A Software Tools Based Application Environment

Steve Landers
scl@fs.com.au

Michael Selig
mike@fs.com.au

Functional Software Pty Ltd
P.O. Box 266
Willetton WA 6155
Australia

ABSTRACT

UNIX[1] is sometimes criticised for being too cryptic, or unsuitable for commercial users. It is argued that a 'full screen' approach is required rather than the 'software toolkit' approach used in UNIX. Yet it is this toolkit approach that can produce benefits to the commercial user by enhancing software consistency, quality and performance. This paper describes one solution that seeks to provide a 'commercial quality' screen based applications environment without losing the benefits of the software tools approach embodied in the UNIX programming paradigm.

Introduction

Functional Software builds products that address the "mainframe expectations" of large UNIX Data Centres. Yet the design considerations for such software are, or should be, similar to that for most other software designed for Open Systems environments.

- portability, which relates to at least the following areas:
  - portability of the application code itself, including considerations such as differences between language compilers
  - independence from underlying system data and filesystem structure
  - independence from differences between host operating system commands, their parameters and their output formats
- performance, or more appropriately, perceived performance
- the impact on other users of the system
- the maintainability of the software
- the speed at which the software can be developed

[1] UNIX® is a registered trademark of UNIX System Laboratories in the United States and other countries.

- consistency of the user interface, whether Graphical User Interface (GUI) or character interface, or both

This paper describes one method of addressing these issues, and some of the design decisions that were made along the way. The paper is presented in two parts - the first describes the design of the software architecture whilst the second discusses some of the more interesting implementation issues.

The paper seeks not just to say "Hey! Look what we have done" but also to show that, given appropriate design and careful implementation, it is possible to implement a "Commercial Quality" screen based applications environment that adheres to the UNIX philosophy.

The UNIX Philosophy

What then is the UNIX philosophy, and why is it desirable to adhere to it? It could be summarised in the following points:

- applications are built from a series of software tools
- each tool performs one clearly defined function and is optimised for that function
- tools are regarded as building blocks that may be combined without constraint
- software development is reduced, as far as possible, to assembling such building blocks
- if no suitable tool is available then consider building one rather than an application specific program

Why is it desirable to adhere to the UNIX philosophy? First, it must be stated that the philosophy is not an 'end' in itself, but rather a means to an end. If we look at the design considerations given previously, we can see how the toolkit approach facilitates each of the points.

Portability

Once a set of tools have been ported to a new environment any application built using the tools will also have been ported. Given that a well designed, orthogonal set of tools is available, and that these are used extensively in an application, there is usually significantly less effort in porting such an application than a 'traditional' compiled application.

If the tool performs one clearly defined function and is designed so that its input and output are well specified, then it is often possible to test each tool in isolation, rather than a monolithic application program.

Performance and impact on the system

If performance of the application does impact the system, then the tool causing the problem may be identified using standard UNIX commands (eg: *ps*). Conversely, a monolithic

compiled application program will require sophisticated debugging and tracing facilities, to identify the section of code causing the problem. These are not always available and are often difficult to use.

Since each tool is designed to perform one task it is possible to tune the tool for that task. Performance benefits then accrue to any application using the tool. If the tool is widely used, as is the case with the standard UNIX tools, then it is far more likely to have been tuned than a one-off piece of application code.

Support

The use of a development toolkit enforces many standards, and therefore has the potential to result in more consistent software. An extensively used toolkit is more likely to be debugged than one-off application code. Such reuse of software components greatly improves the quality of applications and hence reduces the support load.

Development Speed

The reuse of standard software components can speed application development by freeing the developer from 're-inventing the wheel'. But also by enforcing standards it reduces the number of decisions to be made during the design stage.

The use of a toolkit also facilitates rapid prototyping, increasing the likelihood that the final software product will meet the users' expectations.

Goals for a Software Toolkit

As would be expected, UNIX provides a number of software tools that may be used in applications. These tools typically address data management and manipulation and are suitable for embedding within applications. Other UNIX tools are designed to support the developer (eg: *SCCS* and *make)* rather than be used for constructing applications.

Generally, UNIX does not provide software tools that address the needs specific to commercial applications, such as user interface tools. As a consequence much of the commercial application development under UNIX has been based upon the more traditional monolithic compiled programs written in languages such as C.

As more and more software companies are moving their applications to UNIX many developers just bring their existing development methods and mind set from proprietary systems. Sometimes Unix is not as good at supporting such applications as some proprietary systems (this is bound to start a fight!) or the application may have already been tuned for the proprietary system.

What then should a software toolkit suitable for commercial applications look like?

If we were willing to accept gross generalisations, we might represent an application program structure as shown in Figure 1.

This division is somewhat arbitrary and disregards such issues as networking and ad-hoc

reporting. In addition, the boundaries between the various components are rarely so clear cut. Nevertheless this does provide a useful model for categorising existing software tools.

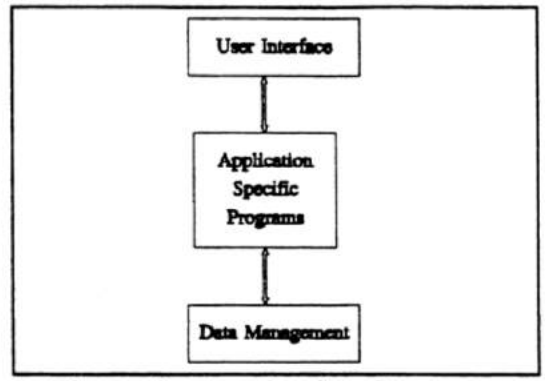

Figure 1 - Application Structure

The user interface tends to be the single greatest factor in the users' perception of the worth of an application. If the user interface falls down aesthetically the potential user may not even consider what may otherwise be an outstanding piece of software. Equally, the user interface must be consistent or users will become frustrated and small problems will be perceived as major.

The data management component may just be access to standard files and as such can be met by existing UNIX tools. Alternatively it may be access to structured data such as that provided by a database management system. If a data management and manipulation language (such as SQL) is provided then this has the potential to greatly improve the development speed and quality of application software by allowing the developer to work at a higher level.

