Analgesia Database:
Perl Program

Version 0.90

J.M. van Schalkwyk

August 3, 2005

Contents

1 Introduction **8**
1.1 Database used 8

2 Initialisation **9**
2.1 Packages 9
2.2 Housework 10
2.2.1 The log file 10
2.2.2 Constants 10
2.2.3 A few noteworthy variables 11
2.2.4 Endian state 11
2.3 ODBC connection 12
2.4 A few frills 12

3 Main Window **14**
3.1 geometry of main window 14
3.2 variables 14
3.3 Control buttons 15

4 Utilities **17**
4.1 Constants 17
4.2 General purpose interaction 17
4.2.1 Alert 18
4.2.2 Print to file or console 18

4.2.3 Warn 18
4.2.4 Confirm 19
4.2.5 Ask 19
4.2.6 Choose 20
4.3 Date handling routines 21
4.3.1 Convert date to seconds 21
4.3.2 Local time 21

5 SQL/database section 22
5.1 Perform SQL commands 22
5.1.1 SQL execution 22
5.1.2 Retrieving multiple values using SQL 23
5.1.3 Fetch a script line 24
5.1.4 SQL batching 25
5.1.5 SQL commit and rollback 26
5.1.6 Key generation 26
5.2 Database creation or deletion 27
5.2.1 Creating the database 27
5.2.2 Kill the database 28
5.3 Menu population 29
5.3.1 MakeMenus 29
5.3.2 ClearMenus 29
5.3.3 MakePeople 30
5.3.4 KillPeople 30
5.4 Submit SQL script queries 30
5.4.1 QUERY 30
5.4.2 SQLMANY 31
5.4.3 DOSQL 32
5.5 Storage of table metadata 32

6 Menu creation 37
6.1 GoMenu 37
6.1.1 Entering GoMenu 37
6.1.2 Should we pop? 37
6.1.3 Clear local variables?! 38
6.1.4 Run associated script 38
6.1.5 Prepare to create 39
6.1.6 Destroy old widgets 40
6.1.7 Do it! 40
6.1.8 Close prior window 41

6.2 Subsidiary functions . . . 41
6.2.1 SubMenu . . . 41
6.2.2 Find all items . . . 42
6.2.3 Create items one by one . . . 42
6.2.4 Identify a monomorphic table . . . 43
6.2.5 Sub-menu . . . 44
6.2.6 Identify a polymorphic table . . . 44
6.2.7 Just make an item! . . . 45
6.2.8 ExitWindow . . . 46
6.3 Grouping . . . 46
6.3.1 ClearGroups . . . 46
6.3.2 FixGroups . . . 46

7 Tables **48**
7.1 Polymorphic tables . . . 48
7.1.1 MakeTable . . . 48
7.1.2 Identify columns . . . 48
7.1.3 Get row V values . . . 49
7.1.4 . . . 49
7.1.5 Get column details . . . 50
7.1.6 Make column header . . . 51
7.1.7 . . . 51
7.2 Monomorphic tables . . . 52
7.2.1 Make1Table . . . 52
7.2.2 [to fix] . . . 53
7.2.3 [to fix] . . . 53
7.2.4 [to fix] . . . 54
7.2.5 [to fix] . . . 55

8 Items **56**
8.1 CreateOneItem . . . 56
8.1.1 Startup . . . 56
8.1.2 Minor initialisation . . . 56
8.1.3 Create a label . . . 57
8.1.4 Create a button . . . 58
8.1.5 Create a checkbox . . . 58
8.1.6 Create a pushbutton . . . 59
8.1.7 Create a text field . . . 60
8.1.8 Create a poptrigger . . . 61
8.1.9 Create a scrollbar . . . 62

8.1.10 Exit 62
8.2 Subsidiary ‘Item’ routines 63
8.2.1 RunClearScript 63
8.2.2 FlipButton 63
8.2.3 ClearButton 64
8.2.4 Mutex2 65
8.2.5 GreyGet 65
8.2.6 DoButton2 66
8.2.7 Dopoptrigger6 66
8.2.8 DoCheckbox3 66
8.2.9 CheckEntry5 67

9 Scripting **68**
9.1 Complete script execution 68
9.1.1 RunWholeScript 68
9.1.2 Pull out commands 69
9.1.3 Other skip values: MARK 69
9.1.4 skip -3: LIST 70
9.1.5 skip -4: URZN 70
9.1.6 Other negative skip values 71
9.1.7 Returning 71
9.1.8 InterList 72
9.2 Command execution 72
9.2.1 DoCommand 72
9.2.2 Insert stack pops 73
9.2.3 Parenthetic argument 73
9.2.4 Invoke a routine 74
9.2.5 Implement REPEAT 74
9.2.6 A quoted item 74
9.3 SQL commands 75
9.3.1 QUERY 75
9.3.2 DOSQL 75
9.3.3 SQLMANY 75
9.3.4 KEY 76
9.3.5 OKSQL 76
9.3.6 COMMIT 76
9.3.7 ROLLBACK 76
9.3.8 ME 76
9.4 Arithmetic and Logical commands 77
9.4.1 ISNULL 77

9.4.2 NEG 77
9.4.3 NOT 77
9.4.4 ADD 78
9.4.5 SUB 78
9.4.6 DIV 78
9.4.7 MOD 78
9.4.8 MUL 79
9.4.9 SAME 79
9.4.10 GREATER 79
9.4.11 LESS 80
9.4.12 AND 80
9.4.13 OR 80
9.4.14 ISNUMBER 81
9.4.15 INTEGER 81
9.4.16 BOOLEAN 81
9.4.17 NULL 82
9.5 Flow of control and stack commands 82
9.5.1 RETURN 82
9.5.2 STOP 82
9.5.3 FAIL 82
9.5.4 SKIP 82
9.5.5 COPY 83
9.5.6 DISCARD 83
9.5.7 LIST 83
9.5.8 SWOP 84
9.5.9 BURY 84
9.5.10 DIGUP 84
9.5.11 MARK 84
9.5.12 UNMARK 85
9.5.13 URZN 85
9.5.14 RUN 86
9.6 Single letter commands and their friends 86
9.6.1 X 86
9.6.2 V 86
9.6.3 SETX 87
9.7 General purpose/text commands 87
9.7.1 IN 87
9.7.2 SPLIT 87
9.7.3 LENGTH 88
9.7.4 UPPERCASE 88

9.7.5 LOWERCASE 88
9.8 Date and time 88
9.8.1 NOW 88
9.9 Menu-related commands 89
9.9.1 ALERT 89
9.9.2 ASK 89
9.9.3 CONFIRM 89
9.9.4 QUIT 89
9.9.5 MENU 90
9.9.6 ENABLED 90
9.9.7 POPMENU 90
9.9.8 PUSHMENU 91
9.10 Local variables 91
9.10.1 NAME 91
9.10.2 ISNAME 91
9.10.3 $[name] 92
9.10.4 SET 92
9.11 Graphical 92
9.11.1 PAPER 92
9.11.2 INK 93
9.11.3 LABEL 93
9.12 Communication (experimental) 94
9.12.1 IAM 95
9.12.2 SEND 95
9.13 Experimental, obsolete and debugging routines 96
9.13.1 PRINT 96
9.13.2 DEBUG 96
9.13.3 REDRAW 96
9.13.4 REGEX 97
9.13.5 CHOOSE 97
9.13.6 MAP 97
9.13.7 RECIPROCAL 98
9.13.8 FORCEX 98
9.13.9 SETZ 98
9.13.10 TEXTBEFORE 99
9.13.11 PAD 99
9.13.12 JOIN 99
9.13.13 SECONDS 100
9.14 End of a long run 100
9.15 Subsidiary routines 100

9.15.1 XPrint 100
9.15.2 Invoke 101
9.16 Local variables 101
9.16.1 ClearLocalNames 101
9.16.2 KeepLocalNames 102
9.16.3 RestoreLocalNames 102
9.16.4 CreateLocalName 102
9.16.5 SetLocal 103
9.16.6 IsLocal 103
9.16.7 FetchLocal 104

10 PDB Creation **105**

10.1 PDB file format 105
10.2 Our own header 106
10.2.1 Column data types 107
10.3 Data row format 108
10.4 PDB creation routines 109
10.4.1 MakeAllPDBs 109
10.4.2 MakeOnePDB 110
10.4.3 MakeOurHeader 111
10.4.4 FetchAllColumns 113
10.4.5 MakeColDescriptor 114
10.4.6 FetchAllRecs 117
10.4.7 FormatDatum 120
10.4.8 SqueezeDate 122
10.4.9 SqueezeTime 122
10.4.10 MakePalmDBHeader 123
10.4.11 Print0 125
10.4.12 Print4 126
10.4.13 Print2 126

1 Introduction

In previous documents (PDA Implementation Details, Part I and Part II) we described SQL code underlying our database, as well as how we dynamically create user menus from data stored as SQL.

This documente shows how we flesh that code out into a Perl program. Using ActivePerl for MS Windows [1] we establish an ODBC connection, create and populate our database using SQL scripts, and then create a user interface with the help of menu data stored in the database.

Readers of this document are assumed to have a fair working knowledge of SQL, as well as being fairly fluent in Perl. Minor familiarity with ODBC (SQL/CLI) will be an asset. Although we use Perl/tk to create a graphical user interface we don't assume familiarity with Perl/tk or Tcl/TK.[2]

This documentation and all associated code is released under the GNU Public Licence (GPL). Please note the conditions of this licence, a copy of which can be obtained at: http://www.gnu.org/copyleft/gpl.html. This document is Copyright ©J van Schalkwyk, 2005, as is the associated perl program (pain.pl) released together with this document.

1.1 Database used

For initial implementation, we decided to use the little-known Ocelot SQL. Reasons for this choice include:

- Availability of the Ocelot database under the GPL;
- Close conformance to reasonable SQL standards;
- The small size and fast speed of the database;
- Ready ODBC connectivity.[3]

Because we have eschewed vendor-specific SQL constructs to the best of our ability, we hope this will facilitate use of our SQL on other platforms. Programs such as Access [4] will need an extra layer between our code and the ODBC.[5]

[1] Versions 5.6 or 5.8. Heck, if you can get it to work under Windows, it'll work anywhere :-)

[2] not that such familiarity would be without benefit!

