THE ANALYSIS AND DESIGN
of a
FOURTH GENERATION LANGUAGE

A Thesis

Presented to the

Division of Mathematics and Physical Sciences

EMPORIA STATE UNIVERSITY

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Karen I. Craft

May 1988

AN ABSTRACT OF THE THESIS OF

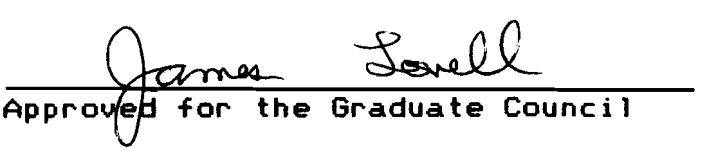
Karen I. Craft for the Master of Science Degree

in Mathematics presented on March 1988

The Analysis and Design of a Fourth Generation Language

The enclosed thesis contains a study of the first three phases of the software engineering process as applied to the project of creating a fourth generation language (4GL). Initially, the 4GL is defined. Three levels of users are also defined - the software engineer who creates the 4GL, the application developer who uses the language to develop a specific application program, and the final end-user who operates the application program created with the 4GL. Included are schematic diagrams to show the logic flow of the language. Sample screens are also included to show the results of using the language for an application. The fourth generation language automates the code writing process with modules to handle menu systems, data entry, screen handling, keyboard handling, validity testing, error handling, data access and searches, report generation, printer control, file and index management, and internal data management of buffers, pointers, and system functions.

Approved for the Major Division



Approved for the Graduate Council

---=== CONTENTS ===---

---=== TABLE of FIGURES ===---

----==== I - INTRODUCTION ====----

Overview --- This document contains a study of the first three critical phases of the software engineering process as it is applied to the project of creating a fourth generation language. Initially, it is necessary to provide some background information which defines the terms *software engineering*, *CASE* (Computer Aided Software Engineering), and *4GL* (Fourth Generation Language). Understanding these terms will aid in better understanding the goals of the project.

Background Information --- Within the past few years, the ever-changing micro-computer software market has presented many new problems to solve. In the world of value-added computer retailers (VARS), specific "target markets" have become the emphasis, such as medical, dental, retailer inventory, accounts receivable, or general ledger applications, to name a few. Off-the-shelf software is too general for these specialized fields, therefore custom software becomes a necessity. With the recent flood of these market needs, the software engineer is forced to develop more efficient methods to produce the volume of

applications needed. Tom Adams, team leader at ASCII (Automated Software Concepts International, Inc.) and the designer of GhostWriter, describes the changes required in the programming approach at ASCII. "We had to find a competitive - and affordable - way to do custom software. We decided to develop a package that automated the code writing process."[1] Custom programmers find it necessary to increase productivity and coding efficiency, to standardize their applications for maintenance efficiency and still retain versatility. Continuity from one application to another keeps the maintenance and training effort to a minimum.

Software Engineering ---

> Software engineering is not just a matter of writing code! ... The software engineering *life cycle* involves a more or less standard sequence of phases. At the outset of a software project, a set of *requirements* are gathered, describing the objectives that the software must satisfy. Next is an *analysis* of the requirements, exposing important patterns and structures. These are called the *design* specifications.
>
> At this point, software design begins. Usually a schematic design phase precedes detailed design. Upon completion of design, code is written --- this is the *implementation* phase --- followed by *testing* and *maintenance*.[1]

Software engineering therefore involves the following phases in the life cycle ---

1 - Requirements - gathering the objectives
2 - Analysis - exposing patterns and structures
3 - Design - schematic and then detailed design
4 - Implementation- actual coding
5 - Testing - verifying correctness
6 - Maintenance - correcting, updating, revising

Advances in technology and techniques to improve the process have not kept up with the demand for new software; or perhaps programmers have been too busy to learn them. Whatever the cause, many programmers have apparently decided to skip designing software in favor of just getting the job done. "It can always be fixed later" seems to be the attitude of the day.

Programming is costly enough when it is done correctly. this haphazard rush to completion only adds to the costs, both in maintaining the software and in producing truly useful applications.[2]

In a software engineering project, the greatest effort is expended in the later phases of the life cycle. Coding, testing, and maintenance take far more time than analysis and schematic design.

In contrast, decisions made early in the life cycle have the greatest impact on the quality and maintainability of the resulting software. Studies have shown, for example, that errors detected during [the] requirements [phase] are corrected in far less time than errors detected during implementation or maintenance.

In other words, the least effort is invested in the most important phases of the life cycle![1]

CASE --- A CASE product is any computer tool that assists any phase of the software engineering process. The definition is quite liberal due to the fact that software engineering itself is a broad activity. "Any computer tool that assists in the process can legitimately claim the CASE label."[2] "CASE tools perform analysis and design, code generation, testing and maintenance. Few, if any, do all these things, however."[3]

CASE can partially automate the coding and testing phases. "This is the goal of application builders and code generation products. CASE will also promote standardization and support better and more accessible documentation, resulting in lower maintenance costs. In this way CASE will, in principle, redirect resources to the critical phases of requirements, analysis, and schematic design."[2]

> Through extensive user surveys (published in *ComputerAided Software Engineering: CASE*), BTR [Business Technology Research of Wellesley Hills, MA] determined that most users have been employing CASE tools for only the past 18 to 24 months. [as of March 1988]
>
> CASE tools break down into two segments: design automation and programming. Programming tools are primarily available on IBM mainframes at prices in excess of $100,000, according to Bayer [David Bayer, an industry analyst with Montgomery Securities, San Francisco, CA]. The

> trend, however, is to less expensive products that can run on micro and minicomputer workstation platforms. He expects more sophisticated programming tools to appear this year [1988].[2]

The project included herein is following the trend toward the micro market.

The CASE definition includes two distinct technologies. *Front-end or upper-case tools* include the analysis and design aids. *Back-end or lower case tools* include application generators. This project centers around the back-end or lower case tools which are also labeled 4GLs or Fourth Generation Languages.

4GLs --- "These application generators assist the later phases of the [software engineering] life cycle, from detailed design through coding, testing, and maintenance. They focus on process, format, and documentation disciplines, not information or project management."[2]

A 4GL is actually a type of CASE, but is more easily thought of as a subset of the CASE technology, and also as the parent of the CASE. In its broadest sense, it is the the back-end or lower CASE tool mentioned above.

As a response to many new needs, fourth generation languages (4GLs) are becoming more prevalent and are quickly replacing third generation languages (3GLs) such as

BASIC, PASCAL, COBOL, FORTRAN, etc. A 3GL requires the application programmer to specifically use the language to tell the computer *how* to do every detail of each task. Using a 4GL, the programmer need only tell the 4GL program *what* to do. This is due to the fact that a 4GL is an organized collection of pre-written code which contains about 90% of all standard management code needed for all applications. Just as an operating system is a collection of standard low-level menial tasks that all users need, such as device management, character manipulation, screen management, etc., the 4GL contains a higher level collection of management "macros" that automate data, file, record, field, screen, and report management tasks. The application programmer no longer needs to "re-invent the wheel." All programs developed with the 4GL then become standardized in these tasks and all standard or common tasks are handled the same from one application to the next. A 4GL does the same thing for application programs that the operating system does for hardware, only at a higher level. The operating system standardizes the interface between the programmer, operator and hardware devices, and the 4GL standardizes the interface between the programmer, operator and the application. A truely

efficient 4GL will automate as many standard or normal tasks as possible. All that is left for the application programmer to do is to add the non-standard tasks specific to the application.

4GLs have also been described as "a continued evolution of languages"[4], "a specialized language that has been designed to do a specific function"[4], as being "able to do a task with roughly one-tenth of the code needed in a 3GL."[4]. "With 4GLs, the precise instructions have been automated. The language has been demystified. 4GLs employ a dialogue between user and computer, interacting to solve the user's problem. The focus is on the task, not on the computer."[5]

Many levels of 4GLs exist, from simpler 4GLs which allow an end-user to create a simple database to powerful system-house 4GLS used by software engineers. The difference between the two is the degree of flexibility and of the detail allowed in its use. The simple 4GLs allow for only very basic, standard functions, while the powerful 4GLs allow the versatility of adding the unusual, nonstandard functions when needed using the host language and the extended "macro" language of the 4GL itself. This study considers the later, the powerful system-house 4GL.

ASCII, the producers of GhostWriter, define their product as (1) an automated code writing process, (2) an application generator, (3) an application development system,[4] and (4) a CASE (Computer Aided Software Engineering) product for program development.[5] The engineering approach contained herein echoes these four definitions and considers the 4GL as an extension of the Pascal language with the creation of "macro" procedures and functions for system-house use to create relational database management applications.

Summary --- In summary, this document contains the study of the first three critical phases of the software engineering process (the requirements, analysis and design phases) for a fourth generation language. Briefly, this involves a subset of the CASE technology called the application generator technology (also called back-end or lower CASE). It can also be described as an automatic code generator with screen/report creation, dictionary definitions, data base management, procedural language, and functional integration as described in "The James Martin Productivity Series"[2]

The creation of a fourth generation language (4GL)

itself requires quite an involved programming project. The ASCII programming team spent three-and-a-half years work on their GhostWriter written in Turbo Pascal.[1] This document analyzes the type of product produced by this company.

II - ANALYSIS

A high-level analysis at this point involves several directions. The first step is to define the users, and their respective needs.

The User --- Defining the needs or requirements of a software project also initially requires defining the user. In order to avoid any confusion, three terms are used consistently within this document ---

(1) The software engineer who creates the 4GL.
(2) The application developer who uses the 4GL.
(3) The client/end-user or operator who uses the application created with the 4GL.

In this situation, the application developer (also known as a system developer) is the user of the 4GL, however, while using the 4GL, this developer produces applications for a variety of clients or end-users. This requires that the 4GL be written in such a manner as to allow for many different variations in database applications. Therefore analyzing the needs of the application developer, also involves analyzing the needs of as many client applications as possible. The application developer becomes the "middle-man" for the client, the final application end-user. Thus, two levels of analysis are considered, the needs of the

application developer and the needs of the developer's clients.

James Hughes explains a standard approach to the analysis process --- "At project initiation, a project team - consisting of systems analysts and users assigned full time to the team - must define the preliminary requirements of the system."

"Traditionally, at this point analysts would interview dozens of users to determine requirements. This often produces a long list of wants and needs that are difficult to analyze and use for system development."

"A better approach is to involve a few experienced users in the definition of preliminary requirements and major system externals - such as menus, data-entry screens, on-line query displays and reports. These system externals should then be incorporated into a horizontal prototype."

"Users can review and rank the functions in the horizontal prototype to determine which functions should be automated."[8]

Software Engineer and Application Developer Needs --- The software engineer (the 4GL developer) and application developer (user of the 4GL) have much in common. They are

both developers for a client. The software engineer creates general applications for the application developer, and the application developer creates specific applications for the client/end-user. As mentioned in the introduction, several quality requirements include increased productivity, code efficiency, maintenance efficiency, standardized applications, and versatility. Further clarification is now needed.