Some application specific procedural code will still be necessary, even with the most advanced software tools. The issue is whether this should be a compiled language such as C, or a higher level interpreted language.

If we summarised the desirable features of a software environment suitable for commercial applications development, we might get a list like:

- ability to interface with existing UNIX tools
- full screen user interface tools
- a procedural programming language, preferably interpreted for speed of development and portability
- higher level data management tools, probably including a database of some sort

Given that Functional Software is in the business of providing Data Centre Management tools to support the management of a commercial UNIX installation, then providing the above facilities with the smallest impact upon other users of the system was also considered critical.

What is out there?

There is a large amount of software that attempts to facilitate development of applications. Unfortunately, much of it did not start life under UNIX and so does not aim to provide the benefits available from development using the UNIX philosophy.

Available software can be divided into three main categories

- database management systems with forms based front end and 4GL's
- screen, form and database libraries
- interpreted higher level languages

Each will be dealt with in the following section.

Database Management Systems

Most of the 'big' commercial relational database vendors also provide front-end tools for developing applications. These are generally described as 'integrated environments', which usually means that they don't interface well with other tools. This is not to say that these products are bad - just that they do not fit particularly well with the UNIX philosophy and do not facilitate the reuse of software components. Amongst the problems are:

- not all support both Graphical User Interfaces (GUI's) and character based terminals
- most do not use the Unix **curses** and **termcap/terminfo** facilities. The reasons for this tend to be historical - when these products were originally designed the versions of **curses/terminfo/termcap** were buggy or inconsistent (and often both). Also, the vendor might want to port to proprietary systems that don't support these facilities. The net result is the development of specific terminal handlers. In almost every instance these are inferior to the latest **curses** versions, both in terms of functionality (such as function key labels) and performance. Worse still, it means that an additional set of terminal definitions must be supported at each site
- the front end tools tend to be monolithic - once one enters the forms and menus it is not possible to use other tools that may be available and so one is limited to the features provided by the software vendor
- the front end applications tend to be huge, and consequently place a significant load on a system. This is especially so for the interpreted 'Forms' packages, although there are compiled interface builders that are just as bad. It is not always apparent why this is so - it may be because these interfaces were built in environments less sensitive to high memory usage than UNIX

Some database vendors also provide 4GL's (as an aside, has anybody been able to define what a Fourth Generation Language is?). These have their own set of problems. Many can only access the vendor's database and can't even read normal UNIX files. Others are no more than interpreted languages with limited type checking.

The database management systems themselves tend not to fit well into the UNIX programming environment:

- most commercial database management systems place significant load on the host system, in terms of memory, CPU and disk requirements.
- they claim tremendous transaction speeds but none seem to publish the time from

startup to completion of the first transaction. This becomes a problem if several tools are used in a pipeline and the startup time is considerable

- the interface to shell programs is incomplete or non-existent although most provide interfaces to compiled languages (usually through preprocessors)

Screen, forms and database libraries

There are a number of these libraries that provide higher level functionality. Despite the quality of many of these, they were not seriously considered because they did not support interpreted applications that would facilitate development and portability.

Interpreted higher level languages

There are a number of these available under UNIX , including *awk*, *Perl* and *Icon*. They provide, or can be made to provide, database interfaces and all interface to existing UNIX tools. Unfortunately, most did not provide screen based user interfaces, although some could be modified to do so. However, the main reason they were not considered suitable for application development was that they did not facilitate standardisation. Developers would still be working predominantly with a procedural language rather that combining a series of tools. These languages might be suitable for prototyping or implementing the tools themselves, but not as a tool for the application developer.

Summary

The limitations of available software environments to meet Functional Software's needs fell into the following categories:

- poor use of and interaction with existing UNIX facilities
- increased support cost over and above existing UNIX facilities
- performance and load on the system

Given that these are compelling technical reasons there was also one commercial reason that ruled out many of the existing products - the relatively high cost of run-time licenses.

That left the alternative of designing and implementing a software environment that addressed the stated goals.

Designing the Toolkit

The first step in designing the software environment was to establish which tools were required. The various components of the user interface would need to be identified, for example:

- browsing through output

- choosing a data item from a list
- forms based data input

Similarly, the data management functions:

- selection
- projection
- equi-join
- insert, delete, update

While designing the components it was important to be aware of the utilities and tools already available under UNIX, for three main reasons. Firstly, we don't want to reinvent the wheel, so to speak. Secondly, some tools that might prove inadequate for the final product could be used during the prototyping phase. And finally, the design experience that led to the existing tools might give valuable insight.

Amongst the utilities that were considered (and some of their problems) were:

- the standard UNIX pagers such as *more* or *pg* could be used for browsing output, but do not support left and right scrolling and so were only used for prototyping
- *awk* could be used for selection, projection and reporting but would prove too slow in many instances (such as lookups in screen forms)
- *join* could be used for prototyping database joins but has a number of problems (as discussed later in the implementation section)
- *sort* worked fine and removed the need for implementing a sort utility
- *sed* could be used for insert/update/delete but had no locking mechanism

Having decided on the toolkit components, and ascertained the available utilities to help implement the prototype, the next step was to design the interfaces to the toolkit, both from a user and programmer perspective.

The decisions on the programming language, user interface and database will be presented sequentially in the following sections of the paper, although they were in fact made in parallel.

The Programming Language

The design constraints for the programming language were:

- it should be interpreted, to facilitate portability and development. If a compiler was also available then this would be considered advantageous

- it should provide loose typing or context based typing, again to facilitate development but also due to the type of applications to be developed by Functional Software. Later, it was found that by performing type checking in the user interface the programming language did not require typing at all

- it should provide the common control mechanisms, including alternation and iteration

- it should interface with both the user interface and the database, as well as with normal UNIX files

- it should be able to run interactively for prototyping and debugging

- it should provide debugging tools such as a trace mechanism

In addition, if an existing product was used it would be considered advantageous if the product was both well known and well understood.

In the end the choice was obvious - the Bourne Shell.