[3] There's also the fact that on the couple of occasions where we've contacted the Ocelot chaps, they have been incredibly helpful and friendly, despite the fact that no money changed hands!

[4] where the developers would appear to have gone to great lengths to make a non-standard product

[5] Our use of pipes as delimiters will also need to be translated to a more Visual-BASIC friendly character, or sequence of characters in this context!

2 Initialisation

We start off with standard Perl initialisation, and then establish an ODBC connection. For the Perl program to function in Windows, you will need to have Active Perl, and must install Ocelot SQL.[6] Make sure that you enable ODBC connectivity when you install Ocelot!

You will then need to go to the Windows control panel and click on the 'Data Sources (ODBC)' item, which in Windows 2000 is hidden under 'Administrative Tools'. There, in the User DSN tab, click on 'Add', select OCELOT[7], and after selecting 'Finish', type PAIN04 into the Data Source Name field, and click on OK.

In addition you will need to create a directory to contain the `pain.pl` file. Within that directory, create a subdirectory called `data`. Download the following files from our website,[8] and move them to the `data` subdirectory — now you're away!

- clearmenus.sql
- CLEARPATIENTS.sql
- constants.const
- KILL.sql
- MENUS.sql
- META.sql
- PEOPLE.sql
- SETUP.sql

2.1 Packages

Pretty trivial stuff:

```
#!/usr/local/bin/perl -w
use strict;
use Tk;
require Tk::Dialog;
```

[6] Or you might wish to modify things youself to run under e.g. Linux/PostgreSQL, if you're a whiz kid!

[7] If it's not there then you forgot to install it as an ODBC driver!

[8] You can obtained them zipped at http://www.anaesthetist.com/analgesia/data/data.zip

```
require Tk::Toplevel;
require Tk::Font;
use Date::Calc qw(:all);
```

We use the Tk package (and specific components) extensively. We also use the Date::Calc package within a few minor routines.

2.2 Housework

The following cumbersome but necessary code is used to:

1. Open up a log file, used to log errors, and so forth;
2. Load a few ’constant’ values from a separate file;
3. Establish a few important variables
4. Check the endian state of this machine

Let’s look at each of these in turn:

2.2.1 The log file

```
my $logfile;
$logfile="EDLOG.LOG";
open LOGFILE, ">$logfile" or die
     "*CRASH* Could not open LOG $logfile :$!\n";
```

Trivial. Open up a file to write errors and stuff to. Call it `EDLOG.LOG`.

2.2.2 Constants

```
my %CONST;
&LoadConstants();
my $WARNINGTHRESHOLD = $CONST{’WARNABOVE’};
my $WARNCOUNT=0;
```

An associative array of constants, almost the best I can do with Perl (About the only area where C is conspicuously better than Perl is with constants). I used to place emphasis on the WARNing variables in the above, but these should be de-emphasised. The actual LoadConstants routine is discussed below (Section 4.1).

The file is `constants.const` within the data subdirectory. We open it, and parse each line until we encounter a line with a star as the first character.[9]

[9] Not a good idea to leave out this line.

In my usual fashion, I balance parentheses vertically, and use odd `while` constructs in preference to other loops. We associate constant names with values using the CONST associative array. The alert function simply provides an alert on the screen, with an OK button. It's discussed below in section 4.2.1

2.2.3 A few noteworthy variables

```
my %LOCALNAMES;
my %KEPTLOCALNAMES;     # temp copy of %LOCALNAMES
my @LOCALARRAY;
my $LOCAL;
my $KEPTLOCAL;          # and store of index

my %IAM;
my %KEPTIAMS;
my @INSCRIPT;
my $FRED;
my $SQLOK;
my $ROOTFONT;
my $METASTORE = 0;

$FRED = '1';
$SQLOK = 0;    # clumsy global hack
```

Each menu in our program has associated local variables, stored in LOCALARRAY which we create here. LOCAL is an index into this array; LOCALNAMES an associative array of names. We can keep temporary copies (ugh) of both in KEPTLOCALNAMES and KEPTLOCAL.

IAM, KEPTIAMS and INSCRIPT are very experimental, used for communication between items in a menu. Can largely be ignored. The remaining ugly variables are simply convenience variables, with the *really important* one being SQLOK, used to determine whether a recent SQL statement succeeded or failed! METASTORE determines whether, as we create tables, so we record information about them in 'meta tables'![10]

2.2.4 Endian state

It's nice to know whether we're on a big or little endian machine. Here's a test (although we don't use it much)! We even print the result to the console.

```
my $BIGENDIAN;
if ( unpack ("h*", pack("s2", 1, 2)) == 10002000 )
```

[10] By the way, attempts at such storage would be singularly silly while we were creating the meta-tables themselves.

```
   { $BIGENDIAN = 0;   # on WIN-DOwS system, should be zero.
     print "\n Little-endian";
   } else
   { $BIGENDIAN = 1;
     print "\n Big-endian";
   };
```

2.3 ODBC connection

Here's another Perl package we use — Philip Roth's ODBC. See how we use our first constant — the database name is specified as e.g. PAIN04 in the file `constants.const` referred to above.

```
my $myODBC;
  use Win32::ODBC;
  $myODBC = new Win32::ODBC($CONST{'DATABASE'});
  unless ($myODBC->Connection)
    { die "*CRASH* Failed to connect. Dearie me!\n";
    };
  print "\n ODBC: connection worked\n";
```

2.4 A few frills

Before we move on to creating the main window onscreen, we set up debugging, and a few important variables.

```
my $DEBUG;  # for ODBC debugging
   $DEBUG = 0;
my $BUG;
   $BUG = 16;
   $myODBC->Debug($DEBUG);
my $TODAY = &GetLocalTime();
   print ("\n TODAY: $TODAY\n");
my $CURRENTUSER;
   $CURRENTUSER = 1;
```

The useful BUG variable contains bit flags, each of which triggers a different debugging behaviour. Table 1 contains the flags (and their masks).

Bit	*Mask*	*Meaning*
0	1	general
1	2	Item creation
2	4	SQL
3	8	Tk item debug!
4	16	Scripting
5	32	groups
6	64	-
7	128	-

Table 1: Bit flags for BUG variable

Ultimately we will identify the current user by their unique code; for now, we just use a default value of 1. Section 4.3.2 discusses GetLocalTime.

3 Main Window

The main window is at present a rather clumsy, large menu constructed using Tk. We will rather arbitrarily divide the (continuous) code into three sections:

3.1 geometry of main window

This section is tiny — we just set up the window MAINW. We use the rather cute Perl/tk feature that as we move the mouse around, so the widget focus follows it without any clicking!

```
my $MAINW = new MainWindow;
   $MAINW->geometry('760x500');  # dimensions
   $MAINW->geometry('+10+30');   # screen offset!
   $MAINW->title('Pain Database 2005');
   $MAINW->focusFollowsMouse;
```

3.2 variables

A whole host of variables, arrays and so forth follows:

```
my $BASEX = $CONST{'BASEX'};
my $BASEY = $CONST{'BASEY'};
my $BASEW = $CONST{'BASEW'};
my $BASEH = $CONST{'BASEH'};
my @TKITEMS;  # all active widgets (global)
my @TKVALUES; # and their values
my @VVALS;    # specific to V
my @POPVALUE; # for poptriggers
```

TKITEMS is a record of all widgets created within Tk. We also retain associated values in TKVALUES We later on introduced a specific table (VVALUES) to allow the V command access to row values from within scripts. This coding is ugly and inefficient, and should be fixed! BASEX and so on are ‘baseline’ dimensions which allow us to adjust the size and position of our PDA-like menus.

We also need a separate array for poptriggers (POPVALUE). Next, let’s look at several other important globals (eugh).

```
my $XPARAM;
my $NEWXPARAM; # 'intended XPARAM'
my @X;         # X-values for menus
my @MENUS;
my @CMDSTACK;
my $ICOUNT;
my @GROUPS;
my $TOPGROUP;  # top group number

$XPARAM    = '';
$NEWXPARAM = '';
$ICOUNT    = 0;
```

We turn to X, the *subject* of each menu. This value, passed between menus, is stored in XPARAM, and when pushed as we move from menu to menu (and back), is stored in the array simply called X. We also need an array for menu names (MENUS), a rather hideous stack on which to put commands, a menu item count (ICOUNT), and some way of grouping items, which points to items in TKITEMS. We intialise these as appropriate.