Increased productivity involves the production of an application in less time than with previous methods. This is easily measureable by keeping time logs on all projects, however, one must remember that not all projects are created equal.

Maintenance includes updating the system for changing needs, correcting errors, and adding new capabilities. In most organizations, it is estimated that 70% of the programming is dedicated to maintaining existing systems. With 4GLs 70% of the time is spent on original coding and 30% on maintenance.[4] A ratio of 80% - 20% is the minimum acceptable goal of this project with the ultimate goal of a 95% - 5% ratio. The application developer is required to provide parameters to the 4GL to define the data file

structure, entry screens, and report formats. In other words, the majority of the time for using the 4GL is spent in designing the application database, as is necessary in any application, however, little additional time is needed beyond that stage.

Efficiency involves three major areas of concern - speed, memory, and maintenance efficient code. *Speed efficiency* can be measured with timing tests. The 4GL developer must consider disk access time when using overlays and managing data files. Both the software engineer and the application developer have access to the use of inline and external code to speed up processing. *Memory efficiency* can also be implemented using inline and external code. Due to the fact that the constraints of the project include use of PC/XT equipment, it is necessary to keep the code as compact as possible. The 4GL developer can compare the memory required by one algorithm over another in order to determine the memory efficiency. *Maintenance efficiency* is included here as the top priority requirement. It is not a truely measurable feature. However, all design for the software engineer and application developer must have an underlying purpose of

being easily updateable as new needs surface. This becomes a "gray area". An elaborate algorithm may be quite efficient time- and memory-wise, but too difficult to maintain because of a lack of readability. Readability is based upon the opinion of the programmers and dependant upon their expertise. Readability and maintenance have top priority over speed and memory.

Standardizing applications is important in two major ways. From the programming and maintenance point of view, standardized algorithms prevent reinventing the wheel and save much development time. From the application developer's point of view, this is the major purpose for using the 4GL. The application developer no longer needs to be concerned with how the application handles menus; input, keyboard handling, validity testing, and error handling; output and printer control; screen functions, color coding, and windowing; search routines; buffers; pointers; system functions; or even the processing of data, indices and file management. These are totally in the control of the 4GL. The application developer then is concerned with *what* the system must do for the special application and not *how* it does it.

Versatility is a quality that can really only be measured with time. Due to the vast range of specific application requirements of client/end-users, it is impossible to foresee and allow for all possible features that might be needed by the application developer. After the major design requirements are defined, a further requirement is to allow for direct application programmer code within the program to add the non-standard features. Standard code is code that is protected and not allowed to be changed by the application programmer. The requirement of versatility is provided for with code files accessible to the application developer that contain stubbed procedures in which any unforeseen specific application requirements are coded by the application programmer. Standard Pascal commands and an extended language set of 4GL macros (described in more detail later) are available to the application programmer as the specific code is added. The standard code files created by the 4GL developer are designed to handle 80 - 99% of all application needs. These include automatic handling of the functions listed in the "Standardizing Applications" paragraph above. Additional code added by the application developer is typically 1-5% of the total.

Client/End-User Needs --- Quite often the client/end-user is a novice with respect to computer operation. All design is directed to the needs of the new computer user. As mentioned earlier, maintenance is the top priorty of the entire project, but consistent user-interface is the second priority. Every move required from the operator is to be thoroughly prompted. In more specific terms, this first of all requires a *standard screen layout* in order to assure the operator that all prompts will be consistently seen in the same format throughout the entire program operation. *Color coding* aids in prompting the user also. *Help screens* are available at all times. *Data input* is prompted at the field level with data type, data limitations, and field descriptions. All data is checked for *validity* and any required or coded fields are tested before saving the record to the database. *Searches* of the database allow for multiple searches for any limitations on any or all fields available in the file. The first phase of development allows for searches of concurrent search criteria, however the second phase allows for OR, NOT, XOR, and wild card searches. *Report formats* allow for columnar formats, page numbering, column totals, and counters. Printer attributes, such as bold and underline, can be embedded in

report formats. User-entered subtitles are also allowed.

Summary --- In summary, the functions that will be automated include ---

1 - a menu system,
2 - data-entry/display screen functions
    (I/O screens),
    color coding
    windowing
3 - data entry,
    keyboard handling
    validity testing
    error handling
4 - data access (searches),
5 - report generation (output)
    printer control
6 - file management
    indices
    saving, deleting, reading data file
7 - internal data management
    buffers
    pointers
    system functions
    processing of data

--==== III - DESIGN ====--

The top level design concentrates upon the user-interface. The lower level of design concentrates upon the mechanics and maintenance aspects. This chapter will discuss the top level designing phase in terms of general requirements and of the functions to be automated.

Phased Development --- Development of this project is designed for two major phases. The first phase uses Turbo Pascal 3.x BCD with the top priority features. The second phase uses Turbo Pascal 4 and adds the more detailed features described in the following.

System Requirements --- Design begins with the declaration of the constraints. This project is to be implemented on IBM PC/XT/ATs or compatibles (XT/ATs are preferrable) with DOS 3.1 or higher and 320K of RAM. A hard drive is recommended and mono, CGA, or EGA monitors are supported. The application developer is required to use Turbo Pascal 3.x in the first phase of development and as soon as version 4 is shipped, the second phase will use it as its base.

Initialization --- A parameter passed in with the call to execute the program declares to the program where the data files are located, ie. the drive and subdirectory specifications. This allows the operator to keep sets of files within separate subdirectories. Once loaded, the initialization of the application program created by the 4GL includes an option for immediate update of files and the initialization of the date. Exit from the program is possible even from this point. A personalized logo screen, designed by the application developer, then appears to declare the program name and any copyrights and dates needed. Pressing a key continues into the main menu screen.

Menus --- The main menu for the entire application created by the 4GL includes a list of installed database applications. For example, a simple menu may include ---

0 = Codes
1 = Chart of Accounts
2 = Transactions
3 = Audit Trail

Each menu selected results in the use of the same screen layout described in more detail later. When a menu item is selected, the appropriate files are opened, the I/O

screen is created, and the file and field definitions are initialized.

Command Line --- Once within a menu selection, a commandline appears at the bottom of the screen. Two forms are used. The first allows adding and updating of records and the second allows only reading of data ---

```
Menu; Add/Update; Find/Data/Clear; Report; PgUp; PgDn --->
Menu; Find/Data/Clear; Report; PgUp; PgDn --->
```

Phase 2 includes security password options which allow for read/write access or read only access, therefore determining which commandline the end-user will see.

I/O Screens --- Data entry and display (for browsing through a search buffer) screens are the same. Standard I/O screens contain the following:

1 - the menu title
2 - the current mode of operation
3 - a message area for errors and how to exit
4 - a data entry area
5 - a commandline area
6 - a field-level prompt area
7 - an input prompt for data entry limitations
8 - a status line including --- the number records in the current files, the number of records in the search buffer, the record number of the record currently being entered or viewed.

Help Screens --- Help screens are constantly available by pressing the F1 key. Phase 1 help screens include user-specific information, such as accounting aids, and a user manual displayed sequentially in a circular list as the space bar is pressed. Phase 2 screens allow for going to specific pages in the on-screen manual, for scrolling forward and backward sequentially as desired, and allows for context-sensitive help. The standard user manual is pre-installed, however this manual is accessible to the application developer for editing.

Data Entry is standardized with I/O screens defined by the application developer. All data is entered within these screens and also displayed within them when in browse mode to page through a set of records in a search. Data entry is also dependant upon file definitions defined by the application developer using parameter tables. File sizes, access codes, indexing, and field definitions are contained in these tables. Data entry in a field uses the field definition to determine data type, field length, validity tests, and its location within the record. Each field has field-level prompts displayed as the field is entered by the operator.

Phase 1 includes data types of string, integer, BCD real, byte, date, and character. Automatic sequential fields (such as sequential invoice numbering) and default fields are supported. A key click option is available for keys pressed. A lookup facility is built-in for coded data fields as explained in the next section on the "Coding System". Validity tests are performed at the field level. Coded fields are allowed which require entry of only preinstalled codes. Numeric data is tested for minimum and maximum limits. String data entry is not allowed beyond its maximum length limit. Date entries are also tested for valid dates. When an entry is executed using the F10 key, the record is tested for the existence of required fields. If the record is acceptible it is recorded and any updates to related files are also updated at that time, such as updates to chart of account totals when a transaction is entered.

Phase 2 adds data types for telephone numbers, zip codes, social security numbers, time, and short dates (mm/dd).

Coding System --- Due to the fact that it is storage efficient to use codes for some data fields, a standard for all applications is a built-in coding system. Menu 0 is reserved for Codes. This menu selection allows the initialization of a coding library which contains lists of available codes used in the data entry of the entire program. All coded fields used throughout the program reference this code library to test the validity of the data entry for that field. Therefore, all codes must be entered in the coding library before they will be accepted as valid data in the coded data fields. No duplicates are allowed. An example would be codes for source documents in a transaction, such as "CK' for check or 'IN' for an invoice. These codes are indexed and quickly accessible at any time during data entry (adding or updating) by pressing the F2 key. A pop-on window lists the installed codes and their descriptions in alphabetical order and in a circular list for paging through. The operator then returns to the data entry from whence he came. Once a code is used as a reference in a coded data field that is saved to a file, the code is flagged and will not be allowed to be deleted to guarantee that it can be referenced by the coded field later.

Data Searches --- A data search menu includes a numbered list of all the fields in the records involved with the current menu selection. The end-user has the option to select any or all numbers of the fields for which he wishes to request concurrent special criteria (a logical AND search). After selecting the proper fields, the end-user is prompted for the special search criteria. String searches are case independent and allow for two search type options. "Begin-End" searches locate data matching and between a user-input beginning string to an ending string. Data is matched from the beginning of the data field. It allows for a search such as all names beginning with 'a' to 'm'. "Within" searches locate all data fields which contain the user-input string within them. Numeric and date fields allow the user to input minimum and maximum limits on the search.

Searches involving an indexed field will be implemented using B-tree indexing and other searches will be sequentially implemented. Data search records are saved in a circular linked list called a search buffer. The search buffer can be used for browsing through the data on the screen (paging to the previous or next record) or to print out numerous reports. Once the search has been

used, the search buffer can be cleared.

To see what search criteria have been used for the current search buffer, the operator can enter 'D' for data at the commandline. A pop-on window will display the list of criteria used for the search. The count of records contained in the current search buffer will be displayed at the bottom of the screen.

Phase 2 will add the OR, XOR, NOT, IF-THEN-ELSE and wildcard searches, and also will allow for ascending or descending sorts according to any chosen combination of fields. Search and replace features will be added in this phase. An operator option is available to search for case-sensitive data. Ad hoc searches are allowed in this phase and commonly used searches are saved as standard "macros".