When combined with The Functional Database™ and The Functional Toolset™ (see following sections) it provides a programming environment under which experienced Unix programmers find the development of full screen database applications comes in a natural way, fitting well into the Unix programming paradigm. It is interpretive, and shell compilers are available commercially.

And, as an added bonus, it doesn't require additional run-time licenses!

The User Interface

The user interface should, above all else, provide consistency and portability.

To achieve consistency one could choose to specify a standard and produce a corresponding style guide - as with **OpenLook** for example. However, this has some problems:

- there is no "enforcement" of the standard

- there would need to be extra effort by the developer to understand and conform to the standard

- there would need to be an additional Quality Assurance phase to confirm that the standard was being met

An alternative is to provide software that enforces the standard. This is usually done by providing a library of user interface components which is linked with the compiled application. Some examples of this are the **Motif** and **OpenLook/xview** toolkits. Whilst this would address the consistency issue it would still have problems:

- application programs would need to be compiled, rather than interpreted

- the portability of applications would require more attention
- the development time would be lengthened. Ideally, we are interested in prototyping and incremental development
- another layer of abstraction would be needed to provide consistent interfaces on character terminals as well as GUI's

The method chosen still enforces a standard, but does so by using a set of user interface tools, each a standalone program, which are connected via the shell and pipes.

To enforce consistency, the developer does not specify the layout of the screen - only the contents. Each tool decides on the most appropriate layout for the user's type of display. Portability of an application is achieved through portability of the tools. If input to a tool is sufficiently abstract then portability of an application can be ensured. Because each tool can be used standalone or combined via the shell, existing UNIX tools can be utilised by the developer. Although the tools are interpretive, it was anticipated that careful implementation would result in acceptable performance. If a particular tool did provide unsatisfactory performance then extra effort could be invested to tune that tool.

The next step was to decide on what these tools should look like. An attempt was made to abstract the key components of the user interface of existing full screen applications. The intention was to then develop a tool to implement each of the components.

The first observation was that most applications start with a menu (except for Motif applications which start with a 5 second pause ;-). Clearly a menu management tool was going to be central to the user interface toolkit.

Perhaps surprisingly, it transpired that the key components of a user interface can be abstracted to five tools:

Menu	provides screen based, hierarchical menu structure with the ability to invoke arbitrary UNIX command sequences as well as application specific commands
Scroll	to browse through the output from a program, and optionally print a range of pages
Choose	choose a data item from a list
Prompt	'Fill in the form type' data entry
Confirm	Confirm an action by function key or button press

These tools would cover all the main user interface functions. In fact, *Menu* could be regarded as a specialised case of the *choose* tool, where the data item is a tuple of (option, action), and *confirm* could be replaced by a menu.

Once the toolkit components were established, it was important to consider the 'look and feel' that would be presented to the user. This had to be designed so that it could be implemented

on character terminals as well as GUI's. Many GUI's implement "buttons" for selection or confirmation. To represent these on character interfaces it was decided to use function keys. Fortunately, the System V **curses** library provides support for function keys and labels. It was decided to limit the number of function keys used by the tools to eight, since this was the number of labels supported by **curses**. This also meant that the tools could run on all but the most primitive terminals. For consistency, each tool had to use the same function keys for equivalent functions, so a standard was adhered to, which (incidentally) was also SAA compliant.

Each of these tools presents a uniform user interface. This consists of:

- a title bar on the top line of the screen, with a title in the middle, the customer name at the right, and the tool name or menu name at the left
- a row of 8 function key labels on the bottom line of the screen, with F1 being Help, F2 Extended Help, F3 Exit, and F8 Accept. Other function keys may change depending on the particular tool
- a message line on the second bottom screen line. Error or informative messages appear on this line

An example of the screen layout is the menu shown in Figure 2.

The Menu Manager

Menu implements an hierarchical menu structure. The user positions the cursor on the desired option and presses **Accept**. Cursor positioning is via cursor keys, pressing the first letter of an option and, where available, the use of a mouse or other input device. If the option is an **action** then the Unix command corresponding to that action is executed. If it is a **submenu**, then that menu will be entered. Pressing **Exit** will take the user up a level in the menu hierarchy, or exit Menu completely if it was the top level menu. Pressing the **top** key will take you to the top level menu.

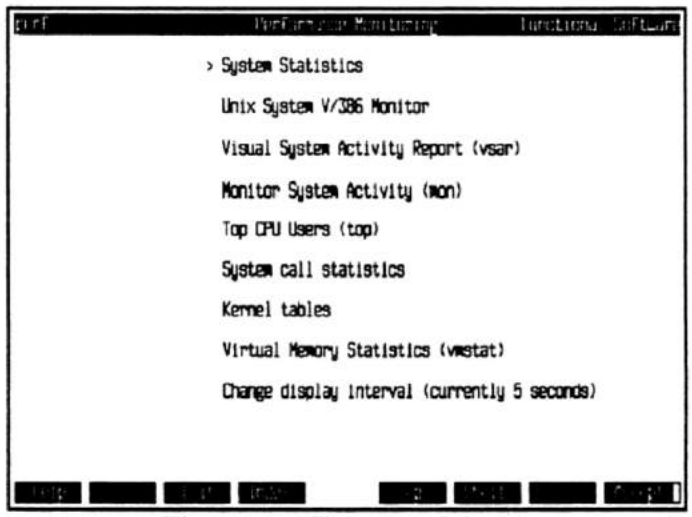

Figure 2 - Example Menu

The menu definition is, as discussed previously, instructive rather than procedural. Part of the definition for the menu in Figure 2 follows.

```
#
#       Performance Monitoring
#

menu          Performance Monitoring
set           interval    5

option        System Statistics
submenu       systat.menu
help          Monitor system performance using SYSTAT(1)
conditional   systat

option        Top CPU Users (top)
action        top -s $interval
help          Monitor the processes with the heaviest CPU usage
conditional   top
when          [ -f /dev/kmem ]
window        none
manual        man top

option        Kernel tables
submenu       pstat.menu
help          Display the contents of kernel tables using PSTAT(1)
conditional   pstat

option        Change display interval (currently $interval seconds)
prompt interval Display interval
```

Note the following features of the Menu description file.