3.3 Control buttons

We distribute a number of badly-placed buttons around the main menu. The most important of these by far is the last — the MENU button which opens up our 'PDA simulation'. We also toss in a new font (ROOTFONT). This is all rather clumsily done, and needs attention.

```
my $bottomFrame = $MAINW->Frame();
my $newBut = $bottomFrame-> Button(
            -text => 'Create Database',
            -command => [ \&MakeDB] );
   $ROOTFONT = $newBut->fontCreate( 'fred',
            -family => 'Helvetica',
            -size =>9);
   $newBut->configure(-font => $ROOTFONT);
   $newBut->configure(-background => 'green');
my $killBut = $bottomFrame->Button(
            -text => 'Kill Database',
            -command => [ \&KillDB ] );
   $killBut->configure(-background => 'red');
my $quitBut = $MAINW->Button(
            -text => 'Quit',
            -command => sub{exit} );
my $menuBut = $bottomFrame-> Button(
            -text => 'Make menus',
            -command => [ \&MakeMenus ] );
my $menuClearB = $bottomFrame-> Button(
            -text => 'Clear menus',
            -command => [ \&ClearMenus ] );
my $ptBut = $bottomFrame-> Button(
            -text => 'Make people',
            -command => [ \&MakePeople ] );
```

```
my $ptClearB = $bottomFrame-> Button(
            -text => ’Clear people’,
            -command => [ \&KillPeople ] );
my $MakePDB = $bottomFrame-> Button(
            -text => ’Make PDBs’,
            -command => [ \&MakeAllPDBs, $myODBC ] );
my $menuB = $bottomFrame->Button(
            -text => ’MENU’,
            -command => [ \&GoMenu, $myODBC, ’MAIN’, $MAINW]  );

   $newBut->configure(-width => 20);
   $killBut->configure(-width => 20);
   $menuB->configure(-width => 20);
   $menuB->pack(-side => ’left’,
                -expand => 1,
                -ipady => 10,
                -pady => 3);
   $killBut->pack(-side => ’left’,
                  -expand => 1);
   $newBut->pack(-side => ’left’,
                 -expand => 1);
   $menuBut->pack();
   $menuClearB->pack();
   $ptBut->pack();
   $ptClearB->pack();
   $MakePDB->pack();
   $quitBut->pack();
   $bottomFrame->pack(-side => ’top’,
                      -fill => ’both’);
MainLoop;
```

The -text values in the above give an indication of their function, which is implemented by linking the buttons to the relevant commands thus:

```
-command => [ \&MakeMenus ]
```

As we’ve already said, the most important of all is:

```
 [ \&GoMenu, $myODBC, ’MAIN’, $MAINW]
```

See how we submit a handle to the database myODBC,[11] as well as the name of the first menu (’MAIN’), and of course, our current window which will be the parent of subsequent windows.

[11] Despite myODBC being global, we retain some dignity and pass it from menu to menu.

4 Utilities

These are largely trivial subroutines. Several of them use Tk to interact with the user, providing confirmation and so forth. First let’s look at such subroutines, then we’ll examine date-handling routines.

4.1 Constants

Perl is mildly retarded when it comes to constants, so we create our own associative array to allow us to use these little creatures. Here’s the routine LoadConstants, already referred to in Section 2.2.2 above.

```
sub LoadConstants
{ my ($CONSTFILE, $ok, $key, $value);
  $ok = 1;
  $CONSTFILE = "data/constants.const";
  open CONSTFILE, $CONSTFILE
    or die "*CRASH* Can’t open $CONSTFILE :$!\n";
  <CONSTFILE>;       #discard first line
  while( $ok )
      { $_ = <CONSTFILE>;
        if ( /^\*/ )  #if last line
          { $ok = 0;
          } else
          { chomp $_;
         if ( /<(.+)>=\’(.+?)\’/ ) #pull out key and value
             { $key = $1;
               $value = $2;
               $CONST{$key} = $value;  # key MUST be unique
             } else
             { print LOGFILE "Bad CONSTANT line: <$_>\n";
                &Alert($MAINW, "Bad constant <$_>. See EDLOG");
      };  }; };
  close CONSTFILE;
}
```

We can now say `$CONST{’FRED’}` to refer to a constant called FRED. The value of FRED can be defined in the file `constants.const` within the data subdirectory of the current directory.

4.2 General purpose interaction

There are several subroutines in this group. The Alert subroutine has already been mentioned above. Here it is:

4.2.1 Alert

```
sub Alert
{ my ($thisW, $msg);
     ($thisW, $msg) = @_;
    my $D = $thisW->Dialog(
                 -title => $msg,
                 -text  => "$msg",
                 -default_button => ’OK’,
                 -buttons        => [’OK’],
                       );
    $D->title(’Note..’);
    $D->Show;
}
```

This function makes use of Tk. We read the input parameters, and create a Dialog in the usual Tk manner, with `-name => value` pairs, separated by commas. I’m sure you’ll agree that after a few glances, Tk is pretty self-explanatory.

4.2.2 Print to file or console

```
sub Print
{ my ($fred);
  ($fred) = @_;
   print LOGFILE $fred; # redirect
}
```

We use this rather than the standard Perl `print`, as it permits us to flexibly redirect printing.[12] Here we redirect all Printing to the LOG file.

4.2.3 Warn

The following is a trivial and simple way of detecting aberrations and logging them. We don’t use it much any more.

```
sub Warn
{ my ($myODBC, $level, $msg);
      ($myODBC, $level, $msg) = @_;
  if ($level > $WARNINGTHRESHOLD)
     { print LOGFILE "\nWARNING ($WARNCOUNT): $msg\n";
       $WARNCOUNT ++;
     };
}
```

[12] There are probably better ways . . . sigh, there always are.

4.2.4 Confirm

We display a message in the given window, and return 1 or 0, for confirmation or not.

```
sub Confirm
{ my ($thisW, $msg);
     ($thisW, $msg) = @_;
  my $D = $thisW->Dialog(
     -title => "Confirm your choice",
     -text  => "$msg",
     -default_button => ’No’,
     -buttons        => [’No’,’Yes’]);
    $_ = $D->Show();
    if ($_ eq ’Yes’)
       { return 1;
       };
    return (0);
}
```

We display a message in the given window, and return 1 or 0, for confirmation or not.

4.2.5 Ask

```
sub Ask
{ my ($win, $title, $default);
     ($win, $title, $default) = @_;
  my ($db, $fred);
  my ($e);
  $fred = $default;
  $db = $win->DialogBox(
        -title => $title,
        -buttons => ["OK", "Cancel"]);
  $e = $db->add(’Entry’,
       -textvariable => \$fred)->pack(-padx => 50,
                                      -pady => 15,
                                      -ipadx => 5);
  my $choice = $db->Show;
  if ($choice eq "Cancel")
     { return ("");
     };
  return ($fred);
}
```

Given a window, a title and a default value, obtain a text string from the user and return this, unless the user hits the ‘Cancel’ button, whereupon we return a zero length (null) string. The ‘Entry’ we add in the above is the text field, bound to the variable fred using -textvariable. This binding allows us access to the value.

4.2.6 Choose

The following is a little more ambitious, as we create a dialogue box containing a pop-list (Optionmenu). We don’t yet use this routine in our coding, but it has potential.

```
sub Choose
{ my ($newW, $title, $lst, $myODBC, $idx);
     ($newW, $title, $lst, $myODBC, $idx) = @_;
  my ($db, $txtvar);
  my ($e);
  $txtvar = "";

  $db = $newW->DialogBox(
        -title => $title,
        -buttons => ["OK", "Cancel"] );
  $e = $db->add(’Optionmenu’,
        -textvariable => \$txtvar)->pack(-padx => 50,
                                         -pady => 15,
                                         -ipadx => 5);
  my (@optns);
  $_ = $lst;
  if ( /^\&(.+)/ )         # if begins with &, run fx!!
     { my($kept);
       $kept = $#CMDSTACK; # preserve cmd stack
       &Invoke($myODBC, $newW, $1, $idx, ’’, -1);
       @optns = @CMDSTACK[$kept..$#CMDSTACK]; #get list
       $#CMDSTACK = $kept; #restore cmd stack
     } else
     { s/\|$//;            # rid of terminal pipe
       @optns = split /\|/;
     };
  $e->addOptions(@optns);

  my $choice = $db->Show;
  if ($choice eq "Cancel")
     { return ("");
     };
  return ($txtvar);
}
```

The if..then..else decides whether we are to run a script to obtain the list, or simply take the submitted values and split it as indicated by the pipes. We still need to standardise the script invocation — best have a leading `->`, rather than just the &. We will later on examine the Invoke subroutine (Section 9.15.2).

4.3 Date handling routines

Most of the routines that formerly existed here have been removed. It will be a good idea to eventually make this section richer, with date- and time-handling routines.

4.3.1 Convert date to seconds

```
sub ConvDate
{ ($_) = @_;
  if (! /^ *(\d+)-(\d+)-(\d+) +(\d+):(\d+):(\d+)\.*\d* *$/ )
     { return -1;
     };
 return( Mktime($1,$2,$3, $4,$5,$6) );
}
```

For easy calculations, we convert all dates in format YYYY-MM-DD HH:MM:SS to a number. The number, should *only* be used for immediate calculations (e.g. finding a duration) and *not* to store a timestamp in any way. Mktime is part of the Date-Calc package. We allow a little leeway in the submitted format (M or MM, for example).

4.3.2 Local time

```
sub GetLocalTime
{ my ($sec, $min, $hour,
      $mday, $mon, $year,
      $wday, $yday, $isdst);
  ($sec, $min, $hour,
      $mday, $mon, $year,
      $wday, $yday, $isdst) = localtime(time);
  $year += 1900;      #fix y2k.
  $mon ++;            #january is zero!
  return ("$year-$mon-$mday $hour:$min:$sec");
}
```

We use the Perl function ‘localtime’ to obtain these values, returning a text datestamp.

5 SQL/database section

There are five subsections here. First we write some fairly primitive ODBC interaction (including batching of a whole SQL script file), then we build upon this functionality to create or destroy a whole database, then we explore some ‘database-populating’ code, and penultimately we introduce the ‘layer’ that allows us to script SQL flexibly. The final section is tricky — it’s about storage of table metadata. By this we mean, information about tables themselves, stored as they are created!

