc_no = stgtranr.doc_no

    and stxtranr.doc_date  progname.4gi    # concatenates all .4go's into a
 .4gi
```

The next step would be to make a backup of the `stgtranr.unl` file.

Passing Information Between Databases

This section explains how you can pass information from database to database, within the same program, and briefly describes how to use `dbload`.

The standard method (a better technique is described below) of exchanging information is to unload the data from the first database to an unload file and then load it into the second database. This is easy to do with ISQL and dbload.

In ISQL select the query option, after having selected your database. Then execute the following command:

```
unload to "/usr/my/unloadfile/stxchrtr.unl"     select * from stxchrtr
    [optional where clause]
```

All rows are selected from table stxchrtr and written out to the indicated file in a pipe delimited format. Here are the first few lines.

```
9998|EXPENSES|MISCELLANEOUS EXPENSES|F|8|N||
9600|EXPENSES|STATE B&O RAILROAD|F|8|N|STATE TAXES|
1370|CURRENT ASSETS|PREPAID TRAVEL|A|1|N||
```

Using `Dbload`

To load the information into a database use `dbload`:

1. you need an unload file, for example `stxchrtr.unl`
2. you need a `dbload.cmd` file. This command file contains the name of the unload file, the default delimiter, the number of columns, and what to do with the data . The pipe "|" is the default delimiter.

   Here are the two lines contained in the `dbload` command file for `stxchrtr`.

   ```
   file stxchrtr.unl delimiter "|" 7;
   insert into stxchrtr ;
   ```

3. the command to run `dbload` is:

`dbload -d <dbname> -c <dbload.cmd> -l <err.out>`

```
-d database name
-c name of the the command file
-l name of file to send error output
```

Obviously unloading the data to an intermediate file and then reloading it is a labor intensive and error prone task. There are better ways of passing information from database to database. The trick is to use temporary tables.

All programs—with the exception of the specialized `dbmerge` programs—have "database standard" at the beginning of `globals.4gl`. This tells the SQL engine (the background process that handles the read and write requests) which database you'll be using. Specifying the database is important for two reasons.

The first time it's important is at compile time. Suppose you define a record as follows:

```
define
    p_stuff record like stxchrtr.*
```

And suppose you have two databases, each with an `stxchrtr` table, but the tables have different column names and data types. The asterisk at the end of `stxchrtr.*` is short hand notation for the column names. At compile time the compiler checks the database to see what columns are there so that it can expand the asterisk.

For example, when using `one.dbs`, `p_stuff` might be expanded as follows:

```
p_stuff.acct_no
p_stuff.acct_type
p_stuff.acct_desc
p_stuff.acct_cat
p_stuff.processing_seq
p_stuff.incr_with_crdt
p_stuff.subtotal_group
```

But when using `another.dbs`, the experimental database your cohorts have been working with, the asterisk might be expanded like this:

```
p_stuff.sleepy
p_stuff.doc
p_stuff.happy
p_stuff.uubob
p_stuff.freddy
p_stuff.bart
p_stuff.poop
```

If you try to insert `p_stuff.subtotal_group` into `p_stuff.sleepy` you will get unexpected results. The benefit of using this "define like" notation is that you can add a new column to the database and continue to run the same programs without making any code changes—although you do have to recompile, so that the new column is included when the asterisk is expanded again.

NOTE: Whenever you add a new column, you must recompile.

The second reason that specifying the database is important is because you might be doing accounting for two different companies. You cannot simply insert debits in one database and credits in another database and expect the sleepy account to balance with the retained earnings account.

When you specify the database in your source code and subsequently execute the program, the SQL engine automatically opens that database for use. This is why all accounting programs require that `standard.dbs` be on the system (and the CASE tools require `stores.dbs`). Even if you always use `one.dbs`, the `standard.dbs` is opened first (automatically by the engine), and then from within the program standard is closed and `one.dbs` (or whatever database was specified on the command-line) is opened for use.

All this leads us to temporary tables and to an undocumented method of exchanging data between `one.dbs` and `another.dbs`.

Temporary tables are created at run time and exist only for the duration of the program. That’s why they are called temporary tables. It does not, therefore, make sense to define a record "like" a temp table:

```
define
    p_fails record like temp freddy.*
```

It doesn't work because at compile time there is no temporary table freddy out there in the database from which to expand the * into column names. It's not there because it hasn't been created yet. Temp tables are built and maintained outside of the currently selected database, and that is the key.

Using Temporary Tables

The trick, then, is to open one database, select the information into a temporary table, close the first database, open the second database, and read the information from the temp table. Here is a simple example.

```
main
    define
        tmpstr char(15)

    # open the first database
    database first

    create temp table freddy (mycol char(15))

    # read the values from the first database
    # in this example we simply insert a new value

    insert into freddy values ('Hello, world!')

    # close the first database
    close database

    # now open the second database
    database second

    # read the information from temp table freddy and
    # insert it into tables in second.dbs
    # in this example we simply select the value and display it

    select mycol into tmpstr from freddy
    display tmpstr

end main
```

This is obviously a simple example. If all you wanted to do was pass a 15 character string you could use a variable, like `tmpstr`. But this technique can be used to read information from several databases, without being limited, for example, by pre-defined array sizes. In future releases of accounting, this temp table technique will be used to allow for reporting on several databases at once.

4

Program Design and Modification

This section of the manual covers the general design of the application code and discusses the basic techniques for modifying that code. Even though a large number of separate INFORMIX-4GL programs are joined together by the menu system to create Fitrix Accounting, each of these programs is created according to very specific rules. Once you learn the general design of these programs, you can apply that knowledge to every program in the package.

As discussed earlier, programs are first generated by CASE tools, the *Screen* and *Report* code generators. Using these code generators assures that all programs have common features and a consistent design. This section discusses the basic code as generated by these products. There are only two program templates used in the package. Once you have learned those, you have learned the vast majority of the code in Fitrix Accounting.

All menu items in Fitrix Accounting, with the exception of that which changes the company database, call INFORMIX-4GL programs. The code generators create the code for these programs and store it in a unique directory for each program. For example, the directory `i_custr.4gs` contains the source code, the

screen formats and the `makefile` for add/modify/query on the customer table. The code for all functions unique to the program are stored in this directory, but this basic code utilizes a number of program libraries stored elsewhere on the system. One library contains the "data independent" functions used in all programs. Others contain the functions and, in some cases, complete programs, shared by various programs in Fitrix Accounting.

Few programs are complete after code generation. Many program specific additions to the code must be made to get the program to perform its specific functions. The code generators create a framework for modification as well as the basic code for entering data or printing reports. After the basic code is generated, the program specific coding is done. These additions are fairly straightforward, usually involving the addition of application specific logic or mathematical operations.

What is explored here are the basic types of programs, the ways in which they are organized, and the typical ways they are modified.

The most complicated programs are those for data entry. *Screen* automatically creates all the code necessary for a fully operational data entry front-end. It requires a screen form specification file similar to the default screens created by FORM4GL of INFORMIX-4GL and the "perform" files of the INFORMIX-SQL (version 3.3). It generates the code to handle all data entry into a table from the screen or into two different tables in a "header/detail" relationship.

The generator creates data entry programs that can:

- add new documents
- update existing documents
- find rows by column queries by example
- browse through data with a screen that shows several rows at once
- view the Next and Previous rows (in the order defined)
- scroll through the detail lines in the detail window of the screen
- run operating system commands without exiting the screen
- navigate to any function or program inside or outside of the current program
- map a number of function keys that can perform a variety of functions

- create, modify, and use help text
- create, modify, and use error text
- define up to 50 new fields per document
- create free-form notes to supplement any document
- create and maintain a personal to-do list
- create on-line feature requests

The other types of programs are for reports and posting. *Report* automatically generates code needed to create an Informix report. The developer defines the report by creating a report format file which contains shorthand commands and picture layouts of a report format. The format file is named `report.ifg`. *Report* interprets and parses the WYSIWYG (What-You-See-Is-What-You-Get) style form and expands the shorthand commands to create source code, which may be immediately compiled into a working report program or to be modified before or after compilation.

The section discussing file organization covered in great detail the specific files generated by these programs and the locations of the functions they use. The following section looks at how these different files and functions fit together to make a working program.

4GL Library Functions

Since Fitrix Accounting source code uses the concepts of libraries extensively, you need to first understand our concept of a library and how it is used by the code.

There are several different kinds of libraries, but the most basic and those that play the biggest part in the functional flow of our programs are called "data independent." This INFORMIX-4GL library is a collection of pre-compiled (object) modules for frequently used functions. These functions perform specific generic tasks that usually don't rely on global variables. The purpose of a library is to place in one location the functions that are used in a variety of application programs.

Why Use Libraries?

There are three reasons for using libraries when working with INFORMIX-4GL. First, libraries allow many programs to share a set of code instead of duplicating code in different applications or parts of a single application. This makes programs smaller and more manageable.

Second, the extensive use of libraries makes changes and debugging easier. Fixing a bug in one library routine takes much less time than finding and fixing the bug as it recurs through several different programs. Using proven library routines tends to isolate program bugs to the "new" code written for that application.

Finally, once the library functions are written, they dramatically reduce development time for new applications. Other programmers can use library routines without having to know the source code. All they have to know is what a function's duties are and the syntax for calling it. Here is an idea of how effective the use of libraries can be as a method of reducing the lines of code it takes to generate an application. Currently, over 62% of the lines of code in applications consists of library routines. Of that library code, 75% is generic, used in all applications. Only 25% is application-specific library code.

Creating 4GL Libraries

Though most modification of the application is not done at the library level, you need to understand how these library routines are created in case you ever do have to modify them.

INFORMIX-4GL libraries are created in a manner similar to the creation of C language libraries. 4GL goes through an initial phase of compiling the `.4gl` files, 4GL source files, into `.ec` and `.c` files, the C language source files. The C compiler then compiles the C source files into the `.o` or object files. These object files are the files that make up the function library. Once you have created the object files, they are then archived to create the library itself. These are the steps to build a library of functions:

1. Compile the 4GL file to C using the command:

   ```
   c4gl -e filename.4gl
   ```

   Example:

   ```
   c4gl -e myfunct.4gl
   ```

   This creates `myfunct.ec` and `myfunct.c` files. The `-e` flag allows you to request just the preprocessor steps with no compilation or linking. In other words, this process converts the `.4gl` files to INFORMIX-ESQL/C and C source files, but does not go any further.

2. Compile the C files to object files using the command:

   ```
   c4gl -c filename.c
   ```

   Example:

   ```
   c4gl -c myfunct.c
   ```

   This creates `myfunct.o` object files. The `-c` flag suppresses the link edit phase of the compilation and creates an object file for that program/file.

3. Move the object files to the archive or library with the command:

   ```
   ar -ru libraryname.a object1.o object2.o
   ```

Example:

```
ar -ru ../lib.a globals.o myfunct.o input.o
```

This takes the object files `globals.o`, `myfunct.o`, and `input.o` and puts them into the library file `lib.a`. The `ar` command is used to maintain groups of files in a single archive file. It creates and updates library files as used by the linker. The `r` flag is used to replace the named files in the archive file. When the command is used with the `u` flag, only those files with modification dates later than the archive files are replaced.

4. On some systems—usually in XENIX systems, the program `ranlib` needs to be run on the archive library file to create a library index.

   ```
   ranlib libraryname.a
   ```

   Example:

   ```
   ranlib lib.a
   ```

   This creates an index for library `lib.a`. The program `ranlib` creates an index/table of contents for a library file. In SCO XENIX the index is called `__.symdef`. It is added to the beginning of the archive so the loader can locate the object file faster. After the library is created, you can use the functions simply by calling them in your `.4gl` programs. During the linking phase of these program object files, you must include the `libfile.a` file. To link your programs to a library, add the library file to your `c4gl` line using the syntax:

   ```
   c4gl program_object_files  library_files  -o  executable_file
   ```

   Example:

   ```
   c4gl program1.o program2.o program3.o lib.a -o program.4ge
   ```

The functions in the library file are then combined with your programs. The linker searches all specified libraries for the functions you call. It links only the functions that you have specified in your program. If it can't find the called function in either the object files you specify or the libraries it was told to search, the linker returns a fatal error message stating that an undefined symbol name or function name was

not found. If you have a function in the "local" code that has the same name as a library function, the linker does not override the local function with the library function. The first function that it finds is the function that is always used.

Function Library & Sample Functions

In our accounting application, we organize libraries into three different types:

1. **User Interface Functions**: These are the functions that allow the entire application to have the same look and feel. Functions in this library handle the flow of data in entry forms and report forms. They include routines such as handling SQL errors, setting system defaults, providing the data entry command line functions (add, update, delete, find, etc.), high-level validation of data, must-fill fields, and many other functions that are called by the local code.

2. **Global Application Functions**: These are functions that are called by different modules within an integrated application. These functions are identical when called by these different "modules." In Fitrix Accounting, these modules include A/R, A/P, G/L, etc. These functions include printing version numbers and logos, accessing different databases, starting transactions logs, certain mathematical calculations, routines for displaying error and help windows, posting routines, and so on.

3. **Module Specific Functions**: We have a library for each accounting module. If functions are only used in a specific module, they are placed in the module library. This library includes functions such as getting control table information, creating customer or vendor selection criteria, zooming into various tables, and so on.

The following pages contain examples from the Fitrix Accounting libraries. The sample code is included to demonstrate the use of library functions.

User Interface Functions

These functions control the basic user interface. They are the "data independent" functions that were mentioned before, which are utilized by the CASE tools in creating the basic applications. As you will see, there are sixty-six such functions that are used or are available for data entry programs, and another thirty-one that come as part of the report/posting function library. Here are some examples of the code from our basic library functions:

`ring_bang`: This function is called from a command line or "ring menu." It allows the users to execute a system call from the command line by typing an exclamation point or "bang." It is a function used by the *Screen* code generator for data entry programs. This function is automatically added to all data entry programs.

```
##########################################################################
# Copyright (C) 1990
#
# Use, modification, duplication, and/or distribution of this
# software is limited to the terms of the software agreement.
# Sccsid:  @(#)  .../scr.4gs/r_bang.4gl  1.5  Delta: 1/3/92
##########################################################################

globals "globals.4gl"

define
    arr_mesgs array[1] of record    # Message text
        mssg_text char(132)
    end record,
    mssg_prep char(1)                  # Y/null Message cursor prepared?

##########################################################################
function ring_bang()
##########################################################################
# This function provides a shell escape
#
    define
        x   char(1)

    # Trap fatal errors
    whenever error call error_handler

    if mssg_prep is null
    then
        let mssg_prep = "Y"
        #1: "System:  Enter command or [DEL] to quit"
        let arr_mesgs[1].mssg_text = fg_message("lib_scr","rbang",1)
    end if

    # intelligent bang routine
    if ok_bang() = false then return end if

    # Open the window (because we don't want to trash the heading)
    whenever error continue
    let status = 0
    open window bangwin at 2,3 with 2 rows, 76 columns
        attribute (border, white)
    whenever error call error_handler
    if status = -1143
    then
        call scr_error("recursive","hot_win")
        return
```

```
        end if

        if fg_prompt(arr_mesgs[1].mssg_text clipped, "")
          then call fg_os_exit(scratch, true) end if

        close window bangwin
        let int_flag = 0
    end function
    # ring_bang()
```

`mustfill`: This function is used to force a user to enter information into a data entry field. It is part of the user interface library, but it is not used by the code generators. It is available for your use when modifying INFORMIX-4GL code.

```
    ########################################################################
    function mustfill(fld,mssg)
    # returning true/false based upon fld=null/not null respectively
    ########################################################################
    #
        define
            fld     char(10),       # field passed
            z       smallint,       # misc number
            mssg    char(50)        # message for error statement

        # Trap fatal errors
        whenever error call error_handler

        if fld is null or fld = " "
        then
            let scratch = " The '",mssg clipped,"' field must be filled. "
            let z = length(scratch)
            call str_error(scratch, z)
            return true
        end if
        return false    # field is not null
    end function
    # mustfill()
```

Global Application Functions

Global application functions are used throughout the Fitrix Accounting package. These functions perform general kinds of tasks that are unique to the accounting system. They may or may not be data independent. In other words, some of these functions are "large-scale" enough to handle their own data flow, but many are not.

autonext: This function serves the same purpose as a serial field. It uses a control field to keep the next sequential number. This function is relatively data independent.

```
###############################################################
function auto_next(table_name,column_name)
# returns the next control number
###############################################################
# This function increments a column in a control table by 1,
# locks the column (fetch first) in the table, then builds the sql
# script to increment the column based upon column_name.  If it
# cannot obtain a lock within 5 seconds, it then pulls up a window
# with the message:
#
#    "Waiting for table: (table_name) to become available..."
#
# It then goes into a try/sleep/try/sleep loop until it can get that
# lock after which it increments the column.
#

define
    stat_flag       smallint,
    find_num        char(128),
    update_1        char(128),
    number          integer,
    table_name      char(8),
    column_name     char(14),
    tmpstr          char(512),
    count_flag      smallint,
    z               smallint,
    thekey          char(1)

    whenever error call error_handler

    if mssg_prep is null then
       #1:" Waiting for table: "
       call fg_message("lib_std","autonext",1)
         returning arr_mesgs[1].mssg_text
       #2:"to become available.."
       call fg_message("lib_std","autonext",2)
         returning arr_mesgs[2].mssg_text
       let mssg_prep = "Y"
    end if

    let stat_flag = true

    let find_num = "select ", column_name, " from ", table_name,
        " for update"

    let update_1 = "update ", table_name, " set ",
                   column_name, " = ", column_name, " + 1"
```

```
prepare select_1 from find_num
prepare update_2 from update_1

declare c_cursor cursor for select_1

whenever error continue
  # normally we don't mask errors, but in this case we expect the
  # SQL statement to fail when the table is locked by another user

open c_cursor
fetch c_cursor into number

case
      # these are the errors we might expect
  when sqlca.sqlcode = -242
  when sqlca.sqlcode = -244
  when sqlca.sqlcode = -250
  when sqlca.sqlcode = -246
  when sqlca.sqlcode = -400
  when sqlca.sqlcode = 0
      # anything else is an unknown error in which case
      # you should check the errlog
  otherwise
      let tmpstr = "Unknown error in auto_next(): sqlca.sqlcode=",
        sqlca.sqlcode
      call errorlog(tmpstr clipped)
      call error_handler()
end case

whenever error call error_handler

# if there is a problem then wait 5
if sqlca.sqlcode != 0
then
    let sqlca.sqlcode = 0
    let stat_flag = false
    open window wt_win at 2,4 with 1 rows, 60 columns
        attribute(white,border)
    let tmpstr = arr_mesgs[1].mssg_text clipped, table_name,
        arr_mesgs[2].mssg_text clipped
    let z = length(tmpstr)
    call str_display(tmpstr, z, 1, 1, "")
    sleep 5
end if

let count_flag = 0
while stat_flag = false
    whenever error continue
    open c_cursor
    fetch c_cursor into number
    whenever error call error_handler
     if sqlca.sqlcode = 0
    then
```

```
            exit while
        else
            let sqlca.sqlcode = 0
            sleep 5
             let count_flag = count_flag + 1
             # if this has gone on for more than a minute
            # ask user if they want to continue or abort
             if count_flag  15
            then
                open window aunx_win at 4,6 with 1 rows, 60 columns
                    attribute(white,border)
                prompt
                "Table still locked.  Continue trying or Abort (C/A)? "
                      for char thekey
                if thekey matches "[Aa]"
                then
                    call errorlog("user aborted program in auto_next()")
                    call error_handler()
                end if
                close window aunx_win
                 let count_flag = 0
            end if
        end if
    end while

    # we got out of the while stat_flag via exit while
    # so if stat_flag was ever set, then close the window
    if stat_flag = false
    then
        close window wt_win
    end if

    execute update_2
    if number is null then let number = 0 end if
    let number = number + 1
    close c_cursor
    return number
 end function
# auto_next()
```

Module Specific Functions

Module specific functions are those used only in a specific module of the Fitrix Accounting package. These functions also perform general kinds of tasks that are unique to that particular module. They tend to be data dependent. Most of these functions are "large-scale" enough to handle their own data flow.

apactvz: This function is from the accounts payable module. It is used to allow a user to see a specific vendor's A/P activity and "browse" through that activity, accessing it if he desires. This function is associated with its own data display form, which is also stored in its source and compiled in the library directory to which it belongs.

Modifying Fitrix Accounting with Custom Functions

Since the introduction of Fitrix Accounting 3.5, the concept of Modifiability by Design has never had more meaning. Zooms, lookups, math and simple custom logic are now easy to incorporate into programs, yet they are not panaceas. Frequently customers will request special features for Fitrix Accounting that require custom functions. When custom functions are needed, there are some simple rules that are still as valid today as they were before the *Screen* 3.0+ came out.

Rules for modifying accounting with custom functions:

1. Don't Reinvent the Wheel.

We've invented the wheel for you already. Instead of coding things yourself, try to find examples of Fitrix Accounting features that will closely replicate what you want to do. Think of Fitrix Accounting source code as an "encyclopedia of capabilities".

2. Keep the Logic, Change the Names

Once you've found a function or functions that fits your needs, it’s usually a matter of leaving the existing logic alone and just changing variable and field names.

An example best explains this concept. Let's say you are a reseller who is becoming familiar with Fitrix Accounting. A customer of yours likes using account groups throughout Fitrix Accounting. Account groups are those codes that tie together ledger accounts, so that the burden of knowing what type of transaction affects what ledger accounts is relieved from the data entry operator. Thus the data entry operator must merely remember "ARINVC" to input an invoice (instead of remembering to retrieve the sales and A/R ledger accounts).

Your customer has a lot of non-A/R cash receipts to process each period. What would be nice is if the input program Update Non-A/R Cash Receipts had an account group field so that the data entry operator could just enter, say, CASHSL and the ledger accounts for cash sales would automatically be retrieved. So in making this modification, you follow rule No. 1, Don't Reinvent the Wheel.

Since you know that in the Update General Journal program there is an account group field that would fit nicely, you can see what was done there and duplicate the capability for your customer in the Update Non-A/R Cash Receipts program.

In the trigger file for Update General Journal, you follow the path of control for the account group field and start copying whatever is pertinent to your work directory. The after_field trigger for account group calls a custom function called `fill_grp_key`. Copy this function and its call into your work directory.

Next, in the .per file for Update General Journal, you search for material that you can steal from it. Copy the lookup named `actgrp_look` into your work directory.

Next in your work directory add the function and its call to the trigger file for Update Non-A/R Cash Receipts. The function `fill_grp_key` will not work as it is, so you must customize it to fit. In doing so, you remember Rule No. 2, Keep the Logic, Change the Names

Go through and change the names to local names in `fill_grp_key`. See `globals.4gl` and your .per file for the local variable, field and screen record names. In this example, you would be changing the p_ record names, the field names and the screen record names.

Finally, enter the Form Painter and add the account group field. You may add an account group description field if you wish. Want to add a zoom to the account group field? Don't do it yourself. There is a zoom for account groups already there for you called `actgrpz`. You can add the lookup here also, but remember back in Update General Journal it had been done already and you stole the code to do it. You can exit the painter and add this lookup manually to the .per file.

Now regenerate, recompile and test. In the custom function there may be calls to functions that are not in your work directory. If so, simply comment out or delete the calls.

Don't forget there is a wealth of ideas for custom functions in the tool libraries (`$fg/lib`) and accounting libraries (`$fg/accounting/all.4gm/lib.4gs` and `$fg/accounting/`*`(module)`*`.4gm/lib.4gs`). You may traverse these directories and grep for the comments for a sampling of what's there. Typical zooms that you'll need for accounting (e.g., `actgrpz.per`) are in these library directories.

Tools to help you relate special features with source code are the Edit Current 4GL Function option on the Navigate Menu, tagging (placing the cursor on a function call in source, pressing the left bracket and having the function appear in front of you), and the Informix RDS debugger.

As you become more and more familiar with Fitrix Accounting, be on the lookout for these capabilities. You'll be surprised at the wealth of opportunities to meet customer requests right there in your source code.

Locating Functions/Displaying Function Descriptions

This section describes some utilities included with Fitrix Accounting which help you to quickly locate functions.

The Tag Utility

During compilation of source code, a database of function calls is created and stored in files named `tags`. The files are found in the local module directory and at the `$fg/lib` level. These `tags` files constitute the paths for all local and library functions called by the executable program. These tags are created by two shell scripts:

- `$fg/bin/itags` creates an INFORMIX-4GL `tags` file in the local source directory.
- `$fg/bin/litags` creates a `tags` file for 4GL libraries and merges it into `../tags`

The idea behind the creation of `tags` files is to allow you to benefit from hypertext-style mobility. If you use vi as your text editor, you can set up your system to edit a function simply by pressing one pre-defined key. For example, if the cursor is currently positioned on a word that is a function call, your pre-designated hypertext key will take you to the file that defines the function. A separate key can be set up to take you back to the departure point. You will find this to be a powerful and convenient feature of the *Screen* source code. The tags feature helps you to examine the source code in a step-by-step fashion.

In order to benefit from the tags feature, a few additions must be made to your `$HOME/.exrc` file. Set `tags` as follows:

```
set tags=tags\ ../tags\ $fg/all.4gm/tags\ $fg/lib/tags
```

The `$fg/lib/tags` is intended to point to the `tags` file in the directory containing the archived library functions. With the above line in your

`$HOME/.exrc` file, you can begin to take advantage of the power of hypertext-style mobility with source code.

To automate the process further, you will need to map keys in your `$HOME/.exrc` file.