- conditional menu items, which may be one of two forms. The first, the **conditional** directive, looks for an executable command in the path and if found, then the option is displayed to the user. The second, the **when** directive, allows the specification of an arbitrary command sequence. If the command returns a non-zero status then the user will not see that option

- comments are specified in much the same way as to the shell

- "self-modifying" menu text. The menu description can contain references to environment variables. If a variable changes, so will any text on the menu that refers to it. Environment variables may be set either by prompting the user to input a value, as in the last option, or by setting a variable to a text string

- support for management and display of error messages and diagnostics via popup windows. By default, standard error (**stderr**) is directed to a popup window. This fits particularly well with the UNIX paradigm, since **stderr** is usually only used for error messages. This may be overridden if the command being invoked does not adhere to this convention

- provision of help messages and extended help facilities (via the **manual** directive)

In addition, a full hierarchy of every menu in the system is available by hitting the **Index** key. The user can navigate the hierarchy or search for a menu containing a specified text string and then invoke the menu.

Scroll

Scroll could be considered a more sophisticated version of *more* or *pg*. It provides up and down, left and right scrolling through any output data. The user can search for arbitrary patterns, and can spool any range of pages to a printer.

As an example, consider the System V command to display all the processes on the system

```
ps -efl
```

This would produce output similar to that shown in Figure 3. Note that the output is wrapped, and the column headings indistinct.

```
 F S      UID   PID  PPID  C PRI  NI     ADDR    SZ    WCHAN    STIME TTY      TI
ME CMD
19 S     root     0     0  0   0  20  14e000     2 d00f2dc0 18:24:22 ?        0:
00 sched
10 S     root     1     0  0  39  20  278000    17 e0000000 18:24:22 ?        0:
01 /etc/init
19 S     root     2     0  0   0  20  27a000     0 d00641f8 18:24:22 ?        0:
00 vhand
19 S     root     3     0  0  19  20  27c000     0 d005ce4c 18:24:22 ?        0:
00 bdflush
10 S     root   102     1  0  28  20  29e000    15 d1017000 18:30:56 console  0:
00 /etc/getty console console
10 S     root   103     1  0  28  24  3c8000    23 d00e05d0 18:30:56 ?        0:
04 /etc/rwhod
10 S     root    99     1  0  26  24  36a000    19 d00b2c4c 18:30:55 ?        0:
00 /etc/inetd
19 S     root    89     1  0  20  20  377000    12 d003bf1c 18:30:51 ?        0:
01 netsched
10 S     root    73     1  0  26  20  334000    19 d00c15ca 18:30:41 ?        0:
01 /etc/cron
10 S     root    77     1  0  26  20  35e000    68 d00c1748 18:30:47 ?        0:
01 /usr/lib/lpsched

[]
```

Figure 3 - ps output

Alternatively, you could pipe the output into *scroll* and be presented with the screen shown in Figure 4:

```
ps -efl | scroll
```

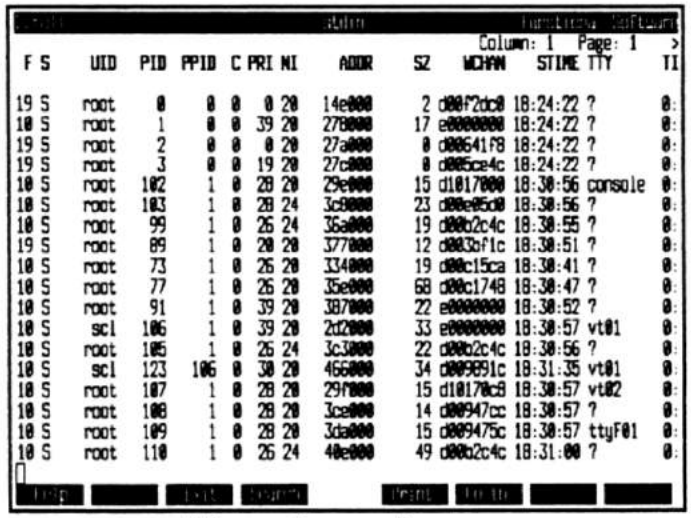

Figure 4 - scroll screen

Note the following aspects of the user interface:

- the title line across the top of the screen
- the function key template across the bottom of the screen
- the arrow on the top right indicating that there is more output to the right of the last column

Several features of scroll are worth noting. Firstly, the first line of output from some UNIX programs is a heading or title line (eg: *who*, *ps*). Using standard pagers (such as *more* or *pg*) this line is lost once the user moves to the subsequent pages. *Scroll* holds this heading, emboldens it, and displays it on every page. Secondly, many UNIX programs produce output that is wider than the screen. Standard pagers either truncate the output, or wrap it around thus destroying the visual effect of the output. In contrast, *Scroll* provides left/right scrolling as well as up and down.

A third feature is that a range of pages may be sent to the printer from within *scroll.*

These features of the *scroll* program lead to some interesting changes in the way applications are built. An application program can output a heading line, followed by the data. The developer does not need to provide options to print, or paginate the output.

Choose

Choose is a "pick-and-point" selection utility.

When choosing a data item, *choose* reads a list of items read from standard input, displays the list in a scrolling region on the screen, and returns the selected items to standard output. *Choose* provides facilities to scroll up and down, left and right (should the data be wider than the screen), and to search for patterns using regular expressions. *Choose* may therefore be regarded as a data reduction filter.