5.1 Perform SQL commands

5.1.1 SQL execution

Executing an SQL statement is a little cumbersome owing to the need to debug and check for errors:

```
sub DoSQL
 { $SQLOK=0; # default ’fail’
   my ($myODBC, $SQLstmt, $bugstmt);
      ($myODBC, $SQLstmt, $bugstmt) = @_;
  $_ = $bugstmt;
  if (/\*/)
     { print LOGFILE "Debug SQL $_:\n$SQLstmt\n" ;
     };
  if ($BUG & 4)
    { &Print ("\n DEBUG: SQL <$SQLstmt>");
    };
  my ($retcode);
  $retcode = ($myODBC->Sql($SQLstmt));
  if ($retcode)
     { my ($sqlErrors);
       if ($retcode < 1)
       { $sqlErrors = $myODBC->Error();
         print LOGFILE "ERROR SQL failed ($bugstmt): \
               return code ’$retcode’ \n\<\<$SQLstmt\>\>\n";
         print LOGFILE "Error message: \"$sqlErrors\"\n\n" ;
         die "*CRASH* SQL statement failed on $bugstmt!\n";
       } else
       { $sqlErrors = $myODBC->Error();
         &Warn ($myODBC, 4,
                "SQL ret code ’$retcode’ \n<<$SQLstmt>>");
         print LOGFILE "Error message: \"$sqlErrors\"\n\n" ;
         if ( $sqlErrors !˜ /\[911\].+\[1\].+\[0\]/ )
           { &Print ("\n WARNING: There was an SQL problem!\n\
                \nSQL return code ’$retcode’ \n\
                SQL STATEMENT\n:<<$SQLstmt>>\n");
             &Alert($MAINW, "SQL error. See EDLOG.LOG");
          };
       };
       $retcode = -1;
     } else
     { $retcode = 0;     # OK.
       $SQLOK = 1;
     };
 $retcode;
 }
```

We receive three arguments: the ODBC connection, the statement to be submitted, and a debugging statement. The debugging statement comes into its own if prefixed by a star (*) — it is then printed out every time the routine is entered. There is another method for debugging SQL: if a flag is set in the variable BUG, then *every* SQL statement will be printed.

The variable retcode is used to determine whether submission of the SQL succeeded or failed. This success or failure is returned, but in addition the global variable SQLOK is set or reset depending on the success or failure of the SQL. A negative value in the return code signals an error, while a positive non-zero value is simply a warning.[13]

5.1.2 Retrieving multiple values using SQL

If we wish to retrieve an array of values (rather than a single row) from an sql query, then we need a special function. Here it is:

```
sub ManySQL
{ my ($myODBC, $SQLstmt, $tag);
      ($myODBC, $SQLstmt, $tag) = @_;
  DoSQL ( $myODBC, $SQLstmt, $tag );
  my($idx, @bigarray);
  $idx=0;
  while ( $myODBC->FetchRow() )
       { $_ = $myODBC->Data();
         $bigarray[$idx] = $_;
         $idx ++;
       };
  if ($BUG & 4)
     { &Print ("\n  ===> { @bigarray }");
     };
  my($itmz);
  $itmz = $#bigarray;  # count is +1
```

[13] Well, more or less. See the coding for how we fiddle things!

```
  if ($itmz <= 0)
     { if ($itmz < 0)
          { $SQLOK = 0;   # signal failure
          } else
          { if ((length $bigarray[0]) == 0)
               { $SQLOK = 0;
                 @bigarray = ();
     };   };   };
  return( @bigarray);
 }
```

There are several issues with the above. The first is a quirk (I'm not sure whether it's ODBC, Perl or Ocelot) where multiple items in a row are inconveniently concatenated. We normally fix this by simply intercalating a pipe character, so instead of saying SELECT a, b we say SELECT a, `'|'`, b.

The second issue is more subtle. Nothing isn't the same as NULL. Now a SELECT statement can easily return no result, but if we use MAX and there's no result, then Ocelot SQL (at least) returns not 'no result', but NULL.[14] The rather complex testing at the end is to look for a null item, and if it's present, force the array to empty!

5.1.3 Fetch a script line

It's possible to read in a complete file containing SQL code, and submit the SQL via ODBC. In doing so, we will use the following small auxiliary function:

```
sub Fetchaline
{ my ($nxit);
  $nxit = 1;
  while ($nxit)
    { $_ = <SQLFILE>;
      if (! defined)
         { die ("\n *CRASH* Unexpected file end, in SQL batch");
         };
      if ( ! /^--/ )  # if not a comment
         { chomp;                   # remove cr+lf
           $nxit = 0;               # force exit
         };
      if ( /^\*/ )                  # last line?
         { return '';
         };
      if (length $_ < 1)
         { $nxit = 1;               # force continuation
```

[14] To my mind, MAX should still return no result, or fail rather than returning NULL.

```
     };  };
 return $_;                 # success
}
```

We have a tiny unconventional wrinkle, in that the last valid line of such files must begin with a star (*) character. We return a null string if the end of the file is encountered.

In more conventional SQL style, we also remove all lines which begin with the characters `--`.[15] A line without any characters in it is ignored and the next line is then fetched.

5.1.4 SQL batching

Here’s the full batching function. We open the file provided, read in statements (which might occupy several lines), and then submit them using DoSQL. The assumption is made that the SQL file name submitted has no ‘.SQL’ suffix, and that it is in the `data` subdirectory.

```
sub BatchSQL
{ my($SQLFILE, $ok, $retcode, $myODBC,
     $errorcount, $longline, $Seek, $Repl);
    ($SQLFILE, $myODBC) = @_;
  $errorcount = 0;
  $ok = 1;
  $SQLFILE = "data/$SQLFILE.SQL";
  open SQLFILE, $SQLFILE
       or die "*CRASH* Could not open File $SQLFILE :$!\n";
  $longline = ’’;

$_ = &Fetchaline;
while ( length $_ > 0 )
   { if ( ( /\;\s*$/ ) )  # terminal semicolon?
        { $_ ="$longline$_";
          # --- START META SECTION -----#
          if ($METASTORE)               #
             { $longline = $_;          #
               &StoreMeta($myODBC, $_); #
               $_ = $longline; # clumsy #
             };                         #
          # --- END META SECTION -------#
          DoSQL ($myODBC, $_, "BATCH");
          print ".";                  #indicate progress
          $longline = ’’;   # clear
        } else              # still more!
```

[15] Anywhere else on a line, these characters are *not* seen as a comment.

```
        { $longline = "$longline$_";
        };
     $_ = &Fetchaline;    # clumsy.
   }
 close SQLFILE;                             #close the file
 return ($errorcount);                      #return this value.
}
```

The above code is clumsy — far more elegant Perl might be to redefine `$/` to, for example, `";\n"`.

One tricky aspect of the above is that, while we are creating tables, we also store information about the tables themselves!! This is the meaning of the META section, which we will soon discuss below, in section 5.5.

5.1.5 SQL commit and rollback

The following transactions are clumsy. For example, *should* be able to say something along the lines of
`$myODBC->Transact("SQL_COMMIT");`
...but this doesn't work[16], so we use the ugly, frowned upon:

```
sub Commit
 { my($myODBC);
   ($myODBC) = @_;
&DoSQL ( $myODBC, "COMMIT;", "Commit SQL");
}

sub Rollback
 { my($myODBC);
   ($myODBC) = @_;
&DoSQL ( $myODBC, "ROLLBACK;", "Rollback to last commit");
}
```

The above routines have the merit of working, but little else to commend them.

5.1.6 Key generation

Autoincrementing keys are not part of core SQL, and every vendor has created a personalised way of generating such keys. As motivated elsewhere, we create our own generator table, and then script key creation. At present, we have not implemented a complex system of semaphores to permit multiple near-synchronous

[16] Look at this, our stupidity, the problem is surely with the 'constant' SQL_COMMIT?

access to the generator table without collisions.[17] Here follows our key generator code . . .

```
sub AutoKey
{ my ($myODBC, $ky);
     ($myODBC, $ky) = @_;
  if ( $ky =˜ /key/i ) # no messing with uKey!
     { die ("Bad Auto Key value");
     };
  my ($SQLstmt, $keyval);
  $SQLstmt = "SELECT u$ky FROM UIDS WHERE uKey = 1";
  $keyval = &GetSQL($myODBC, $SQLstmt, "get key value");
  $keyval ++;                     # bump.
  $SQLstmt = "UPDATE UIDS SET u$ky = $keyval WHERE uKey = 1";
  &DoSQL($myODBC, $SQLstmt, "set new key value");
  $keyval --;
  return ($keyval);
}
```

See how we submit the name of a key which is then prefixed with a `u`. This column is then accessed within the generator table called UIDS, the number currently there incremented by one, and then we return the original number fetched (not the incremented value). We forbid the key string from containing the character sequence `key` in any combination of upper and lower case. We only ever access a single row in UIDS, the one with a uKey value equal to one.

5.2 Database creation or deletion

5.2.1 Creating the database

We now use our new-found SQL batching abilities to create a database. The assumption is that there are two script files in the data subdirectory called META.SQL and SETUP.SQL. We batch these in turn.

```
sub MakeDB
{ my ($ocm);  # outcome!
  print("\n Start meta");
  $ocm = BatchSQL ("META", $myODBC);
  print ("..Meta tables made");

  $METASTORE = 1; # start using meta tables
  $ocm =  BatchSQL ("SETUP", $myODBC);
  $METASTORE = 0; # turn off again (NB)!
```

[17] Ultimately such key generation should be atomic. The tricky issues arise not when things work, but on the rare occasion where a process dies in the ‘middle’ of a transaction.

```
  $ocm = "\n Setup: $ocm";
  print ($ocm);

  # --- START HACK --- #
  my (@rooms);
  @rooms = &ManySQL ($myODBC,
     "SELECT srmID, ’,’, srmWard, ’,’, srmText FROM ROOM;",
     "get list of all room codes and names");
  my($rmdata);
  my($id, $room, $ward);
  foreach $rmdata (@rooms)
    { $rmdata =˜ /(.+),(.+),(.+)/;
      $id = $1;
      $ward = $2;
      $room = $3;
      &DoSQL ($myODBC,
        "INSERT INTO BEDSPACE (sID, sRoom, sName) \
           VALUES ($id" . "00, $id, ’$ward/$room" . "--’);",
        "create a new bedspace");
    };
  # --- END HACK --- #

  &Commit($myODBC);
  Alert( $MAINW,  "\n Database created.");
  print "\n Created!";
}
```

See how we turn off ‘meta-documentation’ of SQL database table creation while we’re making the meta files themselves. The batching of the SETUP file is self-explanatory, but what about the ‘HACK’? This section merely pulls out information about each room just created, and then creates an equivalent ‘generic’ bedspace.[18] The pattern of the room creation is based on the following SQL ‘prototypes’:

```
 # INSERT INTO ROOM (srmID, srmText)
 #   VALUES (3101, ’31/1’);
 # INSERT INTO BEDSPACE (sID, sRoom, sName)
 #   VALUES (310100, 3101, ’31/1--’);
```

See how the room ID is a compound of the ward (times 10000) plus the room (times 100) and 00 to signal a generic bedspace.