Data Reports --- Report formats are initialized by the application developer. In phase 1, these will be implemented using text files. These formats will include field locations and field numbers. Special printer options, such as bold, underlining, and compressed print, will be imbedded within the formats. When requesting a report, the end-user is requested to initialize a search if none exists. A numbered list of installed reports is

displayed in menu fashion. When the operator selects the desired report, he may opt to have the report format displayed before continuing. This allows the operator to be sure he has the proper report. Counters, paging, column totals and numeric grouping (by groups of 5, for example) are available directly through the formats created by the application developer. To implement control breaks, other than numeric grouping, the application developer adds specific code. The page length is set in the report format and is the key that automatically manages pagination and the creation of a title block at the top of each report. One line in the report is required to contain fields from one common file.

Phase 2 allows for a more "free form" type report in which fields on the same line can be contained in different data files. Phase 2 uses the 4GL itself to create the report formats and parameters and also allows a maximum width report of 132 characters. A mathematical formula or a special logic procedure can be manually coded by the application developer.

Printer Support --- Due to the fact that there is little standardization for the many printers on the market, each printer requires its own set of driver codes for special attributes, such as bold, underline, compressed, 6 or 8 lines per inch, italics, and double wide. Printers are supported by a parameter file containing these code numbers. The file is created by the application developer and allows him to install codes for for any or all printer interfaces for which he has codes.

File Management --- File management will be implemented with a B-tree and indexing system. Parameters will be supplied by the application developer to define files and their respective fields. Restructuring of files is not allowed, however, a transfer from one file to another can be made. A re-indexing utility re-indexes files that have been corrupted. It uses the file, field, and index definitions to restructure indices after renaming the old indices to have as a backup until the process has been verified. The operator then has the option to delete the old indices.

User Interface --- User interface is well-served by

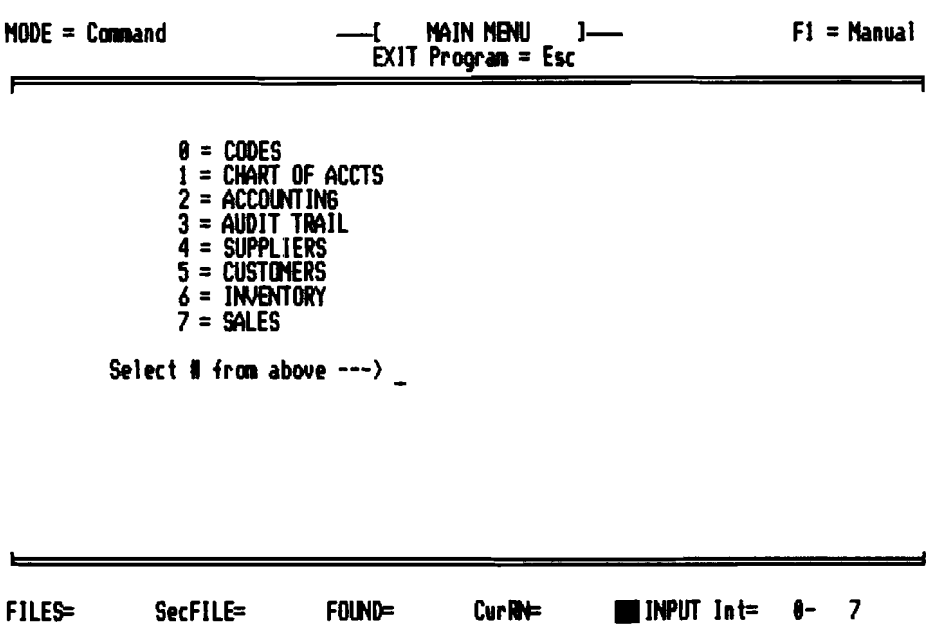
the several requirements previously mentioned - standardized screens, a pop-on code window, a pop-on manual or help window, and constant prompting at the field level. The user will know that he is expected to enter data when the color yellow (or reverse video on monochrome monitors) appears on the screen. Even character input will be prompted by listing all possible characters allowed in the input. Error messages must be preceded by a beep to signal the user to look in a pre-determined message area. At all times, the user will have a message explaining how to exit the current situation. The use of keys will be consistent throughout the application for the operator. Standard key combinations are pre-installed, however, these can easily be re-configured by the application developer with the simple reassignment of key constants.

--==== IV-SCREEN DESIGN ====--

Within the next four chapters are the more detailed aspects of the project. This chapter concentrates upon the I/O screen designs used by the data entry routines and report browsing routines.

Standardized screens --- Standardized screens are not only a benefit to the end-user, but also to both the software engineer and the application developer. End-user interface is of utmost importance, due to the fact that the majority of users are novices. Being able to consistently find error and status messages and prompts in pre-set locations aids greatly in the operation of the application program. However, the major advantage for the developers is the fact that all screen handling can be channeled through one screen formatting module. Any desired changes in the screen layout are made in that one module. Figure 1 is an example of a main menu screen from an application using the 4GL.

FIGURE 1 - Main Menu Screen:



I/O Screen --- When a main menu item is selected, the program then uses the standard I/O screen shown in Figure 2 and described below. The following list contains the description and location of the standard items in the screen layout and their reserved areas.

Name	Line	Description of use
Mode	1	Displays current mode chosen from the commandline menu. Command means it is waiting for a command. Add or Update are data entry modes
Title	1	displays the title of the item selected from the main menu.

Name	Line	Description of use
Manual	1	always shows how to get the pop-on manual or help screen
Message	2	displays messages such as error messages that tell what is wrong and what to do about it, how to exit, special instructions for the current situtation
Work area	4-22	area for entry and data display
Commandline	24	commandline menu field level prompts
Files	25	number of records used in current file
SecFile	25	number of records used in current secondary or relational file such as a file containing sold items related to a primary file of invoice data
Found	25	number of records found in a search and contained in the current search buffer
CurRN	25	record number of record currently displayed on screen (the actual physical location within the file)
Input	25	prompt area for absolutely every operator input in these formats: Byte= ###-### Int= ######-###### Real=########.##-########.## Max Length= ### (used for strings) A,C,D,F,M,R,U,(example of character input for commandlines)

Figure 2 - I/O Screen Format:

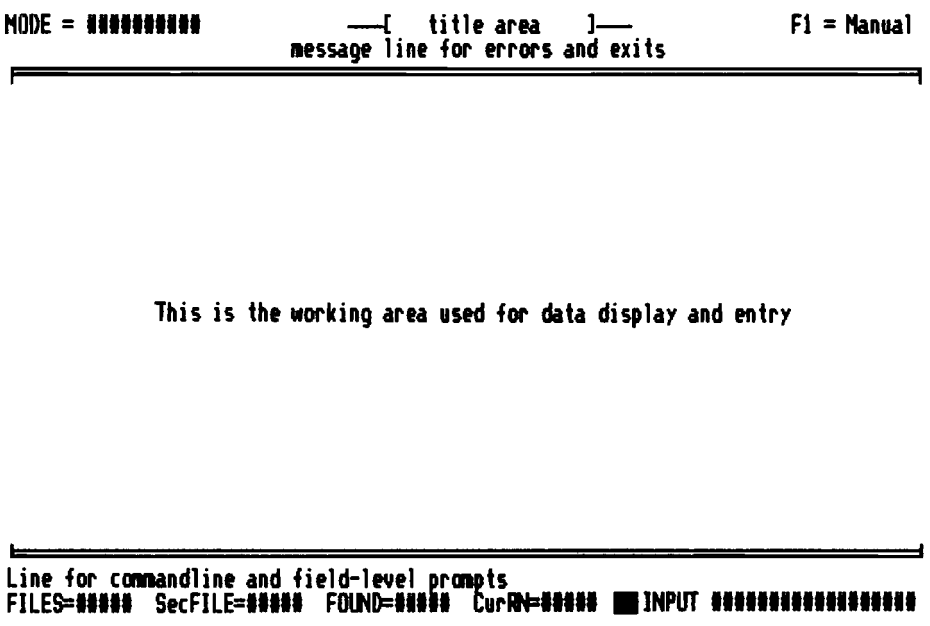
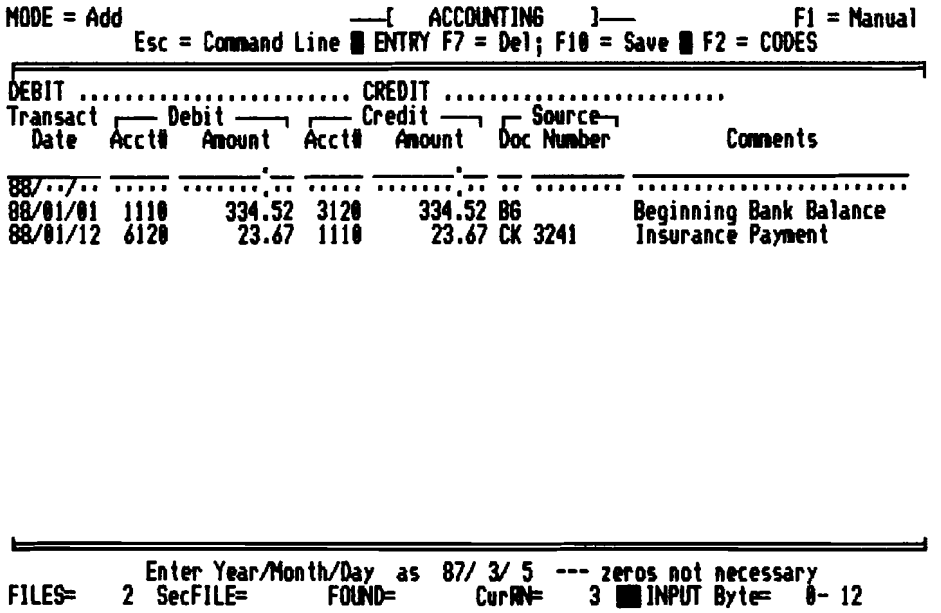


Figure 3 below demonstrates a feature that an application developer may use for columnar-type data entries. It exemplifies an actual entry screen as may be defined by an application developer for the entry of an accounting transaction. The example shows the screen after three transactions have been entered. In this application, all entries are made on the line directly under the column headings. When the transaction is saved by pressing the F10 key, a prompt line is inserted at this location, pushing the previous transactions down, thus keeping them on the screen as templates for further entries. The design of the screen within the entry area, lines 4 - 22, is under the control of the application developer. On line 24, is a field-level prompt describing the current field at which the cursor is located.