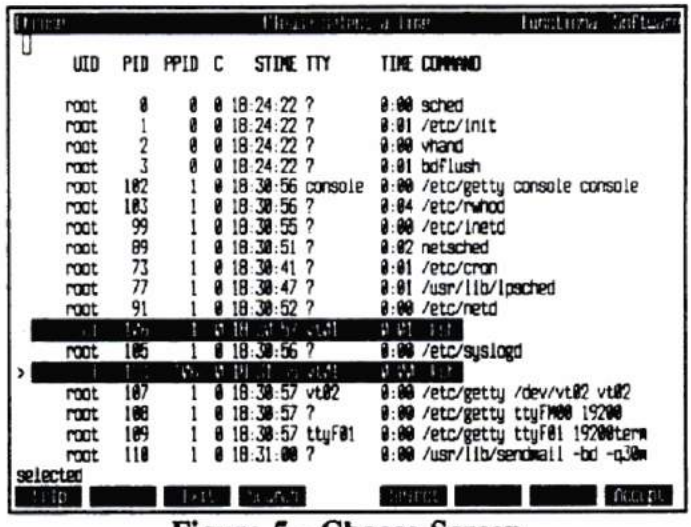

Figure 5 - Choose Screen

A typical use of *choose* would be to select a data item for use as a parameter to another UNIX command. For example, to choose a file to edit from the files in the current directory you could use:

```
vi 'ls | choose'
```

Similarly, to print one or more of the files in the current directory you could use:

```
lp 'ls | choose -m'
```

the -m flag indicating choose should allow selection of multiple lines from the input. In

Figure 5 we can see how a number of the input lines have been selected. The command used to do this was:

```
ps -ef | choose -m
```

The two lines selected would be written to standard output when the use presses the **Accept** function key.

Prompt

Prompt allows a user to fill in a form and when complete, it outputs Shell commands to set environment variables to the values entered in each field. Optionally, *prompt* can also load default values for each field from environment variables of the same name.

This association of shell variables to fields on a form, and the association between database column values and shell variables means that *prompt* can be readily used to input and manipulate database column values.

Figure 6 - Prompt Screen

Prompt is also capable of doing extensive validation of user input, of doing database "lookups" to ensure consistency and to provide useful feedback to the user.

Figure 6 shows an example *prompt* screen. The following is part of the input description. Note that the developer does not specify the layout of the form, but rather a description of the form's contents.

```
#
#       Copyright (c) 1990 Functional Software
#       All Rights Reserved
#

title "$FUNCTION User Account "

field Fullname
      format        30s
      prompt        Full Name:
      match         ^[^:]*$
                    Field cannot contain a colon.
      set Fullname 'echo $Fullname | lower | initcap'
      set Logname  'eval $Default_Logname'
      notnull

field UID
      format        5n
      prompt        User ID:
      entry         ' ... '
      verify        !db_row passwd "UID == $UID"
                    User ID $UID is already in the password file
```

```
        verify      [ "$UID" -ge $minUID -a "$UID" -lt "$maxUID" ]
                    User ID must be less than $maxUID and at least $minUID
        notnull

field Group
        format      10s
        prompt      Group Name:
        choose      'db_choose group Group'
        match       ^.+$
                    Group name cannot be null.
        set GID     'db_row -h group "Group == \"$Group\"" GID'
        verify      [ -n "$GID" ]
                    Group $Group does not exist - use choose key.
        help        Enter the default group name for this user.

field GID
        display
        prompt      Group Id:
        format      5N
        notnull

field Universe
        format      3s
        entry       'echo $Default_Universe'
        select      att\nucb
        help        Enter the default Login Universe of this user.
        when        [ -f "/etc/u_universe" ]
```

Some of the points to note are:

- comments are similar to those in the shell
- each field corresponds to an environment variable unless the field is marked as being local to the form
- the prompt text for the field may be specified or it can default to the field name followed by a colon
- the format of the field must be specified. This allows type checking to occur as the data is input and removes the need for type checking in other parts of the toolkit
- the input data can be matched against an arbitrary regular expression. If the match doesn't succeed then a diagnostic is printed and the user cannot exit the field
- a **default** value for each field can be set, either on input to the form or on entry to the field. An example is the UID field, where the **entry** directive would specify a command to generate a new user ID for the user
- the ability to set environment variables. Since each field corresponds to an environment variable this can be used to change the values in other fields
- an arbitrary command can be specified to **verify** that the data in the field is valid. In the example given, the UID field is checked to see that there is not already a user ID of the same number
- fields can be conditional, as in the Universe field in the example

- the **choose** key can be used to allow the user to choose from a list of available data items. This is usually used to invoke the *choose* program, as is done in the Group field in the example

- fields may be marked **readonly**, as in the GID field

- the **notnull** directive could be replaced by a **match**, as in the Group field

The output from *prompt* is a series of shell variable assignments as shown following. These variables can then be used within any shell script calling *prompt*.

```
Fullname='Alan Main'
Logname='alan'
UID='113'
Group='func'
GID='100'
Homedir='/users/func/alan'
Dirmode='750'
Shell='/bin/sh'
Access='fs'
Profile='shell'
Title='General Manager'
Organisation='Functional Software'
Phone='+61 9 246 3331'
Location='Perth - WA'
Public='yes'
Universe='att'
Printer=''
```

The integration of *prompt* with the database will be discussed further in the Implementation Issues section of this paper.

Confirm

Confirm displays a message (which may consist of a number of lines) on the screen, and asks the user to make a decision. The user is presented with a fixed number of alternatives, each is selectable by a different function key, which is labelled accordingly.

The user indicates the action to be taken by pressing one of the function keys. *Confirm* returns a status code corresponding to the key pressed. The developer can then use this in a shell **case** statement to act accordingly.