5.2.2 Kill the database

In a similar fashion to the above, this routine runs a batch file which will destroy the database. It’s only real use is in the development phase, as for obvious reasons

[18] It would be unwise for room names to contain a comma.

it is not advisable to have this functionality in a ‘production’ version.

```
sub KillDB
{
  $_ = &Confirm($MAINW,
    "DESTROY THE DATABASE? Sure?");
  if ( ! $_ )  # 1 = yes, 0 = no
     { print "\n Database NOT killed";
       return;
     };
  my ($ocm);         # outcome!
  $ocm =  BatchSQL ("KILL", $myODBC);
  $ocm = "$ocm\nKill: $ocm";
  print $ocm;
  &Commit($myODBC);
  Alert( $MAINW,  "\n Database deleted.");
  print "Killed!\n";
}
```

5.3 Menu population

In this trivial section we invoke several more SQL ‘batch files’ or scripts. These are concerned with creating the SQL tables which contain information about our menus, with populating menus with people, and with destruction of such tables (part of the development process only). The routines are trivial.

5.3.1 MakeMenus

```
sub MakeMenus
{ my ($ocm);  # outcome!
  $ocm =  BatchSQL ("MENUS", $myODBC);
  $ocm = "\n Menu creation: $ocm";
  print $ocm;
  &Commit($myODBC);
  print "\n Menus created!";
  Alert( $MAINW, "\n Menus created.");
}
```

5.3.2 ClearMenus

```
sub ClearMenus
{ my ($ocm);
  $ocm =  BatchSQL ("CLEARMENUS", $myODBC);
  $ocm = "\n Menu clearing: $ocm";
  print $ocm;
  &Commit($myODBC);
  print "\n Menus cleared!!";
  Alert( $MAINW, "\n Menus cleared!!");
}
```

5.3.3 MakePeople

```
sub MakePeople
{ my ($ocm);         # outcome!
  $ocm =  BatchSQL ("PEOPLE", $myODBC);
  $ocm = "\n Patient creation: $ocm";
  print $ocm;
  &Commit($myODBC);
  print "\n People created!";
  Alert( $MAINW,  "\n People created");
}
```

5.3.4 KillPeople

The script name (CLEARPATIENTS) is a legacy. We delete all information about everybody in the database.

```
sub KillPeople
{ my ($ocm);         # outcome!
  $ocm =  BatchSQL ("CLEARPATIENTS", $myODBC);
  $ocm = "\n Patient clearing: $ocm";
  print $ocm;
  &Commit($myODBC);
  print "\n All people cleared!!";
  Alert( $MAINW,  "\n People cleared");
}
```

5.4 Submit SQL script queries

In this section we implement the three different types of script SQL commands:

1. QUERY
2. SQLMANY
3. DOSQL

5.4.1 QUERY

QUERY (here implemented as GetSQL) only fetches the first value returned by a query. Contrast this with SQLMANY.

```
sub GetSQL
{ my ($myODBC, $SQLselect, $tagname);
      ($myODBC, $SQLselect, $tagname) = @_;
  $SQLOK = 1;
  &DoSQL ($myODBC, $SQLselect, $tagname);
  my ($newrow);
  if ( $myODBC->FetchRow() )
       { $newrow = $myODBC->Data(); #first data line
       } else
       { $SQLOK = 0;
         return (’’);
       };
  if ($BUG & 4)
     { &Print ("\n  ===> { $newrow }");
     };
 return ($newrow);
 }
```

Failure is signalled by returning a null string, as well as resetting SQLOK to zero. SQL debugging of the result is possible here, by setting a BUG flag bit.

5.4.2 SQLMANY

You might think that we could simply use ManySQL, defined above (Section 5.1.2). The problem is that when we submit pure sql from a script, we can’t expect the code to contain intercalated pipes or whatever, to address our concatenation problem (See ManySQL). So we define the following function, otherwise very similar to ManySQL, which automatically (if clumsily) intercalates the pipes.

```
sub SQLmanySQL
 { my ($myODBC, $SQLstmt, $tag);
      ($myODBC, $SQLstmt, $tag) = @_;
  my ($preamble);
  $_ = $SQLstmt;
  /SELECT( DISTINCT)* (\S+) FROM (.+)/i;
  $preamble = $1;
  $_ = $2;
  $SQLstmt = $3;
  s/,/,\’\|\’,/g;     # add in pipe!
  if (! defined $preamble) { $preamble = ""; };
  $SQLstmt = "SELECT$preamble $_ FROM $SQLstmt";
  DoSQL ( $myODBC, $SQLstmt, $tag );
  my(@bigarray);
  while ( $myODBC->FetchRow() )     #
       { $_ = $myODBC->Data();     #get data
         @bigarray = (@bigarray, split /\|/);
       };
  if ($BUG & 4)
     { &Print ("\n  ===> { @bigarray }");
     };
  my($itmz);
  $itmz = $#bigarray;
  if ($itmz <= 0)
     { if ($itmz < 0)
          { $SQLOK = 0;   # signal 'problem'
          } else
          { if ((length $bigarray[0]) == 0)
               { $SQLOK = 0;
                 @bigarray = ();
     };   };   };
  return( @bigarray);
 }
```

The regex above should probably be diligently examined for potential problems with the pipe insertion. Otherwise, comments are as for ManySQL.

5.4.3 DOSQL

Implementation of this command is straightforward, as we've already defined it as DoSQL above (Section 5.1.1).

5.5 Storage of table metadata

This is sneaky. For relevant tables we actually look at the SQL in a simple fashion, and then record information from this processing in SQL tables! This trick is invaluable when we wish to make our PDA database. In the following we scan for:
`CREATE TABLE tablename ( item , ... , item )`
which we then parse. Otherwise, we simply ignore everything and return.

```
sub StoreMeta
{ my ($myODBC);
     ($myODBC, $_) = @_;
  if (! /\s*CREATE\s+TABLE\s+(\w+)\s*\((.+)\)/i )
     { return;
     };
  my ($tblname, $mor, $id);
  $tblname = $1;
  $mor = $2;
  print LOGFILE  "\n\n Table name: <$tblname>";
  $id = &AutoKey($myODBC, 'xTable');  # make key
  DoSQL ($myODBC,
    "INSERT INTO xTABLE (xTaKey, xTaName) \
       VALUES ($id, ’$tblname’);",
    "Document table");
  $_ = $mor;
  s/\((\d+),*(\d*)\)/\[$1:$2\]/g; # replace (n,m) with [n:m]
  s/\s+/ /g; # fix whitespace repeats
  s/\(\s*(\w+)\s*,\s*(\w+)\s*\)/($1!comma!$2)/ig;
  my ($rest);
  while ( /\s*(\w+\s+\w+[ˆ,]*),(.+)/ )
    { $rest = $2;
      ParseCreateTable($myODBC, $id, $1);
      $_ = $rest;      # move to next item
    };
  ParseCreateTable($myODBC, $id, $_);
}
```

Note that we don’t cover absolutely every option, for example, compound primary keys will muck things up,[19] and obviously we encounter problems with data types we don’t support. Here’s the subsidiary but important ParseCreateTable routine. It’s clunky and too long:

```
sub ParseCreateTable
{     my($myODBC, $tblkey);
      ($myODBC, $tblkey, $_) = @_;
      print LOGFILE "\n-->$_";
  my ($name, $type);
  my ($col, $tbl);
  my ($len, $prec);
  my ($t); # single char type
  my ($chk);
  my ($lid, $licol, $litbl);
  my ($default);
  if (/\s*constraint\s+(.+?)\s+(.+)/i)
     { $name = $1;
       $_ = $2; # PRIMARY KEY, FOREIGN KEY or CHECK:
       if ( /ˆcheck\s+\((.+)\)/i )
          { $_ = $1;
            if ( /([ˆ\s]+)\s+is not null\s*/i ) # ugly
               { $col = $1;
                 print LOGFILE
                   "\n   >CHECK !0: Name:$name Col:$col";
                 $chk = ’X’;
               }
            elsif ( /([ˆ\s]+)\s+is null\s*/i )
               { $col = $1;
                 print LOGFILE
```