```
map ] ^]
map [ ^^
```

NOTE: The characters `^]` represent `[CTRL]-[]]`. The characters `^^` represent `[CTRL]-[^]`. When mapping these keystroke combinations, press `[CTRL]-[v]` prior to typing in the "action" characters.

Thereafter, pressing `]` with the cursor positioned on the name of the desired function will allow you to instantly edit/view that function, wherever it happens to be defined within the current application. Likewise, the `[` key will return you to the file from which you began.

It is not necessary to map keys in your `$HOME/.exrc` file to take advantage of the tags feature. You can use the `tag` command within vi to instantly edit or view any function defined in the program source code. For example, while using vi to edit a source code file (`*.4gl`), the command:

```
:tag func_name
```

will load the source code file containing function `func_name`, with the cursor positioned on the first line of code defining function `func_name`. To return to the file from which the last `tag` command was given, simply type:

```
:e#
```

The tags utility can also be used to start vi from the command line. The command is

```
vi -t func_name
```

NOTE: Some versions of vi do not recognize the `[CTRL]-[]]` shortcut. In such cases, map the `]` keystroke as follows:

```
map ] :tag
```

Upon editing a file with vi, the `]` key will begin the command:

```
:tag func_name
```

Simply complete the command with the name of the function you wish to edit, press [ENTER], and the tags feature will load the file containing the appropriate source code.

Displaying Functions Within Programs

The following shell script displays a list of all of the function names and comments found within those functions in a specified program. Thus by running this script and specifying the name of a program, you can view a description of every function found within that program.

```
#########################################################################
 :
 # awk script to display function name and comments.  specify filename(s)
 # on command line

 awk '
 BEGIN {
     TRUE=1
     FALSE=0
     inflag=FALSE
 }

 {
     if ( inflag == TRUE && $0 ~ /^#/ ) {
         # substitute space for pound sign(s)
         gsub( /#/, " ", $0 )
         print $0
     }
     else
         inflag = FALSE
 }

 /^function/ {
     gsub( /^function */, "", $0 )
     inflag=TRUE
     printf ("\n%s\n", $0)
 }' $@

 #########################################################################
```

Here is sample output.

```
lld_input()

   returning -1 if tab pressed (next window), 0 otherwise

lld_b_field(field_name)

   This function is called from the input function before every field. The
   'prv_fld' variable contains the field we came from. The 'scr_fld' vari-
   able contains the field we're going into. Set 'nxt_fld' if you want to
   skip this field or exit input.

lld_a_field()

   This function is called after every field.

lld_a_input()

   This function is called whenever the input statement exits (except due to
   an interrupt). If you don't want the input session to end, set the
   nxt_fld variable to contain the field to be placed back into.

lld_event()

   This function is called whenever the user presses an event key. The event
   is mapped to the 'scr_funct' variable and processed here.
```

Using the `Make` Utility

You may notice that a file called "`Makefile`" appears in almost every level of directory where we store Fitrix Accounting source code. This is because we use the UNIX `make` utility to simplify the job of compiling the source code.

We have created a special set of rules for `make`, so that it knows what to compile and how to compile it at every level in the program. This is a very powerful tool to use when modifying the code.

UNIX and `Make`

Fitrix Accounting is developed entirely under the UNIX operating system (even DOS applications are developed under UNIX). Through using UNIX, it has been found that `make` (as opposed to using the Informix programmers environment) increases our flexibility and productivity.

In any 4GL application, you are dealing with hundreds of data files including source code, object code, screen, library, and executable files. Whenever you modify a source file (one ending with the extension `.per` or `.4gl`), you must remember to re-compile it. The `make` program knows what files have changed since the last time you compiled. It is designed to maintain a large number of files and knows, based on modification times, what source files need re-compilation. It then links all object files (`.o`) into the executable file (`.4ge`). This is done not only for the source code in the local directory, but for that in the libraries as well.

Location of `Makefiles`

Before we discuss how to implement `make`, you must first understand how the data files are organized within the directory structure. This has been covered earlier in the manual. To review, there are three levels of directory hierarchy: The Application directory (contains all module directories); the Module directory (contains all program directories); and the Program directory (contains all source code for one program, and the `.4ge` executable). Each directory level contains a `Makefile`, so if you invoke `make` at the program directory, your program compiles. (The `M` of

`Makefile` is uppercase so that it usually appears first in listings of files). If you invoke `make` at the module level, all programs for that module compile, and if you type `make` at the application level, all programs for all modules defined in the entire application are compiled in addition to the module library.

The `Makefiles` in the application aren't the ones that actually do the compilation. They are "id" `Makefiles` that identify the programs in their level. These "id" `Makefiles` then invoke `make` again using the "real" Makefile located elsewhere on the system.

We've adopted this two-level approach for two reasons: 1) It allows the "id" files to be very short and easy to maintain, and 2) you only have to change one Makefile (the "real" one) in order to change compiler options and other attributes. This allows you to port your application to different platforms and change only one file for all programs to compile using the different compiler. This advantage becomes more apparent when your applications contain dozens (if not hundreds) of programs.

We place the "real" `Makefiles` into a directory called `$fg/Make` (`$fg` is an environment variable set to the "FourGen directory" on the specific machine. The "real" `Makefile` isn't placed within the application directory hierarchy because all applications for a given machine call on the same Makefile.

In this directory, the "real" `Makefiles` do a variety of different things:

Application Level: The `Makefile` changes directories to each module `.4gm` directory and runs a `make` in that directory.

Library Level: It compiles a library module to its object file and, by way of the `ar` command, places it into a library archive. On XENIX systems, it runs the `ranlib` program on that library archive file.

Module Level: `Makefile` changes directories to each program `.4gs` and runs a `make` in that directory.

Program Level: It compiles all defined `.4gl` files that have a modification time different than that of the object files. It links all defined objects to the executable program.

For reports, we compile all defined `.4gl` files that have a modification time different than that of the object files. We then link all defined library directories, `$fg/lib/report.a` and `$fg/lib/standard.a`. to the executable program.

For screens, we compile all defined `.4gl` files that have modification time different than the object files. We then link all defined objects, `$fg/lib/scr.a`, `$fg/lib/standard.a`, and `$fg/lib/user_ctl.a` to the executable program.

In all of these cases, there is a "local" `Makefile` in the program directory that calls the correct "real" `Makefile` in the Make directory. In doing so, it passes a number of variables.

The variables that get passed to the `$fg/Make/`*`program`* are:

NAME - name of the executable after compilation

OBJFILES - names of the local object files to be linked

FORMS - names of the compiled perform forms

GLOBAL - name of the globals file

LIBFILES - libraries to link in

NOTE: If the `GLOBAL` file is modified, every local file will be recompiled.

Sample `Makefiles`

Below are samples of the `Makefiles` at various levels of the application.

1a. The Application Level "Local" `Makefile`:

```
APPL     =  accounting
MODULES  =  lib all gl ap ar ic oe py pu fa ma pt
all:
        @make -f $(fg)/Make/application APPL="$(APPL)" MODULES="all"
```

1b. The Application Level "Real" `Makefile`:

```
$(APPL): $(MODULES)
        @echo
        @echo APPLICATION: '$(APPL)' complete.
  .DEFAULT:
        @echo
        @echo
        @echo Making Module: $< ...
        @echo "cd $<.4gm ; $(MAKE)"
        @(test -d "$<.4gm" || exit 0;cd $<.4gm ; $(MAKE))
```

2a. The Module Level "Local" `Makefile`:

```
MODULE = gl.4gm
PROGS  = i_chart i_contrl i_genjrn i_stdent\
        o_actdet o_actsum o_balsht o_chthst\
        o_contrl o_genjrn o_income o_stdent\
        o_stdlst o_trlbal\
        p_genjrn p_genled p_new_yr\
        p_newpr1 p_newpr2 p_recalc\
        p_stdent

all:
        @make -f $(fg)/Make/module MODULE="$(MODULE)" PROGS="$(PROGS)"
```

2b. The Module Level "Real" `Makefile`: This program changes to each directory and makes the program in that directory:

```
$(MODULE): $(PROGS)
        @echo
        @echo Module: '$(MODULE)' complete.

 .DEFAULT:
        @echo
        @echo Making $< ...
```

```
        @echo "    cd $<.4gs ; $(MAKE)"
        @(test -d "$<.4gs" || exit 0;cd $<.4gs ; $(MAKE))
```

3a. The Program Level "Local" `Makefile`:

```
NAME =          i_chart.4ge
OBJFILES =      globals.o main.o midlevel.o header.o
                detail.o browse.o options.o
FORMS =         screen1.frm browse.frm
GLOBAL =        globals.4gl

all:
 @make -f $(fg)/Make/screen.fg
     NAME="$(NAME)"OBJFILES="$(OBJFILES)"     FORMS="$(FORMS)"GLOBAL="$(GLOBAL)"
$(NAME)
```

3b. The Program Level "Real" `Makefile`: This file compiles and links all source files for the chart of accounts input program:

```
CFLAGS =        -O
CC = cc $(CFLAGS) $(INCLUDE)
C4GL = c4gl $(CFLAGS)
INCLUDE = -I$(INFORMIXDIR)/incl
LIBFILES = ../../lib.4gm/lib.a $(fg)/lib/scr.a
 .SUFFIXES: .4ge .o .4gl .frm .per

$(NAME):        $(FORMS) $(OBJFILES)
        @echo "    Linking all modules for $(NAME)"
        $(C4GL) $(OBJFILES) $(LIBFILES) -o $(NAME)
        -@(chown informix * >/dev/null 2>&1 ; true)
        -@(chgrp informix * >/dev/null 2>&1 ; true)

 .per.frm:
        @echo "    Making form $*.frm"
        form4gl $* >/dev/null
        @rm -f $*.err

 .4gl.o:
        @echo "    Making program $*.4gl"
        $(C4GL) -e $*.4gl
        $(CC) -c $*.c 2>Make.err
        @rm -f $*.c $*.ec $*.err Make.err

$(OBJFILES):    $(GLOBAL)
```

These are examples of "screen" or input program `Makefiles`. A different `Make-file` is used for report/posting programs, although the only difference is the libraries that are called.

Characteristics of the `Makefiles`

Some important characteristics of these `Makefiles` are:

- They rely upon all levels of libraries.
- They compile all forms and `.4gl` programs that have changed since the last compile.
- You need change only the CFLAGS line for different compilers. The `c4gl` is invoked with the `-e` flag, so the object file is left. No linking takes place until all object files are made.
- The `.c` and `.ec` cadaver files are removed once the `.o` file is made.
- The owner and group of all files in the directory are changed to `infor-mix.` We implement this convention to avoid permission problems for users. If source code UNIX security is desired, these lines can be removed.
- If an error occurs during compilation, the errors are placed in appropriate `.err` files.
- There is a `GLOBAL` declaration that should describe the file that contains the `globals` statement. If this file is changed, the next compilation will compile all `OBJFILES.` Seasoned INFORMIX-4GL programmers should appreciate the need for this. As a convention, we place all globals in a file of their own called `globals.4gl`.

Data Entry Code Design & Modification

Data Entry Code Design Levels

Fitrix Accounting code for data entry code may be categorized into three separate levels and referred to with the terms "low level," "middle level," and "upper level."

Upper Level Functions

Upper level code includes library functions, menu logic, and flow control logic. Upper level code contains no local logic and does not reference specific databases, tables, or columns. All upper level functions begin with the name `ring_`*`function`*. Examples include:

```
ring_add()       add a new row function
ring_update()    update a screen
ring_delete()    delete the current screen (header and detail)
```

The upper level `ring_` functions, with the exception of `ring_options`, are already compiled in the `/usr/fourgen/lib/ring_lib.a` directory and are rarely if ever modified.

Middle Level Functions

If upper level functions are not modified, how do you control this function of the "command line" or "ring menu" commands? This is one of the purposes of the middle level functions. Middle level code contains other functions that make references to the database but are not usually modified, and local code not found in the library functions.

All Fitrix Accounting middle level functions have names that start with `mlh_` (header functions), `mld_` (detail functions), or `ok_` (functions that control upper level functions).

Examples of middle level functions:

```
mlh_cursor()    cursor handling package for the header screen
mld_p_clear()   clears the detail array (p_[].*) of rows

mlh_lock()     locks the head record while screen is being updated or delet
ed
mld_scroll()    the detail line scrolling logic
ok_add()        controls adding of new documents
```

The `ok` functions provide a means for controlling the pre-compiled functions of the data entry ring menu options. An `ok` function may be defined in `midlevel.4gl` for any of the ring menu commands. When that command is selected, the `ok` function is executed. If the function returns a value of "true," the menu command is executed. If the function returns a value of "false," the command is not executed.

Lower Level Functions

Lower level functions pass data between the screen records and the table data, validating data input as they do so. These functions are local code which is often modified. Low level function names begin with `llh_` or `lld_`. Examples include:

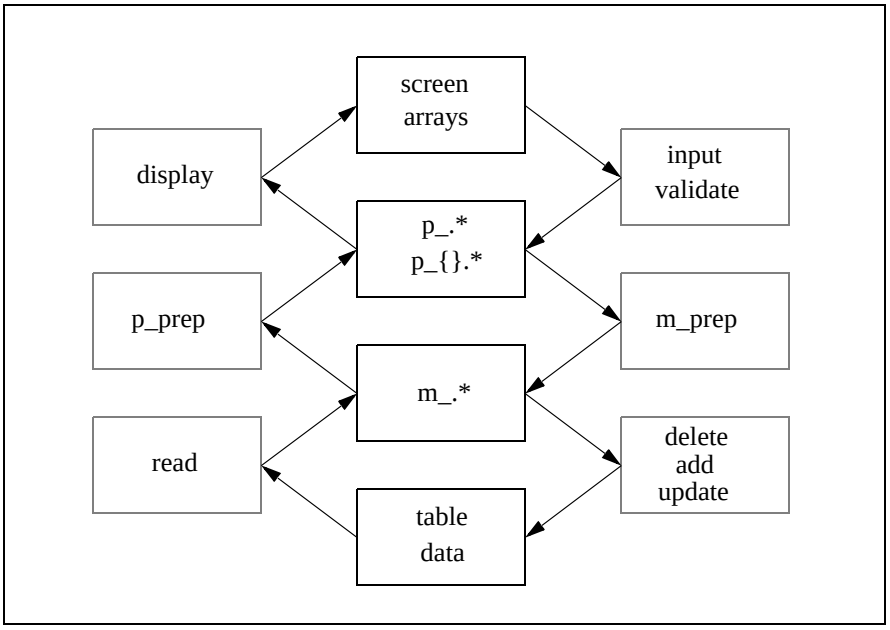
```
llh_input()     input logic for header portion of screen
lld_add()       statements required to add data to the disk
```

Low level functions act as a pipeline between the input to the screen and the data stored. At one end of the pipeline is the screen array(s) defined in the INSTRUCTIONS of the `.per` form specification file. At the other end is the data stored in the computer for the table(s) used. When data is input into the screen, it is first validated and stored in records that look like the screen records. The names of these records begin with `p_.` and end with the last six characters of the table name. If the screen contains detail lines, there is a record array beginning with `p_.[`*n*`]` (*n* is the number of detail lines on the screen).

The `m_prep` function transfers the data from the `p_.` records into records that look like the tables. These records are named with `m_.` and the last six characters of the table. Since the rows of the detail line table are transferred one-by-one, there is only one `m_.` record for detail lines instead of an array.

Similarly, the `p_prep` function transfers data in the opposite direction, filling the screen-like `p_.` records with the contents of the table-like `m_.` records whenever data is read from storage and displayed on the screen.

The flow of data input and display may be represented by the diagram below:



Special Note for Serial Columns

When `serial` type columns/fields are used in Fitrix Accounting, special precautions are taken when adding new documents. Specifically, the field value is set to `0` before the next insert command. This is done in the `llh_add` function of the `header.4gl` file.

If the serial column's value is then displayed, the value is retrieved and inserted into the `p_` record with the command:

```
let p_table.column = SQLCA.SQLERRD[2]
```

To illustrate, the `llh_add` function has the structure:

```
    whenever error continue
    insert into table values(m_table.*)
    whenever error call error_handler
end function
```

If *`table.column`* is a serial column, two lines are added so that the function is in the format below:

```
    whenever error continue
    let m_table.column = 0
    insert into table values(m_table.*)
    let p_table.column = SQLCA.SQLERRD[2]
    whenever error call error_handler
end function
```

(For more on `SQLCA.SQLERRD`, refer to the *INFORMIX-4GL User Guide*.)

Source Code Files Created by *Screen*

Here is a list of source code files and content created by *Screen*:

1. `Makefile` - This file defines make variables and then calls the "true" `make-file`, `/usr/fourgen/bin/Make.4gl`. When you type the `make` command, the `Makefile` in the current directory is used to compile any code that has been changed since that `Makefile` was last compiled.

2. `browse.4gl` - This file contains the low-level logic for operating the Browse screen. If no `browse.per` file exists at the time of compilation, `browse.4gl` will not be created.

3. `detail.4gl` - Contains the low-level code for handling the detail lines in header/detail screens. If no detail line section is included in the `.per` form specification file, `detail.4gl` is not created.

4. `globals.4gl` - Defines all global variables. Since all `.4gl` files depend on this file, changing it will cause `make` to compile all `.4gl` files regardless of the time they were last modified.

5. `header.4gl` - Contains low-level code for handling header data of both flat-file and header/detail screens.

6. `main.4gl` - Initializes variables and windows, then calls the library function `ring_header` if the screen is a flat file screen, or `ring_detail` for header/detail screens.

7. `midlevel.4gl` - Contains all midlevel code. This includes cursor control and some data validation.

8. `options.4gl` - Used to add functions to the screen which may be executed after the `Options` command is run from the data entry ring menu.

Multi-Language Handling

When adding a string to the program, the following needs to be addressed:

1. Assign it a number & put it into the database: (.iX lang_unl)

```
stxmssgr
language char(3),
mssg_module char(8),
mssg_program char(8),
mssg_number smallint,
message char(132)
```

2. Add the string to the STR record in `globals.4gl`:

```
     # Constant strings
STR record
cancel1_message char(60),  # 10 Some lines have been picked.
cancel2_message char(60)   # 11 Still cancel the order?
...
end record,
```

3. Load the string into the record in `str_init()` (`fg_funcs.4gl`)

```
####################################################################
#function str_init()
####################################################################
# This function initializes all of the constant strings from the
# language tables.
#
        let STR.cancel1_message = fg_message("oe","i_order",10)
        let STR.cancel2_message = fg_message("oe","i_order",11)
...

end function
# str_init()
```

Then, wherever program strings are displayed, the variable STR record is used instead:

Old string usage:

```
prompt "Are backorders allowed ?" for char yesno
```

Now string usage:

```
prompt STR.bo_allowed for char yesno
```

This makes it easy to translate the program to multiple languages. Using this technique, the translation occurs in the unload data (keyed to the global 'language' variable).

For controlling strings (like stages), the DATA is always stored in english. The SCREEN will display the localized string. This means that the p_* variable will contain the localized string, and the m_* variable will contain the translated string. Functions are used to translate to/from english & the local language.

Transaction Processing

A Short Discussion of Data I/O

The cursor defined by various Fitrix Accounting programs to access table data contains only the `row_id` plus any columns needed by the "order by" clause for sorting purposes. This cursor serves merely as an index to the rows in the current "active set." When the cursor is defined, the program counts the rows in the active set and displays this number on the screen. All subsequent fetches use the `row_id` to "fetch absolute."

Upper level code decides which absolute row is needed from the cursor at any given time. Whenever data is loaded into program variables, it is fetched directly from the disk with the `row_id` rather than using the open cursor. This insures that all data displayed on the screen is as current as possible.

Detail line rows are loaded into a program array which contains all rows associated with the current header row. The array size (defined by the number entered into the *Screen* screen) must be large enough to contain all these detail rows. When one detail row is being updated, all rows fitting the "where" clause are loaded into the program array. When the user presses the "record" key to store data, all rows matching the where clause are deleted from disk and replaced by the contents of the program array.

Locking Tables and Rows

Fitrix Accounting includes record locking techniques to prevent simultaneous access of data on multi-user systems. If you are making substantial modifications to the code, it is important that you know the rules that the programs follow in locking tables and rows.

Data-Entry Programs

In the case of data-entry programs, our code implements record locking in only two situations:

1. From the time the user selects the Update command of the data entry ring menu to the time that the data has been written to disk.

2. From the time the user enters a Y to confirm data deletion until the time the data is actually deleted from the disk.

Records for data-entry programs are locked by defining an "update" cursor which contains the row_id of the current header row. The cursor is defined, opened, and the row_id is fetched to cause the lock. The cursor is closed to release the lock once changes are written to disk.

Detail rows are not individually locked; the program assumes that they are locked whenever the associated header row is locked. The following rules apply to locking rows in Fitrix Accounting data-entry program code:

1. Lock rows; do not lock tables.

2. Rows being updated on the screen must be locked until a disk write is completed or the update is canceled.

3. When Detail Line rows are being updated on a screen, the header row should be locked, but do not lock the detail rows.

4. Any column used for a unique index should not be filled with nulls. While it is not illegal, nulls in columns used for ordinary indices is discouraged. It is a good idea to check for "not null" when inputting, and to not allow a disk write with a key column value of "null."

Posting Programs

Posting programs do not operate by the same principle of opening an update cursor. Instead, they automatically lock every row that is being updated. The locks are maintained until the transaction is completed (as indicated by the `commit work` command). For more information on the posting process, see "Using Database Transactions for Posting" in section 3.

The following rules apply to locking rows for posting programs in Fitrix:

1. The header row must be locked during any process that "posts" data to another table.
2. Any column used for a unique index should not be filled with nulls. While it is not illegal, nulls in columns used for ordinary indices is discouraged. It is a good idea to check for "not null" when inputting, and to not allow a disk write with a key column value of "null."

The `i_lock` Program - Order Entry

Programs that post a large number of records to many files may pose a "locking" problem to your UNIX system. For some systems, the number of locks required by one posting program may exceed 80% of normal capacity. Under such circumstances, it is preferable to have some of the tables locked during posting. This avoids having to lock individual records.

In particular, the Order Entry posting program `p_pstord.4ge` can require a large number of locks. To deal with this potential problem, the Order Entry module uses a program called `i_lock` to control table locking. The program updates 20 pairs of fields in the `stocntrc` table; each pair corresponds to a table name affected by Order Entry posting, and locking instructions for that table. The instructions can be one of the following: N (do not lock), T (try to lock), or Y (must lock). If the posting program `p_pstord.4ge` cannot lock a table that must be locked, it will terminate without posting. If it cannot lock a table that it must try to lock, it will proceed and lock individual records as needed. For tables that are not to be locked, the "N" instruction is the functional equivalent of a comment or note, having no other effect. Tables are unlocked when posting is complete.

The program `i_lock` resides in the `$fg/accounting/oe.4gm` directory, and is called up from the Setup Order Entry Menu by typing the letter `o`. It is not displayed as a menu option. In a sense, it is a "hidden" menu option.

The i_lock programs in Order Entry and Purchasing can easily be copied and used wherever you need a table locking program.

The following table lists the probable impact of locking tables used by Order Entry posting:

Table 3: Effect of Locking Tables Used by `p_pstord`

Table	Effect
`stxtranr`	one lock per order prevents all other posting
`stitranr`	one lock per line prevents Inventory Control posting
`stiactvd`	one lock per line prevents Inventory Control posting
`stilocar`	one lock per line prevents Inventory Control posting
`stgtranr`	one lock per order prevents all other posting
`stgactvd`	one to six locks per order, three per line prevents all other posting
`strtranr`	zero or one lock per order prevents A/R and Cash Receipts posting
`stractvd`	zero or one lock per order prevents A/R and Cash Receipts posting
`stoordre`	one lock per order prevents addition/maintenance of customer orders
`stoordrd`	one lock per line prevents addition/maintenance of customer orders
`stropend`	zero or one lock per order prevents A/R and Cash Receipts posting

Table 3: Effect of Locking Tables Used by `p_pstord`

Table	Effect
`strcustr`	zero or one lock per order prevents A/R and Cash Receipts posting prevents adding/updating of customers

The Data Entry Form

A data entry program is driven by the 4GL code, but it uses the "Perform" screen. The screen form specification file is defined in the same directory as the program or, in the case of application-wide functions, the directory that holds the functions. For example, the program in Accounts Payable for entering cash is in the directory `.../accounting/ap.4gm/i_cashe.4gs,` which contains the 4gl code for handling the data entry function. The customer information Zoom function that allows the user to see all of the customers has its own screen in the A/P applications library (`$fg/accounting/ap.4gm/lib.4gs`).

The Informix screen form is a text file which may be created with any text editor that creates ASCII files. These forms can also be easily created using the *Screen* Form Painter CASE tool, which allows you to "paint" your screen image. Fitrix Accounting screens are defined exactly as standard Informix screen forms (`*.per` files) with the following restrictions:

- You must define a screen record array for the file.
- The screen record array must be the first one defined in the `INSTRUCTIONS` portion of the file.
- The screen record array must list the columns to include; do not use `TABLE.*` or `table.column THRU table.column`.
- The first table listed in the `TABLES` section must be the one containing the "header" columns of the screen.
- Do not include lines longer than 76 characters (the screen border graphic consumes four characters of each line).
- Do not include more than 18 lines for the screen form.
- Do not include tabs in the screen form file.

A different sort of screen form may be created to display files in a header/detail relationship. When defining a screen form with detail lines, the following additional restrictions apply:

- The table containing the columns for the "detail" lines must be the second table listed in the `TABLES` section.

- You must define a record array for the detail line table.
- The detail screen record array must be the second one listed in the `INSTRUCTIONS` section.
- The detail record name must be followed by a subscript that defines the number of detail lines on the screen (e.g. `SCREEN RECORD ordr[6]` for a customer screen that contains six order detail lines).
- The screen record array must list the columns to include; do not use `TABLE.*` or `table.column THRU table.column`.

The screen must contain each of the sections on the sample and they must be in the order shown. These sections are:

- **DATABASE** - the name of the database.
- **SCREEN** - a visual picture of the screen, a layout of headings, field tags, and delimiters.
- **TABLES** - a list of the tables used for the screen (the "header" table must be listed first).
- **ATTRIBUTES** - field tags mapped to fields and "attributes".
- **INSTRUCTIONS** - must contain the screen record array and that array must be the first one defined (other instructions may be included).
- **FOURGEN**- this section contains special definitions used by the *Screen* Code Generator.

Details on the first five sections and the code that may be used within them are contained in the *INFORMIX-4GL* manuals. The FOURGEN section is explained in the "Code Generator" section of the *Screen Technical Reference Manual*. For descriptions of the different types of perform screens, refer to the *Screen Technical Reference Manual*.

Converting INFORMIX-SQL Perform Files

For those integrating Fitrix Accounting with existing applications written in INFORMIX-SQL you may want to convert your existing applications to the most current version of INFORMIX-4GL by using our code generators, in order to be more consistent with the Fitrix Accounting data entry environment. There are some minor differences between the screen forms. They can be easily converted to INFORMIX-4GL by using *Screen* on the perform screens. The code created by Screen effectively replaces the Perform screen interface with the *Screen* ring menu interface. Many perform commands are not recognized by INFORMIX-4GL, however, so if you have defined many additional instructions in the perform form, you will have to add functions to the *Screen* code to achieve the same effects.

All statements of the Instructions section of a perform screen, with the exception of the `delimiters` command, are ignored.

4GL will not accept form definitions containing more than one screen. If your perform file contains multiple screen definitions these must be removed.

Joins defined in the screen form are ignored.

The `LOOKUP`, `NOUPDATE`, `QUERYCLEAR`, `RIGHT`, and `ZEROFILL` attributes of perform screens have no meaning in 4GL.

INFORMIX-4GL does not use the concept of "current table." An INPUT or INPUT ARRAY statement allows you to enter data into fields that correspond to different tables.

Adding Fields to Data Entry Forms

One common modification is the need to add a new field to a form. Fields can be added several ways:

1. via the Form Painter, then regenerating with the code generator.
2. to the perform file, then regenerating with the code generator.
3. to the perform file, then modifying all necessary files.

The various methods for making this type of modification differ greatly in degrees of complexity. If you have the *Screen* Form Painter, adding fields to your screens is extremely easy. If you do not have *Screen* adding fields can be considerably more complicated.

If for some reason your code is not regenerable, then you will need to use the third option and manually add the necessary code.

Refer to your *Screen* documentation for examples of adding fields to your screens with the Form Painter.

RDS vs. 4GL: Choosing an Informix Environment

If you are familiar with INFORMIX-4GL, you are familiar with the Informix Rapid Development System (RDS). The product is quite similar to 4GL. The corresponding books are identical except for the introduction and the sections that describe how to compile and run 4GL and how to include C functions. RDS is interpretive; that is, if you have written 4GL programs, RDS will run them without compiling. Informix claims that 4GL and RDS are completely compatible.

What Are the Advantages of an Interpreter?

The two major advantages deal with compile time and disk space. RDS actually does have a compiler. When you have created your 4GL source file, you can compile it from the programmer's environment (`rfgl`), or you can compile it from the command line using `fglpc`. The latter command runs in about two seconds and creates a pseudo-code file with an extension of `.4go`. This pseudo-code is portable across any machine that runs RDS and it does *not* require the C development system to compile and run.

You then invoke the RDS "runner" program, `fglgo`, on the `.4go` file. If you broke your source code into several modules, there is an extra step. You compile all your source files and then concatenate the resulting `.4go` files into one `.4gi` file. You can then use `fglgo` on the `.4gi` file. Here is an example:

```
fglpc *.4gl # creates a .4go for each .4gl
cat *.4go > progname.4gi# concatenates all .4go's into7 a .4gi
fglgo progname.4gi# executes the RDS runner
```

This whole process takes a couple of seconds. The resulting `.4gi` file is just a fraction of the size of the corresponding compiled `.4ge` executable. Both compile time and disk space savings are impressive.

Testing has shown that space savings is approximately three to four times when using RDS.

Other Advantages of RDS

RDS allows users the ability to run the INFORMIX-4GL Interactive Debugger. If you have experience with an interactive debugger with COBOL, assembler, or C, you will probably like this one.

The debugger is run in the same fashion as RDS, except the "runner" name is `fgldb.` This command looks the same as `fglgo` used with RDS, displaying the same information on your screen.

At any time you can flip over to the debugger screen. This screen is split into an upper and lower window: one for viewing your source code, and the other for entering the rich set of debugger commands.

You can set traces and break points. You can execute your code one line at a time. You can view any or all of your local and global variables. You can set actions to take place at break points. If your program ends with a fatal error, the debugger screen automatically flips up and highlights the line of source code where the error occurred. In effect, you are supplied with the corpse and the tools with which to perform an autopsy (i.e. values of variables at time of death, list of functions that lead up to demise, etc.).

One of the more valuable functions of a debugger is to allow someone who is new to a language to analyze just what occurs as commands are executed. Another is to allow you to trace through an unfamiliar program to learn how it works so you can maintain it.

Drawbacks of RDS

One drawback is relatively minor: `fglgo` is slower in execution than the compiled executable program. In trials, it could be as much as 40% slower. Of course, when developing software, execution speed is not important. Once the program has been written and debugged with RDS, it can be compiled with 4GL for production. However, RDS is not objectionably slow, even for production use. For complex applications with a large number of programs, the savings in disk space may be worth the trade-off in execution speed.

The drawback considered to be the most severe is that libraries are not supported. This is especially a problem for a shop that develops modular products where consistency of "look and feel" across products is particularly important. In our company's development environment, the products developed are complex, and there are frequently several people working on different programs in the same or companion applications. We depend heavily on functions in libraries to keep features in programs consistent.

All source (including functions common to more than one program) must be compiled into pseudo-code, and all pseudo-code must be concatenated into one file for `fglgo/fgldb` to execute. In short, RDS lacks a linker. There is no linker provided with 4GL either, but it isn't missed since UNIX provides one.

The programmer's environment provided with 4GL (and RDS) is hopelessly inadequate for team development of large applications. We depend on the many utilities in UNIX, such as `make` and `sccs` (source code control system) to control our Informix development. All these utilities work fine with RDS except for `make`, because of the lack of a linker.

For this reason, we provide an RDS linker with both our accounting system and our code generation tools. We also offer a special "make" program (`make.rds`) for using libraries.

Using the RDS Linker

The RDS Linker program is written in UNIX shell scripts and replicates many of the functions of `make`. Although it is not quite as fast as the C compiler `make`, `make.rds` allows you to maintain libraries and links `.4go` object modules from local code with functions from the libraries into RDS `.4gi` p-code modules.

The `make.rds` program uses variables from the `Makefile`s created for make by *Screen*. `Make.rds` uses standard UNIX commands to determine the modules that have been updated since the last RDS compile. It uses UNIX sed and grep commands to gather information from the `.4gl` files. Although this technique does not give `make.rds` the full power and intelligence of `make`, the former may be run in the same manner with the *Screen* `Makefile` files as long as you follow a few simple guidelines.

Rules for Using `make.rds`

1. **All existing `.4gl`** library files must be converted to RDS libraries. The libraries are converted by changing to the directory containing the library source files and typing `make.rds`. The instructions for doing this for the standard libraries of Fitrix Accounting modules are included with the RDS-Linker installation instructions, and are usually performed at the time of installation.
2. **Type** `make.rds` rather than `make` in the program source directory.

Syntax for `make.rds`

The `make.rds` utility replaces the functionality of `make` for use with Informix-RDS (Rapid Development System). It also calls the trigger merge utility in `$fg/codegen/scr.4gm/trigger.4ge` if it is present on the system. The default is for the merge utility to be run.

It reads a `Makefile` in the format of a `Makefile` (one that calls other makes within the `$fg/Make` directory).

It reacts differently based upon which make is called in the `$fg/Make` directory.

Syntax:

make.rds [-l] [-f] [-m] [-mo] [-mf] [*application_name*] [*module_name*] [*library_name*] [*program_name*]

Five options can be used:

link_only: The **-l** flag causes link_only to be set to "Y". Alternatively, link_only can be set to "Y" and exported in the user's environment. When link_only is set to "Y", make.rds skips the fglpc and form4gl parts (i.e., compilation of 4gl files and form files is skipped) and only runs the link part of the make.rds suite of shell scripts. make.rds run in link_only mode always rebuilds the filelist.RDS in the local program directory. also, if a library has been compiled, make.rds run in link_only mode in the local program directory will rebuild depend.RDS, func_map.RDS, and unresolved.RDS in the library. If this option is used the merge utility is not run.

fast_link: The **-f** flag causes fast_link to be set to "Y". Alternatively, fast_link can be set to "Y" and exported in the user's environment. When fast_link is set to "Y", make.rds does not call link.rds to search the 4gl files for function names. It assumes the that a file called "filelist.RDS" in the source directory exists; this is a current list of the files containing all the external library functions. filelist.RDS is created by link.rds. make.rds run in fast_link mode simply cats all the required ~.4go files from libraries as well as the local program directory into the ~.4gi executable. The fast_link will run the merge utility.

merge: The **-m** flag allows control over how the merge utility is run. If the user desires that the make.rds be run without merging of trigger files into 4gl source the -mn can be specified. The environment variable no_merge can be set to "Y" and exported. This will cause the merge utility to never be run. If a merge without compilation is desired the **-mo** flag may be used. If the variable merge_only is set to "Y" and exported to the environment no compilation or linking will ever occur. The time stamping in the merge utility will cause no merge to occur if any 4gl file is newer than its associated trigger file. The **-mf** option can be used to force a merge and override the time stamp comparison logic. The force_merge variable can be set to "Y" to make a "forced merge" the default.

make.rds can be run on several levels: application, module, library, and program.

You can pass `make.rds` arguments. On the application level, you can pass a list of `~.4gm` directories. On the module level you can pass a list of `~.4gs` directories. On the library level, you can pass a list of library functions. On the program level, you can pass a list of `.4gl` files to compile. On the program level, you can pass a list of `.pers` to compile

How to Avoid Recompiling the Entire RDS Library

If you are modifying a library function in RDS, you can avoid the lengthy process of recompiling the entire library provided your modification did not add or delete a function. The procedure is as follows:

1. Modify your function (for this example, we will call the function "`logo.4gl`") with your text editor.
2. Compile it to `p_code` by using `$INFORMIXDIR/bin/fglpc`.

   Example: `fglpc logo.4gl`
3. Move the `logo.4go` file (created by step number 2) to the RDS library.

   Example: `mv logo.4go ../lib.RDS`
4. Copy the function's source code file to the RDS library.

   Example: `cp logo.4gl ../lib.RDS`
5. Change directories so that you are in a program directory that calls the function you have just modified.

   Example: `cd $fg/accounting/ap.4gm/i_vendr.4gs`
6. Update the last modification time of the `globals.4gl` file so that the compiler program will know to recompile all.

   Example: `touch globals.4gl`
7. Finally, recompile this program so as to relink it to the RDS library functions.

   Example: `make.rds`

Converting Libraries to `make.rds`

You may convert any C compiler library to a `make.rds` library simply by typing the `make.rds` command in directory containing the library source files. The instructions for converting the standard libraries used by INFORMIX-4GL products are included with the installation instructions and are usually performed during the installation process. The *Screen* library files are converted by changing to the `$fg/lib/scr.4gs` directory and typing **`make.rds [ENTER]`**.

The `make.rds` library conversion process acquires the name of the *`library`*`.a` archive file from the `LIB` variable set in the `Makefile` of the current directory. It uses this name and creates a directory of the same name with the extension `.RDS` replacing the `.a` extension. The library directory for *Screen*, for example, is `$fg/lib/scr.RDS`. The `.RDS` directory is the functional equivalent of the `.a` archive file.

Next the `make.rds` process then copies library `.4gl` files into the *`library`*`.RDS` directory. Copying the source files to the `.RDS` directory allows make.rds to construct paths of functions called by other functions. It also allows you to use the INFORMIX-4GL Interactive Debugger with only one directory pathname in the `USE` statement, rather than a list of all library source directory pathnames.

The `make.rds` process was created with the idea that libraries are largely static. Whenever new library functions are compiled, the system must search functions recursively for nested function calls and construct a path of functions called by other functions. Since this path is determined and constructed by shell scripts, this process can take some time. However, once completed, the path-building process for a function library does not need to be repeated for subsequent program compiles.

If you change the directory paths to any library, you must re-make all libraries. To re-make the libraries:

1. remove the `$fg/Make/*.RDS` file(s)
2. remove all *`library`*`.RDS` directories (and their contents)
3. run `make.rds` in each library source directory again

Code Conventions for `make.rds`

The `make.rds` system expects INFORMIX-4GL code to follow two conventions that are not required by INFORMIX-4GL itself:

1. The *`function_name`* of `whenever error call` *`function_name`* lines must be the last word on the line.
2. The *`report_name`* of `start report` *`report_name`* lines must be the last word on the line.

For example, the lines below correctly follow conventions and can be properly linked by `make.rds`:

```
whenever error call error_handler
start report my_report
```

The following lines do not follow conventions and cannot be properly linked by make.rds:

```
whenever error call error_handler let a=3
(name of the function is not the last word of the line)

 start report my_report output to screen
(name of report is not the last word of the line)
```

Understanding Program Linkers

This section is intended to address the background behind the program linking phase, and why you might be getting the following informix RDS error:

```
The function "such_and_such" has not been defined in any module in the pro-
gram.
```

The error will have a different disguise under compiled 4GL. It might look something like:

```
Undefined symbols: such_and_such in file this_and_that
```

The reason you would get the above error is because the program calls on a function `such_and_such()`, but that function isn't in the executable (.4ge) or p-code program (.4gi).

There are two reasons the `such_and_such()` function may not have been "linked" into your program:

The less common reason is that you spelled it incorrectly: `call initt()`. This is easy to identify and fix.

The more common reason (and the one that requires an understanding of the linking phase to fix) is that the program "linker" couldn't find the function to link into the program. You might say: Well it's right here in this library. It could find other files in that library, why couldn't it find that one?

In order to answer that question, you have to understand how "C" linkers work, and understand the fact that there are different linkers for different unix flavors. The end result of this exercise is to understand the 'least common denominator' so you will write programs that are portable to other machines.

First, a C executable is little more than just a bunch of .o files concatenated together into one executable. A .o file is just a machine representation of the source .c file.

RDS works very similar to C in that a p-code program (.4gi) is nothing more than a concatenation of several p-code modules (.4go), and p-code modules are simply compilations of .4gl source code.

The program linker (`ld` for C, `make.rds` for RDS) is given the following pieces of information:

1. A list of object files in the local directory.
2. A list of library files to obtain functions that are called from the object files (#1 above).

Example:

```
main.o midlevel.o header.o ../lib.a /usr/four-
gen/lib/scr.a
```

In the above example, a function located in `midlevel.o` may call upon a function called `absolute()` that isn't in `midlevel.o`.

The linker will first determine if it's in `main.o`, `midlevel.o`, or `header.o`. If it doesn't find the `absolute()` function in either of these files, it looks *first* into the library `../lib.a`. (a library is just a group of object (.o) files). If it's not in `../lib.a`, it will *then* look into the library `/usr/fourgen/lib/scr.a`.

If it doesn't find it in any of the local files, or any of the library files listed, it will give you the error that we are talking about.

That's pretty straightforward.

The saga continues however, when the `absolute()` function requires *another* function that isn't in the `absolute.o` object file. That other function may require yet another function. You can see that this procedure is recursive in nature—it can get very deep. By simply calling the `absolute()` function, now your linker must know to load in *several* other object files from potentially several different libraries. Now multiply that by the number of calls to library functions in any given program.

You can see how much work is involved in linking a program together. The linker must, on the fly, build this function dependency information.

Now back to the problem of the linker not being able to find your function.

Let's take the same example:

List of object files and libraries: `main.o midlevel.o header.o ../lib.a /usr/fourgen/lib/scr.a`

A function in `midlevel.o` calls `absolute()` which is located in the `/usr/fourgen/lib/scr.a` library. Lets say also that `absolute()` calls on a `compare()` function. If the first request for `compare()` is from `absolute()`, and if `compare()` is located in a library that has already been processed (`../lib.a` in this `example()`, then the linker will not find the `compare()` function.

Some linkers are smarter than others. This becomes a portability problem. The IBM RS6000 linker, for example, is very smart. It will find the `compare()` function and link it into the program.

When building the `link.rds` program (called from `make.rds`), we had to mimic the most brain dead linkers in UNIX. This isn't the most desirable, but it is what we had to do. This is so we will design our libraries to be portable to all versions of UNIX.

There are two ways of fixing the above problem. First, if you are using RDS and the `make.rds` family of programs, you can go into the `$fg/bin/link.rds` program and remove the 'break' at (around) line 172. This will un-lobotomize the RDS linker so it will act like the smartest linkers in unix (like the RS/6000). This will work, but the result will be a library layout that may not be portable to other versions of unix (by convention, our developers may not use this method).

The second (and most portable) method to fix the problem is to either move the `absolute()` function up to the `../lib.a` library, or move the `compare()` function down to the `/usr/fourgen/lib/scr.a` library (i.e.: rearrange the files in your libraries).

An example of a real-life scenario:

The error message when running the program under RDS is:

```
The function "logo" has not been defined in any module in the program.
```

The following is the list of files and libraries used to create the program:

```
main.o report.o midlevel.o lowlevel.o ../../lib.4gm/lib.a $fg/lib/report.a
```

The problem was that the `logo` function was called from a function that was in `report.a`, but the logo file was located in `../../all.4gm/lib.a` (a library above `$fg/lib/report.a`).

The solution would be to move the `logo` function into the local directory (so it is assured to be loaded), or to move it down to the `$fg/lib/report.a` library.

Since neither of these methods was very satisfactory in this case, (we wanted `logo` to remain in `../../lib.4gm/lib.a`) we decided to call it from another function that we're using in `../../lib.4gm/lib.a`.

The following logic is in the `st_init()` function:

```
if false
  then call logo()
end if
```

This will never be run, but it is enough to fool the linker into knowing that `logo()` is needed. Since `st_init()` is in `../../lib.4gm/lib.a`, the linker found the `logo.a` file because it also resided in the `../../lib.4gm/lib.a` library.

Shell Escapes While Running Fitrix Accounting Applications

When you run any program in Fitrix Accounting, the directory that contains the source code for that program becomes the current directory. This assumes, of course, that you are running it on a system that contains source code. To place yourself in the directory with that program source code, you need only "escape" from the program command line or document command line—this is the command line at the top of the screen that contains the commands for adding, updating, and similar document level functions.

When you are at the "command line" level, simply type:

```
!sh[ENTER]
```

This would run the shell program in the current directory containing the source code for that particular program. The exclamation mark tells Fitrix Accounting that you want to run a system program rather than perform any document access functions. The `sh` tells it that you want to run the shell program.

You can also load, edit, and recompile programs without escaping to the shell. For example, if you wanted to edit the menu that is currently on the screen, you would type:

```
!vi menu[ENTER]
```

If you are in a data entry program, you could edit a screen that is part of that data entry application by typing at the command line:

```
!vi screen1.per
```

Both of these commands load the source for current menu or specified data entry screen into the "vi" editor. There are many files called "menu" and `screen1.per` on the system, but the program knows which one you want by where you are in the application. This is an extremely useful feature, virtually eliminating the need to climb up and down menu trees every time you want to find the appropriate file.

If you have the User Control Library installed on your system then you can also access a shell while running an application by pressing `[CTRL]-[o]`. The `[CTRL]-[o]` sequence has been mapped to a system level prompt and operates similar to the exclamation mark.

Another utility that is helpful when developing or debugging applications is `$ifx`. If you call up a shell after running a program and returning to the menu level, typing:

```
cd $ifx
```

will automatically change directories to the local directory containing the last program run.

Preventing Shell Escapes

You may prevent users from access to the shell by setting the `$SHELL` variable to `false`.

If the following variable is set before loading the *Screen* program, system commands can not be executed.

```
SHELL=false; export SHELL
```

5

System Administration Issues

Setting Up Your Printers

Selecting the "redirect" option from the ring menu when printing a report gives you a menu of printers to select from. This menu is generated in the following way. First, Menus looks for the system variable `$printerlist` which if it exists, points to a file that lists the available printers. For example, `$printerlist` could be set to `/u/account1/printers` for users A, B, and C. When those users select "redirect" while printing they will be given a choice of printers taken from the file `/u/account/printers`. In addition, the first printer listed in the file is used as the default printer for those users. The format of the `printerlist` file is one printer per line. The first printer listed is the default destination.

If Menus cannot find a value for `$printerlist`, it next looks in the directory `$mzcontrol` (`$fg/menueze/control`) for a file called `printerlist`. If this file exists it will be used as described above. If not, Menus will try to automatically generate a temporary `printerlist` file in the `/tmp` directory. To automatically generate a temporary `printerlist` file, Menus searches

`/usr/spool/lp/default` for the default printer (first line of the temporary printerlist file) and `/usr/spool/lp/request` for a list of other spool directories, one for each printer defined to lp. If no `/usr/spool/lp/default` file exists Menus will try using `/etc/qconfig` to generate the temporary `printerlist` (IBM AIX spooler definition file). If Menus cannot successfully create the temporary `printerlist` file from either of the two sources (lp spooler or AIX spooler) it just creates a default temporary `printerlist` with one line, "printer." The temporary printerlist is used just like the `printerlist` described above.

Menus tells the spooler what printer to use by setting the system variable `$LPDEST` to the name of the printer. The lp spooler uses the value of `$LPDEST` to determine the destination. The invocation used by other spoolers must explicitly include `$LPDEST` in order to correctly recognize the destination requested by the user. For example, the `$spooler` variable with AIX is set to **`print $LPDEST -cp`** so that when the spooler is called by Menus it includes the printer name requested by the user.

Finally, the system variable `$lpflag` is used to flag the number of copies to print. For example, with "lp" the way to print 3 copies of a file is to use the flag `-n3` on the command line. When using `lp`, the `$lpflag` should be set to `-n`. With the AIX spooler, you use `-nc=3` which means that the `$lpflag` must be set to `-nc=`. The default values for `$spooler` and `$lpflag` are compiled into Menus when it is ported to a particular platform so they should be set correctly by default. For more information about setting up printers refer to the *-Menus User Reference Manual*.

System Performance Issues - RDS Profiler

This section explains how to build an RDS Profiler using the Informix RDS debugger. A profiler tells you how much time your program is spending in each function. This allows you to locate problem areas which might be running too slowly.

A Typical Problem

Our in-house users were complaining that the order entry program was too slow. Upon inspection we discovered that it occasionally took more than a minute to save an order. This was on the in-house system where the application was actually being used. On the development machine it took only 5 seconds.

Even though the in-house system had more users and fewer resources, the one minute performance time was an indication that something was not right.

Our first step in finding a solution to this problem was to duplicate the in-house database on the development machine (the dev box). We quickly proved that the problem did not stem from the size of the database. Testing the transaction on the dev machine with the same large database yielded the same results. It still only took 5 seconds to save an order.

The natural tendency is to suspect inadequate or mis-managed system resources. However we looked deeper into the problem. Was the problem due to insufficient memory, inadequate disk performance, or what exactly? We needed to know where, specifically, the programs were bogging down.

Our first approach was to count the number of times that each function was executed. We determined this by filtering the "trace functions" output from the RDS debugger through an AWK program.

```
trace functions >> trace.out
tracecount.awk trace.out > output
```

The trace functions command tells what function was called, where it was called from, the arguments passed, and the results (if any) returned. Here is part of trace.out

```
Enter startlog("errlog")  from main line 23
Return
from startlog
Enter put_scrlib(var = "version   ",
value = " 3.70.03 ") from main line 25
Return
from put_scrlib at line 72
Enter put_scrlib(var = "dbname    ", value = "
standard ") from main line 26
Return
from put_scrlib at line 72
```

If `my_func()` is called ten times then there will be ten lines with "Enter `my_func`" in `trace.out`. Thus the design of the awk program was straightforward: count the number of times each function is entered and print the results.

```
##########################################################################
# Copyright (C) 1991
# <Your Company Name>
# Use, modification, duplication, and/or distribution of this
# software is limited to the terms of the software agreement.
##########################################################################
# This script reads a file produced by fgldb when trace
# functions was turned on.  For example:
#          trace functions >> /tmp/trace.out
# it then produces a list as follows, indicating how many times a
# given function was called:
#
#       1 ask_yn
#       1 ok_next
#       2 str_init
#       7 llh_display
#       7 set_count
#       9 window_size
#      74 ring_fetch
#     110 brw_display
#     365 get_scrlib

awk '
    /^Enter / {              # read only lines with Enter at start
        sub( /Enter /, "", $0 ) # remove Enter
        sub( /(.*$/,  "", $0 ) # remove left paren and everything after
        string[$0]++            # count the remaining function name
    }

    END { for ( w in string )   # print out count
```

```
            printf("%4d %s
", string[w], w)
     }' $1 | sort               # and sort it
```

The results of `tracecount.awk` are interesting but not very informative. We should probably have added the heading: Wild Goose Chase. We have more information, but not the information we need. If function Fast executes in one-thousandth of a second and function Slow takes 12 seconds then it doesn't matter that function Fast is called more often; clearly the place to optimize the code is in function Slow.

Indeed this is the idea behind Reduced Instruction Set Computing (RISC). It might take 50 lines of RISC code to do function X, and only 5 lines of non-RISC code to perform the same function, but the RISC code is executed so much faster that the net result is improved performance.

So the output from `tracecount` was in this case virtually useless. What we need to know is how long it takes to execute each function, not how often a given function is called.

Tracing Function with FIFO Pipes

UNIX supports FIFOs, or name pipes. FIFO is an acronym for first-in, first-out. The name of the FIFO resides in the UNIX file system like any other file. What this means is that we have an entity that has the attributes of a file as well as a pipe. Print spoolers frequently employ named pipes to do their work. We'll use a named pipe for our purposes.

Use the `mknod` command to create a named pipe. (On some systems you must be root to execute this command.)