Figure 3 - Sample Entry Screen:



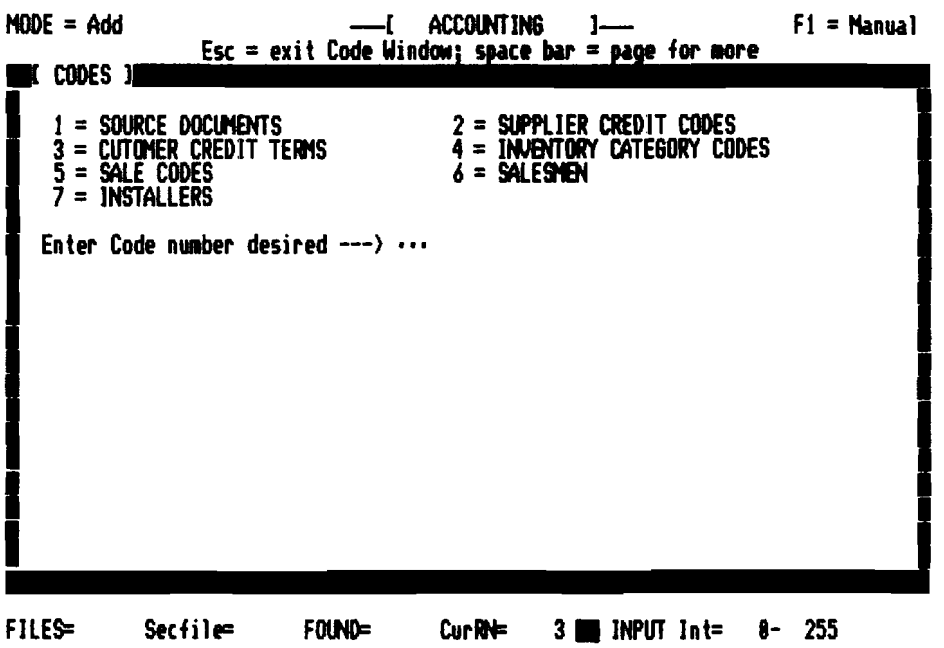
On the fourth line above, the name of the debit or credit is printed in the spaces when the respective account numbers are entered and found to be valid. This is a result of two lines of specific code added by the application developer. This presents an example of what special features can be accomplished with the 4GL.

Color Coding --- When the operator sees this screen on a color monitor, the data entry area for the month is highlighted in yellow to indicate that this information is what is to be entered. Yellow says "do something!". The two bottom lines also verifiy what is to be entered.

Column headings are red. The mode, "F1 = Manual", and status data on the bottom line are coded green. Titles are brown. Entry data that has already been entered is light cyan while the current data entry field is yellow.

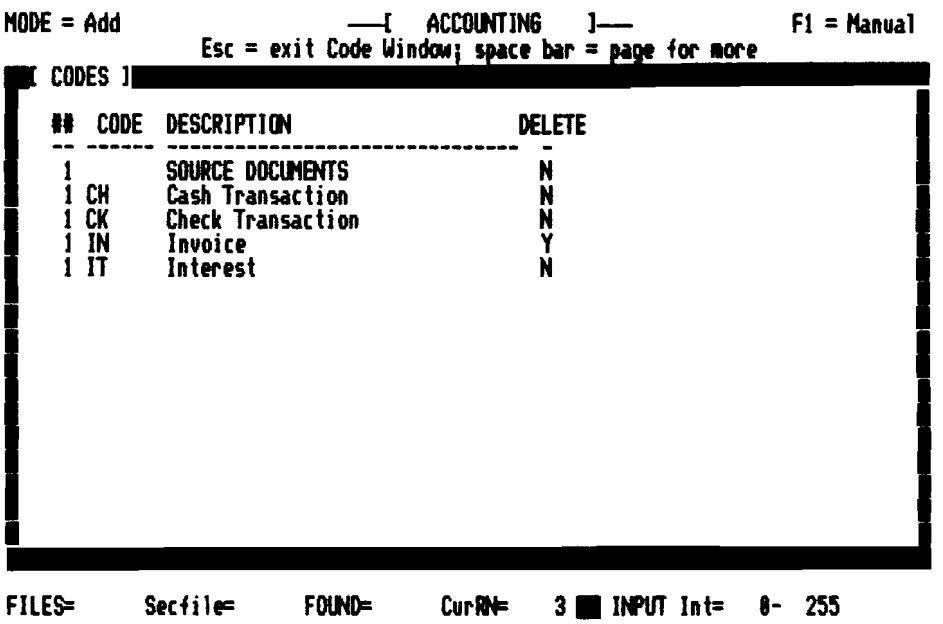
Pop-on Code Screen --- Figure 4 shows an example of a coding menu screen and Figure 5 shows an example of the screen after selecting a particular code to list. This screen is available by pressing the F2 key while in the Add or Update mode.

Figure 4 - Sample Coding Menu Screen:



After selecting the code listing desired (1 for example), a screen similar to Figure 5 is seen.

Figure 5: Sample Code Screen



Manual/Help Screens --- The on-line manual is available at any time by pressing the F1 key. Below is an example of an application specific help screen and of a 4GL general help screen.

Figure 6: Application Specific Help Screen

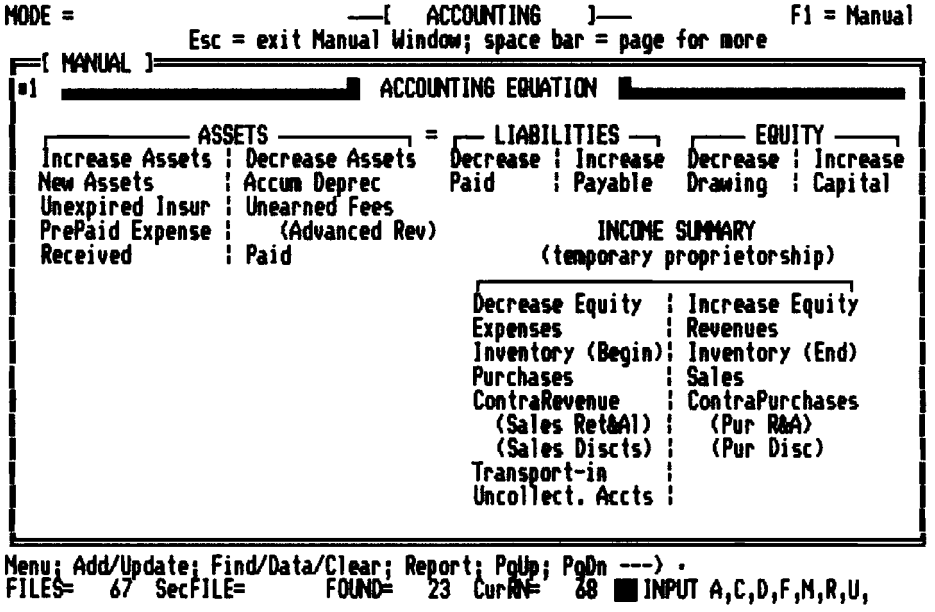
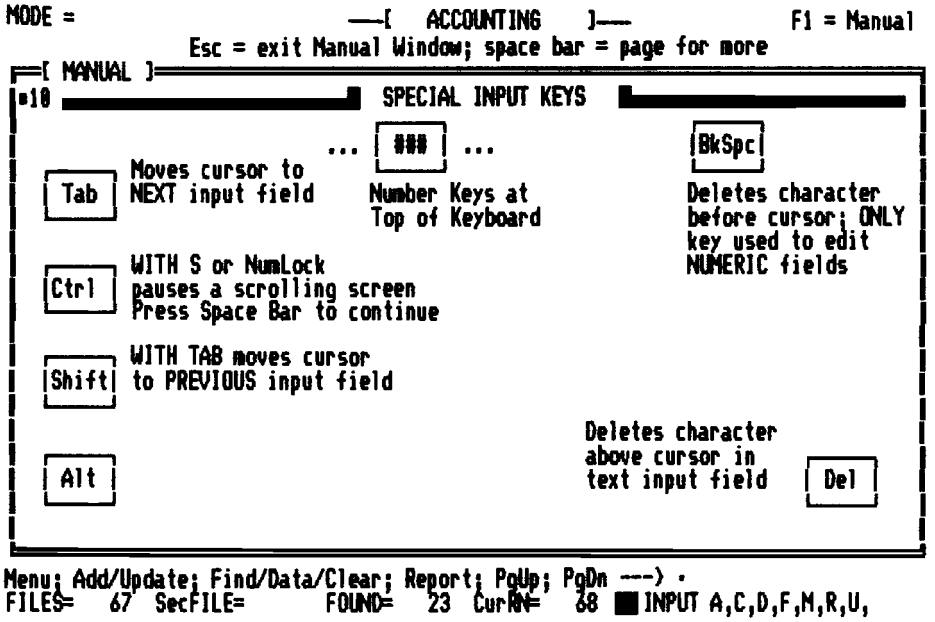
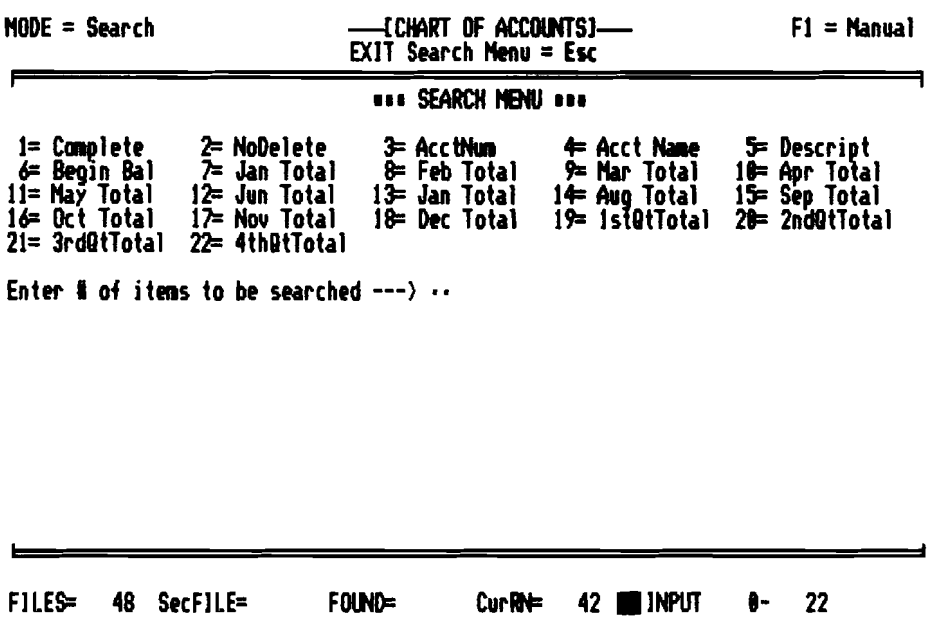