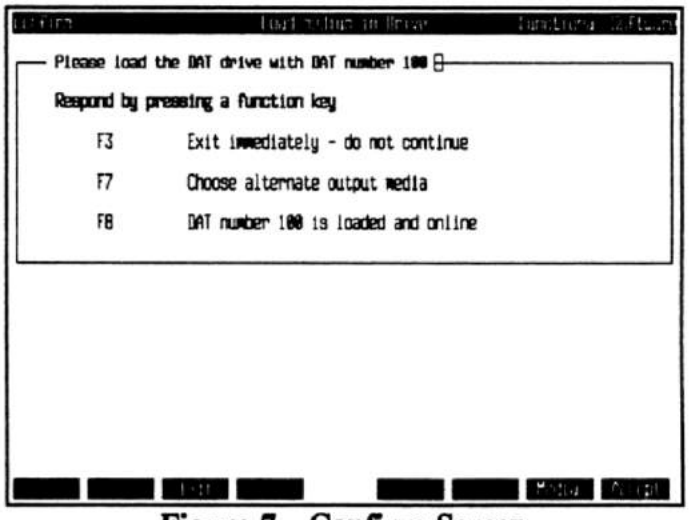

Figure 7 - Confirm Screen

The input commands to generate the confirm screen shown in Figure 7 are as follows:

```
Drive=DAT
Type=DAT
Number=100
PROMPT="Respond by pressing a function key" \
```

```
QUESTION="Please load the $Drive drive with $Type number $Number" \
L8=Accept \
F8="$Type number $Number is loaded and online" \
L7=Media \
F7="Choose alternate output media" \
L3=Exit \
F3="Exit immediately - do not continue" \
confirm
#
# we can now check the return code from confirm and act accordingly
#
case $? in
3)      ... ;;
7)      ... ;;
8)      ... ;;
esac
```

The Database User Interface

Database variants of *scroll* and *choose* were developed (called *db_scroll* and *db_choose* respectively). These provided cosmetic changes required for database integration with the user interface toolkit - for example scroll was modified to retain the primary key column of the table when shifting left and right. Database integration will be discussed further in the Implementation Issues section.

Data Management

One of Functional Software's products (The Admin Manager) required access to and management of several UNIX system files, as well as its own files. In particular, it was recognised that files such as the password file can be regarded as tables within the relational database model. It was decided to treat these as tables and to perform the data management via a relational database.

Adding these requirements to the broad requirements stated previously, we see that the database should:

- fit in with the UNIX philosophy
  - tool based rather than monolithic
  - use pipes to connect component tools
- use and/or interface with existing UNIX utilities where applicable (eg: *awk*, *grep*, *cut*, *sort*)
- use the Bourne Shell as the programming language (or "4GL")
- support the relational database model
- access and update existing UNIX files (eg: **/etc/passwd**, **/etc/hosts**)
- support the joining of these to other tables
- support multiple formats (eg: different column separators)

- be efficient for small tables (< 5000 rows)
- be simple and cheap
- be less concerned with
  - absolute data integrity
  - high transaction rates
- provide table level locking
- facilitate application development at a speed at least comparable with commercial RDBMS's and development tools

Because there were no available databases that meet all these criteria, and (admittedly) because it seemed like an interesting problem to solve, Functional Software developed the Functional Database.

This is characterised by:

- a suite of programs which allow Relational Databases to be built and manipulated under UNIX
- uses "flat" files for tables
- uses the Bourne Shell as the main programming language
- uses a syntax similar to *awk* for selection and reporting
- a data dictionary file for each table in the database which describes the names and formats of the columns and how to access the data in the table

The Functional Database implementation is discussed in some detail in the second part of this paper, which follows.

Implementation Issues

Requirements of the DB Environment

As has been mentioned previously, the overall requirements of the database and development environment were that:

- it should be relational
- must be able to access existing Unix text files, such as **/etc/passwd** and **/etc/hosts**, as database tables
- support several data types: string, numbers, date/time
- the database tools must be able to be easily integrated with the Bourne Shell and the user interface tools to provide a complete database development environment
- have reasonable performance on modest sized tables, say up to 5000 rows

The next step was to design the programmer interface to the database tools. Rather than going through the usage of each tool, their functionality can be understood more easily by example. The examples below demonstrate how we wanted to access **/etc/passwd** and **/etc/group** as database tables.

Database query tools

- Selection - list the users whose UID is greater than 100

```
db_sel passwd "UID > 100"
```

- Projection - show the Login name, UID and home directory of each user

```
db_proj passwd Logname UID Homedir
```

- Join - Join the passwd and group files by group ID

```
db_join passwd group GID
```

- Sort - sort the passwd file by login name

```
db_sort passwd Logname
```

- Combined query - for all users with ID >= 100, display their login group name, and sort the output by group name

```
db_sel passwd "UID >= 100" | \
db_join - group GID | \
db_proj  - Logname Group | \
db_sort - Group
```

Database update tools

- Insert new rows in the password file

```
<Generate data> | db_insert passwd
```

- Delete rows - delete all users whose UID is between 200 & 300

```
db_sel passwd "UID >= 200 && UID <= 300" \
| db_delete passwd
```

- Update rows

```
db_sel passwd <condition> \
| <process data> \
| db_replace passwd
```

- Lock/unlock table

```
db_lock passwd
db_unlock passwd
```

There are a few notes to make about the above:

- there was no requirement for an SQL interface
- *db_sel* uses an *awk* like syntax for its condition. This was also chosen for its string matching operator (~), which is extremely useful for certain queries
- complex queries are connected with Unix pipes. A dash "-" is used to indicate that data is to be read from a pipe rather than from a table stored as a file in the filesystem
- *db_insert/db_delete/db_replace* must handle at least table level locking to ensure that two users modifying the same table will not cause the data to be corrupted
- *db_lock* and *db_unlock* should use locking compatible with other Unix facilities. For example, use **/etc/ptmp** as a lockfile when modifying **/etc/passwd**

Initial Implementation

In order to prototype the design so that the approach could be tested as quickly as possible, the database tools were initially implemented as Bourne Shell scripts using available UNIX utilities. The intention was to prove the viability of developing applications using this environment, and then see which areas required performance enhancements.

The following gives a brief outline of how the initial implementation functioned:

- use *awk* to implement selection (*db_sel*). For example:

```
db_sel passwd "UID >= 100"
```

  is converted to:

```
awk -F: '$3 >= 100 { print }' /etc/passwd
```

- use *cut* to implement projection (*db_proj*). For example:

```
db_proj passwd Logname UID
```

  is converted to:

```
cut -f1,3 -d: /etc/passwd
```

- use *sort* to implement database sort (*db_sort*). For example:

```
db_sort passwd GID
```

  is converted to:

```
sort -t: +3 -4 /etc/passwd
```

- use *join* to implement equi-join (*db_join*). For example:

```
db_join passwd group GID
```

  is converted to:

```
db_sort group GID > temp;
db_sort passwd GID | join -t: - temp -j1 4 -j2 1
```

Besides highlighting the inconsistencies in the command line options in these tools, this does show how much can in fact be implemented using standard Unix utilities.