```
mknod FIDO p
```

That creates a named pipe call FIDO. Let's look at how it behaves (we sent it to obedience school).

If we write something to FIDO, for example,

```
echo "dogs chase cats" > FIDO
```

nothing will appear to happen. The cursor will just sit there and your terminal will be locked-up until you break out with the interrupt key.

If, however, you go to another terminal (or open another X-window, or run the command in the background by appending an ampersand to the command) you can see some action.

```
cat FIDO
```

cat-ing FIDO opens the read end of the pipe and pulls the information on through.

```
dogs chase cats
```

Thus your original terminal (or process) wasn't really locked up. It was simply waiting for someone to pick up the other end of the pipe. This happens at the command-line as well.

```
cat /etc/password |
>
>
```

As long as you hit Enter you will see the secondary prompt. The shell is waiting for you to complete the command, to attach another program to the other end of the pipe.

```
>
> more
```

So FIDO is a pipe that looks like a file. It's this ability to look like a file that we'll exploit with the Informix .03 RDS debugger to see where our programs are spending too much time (later we'll see another technique that works with both Informix 4.0 and .03).

After starting the debugger again to run our program we can trace functions as follows:

```
trace functions >> FIDO
```

Again, this will appear to lock up your terminal unless you already have another process reading from FIDO. (From now on the discussion will assume that you have two terminals.)

Go to your second terminal and cat FIDO. That opens the reading end of the pipe and allows the debugger to continue on at the writing end.

Now run the program.

As it runs you'll see, in real time, a list of functions as they are entered and exited. This list is the same list you would get if you redirected the output from Trace Functions to a normal file, but now we see them as they are written to the file, to our pipe.

Timing Function Execution

The next step is to also record the times at which the functions are entered. We do this by simply writing out the current time as we read in each line. Originally this was done with a shell script and the date command, but a C program was subsequently used because it's faster and thus reduces the granularity of the time sampling. That is, you don't use a sun dial to time a foot race.

The algorithm is straightforward.

```
while there is a line to read
    get the current time
    get the line
    and print them
```

The code below also checks to see if the line read in contains "En" so that we only see the lines in which functions are being entered.

```
/*
   profile.c

   read in lines and write the time before sending it out.  but
   only if the lines start with 'En'
*/

#include <stdio.h>
#include <ctype.h>
#include <sys/types.h>
#include <time.h>
```

```
#define BUFSIZE 1000

/* ============================================================ */
main(argc, argv)
int argc;
char *argv[];
{
    FILE *fp;
    char *pfn;
    char buf[BUFSIZE];
    time_t now, start;

    if (time(&now) == 1|| time(&start) == 1)
        syserr("What time is it?!");

    if ( argc < 2 ) /* read from stdin */
        fp=stdin;
    else {
        pfn=argv[1];
        if ( (fp = fopen ( pfn, "r" )) == NULL ) {
            fprintf ( stderr, "*** could not open: %s
", pfn );
            exit ( 2 );
        }
    }

    while ( fgets(buf, BUFSIZE, fp) != NULL ) {

        if (time(&now) == 1)
            syserr("What time is it now?!");

        if ( buf[0]== 'E'&& buf[1] == 'n' )
            printf("%4d %s ", now - start, buf);
    }

    fclose ( fp );
    exit(0);
}

/* ============================================================ */
int syserr(ps)
char *ps;
{
    fprintf(stderr, "%s
", ps);
    exit(1);
}
/* ============================================================ */
```

We ran the program again, hit the Save key, and watched.

```
 20 Enter lld_high() from lib_screen line 35
 24 Enter lld_a_field() from lld_input line 114
```

```
24 Enter llh_display() from new_doc_no line 512
25 Enter note_newkey(filename = "stoordre

25 Enter note_newkey(filename = "stoordre
32 Enter note_newkey(filename = "stoordrd

32 Enter note_newkey(filename = "stoordrd
40 Enter note_newkey(filename = "stoordre
```

Indeed the system was slow. We noticed however, that it was slow in a particular function; furthermore that this function was called 5 times for each document saved. One function was taking on the average 25-45 seconds to complete! Clearly this was the place to look.

We discovered that the function was checking a table in the database and that the table was missing an index. Thus every time the function was called the SQL engine would have to flip through the data one page at a time, rather than using the index to turn directly to the needed information.

Adding the correct indexes brought the save-time down from 60 seconds to about 5. Quite unexpectedly we received reports from users of other programs as well because that same table was used by several programs.

Running the RDS Profiler with Informix 4.0

Trying to use the same techniques on our development machine, which is running Informix 4.0, was a different story. The debugger refused to write to a named pipe.

Instead we used a little known option of the "tail" program: `tail -f`, as in forever.

```
tail -f trace.out
```

If a new line is written to `trace.out`, `tail` will print it; if not, `tail` waits.

In the debugger we redirected the trace functions output to `trace.out`, and use tail to send the information to profile.c

```
tail -f trace.out | profile
```

Further additions to `profile.c` (we don't need tail) and a new awk program allow us to generate outline-like output. Every time a function is called, the name is indented. Also when the function returns the total time spent in the function is printed. And at the end of the output the total time spent in each function is printed. Below is part of the output. Note that since these functions all took less than a second to execute the times are all 0.

```
line# time   function call

    1    0    startlog
    3    0    startlog: 0
    4    0    put_scrlib
    8    0    put_scrlib: 0
    9    0    put_scrlib
   13    0    put_scrlib: 0
   14    0    init
   15    0        logo
   17    0        logo: 0
   18    0        num_args
   20    0        num_args: 0

           ...

 1122    4                    switchbox
 1123    4                        lib_screen
 1124    4                            llh_setdata
 1125    4                                set_data
 1126    4                                    length
 1128    4                                    length: 0
 1130    4                                set_data: 0
 1132    4                            llh_setdata: 0
 1134    4                        lib_screen: 0
 1136    4                    switchbox: 0
 1137    4                    switchbox
 1138    4                        lib_screen
 1139    4                            llh_display
 1141    4                            llh_display: 0
 1143    4                        lib_screen: 0
 1145    4                    switchbox: 0
 1146    4                    switchbox
 1147    4                        lib_screen
 1148    4                            llh_high
```

```
1150    4                           llh_high: 0
1152    4                       lib_screen: 0
1154    4                   switchbox: 0
1156    4               lib_before: 0
1158    4           llh_b_field: 0
1159    5           llh_a_field
1160    5               auto_zoom
1169    5                   switchbox
1170    5                       lib_screen
1171    5                           llh_setdata
1172    5                               set_data
1173    5                                   length
1175    5                                   length: 0
1177    5                               set_data: 0
1179    5                           llh_setdata: 0
1181    5                       lib_screen: 0
1183    5                   switchbox: 0
1184    5                   length
1186    5                   length: 0
1188    5               auto_zoom: 0

Totals for all functions

   0 startlog
   0 logo
   0 window_size
   1 lib_message
   1 ring_border
   3 lib_chkkey
   3 ring_lladd
   3 udf_check
   3 udf_edit
   4 llh_input
  20 lld_a_field
  27 lld_input
```

The granularity if this type of profiler is fairly coarse. Don’t plan to use it to shave micro-seconds off of your execution times. However if there are glaring problems with execution time, these tools can help point you in the right direction. Also the graphic output of the function calls helps you to see what’s going on.

Below are the C and awk programs used to produce the above output.

For further information (or if you would like an email copy of this article, and the code) contact the author at: uunet!4gen!stevei.

```
/* profile.c
##########################################################################
# Copyright (C) 1991
```

```
# <Your Company Name>
# Use, modification, duplication, and/or distribution of this
# software is limited to the terms of the software agreement.
###############################################################################

 parse the output from fgldb trace functions.  The Enter functions
 and Return from functions are parsed out with the line number from
 the trace.out file and the time in seconds that profile.c got to
 it.
*/

#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <termio.h>
#include <signal.h>
#include <ctype.h>
#include <time.h>
#include <errno.h>

#define BUFSIZE 1000
#define TRUE 1
#define FALSE 0

FILE *gfp;

/* ================================================================== */
main(argc, argv)
int argc;
char *argv[];
{
    char *pfn, buf[BUFSIZE], *newbuf, *ps;
    time_t now, start = 0;
    int from_found = FALSE, int_handler();
    long lineno = 0L;

    signal(SIGINT, int_handler);
    signal(SIGTERM, int_handler);

    if ( argc < 2 )    /* read input from stdin */
        gfp=stdin;
    else {
        pfn=argv[1];
        if ( (gfp = fopen ( pfn, "r" )) == NULL ) {
            fprintf ( stderr, "*** could not open: %s
", pfn );
            exit ( 2 );
        }
    }

    while ( TRUE ) {
        if (fgets(buf, BUFSIZE, gfp) != NULL ) {
```

```
            if ( start == 0 ) /* did we start yet? */
                if (time(&start) == 1)
                    syserr("What time is it?!");

            if (time(&now) == 1)
                syserr("What time is it now?!");
            lineno++;

            if (strncmp(buf, "Enter ", 6) == 0 ) {
                /* chop off left paren and rest of line */
                *(strchr(buf, '(')) = '0';
                fprintf(stdout, "%5ld %4d %s
",
                    lineno, now - start, buf);
            }
            else if ( strncmp (buf, "Return", 6) == 0 )
                from_found = TRUE;
            else if ( from_found ) {
                if ( (newbuf=strstr(buf, "from ")) != NULL ) {
                    from_found = FALSE;
                    /* chop off 'at line' */
                    if ( (ps=strstr(newbuf, "at line")) != NULL )
                        strcpy(ps, "
");
                    fprintf(stdout, "%5ld %4d Return %s",
                        lineno, now - start, newbuf);
                }
            }
        }
    }

    /* never gets here */
}

/* ================================================================== */
int syserr(ps)
char *ps;
{
    fprintf(stderr, "%s
", ps);
    exit(1);
}
/* ================================================================== */
/* the only way to break out of the program is from a termination     */
/* request or interactive attention, e.g., interrupt                  */

int_handler()
{
    signal(SIGINT, SIG_IGN);
    fflush(gfp); /* fclose() should flush it but let's make sure */
    fclose(gfp);
    exit(1);
```

```
}
/* ============================================================== */

:
#########################################################################
# Copyright (C) 1991
# <Your Company Name>
# Use, modification, duplication, and/or distribution of this
# software is limited to the terms of the software agreement.
#########################################################################
# This program takes its input from a file generated by profile.c
# and fgldb with "trace functions" turned on.
#
# the file might look like this.  the first column is the fgldb
# trace.out line number; the second column is the time in seconds;
# and the third column tells what function was entered or exited
#
#   1    0 Enter startlog
#   3    0 Return from startlog
#  15    1 Enter logo
#  20    1 Return from num_args
#
# the output is an outline-like display of the entered and exited
# functions. the time spent in each function is displayed when the
# function exits, and the total time spent in a given function is
# given at the end.
#
# author: stevei.

awk '
    BEGIN {
        printf ("line# time   function call

")
    }
    $3 == "Enter" {
        stack++
        lineno = $1
        time = $2
        funcname = $4
        buf[stack funcname] = time
        printf("%4d %4d", lineno, time)
        indent()
        print funcname
    }
    /Return .*from / {
        lineno = $1
        time = $2
        sub(/^.*from/, "", $0) # note: this resets $variables
        funcname = $1
        elapsed = time - buf[stack funcname]
        global[funcname] = global[funcname] + elapsed
        printf("%4d %4d", lineno, time)
```

```
        indent()
        printf ("%s: %d
", funcname, elapsed)
        stack--
    }
    function indent() {
        # indent four spaces for every stack depth
        for ( i = 0; i < stack; i++ )
            printf("    ")
    }
    END {
        for (w in global)
            printf("%4d %s
", global[w], w)
    }
 ' $1
```

Writing Termcap Entries

Termcap is short for "terminal capabilities," which are descriptions of the various features of a terminal, and instructions on how to use these features, all written in a rather cryptic language in a termcap file. The language which describes the terminal capabilities is interpreted by programs that use terminal I/O in order for the program to correctly control the terminal and interpret input from the keyboard.

The Termcap File

The termcap file, `/etc/termcap`, usually consists of several termcap entries, each one corresponding to a particular terminal or to a particular emulation mode on some terminal, or to a terminal being used in some special fashion. The rest of the termcap file, about 20%, consists of lines beginning with a #. These are comment lines and are generally less intelligible than the rest of the termcap file. These lines are to be mostly ignored. One termcap entry can be separated from another once you understand what an entry itself looks like.

The Termcap Entry

Each entry has the form `label[|label][:capability]:`. This means there are a string of labels by which the entry can be referred to, each separated from the next by a "|" symbol followed by a string of terminal capability codes each separated from the next by a ":". If the termcap entry is longer than a single line (almost all of the time it is) then the symbol "\" is used on the end of a line to indicate that the entry continues to the next line. An easy example of what two termcap entries might look like follows (easy because whoever edited them tried to make them easy to distinguish as entries):

```
n2|7901|NCR 7901:co#80:li#24:bs:am:cl=^L:\
  ti=\E0P^X^L:\te=^O^X\E0@:cm=\EY%+%+:ce=\
  EK:cd=\Ek:kh=^A:kl=^U:bc=^U:kr=^F:nd=^F:\
  ku=^Z:up=^Z:\kd=^J:do=^J:kb=^H:kc=^M:so=^N:\
  se=^O:sg#0:ul:us=\E0‘^N:ue=\E0@^O:ug#0:\
  NM=^O^X\E0@:NB=\E0B^N:NR=\E0P^N:\
  NS=\E0R^N:AL=\E0A^N:AB=\E0C^N:\
  AR=\E0Q^N:
```

```
n3|vwpt|viewpoint|ADDS Viewpoint:co#80:li#24:\
  bs:am:cl=^L:ti=\E0P^X^L:\te=^O^X\E0@:\
  cm=\EY%+ %:ce=\EK:cd=\Ek:kh=^A:kl=^U:\
  bc=^U:kr=^F:nd=^F:\ku=^Z:up=^Z:kd=^J:do=^J:\
  kb=^H:kc=^M:so=^N:se=^O:sg#0:ul:us=\E0‘^N:\
  ue=\E0@^O:ug#0:NM=^O^X\E0@:NB=\E0B^N:\
  NR=\E0P^N:NS=\E0R^N:AL=\E0A^N:\
  NR=\E0P^N:NS=\E0R^N:AL=\E0A^N:\
  AB=\E0C^N:AR=\E0Q^N:AS=\E0S^N:OV#0:\
  k1=\E1:k2=\E2:k3=\E3:k4=\E4:k5=\E5:\
  k6=\E6:k7=\E7:k8=\E8:k9=\E9:MP=\E0P^X^L:\
  MR=\E0@^X:NU=^N:EN=^V:CN=^X:CF=^W:
```

Not all termcap entries appear like the examples above. Sometimes they might look like the following:

```
n2|7901|NCR 7901:co#80:li#24:bs:am:cl=^L:ti=\
  E0P^X^L:te=^O^X\E0@:\cm=\EY%+ %+:ce=\
  EK:cd=\Ek:kh=^A:kl=^U:bc=^U:kr=^F:nd=^F:\
  ku=^Z:up=^Z:\kd=^J:do=^J:kb=^H:kc=^M:so=^N:\
  se=^O:sg#0:ul:us=\E0‘^N:ue=\E0@^O:ug#0:\
  NM=^O^X\E0@:NB=\E0B^N:NR=\E0P^N:NS=\
  E0R^N:AL=\E0A^N:AB=\E0C^N:AR=\E0Q^N:
n3|vwpt|viewpoint|ADDS Viewpoint:co#80:li#24:\
  bs:am:cl=^L:ti=\E0P^X^L:\   te=^O^X\E0@:cm=\
  EY%+ %+ :ce=\EK:cd=\Ek:kh=^A:kl=^U:bc=^U:\
  kr=^F:nd=^F:\ku=^Z:up=^Z:kd=^J:do=^J:kb=^H:\
  kc=^M:so=^N:se=^O:sg#0:ul:us=\E0‘^N:\ue=\
  E0@^O:ug#0:NM=^O^X\E0@:NB=\E0B^N:NR=\
  E0P^N:NS=\E0R^N:AL=\E0A^N:\AB=\E0C^N:\
  AR=\E0Q^N:AS=\E0S^N:OV#0:k1=\E1:k2=\E2:\
  k3=\E3:k4=\E4:k5=\E5:\k6=\E6:k7=\E7:k8=\E8:\
  k9=\E9:MP=\E0P^X^L:MR=\E0@^X:NU=^N:\
  EN=^V:CN=^X:CF=^W:
```

Obviously, it is harder to distinguish one entry from the next in the second example especially when you consider that there may be hundreds of entries formatted together like this.

Notice that the last line of the entry for n2 does not end with a "\" but that every other line of the entry does end with "\". If the last line of the entry ended in "\" like the others then the entry for n2 would continue into the entry for n3. Correct interpretation of the termcap entries relies heavily upon these *very* important "\" characters.

The Labels

The label part of the termcap entry is the mechanism by which a program can find the entry in the termcap file. Usually it consists of a two letter code, a short name or two, and a brief description of the terminal. The termcap entry can be identified by any of the labels in the label section of the entry and the identification is usually based upon the value of the system variable `TERM` (use echo `$TERM` to see its current value). It is highly advisable to use one of the short names rather than the two character code for the value of `$TERM` since the two character code may not be unique and programs find the first occurrence of the label whether or not there is another—yours—further down in the file.

The Capability Codes

The terminal capabilities directly follow the labels in each termcap entry and each code is separated from the next by a ":" and has a two letter "name." There are three different sorts of codes used to identify a capability; a boolean type either the code is there or not with no specific value associated with it; a numeric type an integer value is assigned to the code; a string type a string of characters is assigned to the code.

The **boolean** type codes are used to identify the existence or lack of a certain terminal characteristic such as whether the cursor automatically wraps around the margins of the terminal. In the termcap entry they can be identified because they consist solely of the capability name ("am" for automatic margins).

The **numeric** type code is used to identify countable parameters associated with the terminal such as number of columns and number of rows on the screen. These codes have the form `codename#value` (i.e. number of columns would be `co#80` for an eighty column screen).

The **string** type is used to identify strings of characters sent by certain keys on the keyboard and strings needed by the terminal to perform certain actions such as positioning the cursor in a particular location on the screen. For example, to identify the character string sent from the keyboard by the up arrow key the code might read `ku=\E[1` and would indicate that the keyboard sends the character sequence: `ESC` (octal 33) `[ 1` when you press the up arrow key.

Special Characters

The character strings for a terminal capability often use such characters as `\E`, `^R`, or `^A` which represent ascii `[ESC]`, `[CTRL]-[R]`, and `[CTRL]-[A]` respectively. In the termcap file however, these characters are never given in their literal form because they are generally non-printing characters. Therefore, in order to represent them in text they have a special form.

ASCII `[ESC]` is represented with `\` and `E` (looks like "`\E`").

ASCII `[CTRL]` characters are represented with a `^` followed by the character in upper case, hence `^R` and `^A`.

Characters can also be represented by their octal (base eight) value in cases such as the ":" which is used to interpret the termcap file by separating arguments and can't be included directly as part of a string (it would be interpreted as the end of the string). The octal code for a colon is `\072`.

The Codes

There are too many terminal capability codes to list all of them here and many programs use special sets of codes in addition to the more or less "standard" set. Therefore, you should look in the system documentation for a thorough list of the various termcap codes and their functions. You must refer to your program documentation to find the codes for any special functions used by the program. Here is a list of some of the more common codes:

Table 3: Common Terminal Capability Codes

Code	Function
`cm`	control code for cursor positioning by row and column
`ku`	character sequence sent by the cursor up key
`kd`	character sequence sent by the cursor down key
`kr`	character sequence sent by the cursor right key

Table 3: Common Terminal Capability Codes

Code	Function
`kl`	character sequence sent by the cursor left key
`kh`	character sequence sent by the home key
`k0-k9`	character sequences sent by the function keys
`ho`	control sequence used to position the cursor at 0,0
`do`	control sequence used to move the cursor down a row
`cr`	character sequence sent by the enter/return key
`nd`	control sequence used to move the cursor back a column
`up`	control sequence used to move the cursor up a row
`bt`	control sequence used to back tab
`bs`	boolean code which indicates that backspace is ^H
`am`	boolean code which indicates margins are handled automatically
`co`	number of columns on the display
`li`	number of lines on the display
`so`	control sequence used to turn standout mode on
`se`	control sequence used to turn standout mode off
`sg`	number of characters of display required by the 'so' string
`cl`	control sequence used to clear the display
`ce`	control sequence used to clear to the end of the line

The biggest problem with terminal capabilities is not how to read them but what the various codes mean for the various programs that use them. Unfortunately, the answer to that question often remains in the head of the author of that program and doesn't reach the users of the program nearly often enough, or in an intelligible form. The other complication is that terminal manufacturers seldom produce readable

reference material for their own terminal's characteristics.

Ideally, with a combination of knowing how the program uses termcap and how the terminal behaves, one should always be able to fix or write a termcap entry for any program and terminal (or discover that the program cannot run at all on the terminal). The implementation of the capabilities on various terminals is anything but standard, but the interpretation and use of the codes by a program usually follows certain guidelines.

If a program does any full screen display and entry, if it highlights anything, ever, or if it just clears the screen, then it almost certainly uses termcap to decide how to do its various terminal-oriented tasks. other uses for the termcap entries are: to generate graphics, position the cursor on the screen, and to identify special input from the keyboard (keys with special meaning).

Interpretation and Action

Of the three types of terminal capabilities, the most heavily used and the most complicated are the string type. These, in turn, can be grouped into two classes of applications. The first class, which includes codes for cursor movement (ku, kd, kr, kl, kh, etc.) and many special program codes, is used only to identify keystrokes from the keyboard. For example, when you enter an "a" from the keyboard, you press the "a" key and only one character is sent to the computer; but other keys, such as the cursor keys often will send a sequence of characters to the computer. In order for programs that use special keys to correctly recognize keyboard entry, the program needs to know how to interpret the characters it is receiving from the keyboard. With the aid of the termcap file, a program can recognize complex input and behave accordingly. The group of codes that is used for identifying input is one of the two classes of string type terminal capability codes.

The other class of string type codes is used for directly controlling the terminal screen. These codes include character sequences that invoke a graphics character set, or start a block of highlighted screen, or turn off the highlight or graphics, just to list a few. Others indicate the correct codes to send to the terminal, to move the cursor about the screen, or to enable and disable the terminal's auxiliary port. With a combination of these two classes of codes, a program can both interpret input from the keyboard and perform actions with the terminal such as complex graphics display.

Now you know why programs need termcap files and how they use them, as well as how to read them. The only steps left are testing, modifying, and writing these files. Testing a termcap is not simple because the termcap is only part of the terminal I/O system, any element of which can be to blame for weird or incorrect displays. However, once your terminal is working on the most primitive level (you get a login and can run most system commands without any problem) then specific program misbehavior usually can be attributed to problems with the termcap.

Most of the time, termcap difficulties are related to only a couple of errors in an existing termcap entry for the terminal. It is unusual to not be able to find a termcap entry that provides most of the features for your terminal simply by trying various values for the variable `TERM` and using the different emulation modes available with many terminals. Even when you cannot find such a termcap entry, you can get a substantial head start on developing a new termcap entry by using an existing termcap entry for a similar terminal type.

When you are testing and modifying a termcap entry, it is usually best to make a temporary file that contains only that entry so that there is no danger of corrupting the other termcap entries. Then, in order to direct the system to use that file you can set the system variable `TERMCAP` equal to the full path name of the temporary file and then export `TERMCAP`. Once you have this special file setup you need the proper documentation in order to identify and correct problems within the termcap entry.

You will need the system documentation on the various termcap codes and program documentation if the programs you will be running require any special termcap entries. Also you must have technical documentation for the terminal and be sitting at the terminal ready to go. It helps to have a second terminal available which already has a functioning termcap so that you can edit files without having to rely on the terminal for which you are writing the termcap.

Finally, you must have the UNIX editor "vi" and/or the program "od" on your system in order to read the various character codes sent by the keyboard. With these tools (and some time) at hand you are ready to go.

Testing the Keys

The most reliable way to find what characters are being sent by the special keys on your keyboard is to directly collect and view the output of those keys in an uninterpreted form. "vi" and "od" both provide excellent ways to see what a key is sending from the keyboard ("vi" is better if your termcap is already good enough to support it).

To use "od" (od stands for octal dump) you simply type `od -bc [RETURN]` at the command line. At this point the program 'od' is waiting for input from the keyboard. Now when you press a key followed by a `[RETURN]` and `[CTRL]-[d]` the character and octal representation of the characters sent by the key will be displayed in two rows. The first row is the character representation (if any) for each character and the second row is the octal value for each character (ignore the first string of digits on the first row).

By comparing the octal values with a table of ascii characters, you can determine exactly which characters are being issued by the key. ("od" will be waiting for your next input followed by `[RETURN]` and `[CTRL]-[d]` but can be terminated by an additional `[CTRL]-[d]`.)

To use "vi" instead of "od", you can invoke "vi" without a file name, type `i` to get into insert mode, then type `[CTRL]-[v]` followed by the key you want to test followed by a return. The characters displayed on the screen are the ascii representation for the characters sent by the key - `^[` is `[ESC]`, `^A` is `[CTRL]-[a]`, etc. To exit "vi" type `[ESC] [:] [q] [!]`.

Action Codes

Certain terminal functions require a control code sent by the program to the terminal which causes the terminal to perform the function desired. An example of this type of function is highlighting. In order for the terminal to begin highlighting a certain block of text, first the cursor must be positioned at the beginning of the block and then a code needs to be sent to the terminal to begin highlighting. Then a code needs to be sent after all the characters to be highlighted have been sent in order to stop highlighting. The program looks in the termcap for the string code `cm` to use for positioning the cursor on the screen at a certain row and column. Then it gets the string code `so` to turn highlighting (standout) on. Then it looks for the code `se` to

turn highlighting off. If any of these codes are incorrect, the highlighting action will fail and may also wipe out the rest of the display.

There are many other control codes that are sent to the terminal which can disturb the display seriously if they are incorrect. These codes can only be found in the technical manual for the terminal. For example, `so` must be set to the manual entry for "start standout mode" and `cm` must be set to the manual entry for "direct cursor addressing (cursor movement)." (for `cm` only, there are additional characters explained in the system documentation that refer to the format for cursor and row numbers required as a variable part of the `cm` string.)

By using the terminal manual for the control strings and the system documentation for the termcap codes, you should be able to fix and add needed controls (not all terminals will have all of the possible capabilities). Often, but not always, there will be an appendix in the technical documentation for the terminal which gives the control codes for the available emulation modes for each terminal function. This is often the only place in the documentation where the codes are explicitly given in an understandable form.

The Other Codes

The two classes of string type codes have been discussed so far. By comparison the other codes are simple to understand and work with. The boolean type are either in the termcap or not and indicate the existence of a particular terminal characteristic. There are only two common numeric type entries, `li` for number of rows and `co` for number of columns.

Observations

Things to keep in mind when working with termcaps:

1. It takes time to eliminate all of the possible bugs from a termcap but often the solutions are simple. The only way to effectively work with termcaps is with a substantial dose of patience.

2. Usually the best way to start solving a termcap problem is not with the termcap. Make sure that `TERM` is set to a label in the label string and that the label is unique.

3. Always double check your terminal setup and emulation mode before working on a termcap for that terminal.

4. Before changing a termcap make a copy, and then use a fine tooth comb for syntax errors. All labels should be separated by "|", all entries separated by ":", all but the last line ends in "\".

5. A lot of work can be saved by using at least part of an existing termcap.

No termcap entry (all of the labels and codes combined) can be over 1024 characters long. Usually the entries after the 1024th character will be ignored.

6

Programming Conventions

This style guide describes standards and conventions for designing an application. In programming, the developer makes a number of decisions about how things are presented. The relative merits of a certain "look" are less important than a rigorous consistency in maintaining that look.

Over the last few years there has been an increased interest in consistency of graphical user interfaces. But the fact remains that most people working in or developing a financial package do their work on character-based terminals. Standards for these users are critical.

The following rules have been developed to present input and output to the user in the most consistent way possible.

Note: These rules are intended for programs and system files. Several of the rules vary for typeset documentation.

Naming Conventions

In creating Fitrix Accounting, a number of naming conventions have been developed. These conventions determine how various types of names used in the system are constructed.

Once you understand the naming conventions, you should be able to identify the role of particular element you are looking at by its name alone. These elements may be system file names, tables (database files), rows (database fields), function names, screen variables, and program variables. All of these element names are constructed according to a set of rules.

System File Naming Conventions

We have a number of conventions for naming system files; that is, files at the operating system level rather than those in the database (which we refer to as "tables"). In Fitrix Accounting, you can usually tell the type of file and the role it plays in the application simply by looking at its name. In this section of the documentation we explain the method behind the naming conventions.

On the file system level, these conventions help us manage our files more efficiently. Consistent names for files allow users to identify the purpose of each file by reading its name. If a directory contains many files and directories, having and using the file-naming conventions allows us to list only the files/directories that we want to see when working with that directory.

Using MenuShell

The consistent use of file names helps even more when using the MenuShell program while developing 4GL software. MenuShell is a character-based user interface for UNIX that makes movements between directories easier and displays file names and types on a menu screen.

One of MenuShell’s many advantages is that you can "teach" it to recognize various types of files if they are named according to a convention. MenuShell can then display the files sorted by their various types, grouping files in an orderly manner or filtering out unwanted file types from the directory display.

MenuShell can also be taught how to automatically handle any of these file types when they are selected. For example, MenuShell can be told that when a Perform form in a file name ending in `.per` is selected, first load that form into the editor and, after editing, run the form compilation program in the background to produce the `.frm` file. Teaching MenuShell to do these repetitive tasks allows the developer to concentrate on other things. We strongly recommend MenuShell for all developers because of this kind of automation. By recognizing different file types, MenuShell also serves to enforce your internal naming conventions.

Standard File Names Construction

By convention, the file name is divided into two parts: the name and the extension. The name may be divided into a prefix and a basename. The prefix and the extension are used to convey certain information about the type of file. Prefixes and extensions are selected from a predefined set of names. The name and especially the basename are simply descriptive strings.

Names are formulated according to certain rules. The base portion of the filename, including the prefix, contains eight characters or less. It contains only alphanumeric characters [a-z][0-9] or an underscore. The name is in lowercase letters. This is done to eliminate keystrokes, since most of the work in UNIX/XENIX must be done with the CAPs key off.

Names are, as much as possible, descriptive. Once the prefix and extension are removed, the remaining characters describe in some direct way the content of the file.

There are some special filenames that are used consistently everywhere in the system. The following are examples:

The use of `menu` in any directory name indicates a menu directory.

The file name `menu` by itself indicates a menu "image" file.

The name `sccs` indicates a "source code control program."

A file named `Info` contains information for the programmer.

A file named `Makefile` contains the instructions for "making" the program using the UNIX "make" utility.

A file named `README` is a note for users to read.

There is also a number of standard names for the files that contain generated source code, these are explained in detail in the previous section.

Module Naming Conventions

At the "module" level of either application or menu directories, the following prefixes are used:

`ar` - Accounts Receivable

`ap` - Accounts Payable

`gl` - General Ledger

`oe` - Order Entry

`ic` - Inventory Control

`py` - Payroll

`pu` - Purchasing

`fa` - Fixed Assets

`all` - Base Files

Menu Item Naming Conventions

In the menu system, there is a separate file for each menu item. This file contains the instructions that tell the system what to do when that item is selected from the menu. The Item Instruction files are named in a very specific way.

The first character(s) of the filename make up the item "key." Any alphabetic character(s) must be lowercase, and the combination of characters must correspond to the keystrokes that a user types at the Menu Selection Prompt to select the item. On the displayed menu, the user may see some combination of the digits 0-9 or alphabetic characters (a-z, A-Z) representing the item key. Though the menu system supports any number of characters as the key for selecting a menu item, in Fitrix Accounting we use only a single character to select any menu item.

Fitrix Accounting uses numeric items to call up main and general menus. Since main menus are used only to call general menus, all items on them are numeric.

Alphabetic characters are used on a general menu to call up "detail" menus, Informix programs, or shell scripts. Hence, you can tell when you look at a menu directory that the item `1-journ` loads a general menu, since it begins with a number. You can tell that the item `a-upinv` calls an item on a general menu because it begins with a letter.

When naming the file, use only single numbers 1, 2, 3 etc. for naming menus and sub-menus, and use single alphabetic characters, such as a, b, c etc. when naming the actual menu selections. Use the numbers or letters that correspond to the actual menus and/or selections of the program.

When the menu selection `a` is made by the user, under UNIX (or XENIX) the Menus program looks for an Item Instruction file named `a` or a file beginning with the characters `a-`. Each Item Instruction file must begin with a unique keystroke sequence. If there is more than one file that begins with this character sequence, Menus executes only one.

Database Table Naming Conventions

Database tables (also called database "files") are entities which store a company's financial data. In Fitrix Accounting, table names are constructed according to specific conventions. Once these conventions are understood, examining a table name should clarify that table's purpose.

The name of a database table always contains eight characters. These characters are all lowercase alphabetic letters [a-z]. No numbers or special characters (underscore, underline, period, etc.) are used. As previously mentioned, lowercase letters are used to eliminate keystrokes in UNIX/XENIX.

This eight character name is comprised of four discrete parts. For example, the table stxtranr can be broken down as follows:

```
st x tran r
```

Each of these parts tells you something about the purpose of the table.

The first two characters represent the name of the application system associated with the table. In Fitrix Accounting, these two characters are always `st`. This name corresponds to the "application" directory. Any tables that are utilized entirely by a different application should start with a different two characters.

The third character indicates the "module" of the application to which the table belongs. The following page contains a table of characters used in the third position of Fitrix Accounting names.

Characters used in the third character position of Fitrix Accounting table names:

g - General Ledger

r - Accounts Receivable

p - Accounts Payable

o - Order Entry

i - Inventory Control

y - Payroll

f - Fixed Assets

u - Purchasing

x - several different modules

The next four characters of the table name are descriptive of the type of information the table stores. Most tables can be classified as one of five types. This part of the name can be used to identify these types.

The first type of table is referred to as a **reference table**. Reference tables store information that is "referenced" throughout a Fitrix Accounting module. When the end-user first sets up a module, these tables are used extensively. Reference tables are named according to the type of information that they store; characters 4-7 in the name describe the information stored. For example, `str-custr` is a reference file that stores customer information. Control tables are a specific type of reference table. These tables store a single record. Values that are used to provide default values throughout a module are stored in the module's control table. They may store additional information as well and may have one or more fields that are updated by a posting process. Control tables are easily identified by their names: characters 4-7 are always `cntr`. There is a multipurpose reference table that stores information accessible to all of the Fitrix Accounting modules. This is the `stxinfor` table.

The term, "document," is used to refer to a particular type of business transaction. Sales orders, invoices, and cash receipts are all examples of documents. **Entry tables**, the second major type of table, are those that store the initial entry of such documents. Documents are stored in header and detail entry tables. Entry header tables have an eighth letter of "e". Entry detail tables have an eighth letter of "d". The entry header table stores the general information about the document; the entry detail table stores the detail lines that make up the document. Characters 4-7 of an entry table name describe the type of document that is stored. Examples are `invc` for invoices and `cash` for cash receipts.

The next major type of table is the **transaction table**. These tables are used to record documents when they are posted, storing information such as the document number and posting date. This part of the name is always `tran.` An example of this type of table is `stxtranr`.

The fourth type of table is the **activity table** that records document detail information such as the debit/credit amounts for each ledger number, the inventory item, amount, and price. These are always named `actv`. Rows are inserted into activity tables at the time of posting.

The final type of table is the **open item table** which stores document information that is updated by subsequent documents. These tables are used by accounts payable and receivable to track what invoices are due and when. These tables are always named `open`.

(varies) - Reference tables containing reference information

(varies) - Entry tables containing initial entry of information

`tran` - Transaction table containing document information

`actv` - Activity table containing transaction detail

`open` - Open table item containing invoices with balances

The last character in the table name provides additional information on the nature of the table. In other words, it indicates the role of the table in the database structure. We will be covering database structure and the terminology used in much greater detail in the next section. The characters used to indicate type of tables are as follows:

`c` - Control table containing module or system defaults

r - Reference table such as a customer or vendor table

v - View an Informix "View Table", columns are actually selected from other tables

e - Entry table which is the "header" information for a document such as a general journal entry or an invoice

h - Header file for system reference files such as help and error tables

d - Detail table which contains all the "detail" information for a document or header table

Using these conventions, you can generally identify the role of a given database table simply by knowing its name. For example, when you see the table `str-custr`, you know that this is a table used by the Accounts Receivable module which contains reference information. The descriptive section, `cust`, would indicate that it is the customer table. The table `stractvd` would also be used by the receivable package, but it would contain transaction activity.

Program Naming Conventions

We have already discussed this convention in some detail under the section covering directory structure. In summary:

The prefix of the program name indicates the type of program:

i - data entry program (input program)

o - output or printing program

p - posting program

The basename is descriptive and usually describes the type of reference information or document managed by the program.

The suffix indicates whether the name refers to the directory in which the source code is stored or the actual program itself.

Standard File Name Extensions

The extension is a maximum of four characters long, including the initial period (e.g. `.per`).

Following the period are three characters which define the file type. Like the name, these three characters are always alphanumeric and lowercase. There is one exception to the "lowercase" rule: The compress program that we use adds `.Z` to the end of the compressed file.

Suffixes within the module directory:

- **`.4gs`** - INFORMIX-4GL source directory
- **`.4gc`** - customized INFORMIX-4GL source code directory
- **`.xxx`** - Any version of customized source code (can be any 3 character combination)

Suffixes within the "source" directory:

- **`.4gl`** - INFORMIX-4GL code
- **`.4ge`** - INFORMIX-4GL executable
- **`.4go`** - INFORMIX-4GL RDS (Rapid Development System) object
- **`.4gi`** - INFORMIX-4GL RDS executable
- **`.per`** - INFORMIX perform screen
- **`.frm`** - INFORMIX compiled screen
- **`.c`** - C source code
- **`.o`** - object code
- **`.ec`** - embedded C source code
- **`.ifg`** - *Report* file
- **`.sql`** SQL script

In Menu directories, the following extensions are used:

.act - Indicates a menu “action” file

.hlp - Menu item help file.

The following extensions are typiclly used within various special directories:

.sh - shell script programs

.doc - document files

.txt - text files

.tmp temporary files

.spl - spool file

.dbs - Informix database

.unl - unloaded database file

.a - archive file

.out - output format

.Z - compress format

.tar - tarfile format

.lck - lock file

Function Naming Conventions

There are four types of functions: upper level, middle level, lower level, and application library functions. Upper level functions are data independent. Lower level functions control the data flow. Middle Level functions control the flow between them. Application library functions are those specific to the accounting application or to modules within it.

Since functions are stored within a file (not at the system file level), they, like row or field names, are not limited in name length by the possible limitations of the file system. Therefore functions can have names of any length.

The full names of the upper, middle, and lower level functions contain a prefix, indicating the type of function, followed by an underscore (_), and then by a name describing the function. The following tables contain the standard prefixes.

Upper level functions:

`ring_` - all upper level data entry directories

Middle level functions:

`mlh_` - middle level "header" control functions

`mld_` - middle level "detail" control functions

`ok_` - middle level functions that control upper level functions

Lower level functions:

`llh_` - lower level "header" control functions

`lld_` - lower level "detail" control functions

Application Library Functions

Most functions that are called from the upper, lower, and middle level functions and are written specifically for the Fitrix Accounting packages, have a slightly different naming scheme. These function names contain a two-character prefix which indicates the accounting module using the function, followed by an underscore, then by a four-character name which is usually related to the principal table with which the function works, followed by a one-character suffix indicating the basic type or purpose of the function. One exception to this rule occurs when naming functions that do not relate to generic table access, in which case the four-character name may not relate to a table name.

Function name prefixes:

`al_` - functions used by all modules

`ap_` - functions used by Accounts Payable

`ar_` - functions used by Accounts Receivable

`fa_` - functions used by Fixed Assets

gl_ - functions used by General Ledger

ic_ - functions used by Inventory Control

oe_ - functions used by Order Entry

pu_ - functions used by Purchasing

py_ - functions used by Payroll

Function name suffixes:

c - “Check” - checks existence of data and returns true or false

d - “Detail” - provides extended data for Zoom form lines

n - “Name” - validates a code and returns a description

z - “Zooms” into another set of data, fills a screen array

Functions ending with `c` always return a `true` or `false`. Generally, these check for the existence of data referenced by a code entered by the user.

The functions ending in `n` take a code and return a name or description, and often some related data. For example, the `al_chrtn` function takes an account number and returns the description of the account, as well as a `DB` or `CR` indicating whether the account is increased with a credit or a debit. Most `n` functions return `NOT FOUND` if the search for the code fails, and thus may also provide data validation.

When a function ends with a `z` it is used to fill a screen array. "Zoom" functions allow the user to access a different, more detailed set of data than is normally displayed on the form.

The functions ending in `d` are "Detail" functions. These are often called from "Zoom" functions, providing more detail about the data displayed on one line of the zoom form. Detail functions sometimes call a sub-zoom function, which would also end with a `z`.

There are a few additional standard names used to indicate certain function types: The word `post` is used for functions that post data from location or state to another. Also, there is a set of "control" functions that return all values from the control table of a certain accounting module. The control functions and the tables they access are listed below:

Function - Table

fcontrol - stfcntrc

gcontrol - stgcntrc

icontrol - sticntrc

ocontrol - stocntrc

pcontrol - stpcntrc

rcontrol - strcntrc

ucontrol - stucntrc

xcontrol - stxcntrc

ycontrol - stycntrc

4GL Program Variable Naming Conventions

Variables in a 4GL program are also constructed according to a strict set of rules.

If the variable is part of the data flow, it contains a prefix indicating whether it belongs to the array that contains the screen record or the array that contains the database record. Such variable prefixes are named as follows:

p_ - contains data from the screen record

m_ - contains data from the database record

s_ - the screen record

t_ - also the screen record

The rest of the name is divided into two parts, the screen record name and the row name are represented in the database. Both of these names are highly descriptive. Thus, p_pinvcd.acct_num is the variable that contains data from the screen record on the payables invoice account number.

There are also some standard variables that are in all data entry programs. The table below displays these types of variables:

`progid` - program identification number unique to program

`menu_item` - can indicate whether in add or update mode

`scr1_max` - number of elements in screen array 1

`rec1_max` - number of elements in record array 1

`rec1_cnt` - number of active elements in record array 1

There are also some standard variables that are used in all report and posting programs. They are:

`prcname` - “process name” that appears when report prints

`rtmargn` - starting position of upper right margin

`prc_only` - output is produced if set to N. If set to Y, no report.

`allow_int` - allow interrupts during process. Y or N.

`quiet` - increment at which items count is displayed

`destin` - report destination:screen, printer, pipe, or filename

`sel_join` - join portion of select statement

`sel_filter` - filter portion of select statement

`sel_order` - order by clause for select statement

User Interface Style

It is important to maintain consistency when presenting data. Consistency allows the user to more readily understand the operation of the application. This is especially true when what the user sees refers to parts of the application software itself, in either the documentation, the on-screen help prompts, or the help text files.

Product Names

Informix

INFORMIX-4GL (all caps)

INFORMIX-OnLine

INFORMIX-SQL

PostScript (note uppercase "S")

UNIX (all caps)

XENIX (all caps)

Menus ("Menus" oblique)

Screen ("Screen" oblique)

Report ("Report" oblique)

MenuShell

database (not "data-base" or "data base")

Demonstration (headings, titles and the first reference to demonstrations in text should use the long form of the word rather than "demo")

OK (both characters in uppercase, i.e. don't use "ok" or "Ok")

payor (although this spelling is not found in the dictionary, convention maintains that "payer" sounds a bit awkward; the "o" spelling is preferred)

Setup (noun, adj.) (not "Set-up") e.g. Setup Menu, system setup

sub-directory (not "sub directory" or "subdirectory")

submenu (not "sub-menu" or "sub menu")

-to-date Year-to-date, period-to-date, etc., (not "Year to date")

Zoom feature (not "ZOOM") e.g. Use [CTRL]-[z] to Zoom into the reference file.

Specific Terms

Code: Use the term "Code" when non-numeric codes may be used (even if we don't), instead of the terms Key or Number.

Example: Customer Code, Dept. Code.

Number: Use the term "Number" when the key must be a number.

Example: Account Number.

Capitalize the second of two words joined with a hyphen.

Example: Ship-To (not Ship-to), Bill-To Address.

Command: Use "command" to describe an action on a ring menu of a screen or menu. Do not refer to ring menu commands as "menu options".

Menu Option: Use this term to describe the items of individual menus.

Example: the a-z menu options.

Function: In data entry, this refers to a control-key or special key action.

Example: [CTRL]-[z] is the Zoom function.

Use "Name" or another term when this is preferable to "Description."

Example:

---- Business Name ---

--- Shipping Instructions ----

Debits and Credits: Whenever debits and credits are listed together, put "debits" first (i.e. to the left).

Describing Keystrokes

Single keys should appear with square brackets around them, e.g.

[e] [TAB] [ENTER] [CTRL]-[w]

If more than three consecutive keys are referenced, they should be enclosed in quotation marks, rather than square brackets, to set them off from other text.

Examples:

Type: [s] [h] [ENTER]

Type: "kill -9 $$ [ENTER]"

(The kill command does not surround characters with brackets, with the exception of the [ENTER] key since this key is represented by more than one character).

Keystrokes that are represented by more than one character should have characters in uppercase, e.g.

[ENTER] [RETURN] [CTRL]

Keys should be named the way they appear on the most common keyboard.

The IBM PC is the standard, with the exception of the [ENTER] key.

This means that it is [CTRL] rather than [CONTROL], and [ESC] rather than [ESCAPE].

The following keys should be shown in their most general form, as shown:

[ENTER] - (not [RETURN] or [RTRN])

[ESC] - (not "store" or "escape sequence")

[DEL] - (not [BREAK], [CTRL]-[c], or [CTRL]-[BREAK])

[CTRL] - (not [CONTROL])

[SPACEBAR] - (not [SPACE])

Representing User Input

User input that may be in either uppercase or lowercase should be shown in lowercase, e.g.

Type [r] [d] [ENTER] to run the "Run Data" option.

When the program requires caps, caps should be shown,e.g.

[A] [B] [ENTER].

Keys pressed simultaneously should be separated by a dash, e.g.

[CTRL]-[w]

Keys pressed consecutively should be separated by a space, e.g.

[ESC] [1]

All alphanumeric data should be converted to all uppercase by the program unless there is an application-specific reason for not doing so. One such exception would be in the text of a letter or in a mailing list where the names are going to appear in the text of a letter.

Adding or Modifying Tables

Table Naming

- 1st 2 chars: identify authors of modifications
- 3rd char: identifies application (ex: "g" for g/l)
- characters 4-7: type of file (ex: cntrc for control)
- last character (e: entry, d: detail, c: control, r: reference)

Provide fully annotated schema files for all tables. Describe the structure and usage of the table and each column within the table.

Example of the desired documentation:

```
(from the stoordre table)

pay_method char(6)

     This code is defaulted from the customer table. If it
     is not present in the customer table then it is
     defaulted from the stocntrc table and validated from
     the stxinfor table. If the customer is the one-time
     customer (see cust_code), then the pay_method defaults
     to "CASH". CASH/VISA/AMEX/MC/ONACCT/3RDPTY are
     pay_method examples.
```

It is also helpful if the column descriptions can be grouped by function. In the `stoordre` table, for example, we group columns in the following categories:

Order numbers:

Order Control:

Master Order Information:

Date Stamps:

Line Default Information:

Customer Information:

Order Terms:

Order Total Amounts:

System Maintained columns:

Some concepts or constructs will need more detailed explanation, or will logically be referenced from many places in the documentation. Please provide these explanations in separate files, or a single "expanded info" file. These expansions will then be referenced in the schema documentation. In the example above we have referenced additional information regarding one_time_customers.

This information is stored in the expanded info file:

```
One time customers: If stocntrc.one_time_cust matches
the customer on the order, the customer is identified
as a one time customer. Sample data comes with the
one_time_cust = "CASH". If no one-time customers are
allowed, the column is left null. If the customer on
the order matches the one-time customer code, a window
will be provided to allow for the entry of
name/address. One-time customers will not be allowed
payment types of ONACCT. The payment type will be
required at the time the one-time customer is identi-
fied.
```

Testing Modifications to Tables

Adding a Column

If a column is added to a table the results need to be carefully considered. Installing on an existing database with rows in this table may create problems.

If null cannot be stored in this column, it is preferable to run a script at installation time that fills the column and informs the installer what is happening. Addendums instructing the installer to fill the column may be lost.

Are there any joins based on this column? If so, the installer will not be able to fill this "non-null" column because the join (based on one or more null columns) will fail.

Modifications to Tables

Do an install over an existing database to test changes to `dbmerge`. Test every program that accesses newly added columns. Test it from the end-user's perspective. If they have an existing database with the old version of the table, now their table has been modified. Does the program work as the developer expected it to work?

Input Program Conventions

Primary Screens

The release of the 3.6 version of Fitrix Accounting, marked a major change in how the input programs are displayed and how they run. The new style can be found in `oe/ic.` The look reflects the new concept that the main screen should contain the common information, while the more infrequent information is to be entered via Entry-By-Exception screens to the main header/detail screens. These contain the rest of the various pieces of data entry (document) information.

Starting with `oe/ic`, and any new system, the entry tables should contain all the information required to do the posting routines. All the data need not be on the main screen, but may be distributed among the Entry-by-Exception screens as well.

Use generated Zooms on discrete fields, and Picker Zooms on entry fields where more than one piece of information may be found. It should be noted that Zooms need not be regenerable, but that obvious single table Zooms should be. The Entry-By-Exception screens are derived from header input screens which have been constructed as "custom" screens. This is due to the record-locking limitations of the "add-on" header screens.

The first set of conventions will cover the basic input screen, using an example `screen.per` in `ic.4gm/i_invtr.4gs`.

Example `screen.per` in `i_invtr.4gm`:

```
{#########################################################################
# Copyright (C) 1991

# Use, modification, duplication, and/or distribution of this
# software is limited to the terms of the software agreement.
# Sccsid:  @(#)  .../ic.4gm/i_nvtr.4gs/screen.per  1.9  Delta: 10/24/91
#########################################################################
# Screen Generator version: 3.51.00 }
DATABASE standard