[19] One of our idiosyncratic conventions is that we don’t allow these, anyway

```
         "\n   >CHECK =0: Name:$name Col:$col";
       $chk = ’N’;
     } else
     { print "\n UNSUPPORTED CHECK: $_";
     };
  $licol = &GetSQL ($myODBC,
    "SELECT xCoKey from xCOLUMN WHERE xCoName = ’$col’ \
      AND xCoTable = $tblkey;",
    "identify relevant column");
  if ($licol > 0) # fail if not found
  { $lid = &AutoKey ($myODBC, ’xLimit’);
    &DoSQL ($myODBC,
     "INSERT INTO xLIMIT (xLiKey, xLiName, xLiType, xLiColumn) \
                  VALUES ($lid,   ’$name’, ’$chk’, $licol);",
     "record CHECK constraint"); # ’N’= is null, ’X’ = not
     print ("+");
   } else
   { print ("\n ERR: Column not found: <$col>");
   };
} elsif (/ˆprimary\s+key\s+\((.+)\)/i)
{ $col = $1;
  print LOGFILE  "\n   >1ARY: Name: $name Col:$col";
  $licol = &GetSQL ($myODBC,
    "SELECT xCoKey from xCOLUMN WHERE xCoName = ’$col’ \
      AND xCoTable = $tblkey;",
    "identify relevant column");
  if ($licol > 0)
  { $lid = &AutoKey ($myODBC, ’xLimit’);
    &DoSQL ($myODBC,
     "INSERT INTO xLIMIT (xLiKey, xLiName, xLiType, xLiColumn) \
                  VALUES ($lid,   ’$name’, ’P’, $licol);",
         "record primary key constraint");  # ’P’ = 1ary key
     &DoSQL ($myODBC,
      "UPDATE xCOLUMN SET xCoType = ’I’ WHERE xCoKey = $licol;",
      "adjust key type"); # our 1ary keys all type I (!!)
     print ("+");
   } else
   { print ("\n ERR: Column not found: <$col>");
   };
} elsif (/ˆforeign\s+key\s+\((.+)\)\s+references\s+(.+)/i)
{ $col = $1;
  $tbl = $2;
  print LOGFILE
    "\n   >FOREIGN: Name:$name Col:$col Table:$tbl";
  $licol = &GetSQL ($myODBC,
    "SELECT xCoKey from xCOLUMN \
      WHERE xCoName = ’$col’ \
      AND xCoTable = $tblkey;",
         "identify stated column");
       $litbl = &GetSQL ($myODBC,
         "SELECT xTaKey from xTABLE WHERE xTaName = ’$tbl’;",
         "identify table reference");
       if (($licol > 0) && ($litbl > 0)) # fail if not found
       { $lid = &AutoKey ($myODBC, ’xLimit’);
         &DoSQL ($myODBC,
          "INSERT INTO xLIMIT \
             (xLiKey, xLiName, xLiType, xLiColumn, xLiTable) \
             VALUES ($lid,   ’$name’, ’F’, $licol, $litbl);",
          "record foreign key constraint"); # ’F’ = foreign key
          &DoSQL ($myODBC,
           "UPDATE xCOLUMN SET xCoType = ’I’ \
              WHERE xCoKey = $licol;",
           "adjust key type");
          print ("+");
        } else
        { print ("\n ERR: Column not found: <$col>");
        };
     } else
     { print "\n UNKNOWN: <$_>";
     };
} else
{ /\s*([ˆ\s]+)\s+([ˆ\s\[]+)(.*)/;
  $name = $1;
  $type = $2;
  $_ = $3;
  if (/\[(.+)\:(.*)\]/)
     { $len = $1;
       $prec = $2;
     } else
     { $len = "??";
       $prec = "??";
     };
  $default = 0;
  if ( /default\s+(.+)/ )
     { $default = $1;
       print LOGFILE ("\n  DEFAULT: $default");
     };
  if (length $prec < 1)
     { $prec = 0;
     };
  if ($type =˜ /ˆfloat$/i)
     { $len = 8;
       $prec = 0;
       $t = ’F’;
     }
  elsif ($type =˜ /ˆdate$/i)
```

```
          { $len = 8;
            $prec = 0;
            $t = ’D’;
          }
       elsif ($type =˜ /ˆtime$/i)
          { $len = 12;              # ours
            $prec = 6;              #
            $t = ’T’;
          }
       elsif ($type =˜ /ˆtimestamp$/i)
          { $len = 20;
            $prec = 6;             # ???
            $t = ’S’;
          }
       elsif ($type =˜ /ˆvarchar$/i)  # ? CHARACTER VARYING?
          { $t = ’V’;
          }
       elsif ($type =˜ /ˆdecimal$/i)
          { $t = ’N’;
          } else
          { print "\n BAD TYPE: ’$type’ forced to V";
            $t = ’V’;
          };
      print LOGFILE  "\n  > Name:$name Type:$type Len:$len Pr:$prec";
      # MUST still fix: sort out DEFAULT ...
      my($id);
      $id = &AutoKey($myODBC, ’xColumn’);
      DoSQL ($myODBC,
        "INSERT INTO xCOLUMN  \
          (xCoKey, xCoName, xCoType, xCoSize, xCoScale, xCoTable)\
          VALUES ($id, ’$name’,    ’$t’,    $len,    $prec,    $tblkey)",
        "record column");
      if ($default)
         { &DoSQL ($myODBC,
           "UPDATE xCOLUMN SET xCoDefault = ’$default’ \
             WHERE xCoKey = $id",
           "fix default");
         };
      print ("+");
     };
};
```

The whole of the above could profitably be rewritten, devolving parts to simpler subsidiary functions. Someday. See how we print useful debugging information to LOGFILE.

6 Menu creation

6.1 GoMenu

The principal routine. Cumbersome and badly written. We will break it up into bite-sized chunks as follows:

6.1.1 Entering GoMenu

GoMenu accepts a handle on the database, the name of a menu (menu1), and a Tk window (newW). If the submitted ’name’ is numeric, GoMenu uses the number to pop the stack of menus appropriately, going *back* and enabling the relevant previous menu. Otherwise it just queries the database, finds menu components and displays them in a newly created menu. X, the menu subject, is pushed or popped appropriately.

```
sub GoMenu
{ my ($myODBC, $menu1, $newW);
     ($myODBC, $menu1, $newW)=@_;
  if ($BUG & 8) { &Print ("\n\n NEW MENU <$menu1> "); };
  if ($menu1 eq ’MAIN’) # if at start
     { $MAINW->geometry(’40x20’);  # make MAINW tiny
       $MAINW->geometry(’+600+0’); # move it.
     };
  my ($menuname);
  $menuname = $menu1;
  @CMDSTACK = (); # destroy stack!!
```

The above introductory code allows for debugging (using a bit flag in BUG), and makes MAINW really tiny, if at the start. See how we delete everything on CMDSTACK, which constrains us to really tight, modular menu coding!

6.1.2 Should we pop?

Here we address the issue of a numeric ‘menu name’, actually a command to go back to a prior menu.[20] MENU(1) takes us back one menu, MENU(2) takes us back two, and so on. The popped menus are discarded completely. MENU(0) *reloads* the current menu!

[20] Initially we had this as a negative number, but, mainly for reasons of PDA program design, we have now revised this convention. From now on, the use of negative numbers is deprecated (but, for now, tolerated). If you want to go back one menu, you submit just one, not negative one.

```
$_ = $menuname;
if ( /ˆ-*(\d+)$/ )  # if numeric, usually 1
   { my ($mc);
     $mc = $1;      # trim off -ve
     $mc ++;
     while ($mc > 0)
       { $menuname = pop (@MENUS);
         $NEWXPARAM = pop (@X);
         $mc --;
       };
   }
elsif ( /ˆ0$/ )  # MENU(0) reloads!
   { $menuname = pop (@MENUS);
     $NEWXPARAM = pop (@X);
   };
```

6.1.3 Clear local variables?!

Here we call two subsidiary routines, first storing away the local name values (for possible later use), then clearing them.[21] And that’s the whole of this tiny section.

```
&KeepLocalNames();
&ClearLocalNames();
```

6.1.4 Run associated script

If the menu has an associated initialisation script, now’s the time to run it:

```
my($r);
$r = 0; # default ok
$_ = &GetSQL ($myODBC,
   "SELECT iInitial FROM ITEM WHERE iName = ’$menuname’",
   "get menu startup script");
if (length $_ > 1)
   { $XPARAM = $NEWXPARAM;
     if ($BUG == 16)
        { &Print ("\n DEBUG: MENU INI SCRIPT <$_> \
                     menu <$menu1> ($XPARAM)");
        };
     push (@CMDSTACK, $menu1);
     $r = &RunWholeScript ($myODBC, $_, $newW, -1, -1);
     pop(@CMDSTACK);
     if ($r < 0) # if script execution failed
        {   $menuname = pop (@MENUS);
            push (@MENUS, $menuname);
```

[21] Look into moving this down a bit later? NO!

```
            $XPARAM = pop (@X);
            push (@X, $XPARAM);
            $NEWXPARAM = $XPARAM;
            &RestoreLocalNames();
            return; #fail!
        };
   };
$XPARAM = $NEWXPARAM;
```

We obtain the iInitial script if it exists[22] We then move in the new X (subject) who is waiting in the wings, push the *menu name* onto the stack, and run the full script.[23] After the script has run, we pop the stack.[24]

See how, if the script *fails*, then loading of the menu is completely aborted! In the process of ‘aborting’ we also restore the menu name and X. We even restore the local names (that’s in fact why we stored them above) and NEWXPARAM, which could conceivably have been altered by the script before it died.

Finally in this section, we again move NEWXPARAM into XPARAM. This allows the initialisation script to alter the XPARAM value!

6.1.5 Prepare to create

We next pull out basic menu parameters, after a bit of cleaning up (ClearGroups).

```
my($mTitle);
my($mCode);
&ClearGroups();
$_ = &GetSQL($myODBC, "SELECT iID, ’|’, iText FROM ITEM \
  WHERE iType = 20 AND iName = ’$menuname’",
  "Get menu ID");
/(.+)\|(.+)/;
$mTitle = $2;
$mCode = $1;
my ($mX, $mY, $mW, $mH);
$_ = &GetSQL($myODBC,
    "SELECT miX, ’,’, miY, ’,’, miW, ’,’, miH FROM MENUITEMS \
      WHERE miMenu = $mCode AND miItem = $mCode;",
    "get menu parameters");
/(.+?),(.+?),(.+?),(.+)/;
$mX = $1;
```

[22] There is a potential problem if we have been unwise enough to create two menus with the same iName (which we haven’t forbidden in the database design). Also note that a single-character script will fail.

[23] Pushing the name allows the script to know which menu it’s in!

[24] At present, we don’t check that the menu name is still on the stack. We might enforce this constraint!

```
  $mY = $2;
  $mW = $3;
  $mH = $4;
$mX += $BASEX;
$mY += $BASEY;
$mW *= $BASEW;
$mH *= $BASEH;
$mX = int($mX);
$mY = int($mY); # pixels
$mW = int($mW);
$mH = int($mH);
my ($oldW);
$oldW = $newW;
```

We identify the menu coordinates, width and height within MENUITEMS using our peculiar convention that a 'self-referential' menu item is actually a note about these parameters!

We adjust the coordinates and dimensions using the BASE values, then converting these adjusted floating point values to pixel values.

We also keep a record of the previous window. Later, we will close the previous window, unless of course it was the main window.