Figure 7: 4GL General Use Help Screen



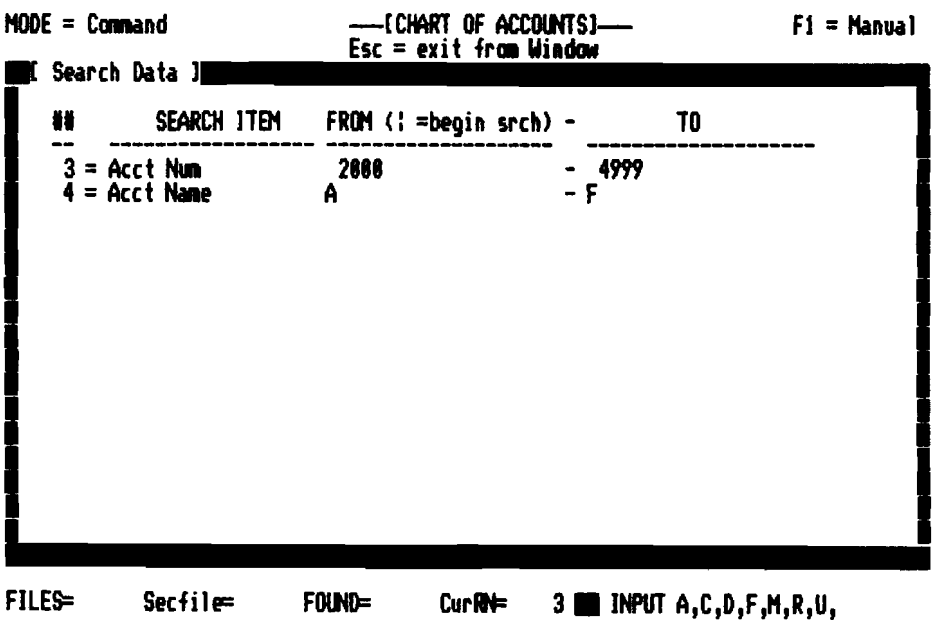
Data Searches --- From the commandline, the operator may select "F" for Find to initiate a data search. Figure 8 shows an example of a data search menu for a Chart of Accounts application. Every possible field is listed and the operator may select one or all of the fields to search by. If a string field is selected, the operator has the option to select a "Begin-End" or "Within" search as described earlier. If a numeric or date field is selected, the operator is prompted to enter the minimum and maximum desired in the search.

Figure 8: Data Search Menu Screen



Search Data Window --- The operator may request a listing of the search criteria used to collect the current search buffer (just in case one forgot) by pressing "D" in the commandline. A pop-on search criteria screen appears listing all limits of the search as in Figure 9 below.

Figure 9: Sample Data Search Screen



Data Reports --- A report may be initiated after a search. If a report is requested and a search has not been executed, then one is automatically requested from the operator. A search menu is then listed as in Figure 10 below. The operator has the option to see the report format (Figure 11) before executing the report in order to be sure it is the correct one. Following that, the operator may opt to have the search criteria listed in the heading of the report, to have page numbers, and to send the report to the screen, printer, or both.

Figure 10: Sample Report Menu Screen

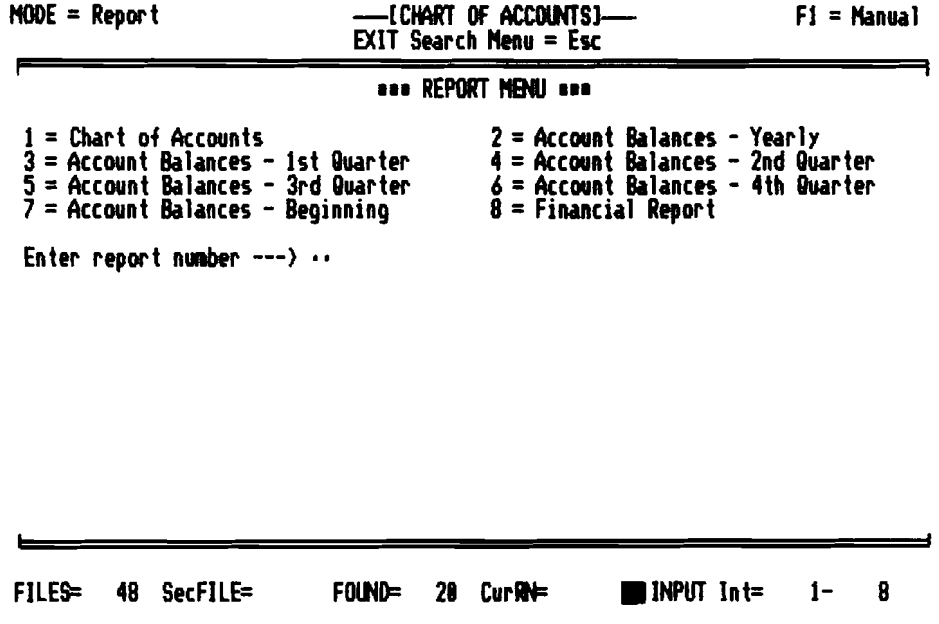
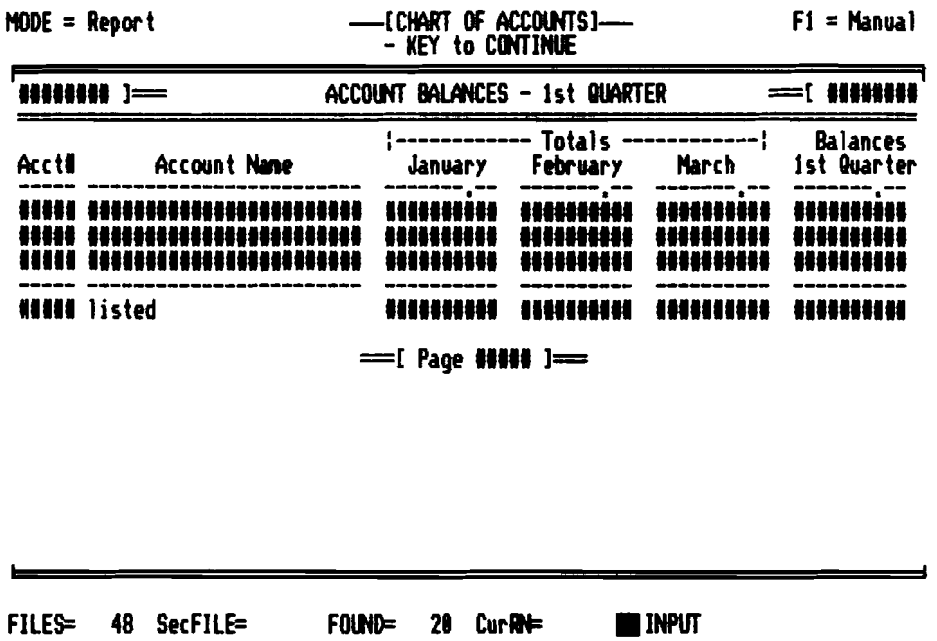


Figure 11: Sample Report Format



-=== V-DEVELOPER INTERFACE ===-

The third priority of a 4GL must be the interface between the application developer and the 4GL itself. The discussion in this chapter involves the communication between the 4GL and the application developer in terms of parameters which define I/O screens, output report formats, and the database itself (records and fields). The major time spent by the application developer should now be in the area of the database analysis and design. Once the design is established, the installation of the parameters involves the following steps:

1 - Parameter tables define the base application for ---
    A - the file data structure
    B - the entry screens
    C - the output or report formats
2 - Specific, non-standard code is added using ---
    A - the 4GL "macro" language described later
    B - the Turbo Pascal language
    C - inline and external code allowed through Turbo Pascal.

In phase 1, the implementation of the parameter tables is through tables set up in standard text files. Phase 2 uses the 4GL itself to create these interface definition files. The structure and content is the same with each method. The phase 2 method allows for greater security and speed in reading the files and initializing the file definitions internally within the program.

Initialization --- The application developer must provide parameters for initialization of the system. These include ---

1 - the location of the data files
2 - The computer type (PC/XT/AT)
3 - video mode
4 - the filler character for entry areas
5 - the client/end-user name
6 - the program name
7 - the logo screen with copyright and date
8 - the option to have the click sound for keys
9 - a table of the main menu selections and their related files

The menu title is used in the main menu, and when a selection is made, it is used in the title area of the entry screen. The related files are automatically opened, managed and closed during the use of that menu item.

I/O Screen Definitions --- Screen definitions start with the definition of the window environment. This includes ---

1 - The related file number
2 - The upper left corner location
3 - The lower right corner location
4 - Background color
5 - Foreground color
6 - The window frame type number
7 - The title color

Screen definitions include a table of the following data per source field ---

1 - The source file number
2 - The entry order number (3 for 3rd field entered in the input sequence)
3 - Field location on the global screen
4 - Field number within the data file
5 - Location of the lower right corner for a windowed field (as for a multiple line comment field)

The final step in defining a screen is the actual contents of the working or data entry area of the screen, such as the entry screen in Figure 3. The column headings and the data entry area (line 8 in the example) must be created.

File and Field Definitions --- The file structure of the database is the basis of all specific data input routines, file and record management routines, and output/report routines. The file and record definitions are created in table form in a text file called File-Def.PAS. The file contains the following data pertaining to a specific file ---

1 - File security code (read/write; read only)
2 - Record size
3 - Number of fields per record
4 - Set of indices indexing the defined file
5 - The major index for the file

It must be remembered that the 4GL does reserve menu 0 and file 0 for the code library described earlier.

File-Def.PAS also contains a table of the following data pertaining to specific fields within a file ---

1 - Field data type
2 - Field name (used in the search menu)
3 - Field-level input prompt
4 - An upper case flag for string data (yes/no)
5 - Field length internally within the record
6 - Field offset internally within the record
7 - Field minimum and maximum acceptable values
8 - Character set possible for character data

The final table included in File-Def.PAS is another table containing the definition of the indexing system ---

1 - The source file for the index data
2 - The total number of fields used in the key
3 - The set of ordered fields to create the key
4 - The key size in bytes
5 - A duplicate key flag (yes/no)

Output/Report Definitions --- Output formats for reports use the file, field, and index definitions entered as parameters by the application developer to create the proper data format to pass to the report routine. The report formats are contained in a file called Rep-Def.PAS. It is initialized with a table of ---

1 - Report name
2 - Related menu selection number
3 - Report number

The initialization table is followed by the report formats. A format is initialized with ---

1 - The report menu number
2 - The report number
3 - Page length of the report/document
4 - Body length of the report/document
5 - The number of records per group
6 - The line spacing between groups

Items 5 and 6 allow, for example, for the grouping of data in sets of 5 lines and then double spacing between them for readability.

The actual body of the report is created in the following manner by the application developer. Figure 12 gives an example of an accounting report format. The first character per format line describes the use of the line. The codes are ---

T - Title line (No data input into this line)
D - Divider line (No data input)
C - Column heading (No data input)
1 - Primary file data line
F - Footer line (fields are totals or counters)
P - Page number line

The # symbol represents the location of a field within the report line.