SCREEN
{
------------------------ Maintain Inventory Item --------------------------
             Item Code:[A1                  ]
           Description:[A2                          ]
                       :[A3                          ]
            Item Class:[A4    ][A5                          ]
            Serialized:[A]  Price Group:[A6    ]  Market Price:[B]
         Stocking Unit:[A7]   Weight:[A8          ][A9] Volume:[B1          ]
          Selling Unit:[B2]    Conversion Factor:[B3           ]
       Purchasing Unit:[B4]    Conversion Factor:[B5           ]
       Inventory Acct.:[B6       ][B7                            ]
    Cost of Goods Acct.:[B8       ][B9                            ]
          Sales Acct.:[C1       ][C2                            ]
-- Warehouse --Location --- Vendor - Qty. on Hand ---- Cost ------ Price --
[C][C3        ][C4          ][C5    ][C6         ][C7         ][C8         ]
[C][C3        ][C4          ][C5    ][C6         ][C7         ][C8         ]
[C][C3        ][C4          ][C5    ][C6         ][C7         ][C8         ]
[C][C3        ][C4          ][C5    ][C6         ][C7         ][C8         ]
[C][C3        ][C4          ][C5    ][C6         ][C7         ][C8         ]
}
TABLES
     stiinvtr
     stilocar
ATTRIBUTES
A1 = stiinvtr.item_code, upshift,
     comments = " Enter the code for this inventory item.";
A2 = stiinvtr.desc1, upshift,
     comments = " Enter the description for this inventory item.";
A3 = stiinvtr.desc2, upshift,
     comments = " Enter the description for this inventory item.";
A4 = stiinvtr.item_class, upshift,
     comments = " Enter the class for this inventory item.";
A5 = formonly.class_desc type char, noentry, comments = "";
A  = stiinvtr.serialized, include = ("",S,B,L), upshift,
     comments = " Enter [S]erial number, [L]ot number, or [B]oth if this item is
     serialized";
A6 = stiinvtr.price_group, upshift,
     comments = "Enter price group code.";
B  = stiinvtr.market_price, include =(Y, N), upshift,
     comments = "Is the price for this item subject to change base on the market
     value?";
A7 = stiinvtr.stock_unit, upshift,
     comments = " Enter the unit type this item will be stocked by."; A8 =
     stiinvtr.weight,
     comments = " Enter the weight of this inventory item.";
A9 = stiinvtr.weight_unit, upshift,
     comments = " Enter the weight unit for this inventory item.";
B1 = stiinvtr.volume,
     comments = " Enter the volume of this inventory item.";
B2 = stiinvtr.sell_unit, upshift,
     comments = " Enter the unit type this item will be sold by.";
B3 = stiinvtr.sell_factor, format = "------.######",
     comments = " Enter the selling conversion factor (negative for
reciprocal).";
```

```
B4 = stiinvtr.purch_unit, upshift,
     comments = " Enter the unit type this item will be purchased by.";
B5 = stiinvtr.purch_factor, format = "------.######",
     comments = " Enter the purchase conversion factor (negative for
reciprocal).";
B6 = stiinvtr.inv_acct_no,
     comments = " Enter the Inventory Account Number for this item.";
B7 = formonly.inv_desc type char, noentry, comments = "";
B8 = stiinvtr.cog_acct_no,
     comments = " Enter the Cost of Goods Account Number for this item.";
B9 = formonly.cog_desc type char, noentry, comments = "";
C1 = stiinvtr.sales_acct_no,
     comments = " Enter the Sales Account Number for this item.";
C2 = formonly.sales_desc type char, noentry, comments = "";
C  = formonly.cmd_line type char, upshift,
     comments = " [CTRL]-[Z] to Zoom into warehouse.";
C3 = stilocar.warehouse_code, noentry, comments = "";
C4 = stilocar.stock_location, noentry, comments = "";
C5 = stilocar.vend_code, noentry, comments = "";
C6 = stilocar.qty_on_hand, format = "------&.&&&", noentry,comments = "";
C7 = stilocar.purch_unit_cost, format = "------&.&&&", noentry, comments = "";
C8 = stilocar.price, format = "------&.&&&", noentry, comments = "";

INSTRUCTIONS
screen record s_iinvtr (stiinvtr.item_code, stiinvtr.desc1,
     stiinvtr.desc2, stiinvtr.item_class, formonly.class_desc,
     stiinvtr.serialized, stiinvtr.price_group, stiinvtr.market_price,
     stiinvtr.stock_unit, stiinvtr.weight, stiinvtr.weight_unit,
     stiinvtr.volume, stiinvtr.sell_unit, stiinvtr.sell_factor,
     stiinvtr.purch_unit, stiinvtr.purch_factor, stiinvtr.inv_acct_no,
     formonly.inv_desc, stiinvtr.cog_acct_no, formonly.cog_desc,
     stiinvtr.sales_acct_no, formonly.sales_desc)

screen record s_ilocar[5] (formonly.cmd_line, stilocar.warehouse_code,
     stilocar.stock_location, stilocar.vend_code, stilocar.qty_on_hand,
     stilocar.purch_unit_cost, stilocar.price)

delimiters "  "
{
################################################################################
FOURGEN
################################################################################

defaults
     module     = ic
     type       = header/detail
     attributes = white
     location   = 2, 3
input 1
     table   = stiinvtr
     key     = item_code
     order   = item_code
     lookup  = name=stxchrtr1, key=inv_acct_no, table=stxchrtr,
     from_into=acct_desc inv_desc, filter=acct_no = $inv_acct_no
     lookup  = name=stxchrtr2, key=cog_acct_no, table=stxchrtr,
     from_into=acct_desc cog_desc, filter=acct_no = $cog_acct_no
     lookup  = name=stxchrtr3, key=sales_acct_no, table=stxchrtr,
     from_into=acct_desc sales_desc, filter=acct_no = $sales_acct_no
     lookup  = name=stxinfor1, key=item_class, table=stxinfor,
     from_into=src_desc class_desc,
     filter=src_type = "P" and src_key = $item_class
     zoom    = key=inv_acct_no, screen=chrtz, table=stxchrtr, noautozoom
     zoom    = key=cog_acct_no, screen=chrtz, table=stxchrtr, noautozoom
     zoom    = key=sales_acct_no, screen=chrtz, table=stxchrtr,
     noautozoom
     zoom    = key=item_class, screen=clssz, table=stxinfor, noautozoom,
     filter=stxinfor.src_type = "P"
     req'd = item_class
```

```
input 2
     table   = stilocar
     order   = warehouse_code
     join    = stiinvtr.item_code = stilocar.item_code
     arr_max = 100 }
```

Input Screen Conventions

Title

- Dashed line extends all the way to the edge of window borders.
- Space before & after title.
- Centered.
- Don't use the word "screen" or "document".

Field Titles

- Capitalized.
- Terminated with ":".
- Evenly spaced across screen.

Screen Divider Lines

- Indent one column in from main title, or 3 blank columns separated dashed line from window borders.

Columns

- Character & fixed length columns, left justify the heading.
- Numeric (non-fixed length fields), right justify the heading.

Bottom of Screen

- Extends all the way to edge of window.

Attributes of Screen, Upshift, and Comments

- Upshift all alphanumeric fields.
- Comments on all input fields should begin with a Capital letter.

- Comment should describe the operation needed.

Attributes of Screen, Format

- Amounts, and quantities need to be formatted.
- Usually either "--------&.&&", or "-------&.&&&".

Attributes of Screen, Delimiters

- Delimiters clause is always " "

FourGen Section, Defaults

- Attributes = white is the color of the background
- On header and header/detail, and add-on header screens

FourGen Section, Input1 Lookups

- Lookups always have unique names when used against the same table, using different fields.
- Use "from-into" clause.

FourGen Section, Input1 Zooms

- Discrete Zooms should be generated.

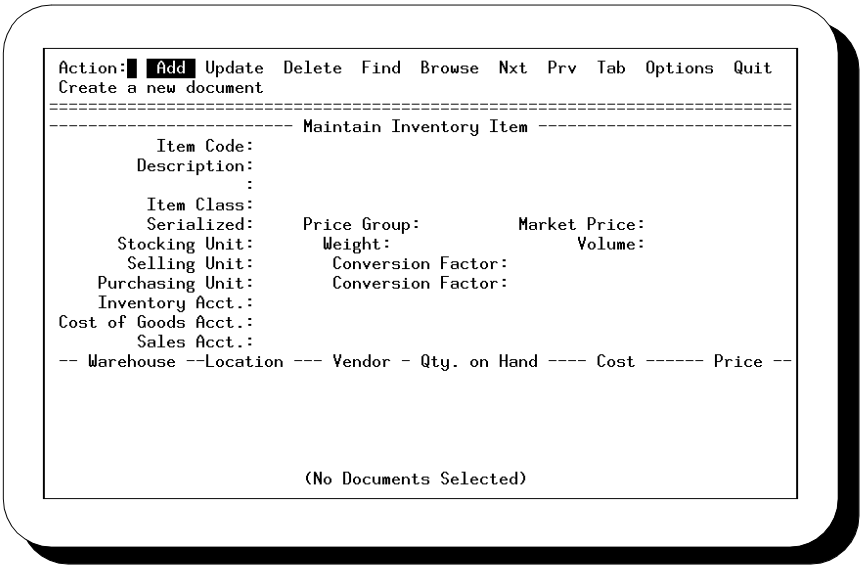
FourGen Section, Input1 Req'd

- Required fields when possible should be generated.

FourGen Section, Input2 Array Size

- Should be 100.

This is how the screen appears:



Data Entry Screen Design

The Screen or form title should appear in the center of the first line of the screen. It should start with a cap, but should not be all caps. It should not contain the words "Screen" or "Form." It should be a noun describing the document or the contents of the file.

It should not contain a verb unless the menu item that calls the screen only allows one possible action and other menu items allow different actions on the same file.

Screens and menu items are named for the contents of the file or table they access, not a report or the file itself. For example, the screen should say, "Ledger Accounts" not "Chart of Accounts."

When a file contains documents, they should be called by the most widely recognized name. For example: invoices, orders, checks, etc...

If there is one document in a file, the name appearing on the screen should be singular. If there is more than one, the name should be plural. For example, Company

Information, Account Groups, and Customers (not Customer).

If an identifying document number is assigned by the system and cannot be changed by the user, it should appear on the same line as the heading in the right hand corner.

If the screen contains rows of detail, the detail should appear in the middle of the screen. Separate this detail from the "header" information by a line.

If the screen contains totals from the detail, that information should appear below the detail, separated by a line in a special footer section.

If a form has no detail section and has room for another line, there should be a solid line of dashes from one side to the other just above the comment line at the bottom of the form.

Whenever possible, fields on a document header should appear on the form one to a screen row.

Fields read in from a detail table should appear in rows and columns.

A "field descriptor" is used to indicate or "label" the contents of a field whenever possible. Descriptors for fields in a heading should appear before the field. Descriptors for fields arranged in columns and rows should appear at the top of the column.

Descriptors should be Capitalized, NOT ALL CAPS.

Descriptors for data entry fields in which the field follows to theright of the descriptor should be followed by a colon.

Descriptors and headings should not be abbreviated unless this is required for spacing.

Abbreviations should be followed by a period, such as Ref. or Salesprsn.

Do not use the symbol # to indicate numbers if No. will fit.

Do not use No. if Number will fit.

Data entry fields should be stacked, each starting in the same column. Their headings should be placed so that their ending colon is one character to the left of the beginning of the field.

Display-only fields should not start in the same column as data entry fields. Display-only fields should form their own column, preferably to the right of the data entry fields.

Row descriptors start, if an area is large enough, over the first character of the value in a text field. In a numeric column, they end over the last character of data (right justified).

If a column is narrower than the heading, the heading is centered.

Headings are not followed by colons or other punctuation.

Screens should not include technical information or terms such as program names or filenames, which is of no interest to the users.

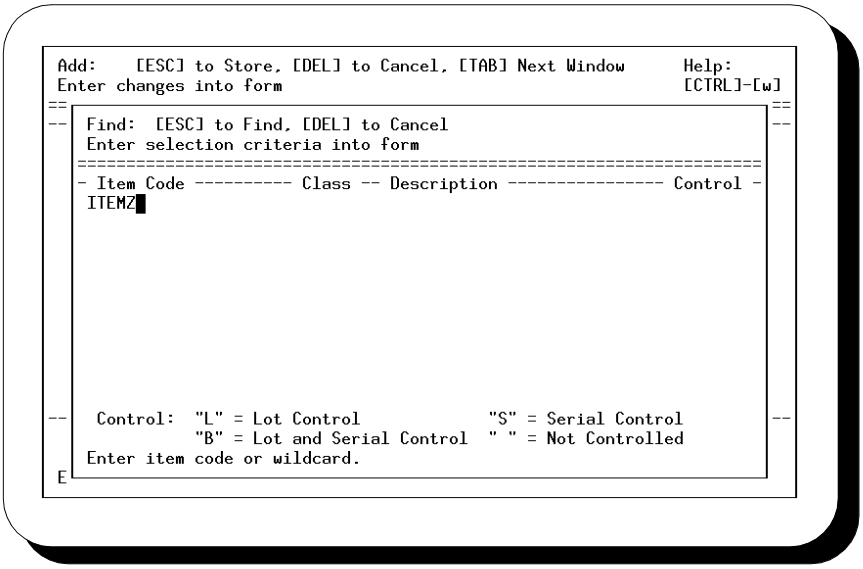
Zoom Screens

Example `itemz.per` from `ic.4gm/lib.4gs`

```
{########################################################################
# Copyright (C) 1991
#
# Use, modification, duplication, and/or distribution of this
# software is limited to the terms of the software agreement.
# Sccsid:  @(#)  .../ic.4gm/lib.4gs/itemz.per  1.2  Delta: 5/10/91
########################################################################
# Screen Generator version: 3.60.00 }
DATABASE standard
SCREEN
{ - Item Code ---------- Class -- Description ---------------- Control - [A1
][A2    ][A3                              ] [A]
[A1                  ][A2    ][A3                              ] [A]
[A1                  ][A2    ][A3                              ] [A]
[A1                  ][A2    ][A3                              ] [A]
[A1                  ][A2    ][A3                              ] [A]
[A1                  ][A2    ][A3                              ] [A]
[A1                  ][A2    ][A3                              ] [A]
[A1                  ][A2    ][A3                              ] [A]
[A1                  ][A2    ][A3                              ] [A]
[A1                  ][A2    ][A3                              ] [A]
[A1                  ][A2    ][A3                              ] [A]
   Control:  "L" = Lot Control            "S" = Serial Control
             "B" = Lot and Serial Control  " " = Not Controlled }
TABLES
     stiinvtr
ATTRIBUTES A1 = stiinvtr.item_code, upshift,
     comments = " Enter item code or wildcard.";
A2 = stiinvtr.item_class, upshift,
     comments = " Enter product class or wildcard.";
A3 = stiinvtr.desc1, upshift,
     comments = " Enter first description line or wildcard.";
A  = stiinvtr.serialized, upshift,
     comments = " Enter serial control type.";
INSTRUCTIONS
screen record s_itemz[10] (stiinvtr.item_code, stiinvtr.item_class,
     stiinvtr.desc1, stiinvtr.serialized)
```

```
delimiters "  "
{
########################################################################
FOURGEN
########################################################################
defaults
module     = ic
     type       = zoom
     attributes = white, border
     location   = 5, 6
     returning  = item_code
input 1
     table   = stiinvtr
     key     = item_code
     order   = item_code
     arr_max = 100
}
```

The `itemz.per` file looks like this on the screen:



Zoom Conventions

Zoom Screen Title Line

- Dash followed by space and column descriptions.
- Don't use the word "screen" or "document".

Zoom Screen Array

- Ten lines if possible.
- Character & fixed length columns, left justify the heading.
- Numeric (non-fixed length fields), right justify the heading.

Zoom Screen Information Lines

- May contain operator prompting information.

FourGen Section, Defaults

- Header Zooms usually start at 5,6 since 2,3 is the top of the Zoom message block.

Entry-By-Exception(EBE) Custom Input Screens

The typical Entry-By-Exception screen is hand-coded, after generation as a header screen. Access to Entry-By-Exception and Zoom screens are now done by pickers. When the user presses [CTRL]-[z], a picker pops up allowing the ability to access other input/Zoom screens.

The EBE programs as found in Order Entry are to be used as templates for programming EBE screens. They are chiefly found in `oe.4gm/i_order.4gs` directory. There are several used such as `shipd.4gl`.

Example `locau.per` from `ic.4gm/i_invtr.4gs`:

```
{########################################################################
# Copyright (C) 1990
# FourGen Software Technologies, Inc. Edmonds, Washington USA
# Use, modification, duplication, and/or distribution of this
# software is limited to the terms of the software agreement.
# Sccsid:  @(#)  .../ic.4gm/i_invtr.4gs/locau.per  1.6  Delta: 10/24/91
########################################################################
non_source_form
}
DATABASE standard
SCREEN
{
------------------------- Item Warehouse Detail ---------------------------
     Item:[A1                ][A2                            ]
```

```
Warehouse:[A3        ][A4                            ]
---------------------- Cost and Price Information ----------------------
Purchase Cost:[A6          ] Last Cost:[A7          ] Qty.:[A8          ]
 Average Cost:[A9          ] Last Date:[B1      ]
        Price:[B3          ] Sold Date:[B2      ]
-------------------- Location and Count Information --------------------
     Location Aisle:[B4  ]        Row:[B5 ]           Bin:[B6 ]
   Count Cycle Code:[A]    Last Count:[B7      ]  On Hand:[B8          ]
------------------------- Vendor Information --------------------------
             Vendor:[B9    ][C1                              ]
        Vendor Item:[C2                  ]
------------------------ Selling Information --------------------------
     Minimum Sell Qty.:[C3        ] Allow Backorder:[B]      Taxable:[C]
Subject To Terms Disc.:[D]                    Subject To Trade Disc.:[E]
       Commission Code:[C4    ][C5                              ] }
TABLES
   stilocar
   stiinvtr
ATTRIBUTES
A1 = stiinvtr.item_code, noentry, comments = "";
A2 = stiinvtr.desc1, noentry, comments = "";
A3 = stilocar.warehouse_code, upshift,
     comments = " Enter the code for this warehouse.";
A4 = formonly.wh_desc type char, noentry, comments = "";
A6 = stilocar.purch_unit_cost, format = "-------&.&&&",
     comments = " Enter the cost for the purchase of one stock unit.";
A7 = stilocar.last_cost, format = "-------&.&&&", noentry,comments = "";
A8 = stilocar.last_qty, format = "-------&.&&&", noentry,
     comments = ""; A9 = stilocar.avg_unit_cost, format = "-------&.&&&",
     comments = " Enter the average unit cost for one stock unit.";
B1 = stilocar.purchase_date, format = "mm/dd/yy", noentry, comments = "";
B2 = stilocar.sold_date, format = "mm/dd/yy", noentry, comments = "";
B3 = stilocar.price, format = "-------&.&&&",
     comments = " Enter the sale price for this item.";
B4 = stilocar.loc_aisle, upshift,
     comments = " Enter the location aisle.";
B5 = stilocar.loc_row, upshift,
     comments = " Enter the location row.";
B6 = stilocar.loc_bin, upshift,
     comments = " Enter the location bin.";
A  = stilocar.count_cycle, upshift,
     comments = " Enter the code for this item's count cycle [A-Z, 0-9].";
B7 = stilocar.count_date, format = "mm/dd/yy", noentry, comments = "";
B8 = stilocar.qty_on_hand, format = "-------&.&&&",
     comments = " Enter the quantity on hand for this item.";
B9 = stilocar.vend_code, upshift,
     comments = " Enter the vendor code.";
C1 = formonly.vend_desc type char, noentry, comments = "";
C2 = stilocar.vend_prod_no, upshift,
     comments = " Enter the vendor product number.";
C3 = stilocar.min_sell_qty, upshift,
     comments = " Enter minimum sell quantity.";
B  = stilocar.allow_bo, upshift,
     comments = " Enter [Y] to allow for this item to go on backorder.";
C  = stilocar.taxable, upshift,
     comments = " Enter [Y] if this is a taxable item.";
D  = stilocar.terms_disc, upshift,
     comments = " Enter [Y] if this item should be subject to terms discount.";
E  = stilocar.trade_disc, upshift,
     comments = " Enter [Y] if this item should be subject to trade discount.";
C4 = stilocar.comm_code, upshift,
     comments = " Enter the commission code for this item.";
C5 = formonly.comm_desc type char, noentry, comments = "";
INSTRUCTIONS
screen record s_ilocar (stiinvtr.item_code, stiinvtr.desc1,
     stilocar.warehouse_code, formonly.wh_desc,
     stilocar.purch_unit_cost, stilocar.last_cost, stilocar.last_qty,
     stilocar.avg_unit_cost, stilocar.purchase_date, stilocar.sold_date,
```

```
     stilocar.price, stilocar.loc_aisle, stilocar.loc_row,
     stilocar.loc_bin, stilocar.count_cycle, stilocar.count_date,
     stilocar.qty_on_hand, stilocar.vend_code, formonly.vend_desc,
     stilocar.vend_prod_no, stilocar.min_sell_qty, stilocar.allow_bo,
     stilocar.taxable, stilocar.terms_disc, stilocar.trade_disc,
     stilocar.comm_code, formonly.comm_desc)
delimiters "  "
{
########################################################################
FOURGEN
########################################################################
defaults
     module     = ic
     type       = header
     attributes = white
     location   = 2, 3
input 1
     table   = stilocar
     key     = item_code, warehouse_code
     order   = warehouse_code
     lookup  = name=stxinfor, key=warehouse_code, table=stxinfor,
     lookup  = name=whse_description, key=wh_desc, table=stiwhser,
     from_into=description wh_desc, filter=whse_code = $warehouse_code
     lookup  = name=stxinfor1, key=comm_code, table=stxinfor,
     from_into=src_desc comm_desc, filter=src_type = "C" and src_key = $comm_code
     lookup  = name=stpvendr, key=vend_code, table=stpvendr,
     from_into=bus_name vend_desc,  filter=stpvendr.vend_code = $vend_code
     zoom    = key=vend_code, screen=vendz, table=stpvendr
     zoom    = key=comm_code, screen=commz, table=stxinfor,
       filter=src_type = "C"
     zoom    = key=warehouse_code, screen=whser, table=stiwhser,
       from=whse_code, noautozoom }
```

Input Screen Conventions (Entry By Exeption Screen)

Zoom Screen Copyright Line

- `Non_source_form` added so Entry-By-Exception code will not be destroyed by re-generation.

FourGen Section

- This was used to generate the original code and can be reused if `non_source_form` is removed, and generator run.

Standard Report Design

Like screens, reports should not include technical information or terms such as program names or file names, which are not of interest to the users.

Report headings should not be all uppercase.

Report headings should not contain the word "Report".

Text elements on printouts should not be all uppercase.

Only data should show in all caps.

The heading should be separated from the body of a report by a double line (formed by equals symbols).

All information that repeats on every page is part of the page heading and should appear above the double line.

Date and Time of Report should appear one above the other in upper left corner. Page number should appear in the upper right corner. These words should appear as `Date:` and `Page:`

The first line should be the report name. Nothing else should appear on that line.

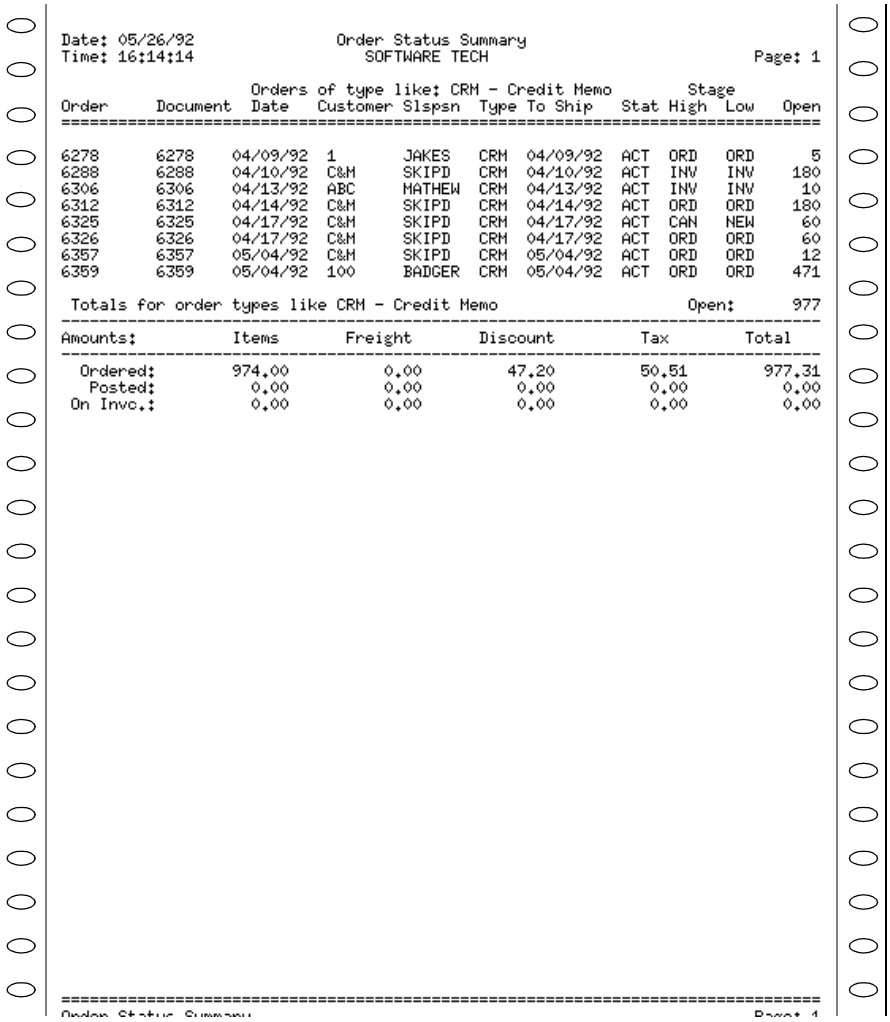
If a trailer is used, it should contain only the name of the report and the page number in the lower right-hand corner, in the format:

```
Report Name Page:
```

Double lines should be used to separate additional information from the body of the report.

Reports that contain codes such as "order status" or "document source" should have a legend explaining the codes at the bottom of the form.

Output Program Modifications



```
Date: 05/26/92                Order Status Summary
Time: 16:14:14                   SOFTWARE TECH                              Page: 1

                      Orders of type like: CRM - Credit Memo          Stage
Order     Document  Date     Customer Slspsn  Type To Ship   Stat High  Low   Open
=============================================================================

6278      6278    04/09/92  1        JAKES   CRM  04/09/92  ACT  ORD   ORD      5
6288      6288    04/10/92  C&M      SKIPD   CRM  04/10/92  ACT  INV   INV    180
6306      6306    04/13/92  ABC      MATHEW  CRM  04/13/92  ACT  INV   INV     10
6312      6312    04/14/92  C&M      SKIPD   CRM  04/14/92  ACT  ORD   ORD    180
6325      6325    04/17/92  C&M      SKIPD   CRM  04/17/92  ACT  CAN   NEW     60
6326      6326    04/17/92  C&M      SKIPD   CRM  04/17/92  ACT  ORD   ORD     60
6357      6357    05/04/92  C&M      SKIPD   CRM  05/04/92  ACT  ORD   ORD     12
6359      6359    05/04/92  100      BADGER  CRM  05/04/92  ACT  ORD   ORD    471

 Totals for order types like CRM - Credit Memo                    Open:      977
-----------------------------------------------------------------------------
Amounts:          Items       Freight       Discount          Tax         Total
-----------------------------------------------------------------------------
  Ordered:       974.00          0.00          47.20         50.51        977.31
   Posted:         0.00          0.00           0.00          0.00          0.00
 On Invc.:         0.00          0.00           0.00          0.00          0.00

=============================================================================
Order Status Summary                                                  Page: 1
```

.ifg Files

The following discussion pertains to this example report:

Below is the `report.ifg` file from `oe.4gm/o_ordsum.4gs` used to generate the above report.