6.1.6 Destroy old widgets

Initially I went through a laborious process of disabling the bindings on widgets in the old menu. It's far easier to simply destroy the lot of them. We won't need them, as we will eventually close their parent, only re-creating everything if we come back to the parent window.[25]

```
  while ($ICOUNT > 0)
    { $ICOUNT --;
      $TKITEMS[$ICOUNT]->destroy;
    };
```

6.1.7 Do it!

At last, we can actually create the menu, and all of its components. The following section calls upon the clumsy SubMenu function to do its dirty work in creating menu components.

```
  $newW = $MAINW->Toplevel();  # create new menu
  $newW->geometry("+$mX+$mY");
  $newW->geometry("$mW" . "x$mH");
```

[25] We only delete the parent later, to prevent an irritating flicker.

```
  $newW->title($mTitle);
  push (@MENUS, $menuname);
  push (@X, $XPARAM);

  $ICOUNT = &SubMenu($myODBC, $mCode, $newW, 0, 0, 0);
  &FixGroups($TOPGROUP);
```

When we create the new menu, we also push X and the menu name. This action allows easy retrieval of these vital parameters, simplifying refreshing of a menu using MENU(0).

6.1.8 Close prior window

```
 if (! ($oldW eq $MAINW) )
        { &ExitWindow ($oldW);
        };
  $newW->focus; # set focus to this window
}
```

Not only do we destroy the old window, we also force the focus to the new one. A wise idea.

6.2 Subsidiary functions

This section is taken up by the monstrous SubMenu, which we'll have to break up into chunks and plod through.

6.2.1 SubMenu

SubMenu accepts a handle on the database, the unique ID of the menu itself (mCode), the relevant window (newW), an X and Y displacement within the current menu used for offsetting components, and the final parameter `i`, which is an index into the global TKITEMS. When first invoked by GoMenu, the last three parameters are all zero.

```
sub SubMenu
{ my ($myODBC, $mCode, $newW, $X0, $Y0, $i);
     ($myODBC, $mCode, $newW, $X0, $Y0, $i) =@_;
  my($j); # record item creation success
```

6.2.2 Find all items

Next we retrieve an array of number pairs. Each element of the array is made up of the id of the ITEM itself, and the unique ID of the row in MENUITEMS describing the item, separated by a comma.[26]

```
my(@items);
  (@items) = &ManySQL ($myODBC,
          "SELECT miItem, ’,’, miUid from MENUITEMS \
            WHERE miMenu = $mCode \
            AND miItem <> $mCode \
            ORDER BY miGroup, miOrder",
          "get items for this menu");
my($item, $miX, $miY, $miW, $miH, $miGroup,
   $iInitial, $iResponse, $iScript, $miEnabled);
my($iType, $iText, $iList, $iLines);
my($both, $iu);
my($miInitial);
```

We exclude the case where miItem is equal to mCode, as this row refers to the menu itself. We also group the items, and order them in ascending order. In the last part of this section we define a whole bunch of variables used in the next one.

6.2.3 Create items one by one

We use the item array to create each item in turn. The `foreach` statement is bulky and offensive, so we’ve broken it up:

```
foreach $both (@items)
  { $both =˜ /(.+),(.+)/;
    $item = $1;
    $iu   = $2;
    $_ = &GetSQL ($myODBC,
          "SELECT miX, ’,’, miY, ’,’, miW, ’,’, miH, ’,’, \
            miGroup, ’,’, miEnabled, ’,’, miInitial \
            from MENUITEMS \
            WHERE miMenu = $mCode AND miItem = $item",
          "get local item attributes");
    if (! /(.+?),(.+?),(.+?),(.+?),(.+?),(.),(.*)/ )
      { &Alert ($MAINW, "Menu parameter error: <$_>");
      };
    $miX = $1;
    $miY = $2;
    $miW = $3;
    $miH = $4;
```

[26] Hey, let’s read this line again :-)

```
$miGroup = $5;
$miEnabled = $6;
$miInitial = $7;
$_ = &GetSQL ($myODBC,
  "SELECT iType, ’@|’, iText, ’@|’, iList, ’@|’, iLines, \
      ’@|’, iInitial, ’@|’, iResponse, ’@|’, iScript \
      from ITEM WHERE iId = $item",
  "get general attribs of this item");
if (! /(.+?)\@\|(.*?)\@\|(.*?)\@\|(.*)\@\|(.*?)\@\|(.*?)\@\|(.*)/ )
   { &Alert($MAINW, "Bad item details, <$_>");
   };
$iType = $1;
$iText = $2;
$iList = $3;
$iLines = $4;
$iInitial = $5;
$iResponse = $6;
$iScript = $7;
if (length $miInitial > 0)  # OVERRIDE
   { $iInitial = $miInitial;
   };
```

Although long, the above is fairly straightforward. We are simply fetching item parameters at this stage. The only part I’m really dysphoric about is the last `if` statement, where we override the item script with the miInitial value.

6.2.4 Identify a monomorphic table

Here we invoke Make1Table if we have a monomorphic table (type 9):

```
if ($iType == 9) # monomorphic
   { $i += &Make1Table ($myODBC, $i, $newW,
           $item, $iLines,
           $miX+$X0, $miY+$Y0, $miW, $miH,
           $iInitial, $iResponse);
   }
```

There is scope for improvement of monomorphic tables. Should we, for example have two variants, one which fits into a certain number of lines, another which is of invariant size?[27] See how we make (x, y) relative to X0 and Y0 — this is a theme throughout the following code.

[27] Another question is whether we should submit iScript to Make1Table.

6.2.5 Sub-menu

Next, is it a menu contained within a menu? If so, we not only have to create that menu as a component within the current one, but also run the associated script! Tricky.

```
elsif ($iType == 20) # a (sub)menu!
   { my($scrp, $snam, $r);
     $r = 1; # default to ’ok’
     $scrp = $iInitial;
     $snam = &GetSQL ($myODBC, "SELECT iName FROM ITEM \
                         WHERE iId = $item",
                      "get menu name");
     if (length $scrp > 1)
        { if ($BUG == 16)
             { &Print ("\n DEBUG: Submenu ini script: <$_>");
             };
          push (@CMDSTACK, $snam); # push name
          $r = &RunWholeScript ($myODBC, $scrp, $newW,
                                -1, -1);
          pop(@CMDSTACK);
        };
     if ($r != 0)  # [unless stopped]
        { $i = &SubMenu($myODBC, $item,
                 $newW, $miX+$X0, $miY+$Y0, $i);
   };   }
```

There are wrinkles. We examine the value `r` returned by RunWholeScript. If it’s zero (’STOP’) then we do *not* recursively invoke SubMenu.[28] Otherwise, we do. Clearly to be used with caution.

See how, if we invoke SubMenu we submit `i`, the current item count, and receive an updated copy. Remember that this item count indexes into TKITEMS. We do *not* pass the current height and width as we do *not* scale items within the sub-menu, relative to the current one!

As usual, we have debugging based upon bit flags within BUG. As before, running the script is preceded by a push of the menu name, and we pop the value back off the stack without checking its validity.[29]

6.2.6 Identify a polymorphic table

If a polymorphic table (type 8), invoke MakeTable. Simple, on the face of it.

[28] Previously, we failed on ’FAIL’ but this was too catastrophic, so we now ’STOP’ to prevent submenu creation.

[29] Might be wise to revise this approach, see similar note above!

```
      elsif ($iType == 8) # poly table
         { $i = &MakeTable ($myODBC,  $i, $newW, $item, $iLines,
                    $miX+$X0, $miY+$Y0, $miW, $miH,
                    $iInitial, $iResponse, $iText);
         }
```

All table creation needs extensive revision (See below).

6.2.7 Just make an item!

In the following we simply create one item that is *not* a menu or table. The CreateOneItem routine is another large chunk of code. The variable `j` is used to test for success or failure of this routine.

```
   else  { $j = &CreateOneItem ($myODBC,  $i, $newW,
               $miX+$X0, $miY+$Y0, $miW, $miH, $miGroup,
               $iInitial, $iResponse, $iScript, $iType,
               $iText, $iList, $iLines, $miEnabled,
               $iu, $iText, '');
           if ($j)  # 0 signals failure of item creation.
              { $GROUPS[$i] = - ($miGroup);
                if (($BUG & 32) && ($miGroup > 0))
                   {&Print (" grp($i) '$iText'->$miGroup ");
                   };
                if ($miGroup > $TOPGROUP)
                   { $TOPGROUP = $miGroup;
                   };
                $i ++;
    };   };   };
 return $i;
}
```

If we managed to create an item, then we bump the variable `i` by one to move to a new index in TKITEMS.

The *grouping* is quite quaint. We store a *negative* value in the GROUPS array for the current item. At the *end* of everything we invoke FixGroups, which then creates links all related items in a single group together, daisy-chaining each item to the next one within that group. The reason why we store a negative value now becomes apparent — when we have resolved an item, it will be clearly identified as such by having a *positive* value associated with it. The positive value is the index of the next item in the chain!

At the end of everything we return the value of `i`, because we must do so in the recursive invocation of SubMenu. And that's it for SubMenu. Whew!

6.2.8 ExitWindow

The following is an atavistic remnant, left over from a much larger precursor routine. Can probably be disposed of.

```
sub ExitWindow
{  my($thisW);
   ($thisW) = @_;
   $thisW->destroy(); #
}
```

6.3 Grouping

We introduced the idea of grouping above in section 6.2.7. Let's flesh this concept out and examine the coding. It's important to us to be able to group menu items together. For example, we need to be able to group pushbuttons, so that if we click on one, the others are all cleared.

The logical way to group items is to make each item in a group refer to the next in that group, in a cyclical fashion. The array GROUPS stores these references. We have only two grouping functions, the trivial ClearGroups, and the more involved FixGroups which associates grouped items within GROUPS.