Figure 12: Sample Report Format:

```
T ######## ]==          ACCOUNT BALANCES - 4th QUARTER          ==[ ########
D ═══════════════════════════════════════════════════════════════════════
C  Acct#      Account Name          October   November   December  Quarter Bal
C  -----  ------------------------  -------,-- -------,-- -------,-- -------,--
1  #####  ########################  ########## ########## ########## ##########
F  -----                            ---------- ---------- ----------
F  ##### listed                     ########## ########## ##########
F
P                          ==[ Page ##### ]==
    Source Data
  Line  Source  Num
  Num   FileSub Flds  Source Field Array
  ----  ------- ----  -- -- -- -- -- -- -- -- -- -- -- -- -- --
> 5       1     6     2  3 15 16 17 21
> 7       1     4     0    15 16 17
```

The lower source data part of the table above defines the field content of each input area for the fifth and seventh lines in the format. The example defines the primary file as the source for the fields listed in the source field array. The fifth line, for example, has 6 fields which are numbered 2, 3, 15, 16, 17, and 21 input into the format. This works much like the FORM command in Turbo Pascal or like the PRINT USING in BASIC. Field number 0, the first source field on the seventh line represents the record count at the bottom of each page of the report. If the page numbering option is selected at the time of the report, then the page number will be printed using the format line preceded by the "P".

Specific Code --- A standard logic for IPO (input, processing, output) is embedded into the system, however it is impossible to foresee all uses for a program (as the many versions on the market prove). Therefore, the use of unprotected stubbed procedures within the 4GL system allow for the application developer to further customize the final application. Turbo Pascal commands, inline code, external code, and internal 4GL "macros" are available for the application developer to complete the specific code. The following chapter describes the "macro" language in more detail and lists the major "macro" library procedures and functions.

This specific code should be the only part of the final application that would possibly require maintenance, thus greatly reducing the cost in time and manpower for maintaining each application.

---=== VI - CODE DESIGN ===---

This chapter concentrates upon the design and organization of the actual 4GL code modules. A more detailed level of design requires a data structure, an organization of the code files, and system diagrams.

4GL Data Structure --- The internal data structure of the 4GL must be well designed in order to be parameter-driven, compact enough for PCs, and as unfettered with limitations as possible. The major structures designed within the system are ---

1 - The use of the B-tree indexing structure
2 - The use of pointers and linked lists
3 - The direct or absolute addressing of data
4 - The allowance for subscripted arrays of pointers to records for menus, files, fields, indices, and buffers

Dynamically created definition records are used to contain all current working data needed for the current menu item selected.

Organization of Code Files --- Due to the fact that the Turbo Pascal version 3 editor can only handle 64K of code per source code file and for organization purposes, the coding organization of the program includes the following separate protected code files. Listed with the

file names are the contents of each.

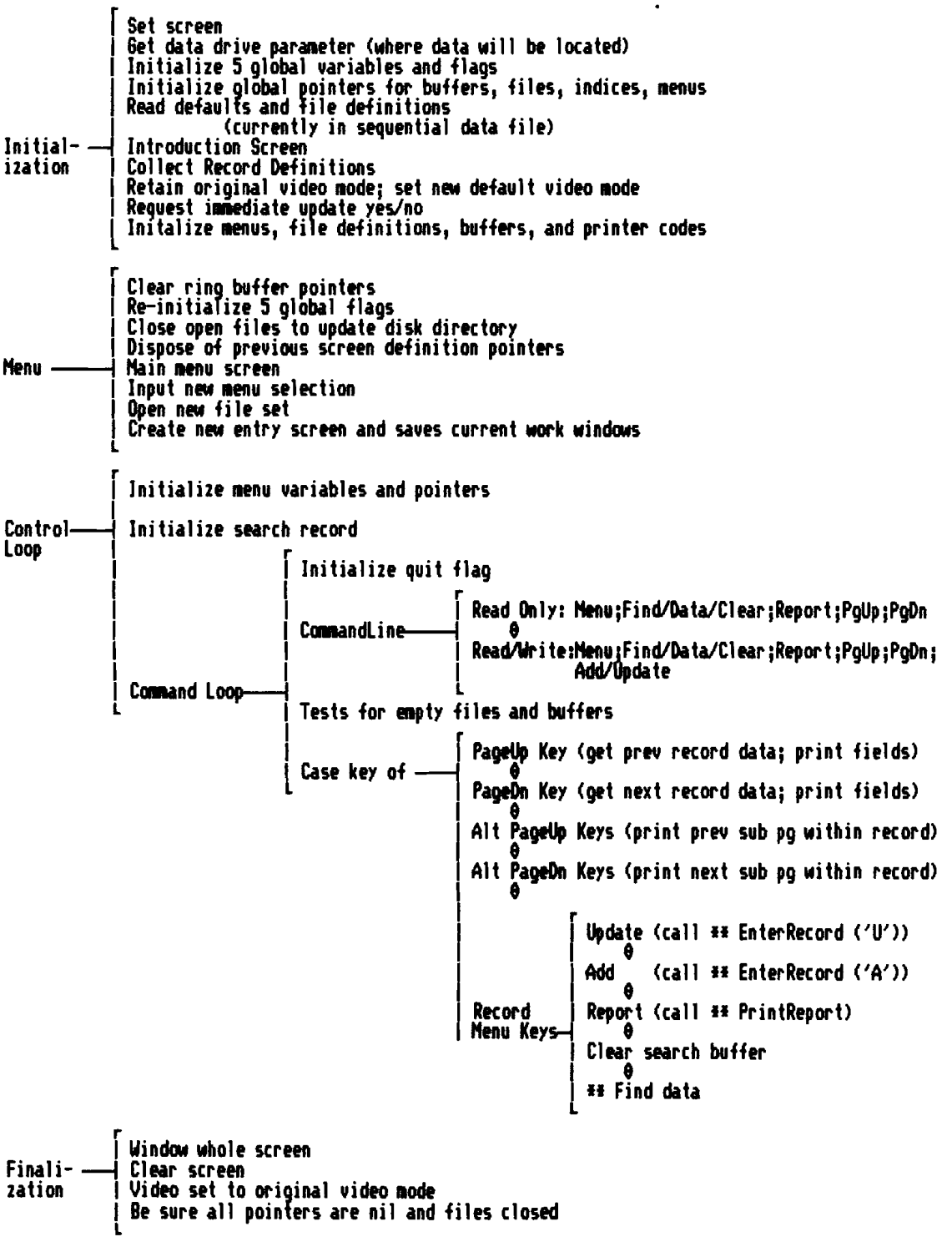
Type-Def.Pas -	all type declarations and a minimum of global variables
Library .Pas -	a universal supporting library containing independent modules used by the entire 4GL program
FileMang.Pas -	code for index keys file management
Pointers.Pas -	pointer management including queues, double linked lists, trees, and buffers
Support .Pas -	search menu print search data window get report option data report selection module data input module record management related to data entry
Controls.Pas -	the 3 major controling modules which call the supporting modules in Support above EnterRecord FindData PrintReport
Menu-Sys.Pas -	menu system
Init-Fin.Pas -	initialization and finalization modules

Also included are two application-specific files accessible to the application developer. These are the files that contain the stubbed interface modules for the developer. They are ---

Special1.Pas -	UpDateAcct for accounting modules any data entry specific tests
Special2.Pas -	any tests required for deletions specific report tests

System Diagrams --- The diagrams that follow present the high-level logic and design in a form similar to a Warnier-Orr diagram. Figure 13 contains the most general logic of the program. Figures 14 and 15 continue with more detail for the Record Menu Keys section in the lower right corner of Figure 13.

Figure 13: System Structure Diagram ---



θ represents "exclusive or" or XOR logic
** continued in more detail on following pages

Figure 14: System Structure Diagram (continued)

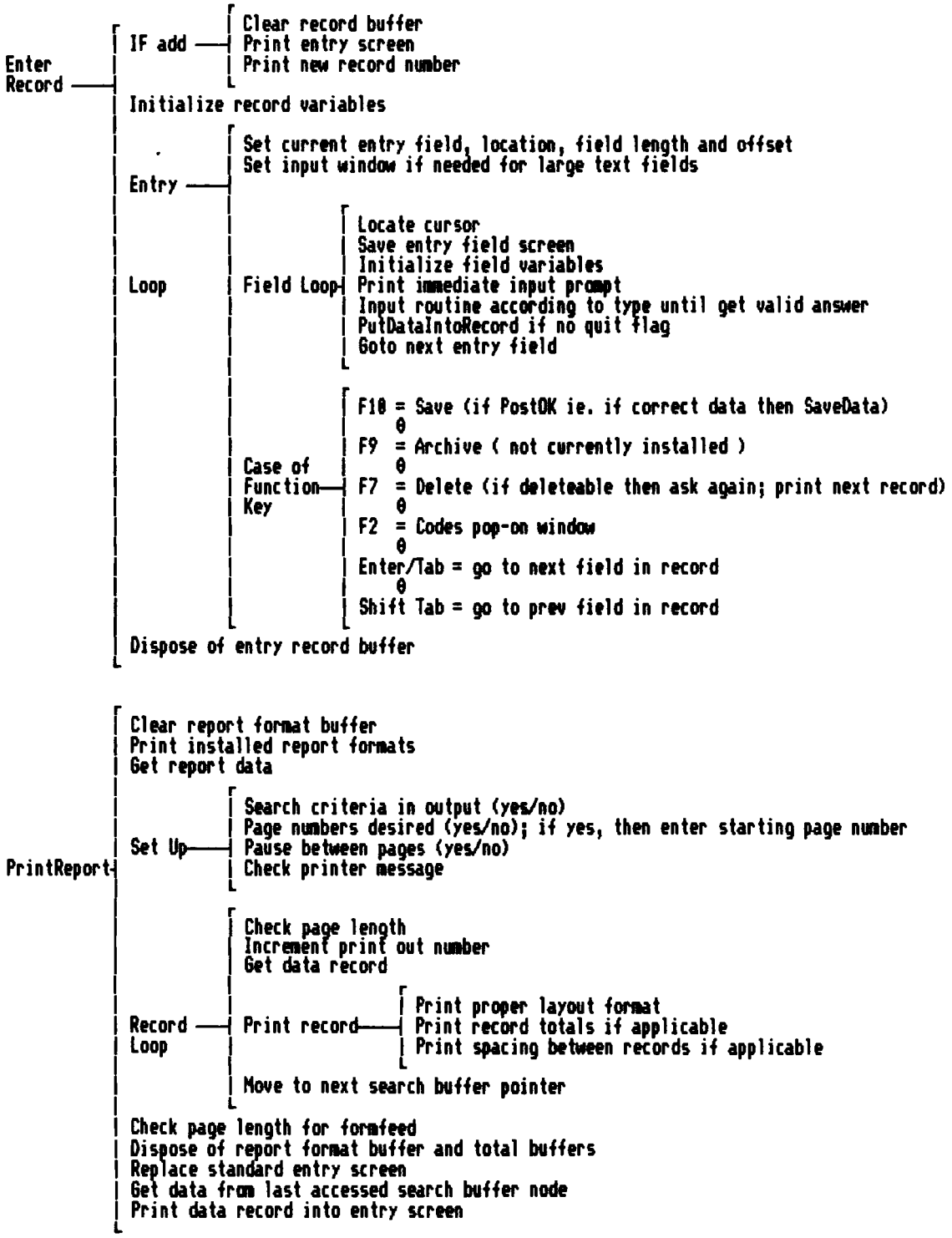
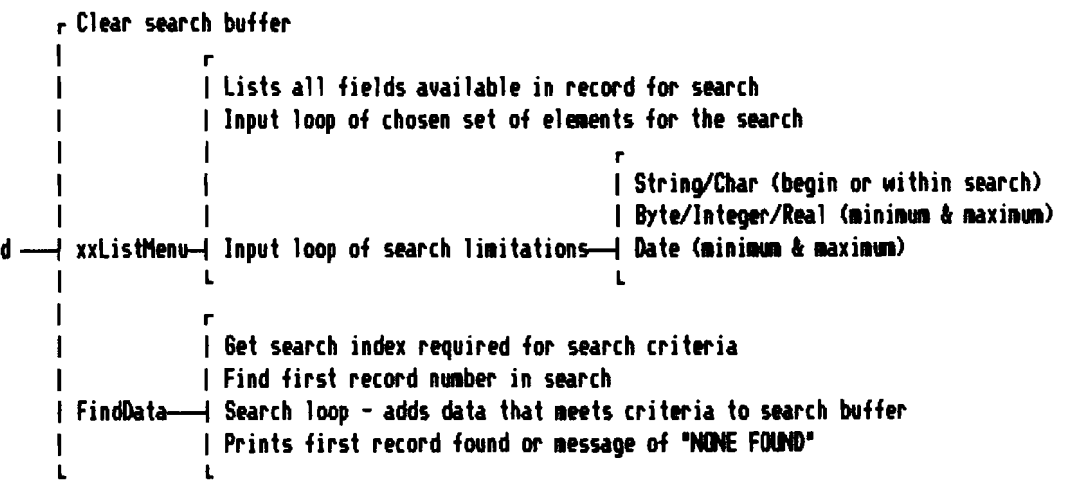