```
################################################################################
# Copyright (C) 1990
# FourGen Software Technologies, Inc. Edmonds, Washington USA
# Use, modification, duplication, and/or distribution of this
# software is limited to the terms of the software agreement.
# Sccsid:  @(#)  .../oe.4gm/o_ordsum.4gs/report.ifg  1.4  Delta: 4/1/91
################################################################################

database standard

output
     top margin     3
     bottom margin  3
     left margin    0
     right margin   80
     page length    66
page header
{
Date: [h1    +h2    >h3
Time: [h4    +h5    >h6
+h7                                                              Stage
Order     Document  Date   Customer Slspsn  Type To Ship   Stat High  Low   Open
================================================================================
==}
on every row
{
[e1       [e2     [e3       [e4     [e5     [e6  [e7       [e8  [e9   [e [e11 }
page trailer
{
================================================================================
==[h2           >h6
}
before group of stoordre.like_type
page
{
}
after group of stoordre.like_type
{
Totals for order types like [g1 - [g2                          Open:[g3
--------------------------------------------------------------------------------
-
Items       Freight       Discount         Tax        Total
--------------------------------------------------------------------------------
-
 Ordered:[g4           [g5           [g6           [g7           [g8
  Posted:[g9           [g10          [g11          [g12          [g13     [*
On Invc.:[g14          [g15          [g16          [g17          [g18     [* }
note to programmer:  put the following comments and code in report.4gl
immediately after the line "after group of stoordre.like_type
     # This code handles the "wrong header" problem (when the
     # exact number of lines on the page is met, the wrong header
     # is placed on the next page)
     # The 53 was calculated using the following formula:
     #     lines per page: 66 -
     #      bottom margin:  3 -
     #       page trailer:  2 -
     #   after group need:  8
     #                      --
     #                      53
     if lineno = 53
     then
       need 7 lines
```

```
     else
       need 8 lines
       print
     end if
attributes
     h1  = date, using "mm/dd/yy"
     h2  = constant "Order Status Summary"
     h3  = formonly.select_msg type char(30)
     h4  = time
     h5  = formonly.co_name type like stxcntrc.co_name
     h6  = pageno using "Page: <<<<<"
     h7  = formonly.hdr_type type char(60)

     e1  = stoordre.order_no
     e2  = stoordre.doc_no using "<<<<<<"
     e3  = stoordre.order_date using "mm/dd/yy"
     e4  = stoordre.cust_code
     e5  = stoordre.sls_psn_code
     e6  = stoordre.order_type
     e7  = stoordre.to_ship_date
     e8  = stoordre.order_status
     e9  = stoordre.hi_stage
     e   = stoordre.lo_stage
     e11 = formonly.open_amount type like stoordre.total_amount using "------&"

# the following 13 fields are dummy fields intended to produce the
# corresponding rpt record variables.
     d1  = formonly.pst_item_amount type like stoinvce.item_amount
     d2  = formonly.pst_frght_amount type like stoinvce.frght_amount
     d3  = formonly.pst_trd_ds_amount type like stoinvce.trd_ds_amount
     d4  = formonly.pst_tax_amount type like stoinvce.st_tx_amount
     d5  = formonly.pst_total_amount type like stoinvce.total_amount
     d6  = formonly.inv_item_amount type like stoinvce.item_amount
     d7  = formonly.inv_frght_amount type like stoinvce.frght_amount
     d8  = formonly.inv_trd_ds_amount type like stoinvce.trd_ds_amount
     d9  = formonly.inv_tax_amount type like stoinvce.st_tx_amount
     d10 = formonly.inv_total_amount type like stoinvce.total_amount

     g1  = stoordre.like_type
     g2  = formonly.type_desc type like stootypr.description
     g3  = sum(formonly.open_amount) using "--------&"
     g4  = sum(stoordre.item_amount) using "---,---,--&.&&"
     g5  = sum(stoordre.frght_amount) using "---,---,--&.&&"
     g6  = sum(stoordre.trd_ds_amount) using "---,---,--&.&&"
     g7  = sum(stoordre.tax_amount) using "---,---,--&.&&"
     g8  = sum(stoordre.total_amount) using "---,---,--&.&&"
     g9  = sum(formonly.pst_item_amount) using "---,---,--&.&&"
     g10 = sum(formonly.pst_frght_amount) using "---,---,--&.&&"
     g11 = sum(formonly.pst_trd_ds_amount) using "---,---,--&.&&"
     g12 = sum(formonly.pst_tax_amount) using "---,---,--&.&&"
     g13 = sum(formonly.pst_total_amount) using "---,---,--&.&&"
     g14 = sum(formonly.inv_item_amount) using "---,---,--&.&&"
     g15 = sum(formonly.inv_frght_amount) using "---,---,--&.&&"
     g16 = sum(formonly.inv_trd_ds_amount) using "---,---,--&.&&"
     g17 = sum(formonly.inv_tax_amount) using "---,---,--&.&&"
     g18 = sum(formonly.inv_total_amount) using "---,---,--&.&&"

select
     more   = stoordre.item_amount
     more   = stoordre.frght_amount
     more   = stoordre.trd_ds_amount
     more   = stoordre.tax_amount
     more   = stoordre.total_amount
     tables = stoordre
     order  = stoordre.like_type, stoordre.order_no, stoordre.doc_no

defaults
     progname  = o_ordsum
```

```
prcname   = Order Status Summary
rtmargn   = FourGen Modifiable Accounting
destin    = report.out
quiet     = 10
```

Report Conventions:

Heading Date, Time, Page Number and Field Tags

Date and Time of Report should appear one above the other in upper left corner.

Page number should appear in the upper right corner.

Use field tags beginning with 'h'.

First line should be the report name.

Heading Separation

Separated from body of report by double line (equals symbols).

Field Tags From a Entry Header Table

Use field tags beginning with 'e'.

Page Trailers

Contains, as last line, the name of the report and the page number in the lower right hand corner.

Text Elements on Printouts

Should not be all caps, only DATA should show in all caps.

Field Tags Appearing in the Before or After Group

Use field tags beginning with 'g'.

Data From Company Table (like company name)

Use the function in `all.4gm/lib.4gs` called `al_compn`, which returns the company name.

Called from `ml_defaults`, like `let rpt.co_name = al_compn()`.

Field Tags From an Entry Detail Table

Use field tags beginning with 'd'.

NOT shown:

Field Tags Appearing in the Subtotals Section.

Use field tags beginning with 's'.

Field Tags Appearing in the Totals Section.

Use field tags beginning with 't'.

Report Width

Reports may be either 256, 132 or 80 columns wide depending upon the data to be printed.

Posting Program Modifications

Basic Requirements

Error checking in edit list and posting must be IDENTICAL. If the edit list runs without problem, the posting should take place.

An edit list must be printed before the posting of the document.

Any SQL statements which delete rows from header and/or detail table(s) should be done in the `after_group` section.

Never use extra outer joins as lookups (i.e., select business name from `stpvendr` based on vendor code), instead use lookup.

Function calls in `on_every_row`.

OnLine compatible

Rowid for temp tables are not guaranteed to be sequential.

Begin work and commit work will close cursors, therefore use temp tables instead.

Select rowid of the tables involved into the temp table, and in `ml_fetch` do the actual selection of the data by rowid. This allows the program to get the most recent record.

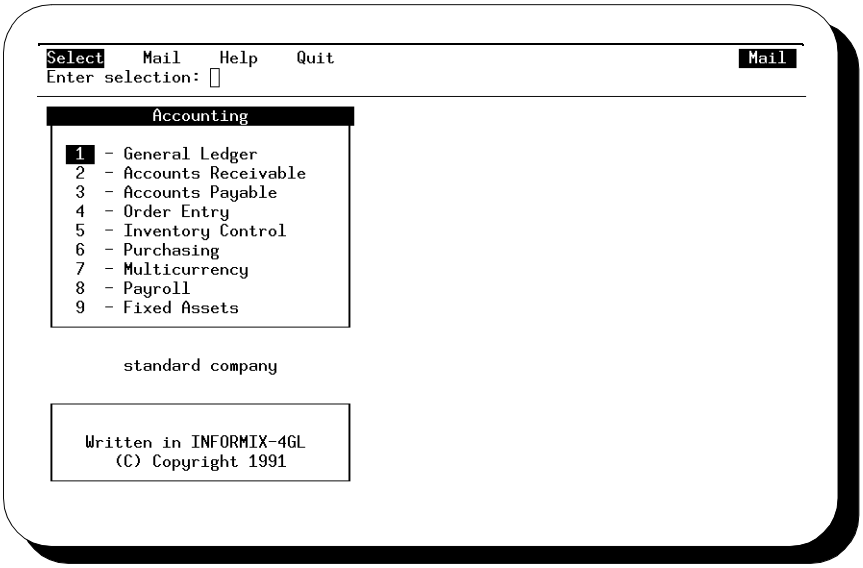
Use `oe.4gm/p_pstord.4gs` for a reference on how to code a posting program using the latest FourGen techniques.

Commit/Rollback Logic

If the document has an error, do not post that document (rollback work) but instead continue posting the rest of the documents.

Menu Design Conventions

Example:



Menu Options

Menu option descriptions should be no more than 30 characters. Menu options should be written with the first letter of important words capitalized, as you would the title of a book. Do not capitalize articles (e.g. a, the, an), conjunctions (and, or, but), prepositions, or infinitives.

Example:

Post Ledger Data to Chart

Titles of menu items should match the title on any screen or report that they call and should also match the description field of the FourGen Menus script :item: line. For example, if the menu item is Update Ledger Accounts, the screen should say, Ledger Accounts not General Ledger Accounts. If it is a general document, it should be

called just document.

The most commonly used options should appear first on the menu, lesser used selections, such as setup options, should appear last.

The report option that is associated with an update option should be directly below the update option.

All submenus of a package should be of identical width (standard width 41 characters, including the borders and one character margins on either side). The margins should contain tildes (in the "menu" file).
For example:

```
~|% a - Update Warehouse Definitions   %|~
```

Menu titles align with the menu option text within the box (not centered).

Menu option descriptions should be no more than 30 characters.

Menu options names: first letter of important words is capitalized.

Menu options names should match screen and report titles.

Menus items should begin with a standard verb that defines the activity

For example:

Create - create something new.

Update - change information in a file manually.

Print - send a list, report or form to the printer.

Post - automatically update info in one file using info from another.

Copy - duplicate records in one file into another.

Delete - purge or erase information from a file

Select - identify records for processing.

Numbered submenus should (if size allows) move from upper left corner toward the lower right corner as numbers increase.

Data in simple reference files should be called "Definitions" (i.e., "Update Warehouse Definitions" not "Update Warehouse Codes").

Never use "FILE" or "TABLE" as the title for a screen, menu, or report.

Screens and menu items are named for the contents of the file or table (i.e., "Update Journal Documents" not "Update General Journal").

If the file contains multiple items, but a description of its contents cannot be made plural, add the word Items (i.e., "Inventory Items" not "Inventory" and not "Inventories").

Menu Item Files

Item instruction files store the FourGen Menus commands. These are the files that are executed when a menu option is selected. They are located in menu directories (one directory for each menu).

Options that print reports should always include an item line and a pause that allows the user to exit without printing.

Options that require user input should inform the user of the required input (from the :item:, :needs:, or :show: lines).

The :item: line should always contain a description of the action the script performs.

The descriptions in :item: lines should be in lowercase, including the starting characters of lines. Single sentences or phrases that extend over more than one line should be indented four characters on all but the top line.

For example:

```
:item:Print Over/Short Report:prints a listing of all counted items that: are
over or short of the amount in the system:
```

(This is all one line in the file.)

FourGen Menus displays the preceding command as:

```
You have selected 'Print Over/Short Report'.

This menu selection does the following:

Prints a listing of all counted items that are over or short of the amount in the
system.
```

Do not write :item: description lines that simply repeat the option name. That is, do not write item descriptions such as:

```
:item:Print Over/Short Report:prints the over/short report:
```

Do not put the :x: exit flag on the end of report or screen lines unless it is absolutely required to prevent subsequent pc lines from executing after failure of the report or screen command.

Main Menu Design

Application or full-screen menus should be as few in number as possible. This means that larger or more general menus with many options are preferable to smaller menus with fewer items. This puts more options on the screen and requires less of the user in determining where a certain function is accessed. Small menus mean complicated menu "trees" that can be confusing for the user.

Menus are always displayed within boxes. The menu box has three sides (left, right, and bottom) drawn with box graphics characters. The top, containing the menu heading, is drawn with reverse video.

Menus with many items should be separated by graphics characters into smaller boxes, grouping options by type of function or file accessed.Each menu box contains a single column of elements. Additional related options are displayed through submenus.

A maximum of 22 options are displayed at one time within a single menu.

Selection keys are always lowercase letters, starting with the first element as "a," the second as "b," and so on.

The option lines of the menu should have the two spaces between the left border of the box and the identifying selection key, then a space, a dash, another space, and the option description.

For example:

```
|% a - Update Warehouse Definitions %|
```

All menus, with the exception of control/main menus, should highlight the entire menu item. The ending highlights should be aligned two spaces from the right border of the box.

For example:

```
|% a - Update Warehouse Definitions %|
|% b - Update Tax Definitions       %|
```

rather than:

```
|% a - Update Warehouse Definitions %|
|% b - Update Tax Definitions %      |
```

A "Control" or "Main" Menu calls up other menus on the screen.

Control Menus have a box of graphics characters around the highlighted heading line containing the title.

The heading line of a control menu should be highlighted with the @ character, so that the highlight remains when you load a submenu (unless another menu can be loaded directly over the control menu).

Menu items on a Control Menu are identified by numbers rather than letters.

All other design rules apply.

An option that returns you to a "higher" level should be identified as a "0" item and appear as the last item of a "control" box.

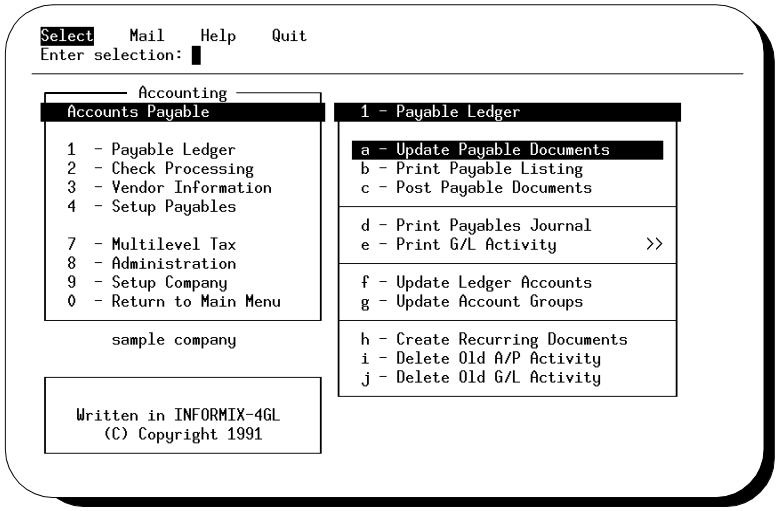
On control menus, the highlight characters (%) should surround the option keys, (not the entire item as other menus do).

For example:

```
|% 0 %- Return to Main Menu |
```

Sample Control Menu:

A "control" or "main" menu is typically displayed on the left side of the screen and calls up submenus menus displayed on the right side of the screen.



Submenu Design

The first submenu of any module should contain the options for the most frequently used transaction processing options. For example, submenu 1 of an Accounts Payable product should contain the option for entering A/P invoices.

The transaction menu(s) of a main menu should be followed by the reference file maintenance menu(s). For example, the transaction processing submenus of A/P should be followed by the menu containing the option for entering vendor information.

Setup menus and other menus containing options that are run infrequently should be put at the bottom of the list of submenus on the Control Menu.

If there is a clear chronological order for running the options of a submenu, the menu items should follow that order (i.e. the first option run should be the top item on the menu, etc.).

The preferred menu organization of data entry and report options, when the reports are basically a list of data entered, is to place the report option directly below the option(s) for entering the data included on the report.

This convention is violated in the new oe, where one detail menu contains all sorts of definition options and a "sister menu" contains all of the print options for those definitions. Placing data entry options on one portion of the menu and report options on another is not acceptable when there is a clear relationship between data entry options and list reports.

Example of good submenu structure:

```
%%- 4 - Setup Payroll ----------------%%
~|                                     |~
~|% a - Update Payroll Defaults       %|~
~|% b - Print Payroll Defaults        %|~
~+-------------------------------------+~
~|% c - Update Income Codes           %|~
~|% d - Update Deduction Codes        %|~
~|% e - Update Obligation Codes       %|~
~|% f - Print Payroll Codes           %|~
~+-------------------------------------+~
~|% g - Update Employee Types         %|~
~|% h - Print Employee Types          %|~
~+-------------------------------------+~
~|% i - Update Tax Tables             %|~
~|% j - Print Tax Tables              %|~
~+-------------------------------------+~
~|% k - Update Ledger Accounts        %|~
~|% l - Print Ledger Account List     %|~
~+-------------------------------------+~
```

As stated earlier, all submenus of a package should be of identical width. The standard width for submenus is 41 characters, including the borders and one character margins on either side. If one submenu needs to be larger, all other submenus for the product should have this same, larger width. Menu titles should align with the menu option text within the box. They should not be centered. The margins should contain tildes (in the "menu" file).

For example:

```
~|% a - Update Warehouse Definitions %|~
```

Rather than surprise the user with "detail menus" that are selected from the submenu, use two greater-than symbols (>>) to indicate menu options that load detail menus.

For example:

```
~|% b - Print Order Entry Defaults    %|~
~+-------------------------------------+~
~|% c - Update Order Definitions   >> %|~
~|% d - Print Order Definitions    >> %|~
~+-------------------------------------+~
```

In this example, both option Update Order Definitions and option Print Order Definitions load detail menus.

Window Menus

Submenu is distinguished from Window menu.

Submenus display on the right side of the screen and are the basic menus that make up a product.

Window menus are small menus that list a few options when a menu option from a submenu is selected.

Submenus are loaded by a :submenu: command in the item instruction file that is executed when a menu option is selected. addmenu is used to display another submenu (sibling menu). Use the existing FourGen product as a guide for when to use :addmenu: and :submenu:

The heading of a window menu should be identically named and displayed in the same place as the menu option on the parent menu.

Window menus should extend one character further, on each side, than the calling submenu.

Window menu options should use the key characters a, b, c, etc.

Menu options should be begin in the same column of the screen as the options of the calling submenu.

Each submenu or window menu directory should contain hidden options (not displayed on the menu) that allow you to move directly to other submenus. Pay attention to the use of :pc:x: (this simply closes the window menu and displays the submenu).

Menu items that are bring up window menus should be listed with two greater than signs ">>". For example, Print Multilevel Tax Analysis provides a detail and summary report.

Window menus should have the ability to change to a submenu simply by pressing the number of that submenu (as you can from other submenus).

Some versions of the *Screen* code generation program automatically center Browse forms and other windows both vertically and horizontally on the screen.

This is the standard location, and should be changed only when it is necessary to avoid covering other information on the screen.

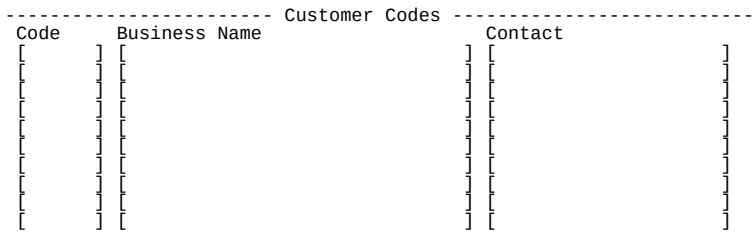
Windows should be as large as required by spacing lines with one space in between fields and one space at the beginning and end of each line, or larger if required for complete headings.

Do not use the first or last character of window lines, except for dashed lines.

Windows stacked on other windows should be centered unless it is necessary to see information from two windows at once.

Browse Windows

The first line of a browse form should contain a heading for each column (not embedded in a line) and the second line should be a line of dashes. The form should contain 10 rows of data. For example:



Detail Menus

Detail Menus are menus that are loaded by a :submenu: command in the menu script of a submenu option (the submenu is called from a control menu with an :addmenu: command).

Detail menus contain various methods of running a basic option, such as printing customer labels in different sort orders or on different size labels.

The heading of the detail menu is identical to the submenu item description that called the menu, including the key character used to execute the option.

Whenever possible, the window heading should appear in the exact position of the menu option used to load the window menu; the option description should not appear to change with the exception of the highlight.

Detail menus should extend one character further, on each side, than the calling sub-menu.

Detail menu options should use the key characters a, b, c, etc.

Detail menu options should begin in the same column of the screen as the options of the calling submenu.

Help/Error Messages

Run spell checker after completing.

Menu help should explain why you would chose the menu option.

Screen help should explain the specifics about using a screen.

Rows should be filled on the 1st page of menu help

This is important for the next screen of menu help to display properly.

If your help extends to the 2nd page, you should fill 18 rows (use blank rows at the end of your help to fill up the 18 rows if necessary).

Columns in width are available.

The title should extend to 56 columns as follows:

```
      12345678901234567890123456789012345678901234567890123456

               10        20        30        40        50

   --------------- Print Company Information -----------------
```

Help for print options should be tied to the related update option if there is one. For example:

```
This option prints the information entered through the
Update Company Information menu option.
```

User Prompts and Messages

All "commands" that appear in a command prompt should be shown with the first letter in uppercase. References to command names in help files, documentation files, and so on should also print the command name with the first character in uppercase and should not use quotation marks.

For example: Use the Find command to select all current documents.

Any prompt requiring the user to wait should be followed by ellipses.

For example: Loading Program to Update Journal Documents...

When referencing a column, field, menu option, record, document, row, command, mode, or similar objects (other than screens, forms or menus), the name of the field is shown exactly as it is shown elsewhere on the screen. If it is in all caps on the screen, show it in all caps in the description.

Do not capitalize words such as "column," "field," "option," or "mode" that follow the name.

Examples:

Description field
Browse mode

Print command
the Update Ledger Accounts option
the CASH IN BANK document

When referencing a column, field, menu option, record, document, row, command, mode, or similar words, if the name describing it does not begin with a capital letter, then use quotation marks around it.

For example: the Browse mode, the Update Ledger Accounts option, the Description field, the "account name" column, the CASH IN BANK document, the ACCOUNTS PAYABLE record.

If a single-character code field is used to identify types, use mnemonic characters (usually the first character of each option), rather than sequential characters.

That is, use something like:

```
[W] Weekly [B] Bi-weekly [S] Semi-monthly [M] Monthly
```

Rather than:

```
[A] Weekly [B] Bi-weekly [C] Semi-monthly [D] Monthly
```

If you do not have italic characters, use uppercase to stress very important text; For example:

```
You MUST post BEFORE you go home.
```

Automatic Data Entry Help from Form Comments

Messages printed on an entry form (such as the comment/help lines that appear on the bottom of Informix forms) should be complete sentences, should end with a period, and should be in lowercase with the exception of the first letter.

Messages should begin with an action (i.e. a verb) and be as explicit as possible in the given space. For example, write:

```
Change to screen for entering warehouse item detail.
```

Rather than:

```
Warehouse Information.
```

For fields that accept a defined set of valid entries, unless the screen lists all possible entries, the comment line must include them.

Example:

```
Apply rate to:[G]Gross [T]Taxable [H]Hours [N]None (flatrate).
```

Error Messages

The message should be short and to the point. In addition, the message should not suggest a failure on the part of the operator (as a phrase like Invalid Entry does), but should suggest a mismatch between the expected entry and the given one (e.g. Reference Information Not Found.)

Capitalize the beginning of each word in the error message with the exception of conjunctions, versions of the verb "to-be", and articles.

For example:

```
Code Number Not Found.
```

Examples of incorrect form:

```
Code number not found.
CODE NUMBER NOT FOUND.
```

Error messages should end with a period.

References to screen names, field names, and menu items must be totally consistent with the use of those names in the application itself.

Whenever possible, limit messages to lines that appear on the screen in the Happens section and Correction section.

Whenever possible, each sentence of the error message should begin on its own line, especially when each sentence is a separate problem or solution.

In the text section, if a sentence wraps around to a new line, indent one character to show continuation from previous line.

For error text, the Happens when section should begin with, "You have...", explaining what the user has done if that has caused the error, or "This document has..." if the problem is with the document.

For error text in the Correction section, begin each sentence with a verb and with the term "you" understood.

Example:

```
Select valid number with [CTRL]-[z].
```

Standard Error Messages

Not Found: Whenever an invalid reference code is entered by the user, the message is: Field Name Not Found.

Example:

```
Customer Code Not Found.
```

Duplicate: When entering a code that should be unique and the system finds a duplicate, the message is: Duplicate Field_Name.

Example:

```
Duplicate Customer Code.
```

Too Many: When there are too many rows to read into a form, the message is: Too Many Line Name in Record Name.

Examples:

```
Too Many Accounts in Account Group.
```

```
Too Many History Lines for Account.
```

There are many error conditions that are identical in different programs. They should have identical error messages.

Examples:

```
Account Number Not Found.