Data Dictionary Information

Implicit in the conversion from one of the **db** commands to one or more Unix utilities is the knowledge of certain pieces of information for each database table:

- the name of the actual Unix file holding the data
- the names and positions of each column in the table
- the format of the table (such as the column separator character)
- the format of the data in each column, needed when data is displayed to the user

For example, the UNIX password file would have:

columns	Logname, Password, UID, GID, etc
filename	/etc/passwd
separator	':'
format	8s, 8s, 6n, 6n, etc
	where 8s is a character string with maximum length of 8
	6n is a number of up to 6 decimal digits

Note that the list of columns gives both the names of each column and also the position in the table.

In the implementation, this **data dictionary** information is stored separately from the data in the table (as it does not change through the life of a table). Each database has a fixed directory containing all the data dictionary information for tables in that database, one table per file.

Integration with the Bourne Shell

As it was decided to use the Bourne Shell as the development language for applications in this database environment, it must be able to read, write and process data easily. In particular, there are times when row-by-row processing is needed, and in these cases the values of the columns must be accessible as variables to the shell program.

To do this a number of environment variables were introduced, corresponding to the data dictionary information just mentioned:

DBTABFILE the name of the Unix file containing the data.

DBSEP the value of the column separator character.

DBCOLS the names of the columns of the table, in the order on file.

DBROW similar to DBCOLS, but each column is separated by DBSEP, and preceded by "$".

DBFMT the output format of each column.

For example, for **passwd** table we might have:

```
DBTABFILE='/etc/passwd'
DBSEP=':'
DBCOLS='Logname:Password:UID:GID:Fullname:Homedir:Shell'
DBROW='$Logname:$Password:$UID:$GID:$Fullname:$Homedir:$Shell'
DBFMT='8s:8s:6n:6n:20s:20s:15s'
```

As an example of the use of these variables, consider the exercise of processing rows from the **passwd** table. In this example, we are setting the GID of every login name starting with 's' to 250:

```
db_sel passwd "Logname ~ /^s/" | {
     IFS="$DBSEP"
     while read $DBCOLS
     do
          # Now each column value has been assigned to
          # a shell variable of the same name.
          # Process the row
          GID=250
          # Write out the row
          eval "echo \"$DBROW\""
     done
} | db_replace passwd
```

Note the "read $DBCOLS". This is expanded by the shell to cause the entire row to be split into columns and assigned to shell variables of the same name. This close association between the database column names and shell variables makes the processing of database information fit in very easily with shell programming.

Also note the way we write out the modified row. "$DBROW" is a template for a row in the password file, and the shell substitutes the current values of each column from their corresponding shell variables.

Passing Data via Pipes

Consider the query:

```
db_proj passwd Logname UID Homedir \
| db_sel - "UID >= 100"
```

For the moment, let us ignore the fact that you would probably write this query the other way around. In the data dictionary information for the **passwd** table, UID is column number 3, but in the data passed down the pipe, it is column 2. How does the *db_sel* know that UID is now column 2? Similarly other "per-column" data dictionary information must be altered whenever columns are projected or tables joined.

Programmatically, the most elegant way to resolve this would be to use two pipes - one to pass the modified data dictionary information, and the other for the data. Unfortunately, the Bourne Shell does not support this, and even if some other shell did have the syntax to allow multiple pipes, we were concerned that it would be clumsy for the user.

Instead it was decided to pass the modified data dictionary information down the pipe, as the first line of standard output. If the table is specified as "-", the database commands would then read the first line of standard input to obtain the data dictionary information. This means that the *db_sel* in the above example knows that column "UID" is in fact column 2 of its input.

Virtual Tables (Views)

A relatively simple addition to the preceding scheme allows the output of a command to be used as input to a **db** utility, instead of a file. For example:

```
DBTABFILE=/etc/passwd
```
- A standard table

```
DBTABFILE="!db_proj passwd Logname UID Homedir"
```
- a view of the **passwd** table

```
DBTABFILE="!ps -efl | dbcols ......"
```
- A table that formats the output from the UNIX *ps* command (*dbcols* is a utility to extract specified columns of text and to convert it to database format by inserting TAB separator characters)

Similarly, the Admin Manager uses **df**, **ls** and other such virtual tables. This enables us to manipulate command output, and present it to the user using exactly the same tools as for database files. It also allows differences between command output to be hidden from the application by using different table definitions for various versions of UNIX.

Problems with the Initial Implementation

After the initial "minimal" implementation, a number of deficiencies and inefficiencies were identified:

- using *awk* with a separate program to convert the selection condition to an awk program is too slow for simple lookups which comprise about 90% of select statements in typical applications
- *join* has several drawbacks which make it poorly suited to commercial applications:
  - it requires both input files to be pre-sorted on the join column
  - it does not support joins on columns in numeric (as opposed to ASCII) order
- most select and join statements (typically over 70%) are immediately followed by a projection, requiring at least 2 extra UNIX processes
- *db_replace* will only work if the primary key of the input rows are unchanged. If the primary key is changed, *db_replace* has no idea how to match the changed row with the original
- the Bourne shell *read* command has a number of problems when used to read rows from a table:
  - it regards multiple IFS (input field separator) characters as a single character causing null fields to be skipped
  - apostrophes or any unmatched quotation marks in the data cause problems

Addressing these Issues

The discussion below describes how these issues were addressed, in the order in which they were presented above.