Figure 15: System Structure Diagram (continued) ---



Overlay Structure --- Due to the limitation of Turbo Pascal, version 3, the management of overlays is necessary in phase 1. Overlays were determined to be more efficient in this project than chaining, due to the code design. It is also possible for the application developer to create a batch file which automatically loads all compiled overlay modules into RAM disk, therefore nearly giving the speed of an EXE program. Four overlay files are used in this project.

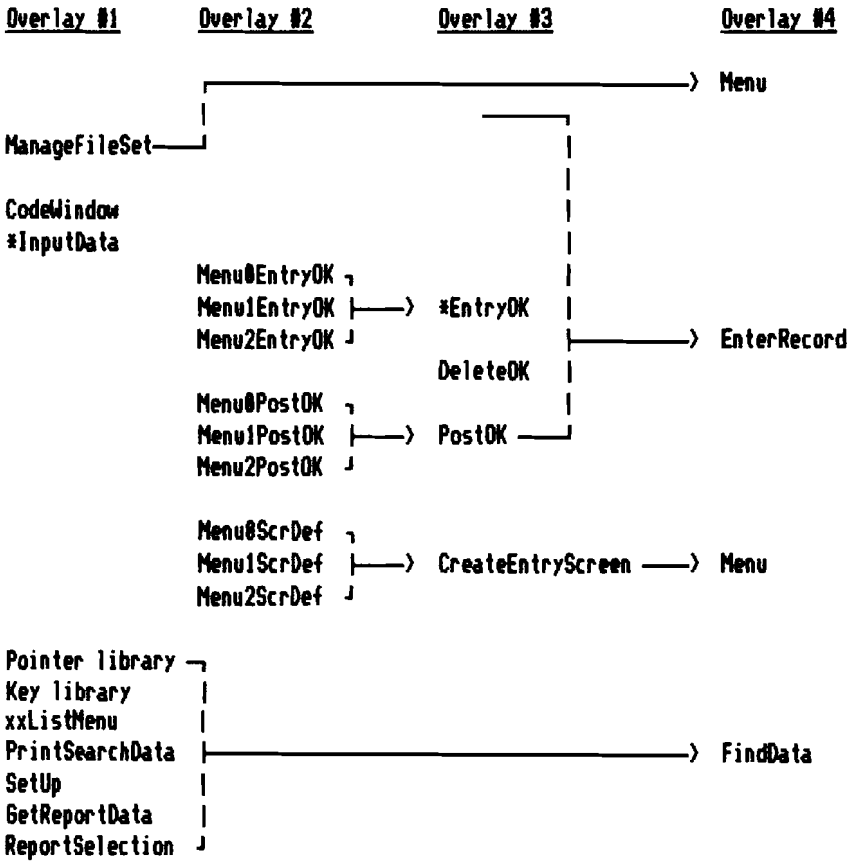
Overlay Design Methodology --- The rules for using overlays can become quite involved when matched with the logic of a program. A single overlay file can not contain modules which call any module within the same overlay code

file, including itself, because they can not exist in memory concurrently. For memory efficiency, the modules should be of similar size and each overlay file should contain as many individual modules as possible. The size of the largest module within an overlay code file is reserved in the main program memory for swapping in the called overlay modules. To retain speed, two or more overlay modules should not exist in a calling loop to prevent continuous trading of the modules into the memory overlay area for each loop.

One way to create an overlay system is to create a columnar table showing the calls or interfaces between only overlayed modules. Global modules are not considered, except for the fact that they must be declared before they are called. The major controlling modules are singled out. In this case, that includes the main module which calls Initialization, Menu, and Finalization. The first and last are ignored since they are only called once at the beginning and the end of the program and therefore can be located in any of the overlay files conveniently. The control loop logic is then the next area to consider. The control loop calls EnterRecord, PrintReport, and FindData which are the three major operations of the entire program.

A column is then created for each overlay file in the design and each column contains the names of the modules in it. To make it easier to design overlays, the software engineer should use color coding with high-lighter pens. For example, use green to high-light the modules involved in the input of data into the program, ie. those used by EnterRecord. One might use pink to mark all modules involved in the output of data from the program, ie. those used by PrintReport. Yellow could be used to mark all modules involved in searches, ie. those used by FindData. Any other modules might be marked with blue. The next step is to draw color-coded lines from one module to another to show what module calls another. To complete the process, each column must be checked to find the dominant module, in other words, the one that will be in memory the most. The dominant module should be marked, perhaps with red. There may be one dominant module in each column for EnterReport, one for FindData, and one for PrintReport. However, there should not be more than one per each of the three major controlling modules. If there is any problem in deciding between several modules, then perhaps a redesign should be considered.

By looking for the dominant module in each overlay file, the speed factor can also be considered. An overlay may be more efficient if it is broken down into more overlays if very many of its modules are called frequently or are contained in loops that conflict. In the diagram that follows, the * indicates the dominant module.



--=== VII - THE LIBRARY ===--

The master support facility for the modules contained in the system diagram is the protected code library of universally used procedures and functions which create a "pseudo language" or "macro language" from which everything else is based. Within this chapter is a list of the major library modules.

--- External Call Library ---

```
PROCEDURE FrameWin (UL,UR,LL,LR,Hor,Ver : Char);
```
FrameWin creates a frame around the current window using the given characters passed into it.

```
PROCEDURE GetScrn (    X,Y,NumChars: Integer;
                   VAR ChArray );
```
ChArray is an untyped variable used to pass in a variable sized array of pixel data. GetScrn gets the pixel data from the screen starting at position X,Y which are global coordinates.

```
PROCEDURE PutScrn (    X,Y,NumChars: Integer;
                   VAR ChArray );
```
ChArray is an untyped variable used to pass in a varing sized array of pixel data. PutScrn puts the pixel data onto the screen starting at position X,Y which are global coordinates.

```
FUNCTION GetVideoMode : Integer;
```
GetVideoMode retrieves the current video mode from DOS.

```
PROCEDURE GotoXYAbs (X,Y: Integer);
```
GotoXYAbs goes to the global coordinate position of X,Y regardless of the current window.

```
PROCEDURE InitVideo (Mode: Integer);
```
InitVideo initializes the video mode to Mode which can be 0-7. Standard settings are: 7 for 80x25 text and 3 for 80x25 color text.

```
PROCEDURE SetCursorSize (StartLine,EndLine: Integer);
```
This procedure sets the cursor size like the BASIC Locate statement.

```
FUNCTION WhereXAbs: Integer;
```
WhereXAbs returns the current global screen column location of the cursor.

```
FUNCTION WhereYAbs: Integer;
```
WhereYAbs returns the current global screen row location of the cursor.

```
PROCEDURE WriteSt   (St: Str255);
```
WriteSt is a fast screen access equivalent to Write.

```
PROCEDURE WriteStLn (St: Str255);
```
This is a fast screen access equivalent to WriteLn.

--- Window Handling Library ---

```
PROCEDURE AddWindow (WinNum : Integer;
                     WTitle : Str80);
```
AddWindow saves the current window contents and current cursor position in a buffer. It can create a frame around the new window and then window inside the frame. It locates the cursor at 1,1 in the new window and then sets window colors and clears the window screen. WinNum is the number of an array of window definitions also set up as initialization parameters by the programmer.

```
PROCEDURE RemoveWindow (NumToRemove: Integer);
```
RemoveWindow removes a given number of layers of windows, resets the final window colors, and relocates the cursor to the final window's last cursor position.

--- Sound Library ---

```
PROCEDURE Beep;
```
Beep merely beeps to get the operator's attention.

```
PROCEDURE Click;
```
Click creates a click sound and is called only by InKey below for each key pressed.

--- String Handling Library ---

```
FUNCTION UpCaseStr (S : Str255): Str255;
```
UpCaseStr changes string S to all upper case using inline code.

```
FUNCTION StrL (Len       : Integer;
               Character : Char) : Str255;
```
StrL returns a string of length Len filled with the given Character.

```
FUNCTION DelFrontSpc (TLine : Str255) : Str255;
```
DelFrontSpc returns a string with all front spaces deleted and is used for strings to be converted to numeric data because spaces will create a run-time error.

```
FUNCTION LSet (Len    : Integer;
               Phrase : Str255) : Str255;
```
LSet left justifies Phrase into a field of length Len

```
FUNCTION Center (Len    : Integer;
                 Phrase : Str255 ) : Str255;
```
Center centers Phrase in a string of length Len.

--- Screen Handling Library ---

```
PROCEDURE StatusLine (Which   : Char;
                      Num     : Integer;
                      Message : Str255);
```
StatusLine handles all reserved message areas and color coding. Any rearrangement of the screen would be done here.

```
PROCEDURE ErrMessage (VAR err     : Boolean;
                          Message : Str255);
```

ErrMessage calls Beep, calls StatusLine ('2',0,Message+' - KEY TO CONTINUE), calls InKey to create a pause and allow for an Esc, and returns the previous contents of line 2.

```
PROCEDURE ManualWindow;
```

This procedure is called any time an F1 is pressed. It displays the manual in a pop-on window and allows paging through the on-line manual.

```
PROCEDURE CommandLine (VAR Command : Char;
                           CSet    : CharSet;
                           Message : Str160);
```

CommandLine calls ColorCodeLine to display a color-coded commandline message on line 24. It then calls CharInput to input a char from CSet and sends Command character back to be used in a case to determine the next mode of operation.