Department Code Not Found.
```

General Conventions-Programming

Programming Style

Our code generators already enforce several standards in terms of code style. We do not consider modularity, a wealth of comments, and similar necessities of code design to be issues of style.

Must-Fill Fields: Any field that must be filled and accepts only certain codes should always be checked for a valid entry. Validation checks are not necessary for non-mustfill fields.

Temporary Files: All programs should allow the user or developer to easily control the storage location used for temporary files. The standard method is to check the `$tmpdir` variable to see if it has been set to a directory that currently exists on the system. If there is a variable for temporary file locations (such as Informix's `$DBTMP`), this should be checked also, but `$tmpdir` should take priority. If no temporary location variable is set to an existing directory, the program(s) should use the `/tmp` directory.

Backslashed Quotes: The use of backslashed quotes is obsolete. For instance:

this use of backslashed quotes is obsolete:

```
when tbl_name = "stxerorh"

  let where_stmt = " where err_module in ",

   "(\"all\",\"lib_all\",\"lib_scr\",\"security\",\"r_menu\") ",

   "and userdef is null and language = \"ENG\""

  let insert_stmt = " insert into stxerorh"
```

please use single quotes:

```
when tbl_name = "stxerorh"
```

```
        let where_stmt = " where err_module in ",

         "('all','lib_all','lib_scr','security','r_menu') ",

         "and userdef is null and language = 'ENG'"

        let insert_stmt = " insert into stxerorh"
```

Code Indentation: Coding style issues are simply arbitrary choices.

The idea is to make the code look a certain way so that people reading it have a clearer idea what is happening. These conventions are simply a method of indenting the code properly.

Lines of single structures, such as "menu" and "input" structures, should be indented, with the exception of the beginning and concluding lines of the structure. The indent should be 4 characters (blanks), with the exception of certain key words that are part of the structure (e.g. "when" and "otherwise" lines in a "case" structure). In the case of keywords, the keyword line should be indented 2 characters, and the subsequent lines contained in the keyword section should be indented 2 more characters.

NOTE: See exceptions in examples (“if then” statements).

"Action" lines of conditional structures (case, if-then) should always begin on a new (indented) line, even if the code allows the first action to be on the condition line, unless 1) every condition has just one action, and 2) every action can fit on the condition line within 80 characters.

Additional exception for SQL lines: Put the "from" clause on one line if it's just one table, even if other clauses are on multiple lines.

Conventions on indentation are shown with the following examples:

NOTE: "no-wrap actions": when the "action" part of the statement does not wrap to the next line.

```
case: (single, no-wrap actions)
    case
      when .... action
      when .... action
      when .... action
      otherwise .... action
    end case
```

```
case: (multiple or long actions on one or more "whens")
case
  when ....
    action
    action
  when ....
    action
    action
  otherwise
    action
end case

Menu:
menu "...."
  command "...." "....."
    action
    action
  command "...." "....."
    action
  command key(w,x,y,z)
    "...." "................................................."
    action
end menu

Input:
input .... from ....
  on key
    action
  before row
    action
  after field
    action
end input

If: (single, no-wrap actions)
if ....
  then action
  else action
end if

If: (multiple or long actions on either the "then" or the "else")
if ....
then
    action
    action
    action
else
    action
end if

If: (very short actions, no wrap allowed)
if .... then action else action end if
if .... then action end if
```

In the case where the following is not feasible:

```
if .... then action action end if
```

the following is acceptable:

```
if condition
  then action end if
```

SQL Indentation:

SQL columns: in general, keep each field/column on a separate line.

For example:

```
select
    tab1.field_1,
    tab1.field_2,
    tab2.field_3,
    tab2.field_4
  into
    rec.field_1,
    rec.field_2,
    rec.field_3,
    rec.field_4
  from
    tab1,
    tab2
  where
    condition and
    (condition or
    condition)
```

Same exception as before: when everything fits on one line, that is acceptable. Additional exception: put the "from" clause on one lineif it is just one table, even if other clauses are on multiple lines.

Unrelated standard: always put the table name as the column prefix in the `select` clause, even if there is no chance of confusion.

The other SQL statements, such as `insert` and `update` need similar attention.

For very simple `selects`, the following is acceptable:

```
select colA, colB
  into varA, varB
  from table
```

General Rules

Program and library file names must be 8 characters.

Program module names start with `"i_"(input), "o_"(report)` or `"p_"(posting)`

Library functions names are descriptive of the function example: `itemz.4gl` (Zoom logic), `sel_cust.4gl` (selection criteria logic)

Functions Referenced in Input Programs

If used by more than one program in the module, the function must reside in the module library.

If used by more than one program in any module, the function must reside in the `all` module library.

If called from the "triggers" (.trg) file, the function should reside in `fg_funcs.4gl`

If called from the Entry-By-Exception screen, the function should reside in `fg_screen name.`

Enclose the function statement in a "flowerbox" (that is, lines of #'s) and specify what if anything the function returns to the calling program.

Design functions so they do not return strings. Numbers are okay, and string info can be passed back via scratch or vararg functions. Passing strings back to calling programs risks the problem of running out of string space.

Other Considerations

Avoid changing anything in upper level libraries; if you must, then create your own upper level lib (example: `$fg/lib/custom.4gs`) and add it to the `Makefile` (this can be done in the `.trg` file).

Comment your code to the extent that a person reviewing the code will not have to contact you for an explanation of why the coding was done in a particular way. Attempt to explain your coding decisions with comments.

Unless there is a good reason to do otherwise, define your decimal columns as decimal(12).

Case statements work correctly if written in the following manner:

Good Form:

```
case
   when boolean1 call do_stuff1()
   when boolean2 call do_stuff2()
   otherwise call do_stuff3()
end case
```

Example:

```
case
    when p_oordre.trd_ds_type = "A" call do_type_a()
    when p_oordre.trd_ds_type = "B" call do_type_b()
end case
```

Bad Form:

```
case expression
    when expression1 call do_stuff1()
    when expression2 call do_stuff2()
    otherwise call do_stuff3()
end case
```

Example:

```
case p_oordre.trd_ds_type
    when "A" call do_type_a()
    when "B" call do_type_b()
end case
```

NEVER UNDER ANY CIRCUMSTANCES USE THE BAD FORM.

Not only can it lead to stuffed up code, it can also lead to at least three other bugs that are extremely difficult to track down, (two in `rds` and one in `4gl`).

Select Statement, and Rowid Handling

Do not use rowid = 1 in where clauses any more. To get info from control tables, use `call ?control() returning m_??????.*` if possible.

For example: `call pcontrol() returning m_pcntrc.*` would be used to get the ap control table.

Do not put control tables in the cursor definition in report/posting programs (unless the program is reporting explicitly on the control table, of course).

String handling should be written in the following manner:

Putting a function call into a string concatenation causes the informix stack to stop up. (Remember, the comma is the concatenation operator.)

Here are some examples of the good and bad form:

```
good: let phil = david, "a real nice string", chauncey
good: let mytemp = round("a", chauncey)
good: let phil = david, mytemp
good: let phil = chauncey using "<<<<<&.&&"
good: let mytemp = chauncey using "<<<<<&.&&"
good: let phil = david, mytemp
bad:  let phil = david, round("a", chauncey)
bad:  let phil = david, chauncey using "<<<<<&.&&"
```

A note: the "using" clause may not look like a function call, but in fact, it is a function that returns a string. We very strongly suspect that the last example labeled "bad" prevents garbage collection from taking place. We suspect that this is the most frequent cause of our running out of string space, because this rule was never explicitly stated, and it is violated in many places in our code.

Zoom Pickers

- To enable logic to create a "picker of Zooms" you need to add the appropriate Zooms to the "switchbox" trigger instead of calling the Zooms from the `.per` FOURGEN section.
- Add appropriate call to execute the "picker" function:
- Add a "picker" function to `fg_funcs.4gl` using the function `oe_Zoom` found in `oe.4gm/i_order.4gs/fg_funcs.4gl` as an example.

Posting Routines

See example of locking related functions in `oe.4gm/p_pstord.4gs/midlevel.4gl` for details.

Input Programs Testing Checklist

LEVEL 1

_____Does the screen have all the data (fields/columns) it should have?
_____Enter incorrect data into every field. Is it handled gracefully?
_____Enter letters into numeric fields.
_____Enter largest value possible.
_____Enter reference codes that don't exist.
_____Can you get the program to "bomb"?
_____Are error messages helpful and appropriate?
_____Enter acceptable data into every field.
_____Test the zoom on every field that has one.
_____Does the zoom work?
_____Do any fields that should have a zoom not have one?
_____Try autozoom in all zoom fields - program shouldn't bomb.
_____Are the comments at the bottom of the screen helpful and appropriate for each field?
_____Check the Help information for the screen.
_____Does it exist?
_____Check it in each header portion and each detail portion.
_____Check for misspellings, grammar, clarity.

LEVEL 2
_____Does the screen follow the style guide?
_____Check for misspellings.
_____Verify that abbreviations are used only when necessary and always concluded with a period.
_____Are columns aligned properly?

LEVEL 3 (full-fledged break-testing)

Add/Update Testing
Max fill character fields:
_____Fill every field on the screen (all character fields should end with "to max" with the "x" in the last character of field: verify all data was stored and

retrieved.)
_____Fill all the detail lines that appear on the screen, and at least 1 more.
_____You must exit the program and refind the data to determine if it is OK.
_____Using ISQL check the record(s) and verify all fields are being stored.
_____Using ISQL make a list of any fields that are null.

Dates:

_____Dates on all screens are displaying properly: check the .pers (each date field should be formatted to "mm/dd/yy").

Max Fill Numeric Fields:

_____For each numeric field, fill the field to the max with 9's (if decimal type fields have not been programmed properly, this will cause a program to bomb).

Balance Testing:

_____If an interactive program updates balances, these balances need to be checked carefully - correct balances need to be verified in addition to all fields being stored properly (max fill testing):
_____Verify that balances are what you expect them to be after an update, add, or delete.
_____You must exit the program and refind the data to determine if it is OK.
_____Detail section: delete a row & make sure the balance is recalculated.

Program Handles Negative Amounts Properly:

_____Enter at least 1 document for each type of transaction with all negative values in each numeric field and make sure the system calculates everything accurately.

Adding or Modifying Tables Checklist

_____Modify `sample` database as well as `standard` database and do not forget to add any new tables to the `pcdtablr` table.

_____Check/modify all records (globals, header, midlevel) that refer to the column that was added/modified.

_____Recompile/relink all functions/programs that reference the table if a function (non-local) is called that loads a record based on this table; even if you don't have to modify that function, you do need to recompile it (i.e., if the function says "like table"). After modifying (touching) the file, make it in the library and then run make back in the program directory.

_____Be sure to Grep in all function libraries for any other functions that access the table.

_____Check all program directories that may reference the table or should now reference the table.

_____Be alert for the creation of a temp table in midlevel (look for a create temp table line) where the temp table parallels the table you have modified - you need to make sure this temp table is in sync (i.e., TEMP TABLES ARE LISTED OUT).

_____Update column level help.

_____Update the unload tables in the appropriate `$fg/data`/(module) directory.

_____Recreate the `dbmerge` executable, using the `dbmerge` generator (see following section).

_____Update the schema definition for the entire module.

Tabs

Tabbing to Different Sections of the Form:

What's supposed to happen: if you modify a field and press [TAB] prior to pressing [ENTER] the new value is not saved, the old value should be redisplayed.

_____Tab from every field in a detail row after modifying the value in a field. Make sure that the original value in the field is immediately restored/displayed.

_____Do the same thing from every section of the form: in the header, try this from one or two fields.

_____Modify a single field in an existing record, save, and refind Note: We found a case where modifying a single record after first running, the program did not save the modified field.

_____For any programs that open additional windows, or use [DEL] to exit from one section of the screen, or to exit from an opened window: test [DEL] in combination with [TAB] carefully.

Required Fields:

_____Save the record without filling any fields: make a list of the required fields. List those fields and pass this list back to the programmer for verification.

Duplicate Testing:

_____In Add mode try to enter a duplicate document (duplicate key field(s)).

_____In Update mode try to modify the key fields to make this a duplicate document.

_____Some documents allow you to modify key fields, some do not. Some are modifiable when setup is not complete only: note the particulars for this document.

_____Any time there is duplicate checking in Update mode, add a record, save it, and then immediately update it. Make sure that duplicates are properly checked for in this unique case.

Collapsing Array Logic:

_____Collapsing array logic works properly (after row __, after input__)

_____Delete both header and detail records are deleted for header/detail type screens. No orphans are left (use ISQL).

[DEL] When Pressed in Add or Update Mode:

_____If header/detail, no orphans left when press [DEL] from header (ISQL).

_____If header/detail, no orphans left when press [DEL] from detail (ISQL).

_____Try to remove all detail lines (using F2) and then post. This is a source of innumerable problems.

_____ [DEL] restores document properly

[F2] Row Delete:

_____For every "update-type" menu option, include the following actions in at least one document. Add a document with some detail lines. Using the [F2] key delete a few detail lines. Then restore the document by pressing the [DEL] key.

Find:

_____Do repeated finds

_____Reorder works. Try reordering on several fields.

Browse:

_____All commands work as expected.

_____Full length of field is displayed in window (see max testing)..if not make a note of this.

Nxt / Prv:

_____Wrap around start & tail end of found selected documents.

Options:

_____Test out any options provided on the Options menu.

L**ocking Logic:**

_____Find out from the programmer if there is any locking logic that is implemented (this information "should" be in the Info file).

_____Test with 2 users trying to access

Lookups:

_____Test for proper loading of the "into" field (if appropriate).

_____Test data-validation of value entered in field

_____Test clearing of "into" field if bad data entered

Comments:

_____Each field has a comment. Test on the Find screen (construct).

Data Validation:

_____Identify those fields where the value entered should be verified by the program to be a valid value.

_____See if the program is doing the type of validation you would expect.

_____Zooms work properly.

_____Lookups work properly (after pressing [ENTER] in a field, the program fills one or more other fields).

Recalculations Occur Properly:

If there is a field on the screen that can be set (to Y or N for example) and the setting of this field results in calculations being done or not being done (Taxable? is an example of such a field).

_____Set the field to one value (Y for example) and verify calculations are being

done properly.
_____Then, with the same document, reset this field and verify that the calculated field is recalculated as you would expect.
_____Now reverse this procedure: start with a fresh document, start off by setting the field to the opposite value (N for example).
_____Then, with the same document, reset this field and verify that the calculated field is recalculated as you would expect.

Spelling:

Run spell check on
_____ each .per.
_____ each unload file for help.
_____ each unload file for error text.

Report Program Testing Checklist

LEVEL 1

_____Verify that report runs correctly with a large amount of data
_____Does the report have all the data (fields/columns) it should have?
_____Does the report have all the totals it should have?
_____If data may extend over one page, are the appropriate headers and subtopics printed on each page?

LEVEL 2

_____Does the report have headers and footers in the standard format (provide examples)?
_____Check for misspellings.
_____Verify that abbreviations are used only when necessary and always conclude with a period.
_____If implemented, verify that scheduled reporting works correctly.

Posting Programs Testing Checklist

_____Enter a header-detail-type document and try to remove all detail lines, and then post. This is the source of innumerable problems.
_____Determine which posting routines are to be tested.
_____Determine how posting affects which tables are updated and which reports you can run to see the results of testing.
_____Enter some documents to test their impact on these tables. Verify that balances, etc., are updated as expected.

A

Problems with [CTRL] - [Y] and [CTRL] - [Z] Keys

This section describes some potential problems with [CTRL] - [Y] and [CTRL] - [Z] keys that are part of the lowlevel terminal control logic (ioctl).

When you execute the command "`stty -a`" it returns a list of the `ioctl` parameters. Many of these are control key mappings. The ones that can cause problems are "susp" mapped to [CTRL] - [Z] and "`dsusp`" mapped to [CTRL] - [Y]. We use [CTRL] - [Z] and [CTRL] - [Y] for our own functions, but when they are part of the terminal control set they are interpreted directly by the terminal device driver and never seen by the program. We have logic in the start up scripts to remap these keys but this does not seem to work on all systems.

For interactive programs the use of "`susp`" and "`dsusp`" doesn't work. "`susp`" and "`dsusp`" are used for job control, to suspend and activate processes. For example, say you start a program, you can press [CTRL] - [Z] to suspend the program temporarily in mid execution in order to do some other task or start some other program in the background before you proceed. This works just fine as long as the program doesn't have a complex user interface (Informix programs and most highly interactive (data input) programs have a complex interface). The problem is that

when you unsuspend a process it doesn't restore the terminal display or lowlevel input processing to the state it was in when the process was suspended. Hence, the program usually will fail to operate correctly. Because of this it is required that the "`susp`" and "`dsusp`" keys be disabled while the interactive program is running as well as other keys that may be mapped to low level (device driver interpreted) terminal control functions.

B

Troubleshooting Products

Required Information

Hardware:

Hardware Platform:_____

OS:_____ Version:_____
Has the kernel been tuned?_____(Y/N)
Amount of memory installed on the system:_____
Amount of available disk space in blocks:_____

INFORMIX:

Informix Version:_____ RDS:_____(Y/N) 4GL:_____(Y/N)
New or Existing Informix install:_____(New/Existing)

Do you have ISQL:_____(Y/N)
Do you have the debugger:_____(Y/N)
Do you know how to use the debugger:_____(Y/N)

BUSINESS:

Business Application Product:_____ Version:_____
Business Application Product:_____ Version:_____

MISCELLANEOUS:

Do you know the "vi" editor?_____(Y/N)

REQUIRED:

Have you followed the steps in the "Troubleshooting Checklist"?
_____(Y/N)
All of the above information must be gathered prior to calling support.

SUGGESTED REFERENCE MATERIALS:

Operating Systems: System Performanced Tuning
By: Mike Loukides
Published By: O'Reilly & Associates, Inc.
632 Petaluma Avenue
Sebastopol, CA 95472
Phone Number: 800-338-6887

VI Editor: The Ultimate Guide to the VI and EX Text Editors
By: Hewlett Packard
Published By: The Benjamin/Cumming Publishing Inc.
390 Bridge Parkway
Redwood City, CA 94065

Debugger: Informix-4GL Interactive Debugger

4gl Code: Informix-4GL Reference Manual Volume-2

Menus: Menus User Reference.

Troubleshooting Checklist

Program Bombs

(displays error: Program Has Stopped Unexpectedly)

1. Use the [CTRL]-[z] command to view the error when the program bombs.
2. Check the "`errlog`" file in the program directory. The errlog displays messages for the SQL error number, ISAM error number, program module, and source code line number. If `errlog` does not show the error, see the instructions on running `errlog` testing described as item #1 in the *Troubleshooting Guide*.
3. Consult Informix manuals concerning the error.
4. Are you using the correct database?
5. Recompile and run the program.
6. If the above does not help, check the environment variables from both your current environment and the menu's environment.
7. Run the debugger using break points to define the exact location of the problem. Write the variable values to a file just prior to the program bombing. If RDS/debugger is not available, use display statements and recompile the program (see item #2 in the Troubleshooting Guide). When correctly placed, display statements will help isolate a problem.

Unexpected/Inaccurate Program Results

1. Check to see whether Menus is calling the program from the expected place--`4gc` or `4gs`, `4ge` or `4gi`. Menus will use a `.4gc` directory before a `.4gs`; it will execute a `.4ge` program before a `.4ge`.
2. Recompile the program.
3. Follow the recommended debugging procedures outlined in the *Troubleshooting Guide*.

4. Collect the versions of the module. This information is located in the `$fg/accounting/install/{module}/def` file.
5. Has the program been modified?
6. Is this a new install or an existing system?
7. Is it a posting program? Has the kernel been tuned?
8. Has `dbmerge` been run successfully on the database? If not, run `fg_load` on the specific modules (`fg_load` is found in `$fg/ubin`).
9. Can you duplicate the problem?
10. Can you duplicate the problem on a different database?
11. Can you duplicate the problem on a different computer?
12. After collecting all of the required information as shown on the first page and in the above questions: Call your assigned Technical Support person.

Troubleshooting Guide

The following is a list of commands or steps to assist in troubleshooting a program.

1. Using the errlog as a way to find the problem and/or isolate the area where the problem is occurring.

   ```
   call startlog("errlog")
   ```

   This will start the error log "errlog" and the current program directory. This should be put in `main.4gl`.

   ```
   call errorlog(variablename)
   ```

   This will write to the error log the value of the variable along with the current date and time stamp.

   ```
   call errorlog("some string")
   ```

   This will write to the error log the value of the string along with the current date and time stamp.

   ```
   let tmpstring = "some string of your choice", variablename
   call errorlog(tmpstring)
   ```

   This will write to the error log the value of the string along with the current date and time stamp. This is a way to write both a variable and string to the error log.

2. Using display statements in the program to see how values are changing while the program is running. You can display a string, a variable or both as shown below.

   ```
   display "Entering lld_read function"
   sleep 2

   display sel_filter
   sleep 2

   display "The value of sel_filter is: ", sel_filter
   sleep 2

   display "The value of sel_filter is: ", sel_filter at 1,1
   sleep 2
   ```

Use the sleep statements to make the display statements a little more readable. As shown in the last example the "`at 1,1`" can be used to position the message on the screen.

3. The following point is useful when your program works from the command line but not from the menu. Using the `-v` and `-p` flags with the menu options. These options are used prior to entering a program from the menu. These options are explained below. These flags are toggles which means the first time you type the flag it turns it on and the next time it turns the function off.

   The "`-v`"(verbose mode) and "`-p`"(pause mode) options will display every command used to execute the program and will wait for a carriage return. Or you want to know what the command line looks like for the program. Running these flags from the main menu may require as many as 20 carriage returns to execute the program.

4. Use `!cat` from the menu. This option is used prior to entering a program from the menu. This option is explained below.

   The `!cat` option is used to find the name of a program, the module for the program and any variables passed to the command line. The syntax is "**`!cat menuoption`**" as shown below:

   "`!cat a`" from the GL menu #1 displays the following:

```
########################################################################
# Copyright (C) 1991-1195
#
# Use, modification, duplication, and/or distribution of this
# software is limited to the terms of the software agreement.
# Sccsid:  @(#)  .../menu/glmenu/journal/a  1.2  Delta: 1/4/91
########################################################################
# Loading Program to Update General Journal
:ifxscreen:gl:i_genjrn::::
```

   As shown above the menu option is a screen(`ifxscreen`), the module is `gl` and the program name is `i_genjrn`. So the program is located in `$fg/accounting/gl.4gm` and the name of the program is `i_genjrn.4gs` or `i_genjrn.4gc`. The menu program will look for a `4gc` program before a `4gs`. And the menu will execute a `4ge` program before a `4gi`.

5. Using `-xv` in shell scripts. This will allow troubleshooting of scripts as they are being executed. The "`set -xv`" is put at the beginning of the script. This will allow you to follow the flow for the script.

```
set -xv
```

   The "`x`" flag is used for "`xtrace`" and traces your way through the script. The "`v`" flag is used for "`verbose`" and displays input on standard error as it is read. Unfortunately, this does not work on scripts that go down to additional shells.

6. If you have RDS, use the Informix Debugger. It is required that all VARS have and understand the Informix debugger. Please read the Informix manual for the use of this tool. There are many flags, break points and variables to use when debugging a program.

   Below you will find some handy commands for using the debugger:

   `$break` ***`programname.linenumber`*** breaks when the program hits the specified function.

   `$break` ***`functionname`*** the debugger will stop when it gets to the specified function.

   `$break if tablename.columnname = value` breaks when the value is equal to the value(>,<,!=).

   `$con(tinue)` continue running the program after a step.

   `$CTRL T` toggles between the screen and the program running in the debugger.

   `$!err errornumber` this will show the error message.

   `$functions` displays functions in the programming window.

   `$help` brings up the help screen w/options.

   `$list` lists the previous commands.

   `$!l filename` lists the file listed or the directory.

   `$nobreak functionname` unsets a break on a function.

$p fieldname	prints the value of the field or variable, also sqlca error handler variable.
$print variable or tablename >> filename	prints the value/s to the named file.
$run (-c database)	run the program.
$s(tep)	step through the program.
$trace variablename	show variable as it is changed.
$view functionname	will view the name of the function that is typed.
$where	shows where you have been in the functions.
$write	writes the value of the screen to the file with the name "programname.4db". to view the file type "!cat programname.4db."

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

Z