- *db_sel* was rewritten in C. The new version parses the selection condition and if it is a recognised simple form (typically about 90% of the time) it is executed directly, otherwise it calls *awk*. This enables full *awk* style conditions to be accepted but results in much better performance in the majority of queries

- *db_join* was rewritten in C. This implementation does not require pre-sorting and handles joins on numeric columns properly

- *db_sel*, *db_join* and *db_choose* (see below) were modified to support built-in projection by specifying the columns to project on the command line. This eliminates the need for separate *db_proj* processes in virtually every case

- a new *db_change* program was written to handle arbitrary changes to a single row. Although not as elegant as *db_replace* it is necessary to support modification to the primary key

- a new command (*db_readrow*) was written to avoid using the Bourne Shell *read* command. This reads a single row from standard input and sets the shell variables specified in DBCOLS to the column values read

Performance

Although performance was not high on the list of goals there are certain operations which must be performed reasonably quickly. Examples of these include:

- simple field validation, such as checking whether a value is already in a table

- single-row table lookups, such as retrieving the full name of a user given an ID

- producing a list of valid inputs for a field

These are typical queries that must be performed between entering fields in a form and are virtually all of the form:

db_sel <*table*> "<*col1*> == <*const1*> && <*col2*> == <*const2*> ..."

It was decided to concentrate on optimising these simple queries initially.

As a goal, it was decided that these queries should be no slower than using the System V Release 3.2 *grep* (which is substantially faster than previous AT&T version, and is in fact faster than *fgrep*). This translates to under one second on a 25MHz 386 for tables of around 1000 rows, or 100KB in size, and under 2 seconds for tables of 3000 rows, or 300KB. Care

was taken to ensure that the file being scanned was not in buffer cache, so that the disk I/O time was included.

With reasonably careful coding this was readily achieved, in fact obtaining performances generally 30 - 40% better than this.

Another simple performance enhancement is possible when it is noted that most of the above queries are known to never return more than one row. A variant of *db_sel* was produced, called *db_row*, which stops after finding one matching row, saving on average scanning half the table if the row is present.

Indices

The implementation of indices on arbitrary columns involves:

- quite a deal of implementation effort
- overheads with each update, insert or delete

Furthermore, as one of the goals was to operate on standard, and hence editable, Unix text files, it is quite conceivable that a user would modify a table with an editor like *vi*, thus invalidating the index.

Bearing these problems in mind, the question is "do we need the performance that is to be gained with indices?" So far the answer has been no.

Alternatives to Traditional Indices

If we restrict the use of indices to the primary key of a table, we can use a form of index that Unix provides for free - the directory. **Terminfo(4)** uses this approach in System V. Instead of having one large file which must be read sequentially, it is split up into a directory of smaller files. In the case of **terminfo**, the splitting is done using the first character of the key. It is easy to think of other schemes:

- one key per file, with the filename being the key
- hashing the key to give a filename

If a sequential table scan is required, it is just a matter of searching the concatenation of all the files in the directory. Furthermore, this scheme could even be extended recursively to take advantage of the directory tree structure that Unix provides.

Integration with User-interface Tools

As discussed previously, the major user interface requirements for commercial applications can be reduced to the following tools:

- hierarchical menus (*Menu*)
- scroll through (and possibly print) output (*scroll*)
- choose one or more data items from a list (*choose*)
- fill in a form (*prompt*)
- confirm an action (*confirm*)

We already had tools like *scroll*, *choose* and *prompt*, but how could these be integrated with the database environment given that they only operated on standard Unix text files?

This turned out not to be difficult. Database variants of *scroll* and *choose* were produced (called *db_scroll* and *db_choose* respectively). The changes required for database integration were in fact really only cosmetic:

- the column names are picked up from the data dictionary information and are displayed at the top of the screen
- similarly, the column formats are used to determine how to display each column
- in order to save a *db_proj* step, projection of nominated columns was built in
- *db_choose* also was given the extra feature of being able to display certain columns to the user, but return others to the program. This is useful when the program, for instance, requires a "key" to a table, but the user wants to choose using more descriptive fields
- left and right scrolling was enhanced to make use of knowledge of the width of columns, and whether the columns were keys. This gives a more pleasing user interface by scrolling whole columns, and by holding left hand key columns on the screen.

Data Formats

Because there is a requirement to store a number of different types of data including character strings, numbers, dates etc, the method in which these are stored on file may be required to be different to the form that a user may enter the data, or want it displayed. For example, internally we decided to store dates in the form YYYYMMDD. This gives a number of advantages:

- dates can be compared & sorted properly

- they can be displayed in one of several forms, depending on the user's requirements. For instance in the USA we may display the date as MM/DD/YY, but in Australia or Europe it would be DD/MM/YY

Thus the concept of "internal" and "external" data formats was introduced. All user interface utilities wishing to convert from one form to the other need only call a well-defined library routine. Furthermore, the addition of new data types is then particularly easy.

Data Entry

The integration of *prompt*, the "fill-in-the-form" tool, with the database came nearly for free. The only addition required was to handle data format conversion to and from "internal" and "external" format.

Using *prompt*, there is a one-to-one relationship between field names in a form and shell variables - *prompt* loads its default value for a field (if any) from a shell environment variable of the same name. Furthermore, it returns the field values entered by the user as shell variables. Bearing in mind that there is a similar one-to-one relationship between database column names and shell variables, this means that database column values can be manipulated easily by *prompt*. A fragment of an example of a typical shell program using *prompt* follows:

```
db_sel <table> <cond> | {
     .....
     while eval "'db_readrow'"
     do
          .....
          eval 'prompt -d form.pr'
          .....
          eval echo \"$DBROW\"
     done
} | db_replace <table>
```

In the above, "form.pr" is a prompt form definition file, specifying the fields to display, the validation for each, and so on.

What has been Learned

- Where possible prototype using available tools
- Performance of the prototype versions is adequate in a surprising number of cases
- Careful design of interfaces is required. These are difficult to change if you get them wrong! These include:
  - interface between tools
  - programmer interface
  - user interface
- When building a development environment, don't try to guess what users need. Development of a real application in parallel gives extremely valuable direction to the toolkit development

Conclusion

We believe that we have built a relational database development environment which fits in nicely with the Unix philosophy of a tool based approach, making best use of available utilities. Furthermore, we have achieved our goals of building a development environment that provides acceptable performance for modest size tables, and enables applications, such as **The Admin Manager™** to be developed at speeds at least comparable with, and in many cases much better than, other commercial products available at considerably greater cost.