--- Input Library ---

```
PROCEDURE InKey (VAR Special : Boolean;
                 VAR Char1, Char2 : Char);
```

InKey loops until a key is pressed. If the click flag is true then it also calls click when a key is pressed and returns the key that was pressed as 2 characters. The Special boolean is a flag that is true if the key pressed is a 2-code key. InKey is implemented with an MsDos call.

```
PROCEDURE ReadStr (VAR TStr          : str255;
                       LMax          : Integer;
                       which         : Char;
                       SpecialWindow: Boolean;
                   VAR BackX, BackY : Integer );
```

Absolutely all user input is entered through ReadStr in string form. ReadStr locates the cursor; tests for actual character input and special keys and key combinations.

```
PROCEDURE IntInput (     Ins,TabOver: Integer;
                     VAR IntNum     : Integer;
                         Bot, Top   : Real;
                         UnderL, NextLine: Integer;
                         Prompt     : Str255);
```

Ins is 0 for a "no insert" option; any other number tells the program to insert a line at the input location. TabOver is the column position of the prompt message. IntNum is the integer to be input; if an original value is passed in, then that value is displayed in the input area. Bot and Top are minimum and maximum values allowed. UnderL is the length of the underline prompt, in other words, the maximum length of the input area or maximum number of characters allowed to be entered. NextLine is 0 to flag for no linefeed/carriage return and any other number sends the cursor to the next line. Prompt is an input message preceding the input area.

The following are similar and related to IntInput:

```
PROCEDURE ByteInput (     Ins,TabOver      : Integer;
                      VAR Bite             : Byte;
                          Bot,Top          : Real;
                          UnderL, NextLine: Integer;
                          Prompt           : Str255);
PROCEDURE LineInput (     Ins,TabOver      : Integer;
                      VAR TLine            : Str255;
                          Bot,Top          : Real;
                          UnderL, NextLine: Integer;
                          Prompt           : Str255);
PROCEDURE DateInput (     Ins,TabOver      : Integer;
                      VAR TDate            : Byte3;
                          Bot,Top          : Real;
                          UnderL, NextLine: Integer;
                          Prompt           : Str255);
PROCEDURE CharInput (     Ins,TabOver      : Integer;
                      VAR TChar            : Char;
                          Bot,Top          : Real;
                          UnderL, NextLine: Integer;
                          Prompt           : Str255);
PROCEDURE RealInput (     Ins,TabOver      : Integer;
                      VAR TReal            : Real;
                          Bot,Top          : Real;
                          UnderL, NextLine: Integer;
                          Prompt           : Str255);
```

--- Output Library ---

```
PROCEDURE PrtStat;
```
PrtStat checks to see if the printer is turned on and select is on. If either is off then it gives an error message and waits for the operator to input to continue or to exit.

```
PROCEDURE PrtPrint (Prt    : Integer;
                    Format : Str255);
```
Prt is 1 for screen only output,
2 for printer only output, and
3 for both outputs.
If Prt is 2 or 3 then PrtStat is called.
Format is the message to be output with no linefeed.

```
PROCEDURE PrtPrintLn (Prt    : Integer;
                      Format : Str255);
```
This is identical to PrtPrint with a linefeed added.

```
PROCEDURE FormFeed;
```
This sends a formfeed to the printer if Prt is > 1.

```
PROCEDURE ColorCodeLine (    Prt      : Integer;
                             L        : Str255;
                             LineFeed: Boolean;
                         VAR lk       : Integer );
```
Prt is 1 for screen output only
2 for printer output only
3 for both outputs
L can contain the characters '^' or '~' or '\'
'^' sets bold attributes for printer output and sets highlighting for screen output.
'~' sets underline on for printer output and sets low lighting for screen output.
'\' turns off all attributes.
LineFeed is true to produce a linefeed.
lk returns an incremented line count if output is to the printer and Prt > 1.

--- File Handling Library ---

```
FUNCTION FileLen (DF : Datafile) : Integer;
```
FileLen returns the number of records contained in file DF. It includes the number of records used and number deleted and ready for reuse.

```
FUNCTION UsedRecs (DatF : DataFile) : Integer;
```
UsedRecs returns the number of used records with current data in the file DatF.

```
PROCEDURE OpInterrupt;
```
This procedure checks for an operator interrupt using the END key. It calls CommandLine to ask "Do you wish to ABORT? (Y/N)". This is used during reports.

```
FUNCTION IndexKey (IndNum : Integer;
                   RN     : Integer;
                   VR     : VariantRec);
```
IndexKey creates the index key for index number IndNum using the record number RN and the data record contained in VR, a variant record used to pass all data records.

```
PROCEDURE NextKey (VAR IndFile : IndexFile;
                   VAR RN      : Integer;
                   VAR Key );
```
NextKey goes to the index file IndFile to get the record number, RN, of the next record in the index and returns the untyped Key for the next record.

```
PROCEDURE PrevKey (VAR IndFile : IndexFile;
                   VAR RN      : Integer;
                   VAR Key );
```
This is identical to NextKey above, but gets the previous record.

```
PROCEDURE ClearKey (VAR IndFile : IndexFile);
```
ClearKey sets the index pointer to the beginning empty node of the index.

```
PROCEDURE FindKey   (VAR IndFile : IndexFile;
                     VAR RN      : Integer;
                     VAR Key );;
```

FindKey must be preceded by a ClearKey command for IndFile. It searches for the given index Key in IndFile. It finds the first occurance of an exact match.

```
PROCEDURE SearchKey (VAR IndFile : IndexFile;
                     VAR RN      : Integer;
                     VAR Key );;
```

SearchKey must be preceded by a ClearKey command for IndFile. It searches for given index Key in IndFile. For example, if IndFile has keys of CATALYST, CATAMOUNT, etc. and Key is CAT, then the RN associated with CATALYST is returned and Key := CATALYST. It finds the first occurance of the first part of a key. If none is found then returns a global flag OK as false;

```
PROCEDURE ManageFileSet (DoWhat : Char;
                         FSet   : FileSetType);
```

DoWhat is either 'O' for open or 'C' to close files. FSet is the set of numbers of the files to be operated upon. Indices are automatically updated and the existence of files and indices are tested; if a file or index does not exist, the operator is asked if the file should be created.

--- Miscellaneous Modules ---

```
FUNCTION Yes (Ins, TabOver, NextLine : Integer;
              Prompt                 : Str255) : Boolean;
```

Yes calls CharInput with character set of ['Y','N'] and returns a boolean value of true if the character is 'Y'.

```
PROCEDURE ClrLine;
```

ClrLine clears the current line where the cursor is located.

```
PROCEDURE IntDate (VAR IDate : Byte3);
```

IndDate returns the system date in three bytes of information (year/month/day).

```
PROCEDURE IntTime (VAR ITime : Byte3);
     IntTime returns the system time in three bytes.

FUNCTION Date : Str8;
     Date returns the system data in string form
     '@@/@@/@@'

FUNCTION Time : Str8;
     Time returns the system time in string form
     '@@:@@:@@'
```

All other procedures and functions are self-managing and will not be called by the application programmer. They are the kernel of the system and are called only by the the program itself.

---==== VIII - TESTING ====---

Testing of the 4GL must be extensive because it is the basis of many applications. To prevent compounding errors, separate testing stages are as follows:

Stage 1 --- File management is tested as a separate module to guarantee that all data is saved, retrieved, and deleted properly.

Stage 2 --- The menuing system and screen handling are tested with stubbed calls to the file management routines. This includes testing the windowing system. It must be verified that all screens retain their consistency according to the design described earlier.

Stage 3 --- Input routines are tested separately before integrating them into the Data Entry system. The major features to be tested include ---

1 - the error handling for invalid entry of data ---
    alphanumeric data entered into numeric fields
    numeric data outside of the minimum and maximum limits
    characters not allowed in the field

2 - the handling of the decimal point in real data fields

3 - the cursor location, especially in string input using both insert and overtype modes

4 - the use and clarity of error messages

5 - the proper return of the input to the calling routine.

6 - the detection of special key combinations

The first 3 stages can be tested simultaneously before merging them together.

Stage 4 --- Testing the merged modules with an actual application while attempting additions, updates, and deletions of every type must be completed before testing the report generation modules.

Stage 5 --- Report modules are now tested with standard screen layouts and then with various columnar layouts. Record counts, columnar totals, and paging are tested extensively.

Stage 6 --- The final testing is a relational accounting application merged with inventory, purchase orders, and point of sale invoicing.

IX - SUMMARY

The analysis and design presented here is now ready for another more detailed stage of design and then coding.

The 4GL is "... not a substitute for good analysis, design and project management."[6] The major coding task has been completed for the application developer, however a thorough analysis and design of the database iteself is the top priority before using the 4GL. It will be only as good as the database design itself. The 4GL has simplified the major functions that have now been automated. The wheel has been invented and the six spokes are the following management routines ---

1 - Menus and commandlines

2 - Screen management

3 - Data entry management

4 - Data search management

5 - Report generation

6 - File and record management

The 4GL now contains from 80 - 98% of the code needed for a relational database application. With the modular design presented herein, the application can be expanded to

include more functions by (1) specific code added by the application developer, or (2) by additional modules or expanded current modules by the 4GL developer. These can both be done without disturbing the structure of the data files themselves. Therefore, the primary goal of maintenance efficiency has been maintained. The secondary goal of consistent user interface has also been maintained with the above management routines. Any additional management capabilities will still be implemented through the existing structure for screen management, data entry management, etc.

The acceptance of 4GLs has grown rapidly within the past few years. It is, however, only the beginning. Even more advancements must be developed to keep up with the never-ending specialized needs of both today and tomorrow. Database management has led the way, but a similar approach is also needed for real-time systems and many other areas. The quest for efficient, well-engineered methods still goes on. There has never been a program that is totally finished. There is always another need that it might fulfill, thus one revision after another. Herein is only the beginning of a new 4GL adventure.

---=== BIBLIOGRAPHY ===---

[1]Turbo GhostWriter: Taking the Time Out of Turbo Pascal, This Month's MENU, Vol. 1, No. 2, August 1987, pp. 4.

[2]Paul Winsberg, CASE: Getting the Big Picture, Database Programming & Design, March, 1988, pp. 54.

[3]Jan Snyders, The CASE of the Artful Dodgers, Infosystems, March, 1988, pp. 28.

[4]The Reality of the Promise, InfoSystems, November, 1986, pp. 32.

[5]Kent Lawson, Thinking About 4GLs, Information Center, January 1988, pp. 28.

[6]brochures, ASCII (Automated Software Concepts International, Inc.), received January 1988.

[7]Pascal GhostWriter amounts to 'instant software', Comdex Show Daily, Vol. 5, No. 6, November 2, 1987, pp. 128.

[8]James R. Hughes, Moving Out of the Middle Ages, Infosystems, October 1986, pp. 76.