6.3.1 ClearGroups

This simply clears the GROUPS array. Later, as we define new items, the magic of Perl creates the elements.

```
sub ClearGroups
{ $TOPGROUP = 0;
  @GROUPS = ();
  return;
}
```

6.3.2 FixGroups

More challenging. We work through GROUPS, associating like items. The routine is currently slow, $O(n^2)$, but optimisation should be fairly easy.[30] At present we submit the highest group number as the sole argument of FixGroups. The usage of the global TOPGROUP is rather clumsy.

```
sub FixGroups
{ my($maxgrp);
```

[30] Helped by prior sorting by group.

```
  ($maxgrp) = @_;  # submit highest group index
  if ($BUG & 32)
     { &Print ("\n Debug: top group is $maxgrp");
     };
  my ($max);
  $max = $#GROUPS;  # get index of last group item
  my($g, $prev, $frst, $i);
  $g = 1;
  while ($g <= $maxgrp)
    { $i = 0;
      $frst = -1;
      while ($i <= $max)
        { if (- ($GROUPS[$i]) == $g)   # if the group
             { if ($frst == -1)
                  { $frst = $i;
                    if ($BUG & 32)
                        { &Print ("\n Debug: group=$g($i");
                        };
                  } else
                  { $GROUPS[$i] = $prev;
                    if ($BUG & 32)
                       { &Print (", $i");
                  };   };
               $prev = $i;
             };
          $i ++;
        };
      if ($frst != -1)
        { $GROUPS[$frst] = $prev; # wrap
           if ($BUG & 32)
              {&Print (")");
        };    };
      $g ++;
    };
  return;
}
```

7 Tables

This section is devoted to the creation of the two types of table we employ - monomorphic, made up of identical elements (e.g. a whole lot of buttons), and the more useful polymorphic table, made up of columns of similar elements. Each element in a *row* of a polymorphic table bears a fixed relationship to a particular database ID, for example, an ID of a person.

7.1 Polymorphic tables

These are more useful than monomorphic tables. The following routine is cumbersome and badly written. For clarity, we have broken it into chunks.

7.1.1 MakeTable

The MakeTable routine is cumbersome. It accepts an ODBC handle, `idx` which is the current number of items already created[31], a Tk window newW, the ID of the table item itself (tbl), the maximum number of table rows displayed (tLines),[32] as well as (x, y), width and height values, and two scripts! The scripts are initialisation (tIni), and response (tResp) scripts. Finally, there is default text (tDefault), to be used if no rows can be found for this table.

```
sub MakeTable
{  my ($myODBC, $idx, $newW, $tbl, $tLines, $tX, $tY, $tW, $tH,
        $tIni, $tResp, $tDefault);
      ($myODBC, $idx, $newW, $tbl, $tLines, $tX, $tY, $tW, $tH,
        $tIni, $tResp, $tDefault) = @_;
   my ($j);     # flag for item creation
```

7.1.2 Identify columns

First we find the columns.

```
   my (@columns);
      (@columns) = &ManySQL ($myODBC,
        "SELECT irItem FROM ICOLTABLE \
          WHERE irTBL = $tbl ORDER BY irOrder;",
        "get all columns");
   if ($BUG & 2)
      { &Print ("\n DEBUG: COL CODES <@columns>");
      };
```

[31] This value is returned after being incremented by MakeTable.
[32] This number includes the header line!

7.1.3 Get row V values

We run the initialisation script to obtain unique values, one per row.

```
my (@rows);
my ($rowcount);
  @CMDSTACK = ();
  &RunWholeScript ($myODBC, $tIni, $newW, -1, -1);
  @rows = @CMDSTACK; # whole!
  $rowcount = 1 + $#rows;
  if ($BUG & 2)
     { &Print ("\n DEBUG: items <@rows>, row count $rowcount");
     };
  if ($rowcount >  ($tLines-1))
     { $rowcount = ($tLines-1);
     };
  @CMDSTACK = (); # clear it
```

If there are more rows than can be displayed (rowcount is over the number of lines tLines, less 1 for the top line), then we limit the row count.

We should probably truncate the `rows` array (cutting the length to rowcount) but don’t do this at present.

7.1.4

Next, we prepare a few variables (the `rowcopy` array is used below to allocate values *within* a column). The pixel height of a row is `iH`.

```
my (@rowcopy);
my ($itmcnt);         # row count down
my ($col);
my ($iX, $iY, $iW, $iH, $irEnabled);
$iH = $tH/($tLines); # row height
$iX = $tX;
if ($rowcount == 0) # no rows
         { $j = &CreateOneItem ($myODBC, $idx, $newW,
           $iX, ($tY+$iH), $tW, $iH, 0, ’’, ’’, ’’,
           1, $tDefault, ’’, 1, 0, 0, $tDefault, ’’);
           if ($j)
              { $GROUPS[$idx] = 0; # headers are not grouped.
                $idx ++;
              };
         };
```

If there are no data rows, then we insert the default message instead, as a text label.

7.1.5 Get column details

Now, for each column we will find its details.

```
foreach $col (@columns)  # for each column
    { @rowcopy = @rows;
      $itmcnt = $rowcount;
      $iY = $tY;   # start at top of column

      # extract column details:
      my ($iType, $iList, $iInitial, $iResponse, $iScript);
      my($iText, $oText);
      $_ = &GetSQL ($myODBC,
            "SELECT iType, ’@|’, iText, ’@|’, iList, \
              ’@|’, iInitial, ’@|’, iResponse, ’@|’, iScript \
              from ITEM WHERE iId = $col",
            "get attribs of COLUMN");
/(.+?)\@\|(.*?)\@\|(.*?)\@\|(.*?)\@\|(.*?)\@\|(.*)/; # [1]
      $iType = $1;
      $iText = $2;
      $oText = $iText;
      $iList = $3;
      $iInitial = $4;
      $iResponse = $5;
      $iScript = $6;

      my($irPaper);
      $_ = &GetSQL ($myODBC,
         "SELECT irFraction, ’,’, \
           irName, ’,’, irEnabled, ’,’, irPaper \
           FROM ICOLTABLE \
           WHERE irTBL = $tbl AND irItem = $col;",
         "get fractional width of item");
      /(.+?),(.*),(.),(.*)/;
      $iW = $1 * $tW;  # pixel width
      $irEnabled = $3;
      $irPaper = $4;
      $_ = $2;  # irName
      if (/.+/) # if column name exists
         { $iText = $_; # overwrite
         };
```

There are several points of interest in the above. In the line labelled `[1]` we separate items with `@|` rather than using a simple pipe, as we also use pipes within iList components, which would cause confusion. See section 5.1.2 for a discussion of why we use pipes, anyway.

The Perl in this kludgey old routine is poor at best. The variable `oText` is used below by CreateOneItem, but only has meaning for pushbutton creation.

7.1.6 Make column header

We now create a column header on screen (a label, i.e. a type 1 item).

```
$j = CreateOneItem ($myODBC, $idx, $newW,
     $iX, $iY, $iW, $iH, 0, '', '', '',
     1, $iText, '', 1, 0, 0, $iText, '');
if ($j)
   { $GROUPS[$idx] = 0; # not grouped
     $iY += $iH; # move down to
     $idx++;     # draw next item
   };
```

There is no response script for the label; at some stage we might consider having a response, so that, for example, clicking will sort the table by the column values in ascending or descending order!

The `if ($j)` section makes sure that headers aren't grouped (have a zero group value).

7.1.7

We next make the rest of the column, item by item. This creation involves determining the value for *each* item using an SQL query — CreateOneItem will perform the invocation if iInitial is not null.

```
my ($row, $c);  # c is row count
$c = 0;         # [2]
foreach $row (@rowcopy)
  { if ( $itmcnt > 0 )   # if more lines
       { $itmcnt --;
         $c ++;
         push (@CMDSTACK, $row);
         my ($iv); # HIDEOUS!
         $iv = $row;
         $VVALS[$idx] = $row;
         $j = &CreateOneItem ($myODBC, $idx, $newW,
                $iX, $iY, $iW, $iH,
                0, $iInitial, $iResponse, '',
                $iType, $iv, $iList, 1,
                $irEnabled, 0, $oText, $irPaper);
         if ($j)
            { if ($iType == 4) # pushbutton!
                 { $GROUPS[$idx] = -($c);
                   if ($BUG & 32)
                      { &Print (" grp($idx)->$c");
                      };
                   if ($c > $TOPGROUP)
```

```
                              { $TOPGROUP = $c;
                              };
                         } else
                         { $GROUPS[$idx] = 0;
                         };
                      $idx ++;
                    };
                 $iY += $iH; # down to draw next
          };   };
        $iX += $iW; #right to next column
      };
return ($idx);
}
```

There are a few clumsy hacks: the variable `iv` becomes the row parameter of CreateOneItem, and we have to initialise the `V` array (VVALS) because a script may reference V. We also push the row onto the stack, as this too may be used by the initialisation script! We also have to group pushbuttons! For each item created we also bump `idx`. The line marked `[2]` is a problem.[33]

All in all, a whole lot of low-grade nastiness makes up this section. At the end, we simply return the new, updated item count (`idx`).

7.2 Monomorphic tables

Here we make the clumsy monomorphic table. Each button (or other component, all of the same type) has its own unique V value and response. In the initial version of this table, we had one column specifier for each column, but such an approach isn't only clumsy — it also goes against the spirit of such a table. We should have only one column, and duplicate it as needed!

[FIX ME: THIS IS STILL UNFIXED]

7.2.1 Make1Table

Make1Table accepts parameters similar to, but simpler than, MakeTable (Section 7.1.1). We similarly have an ODBC handle, item count (idx), Tk window (newW), the table ID, a line count and various coordinates and widths, as well as two scripts.

```
sub Make1Table
{  my ($myODBC, $idx, $newW, $tbl, $tLines, $tX, $tY, $tW, $tH,
        $tIni, $tResp);
      ($myODBC, $idx, $newW, $tbl, $tLines, $tX, $tY, $tW, $tH,
```

[33] What about pushbuttons? Explore.

```
       $tIni, $tResp) = @_;
   my ($i);
   $i = $idx;
   #### MUST STILL FIX THIS SECTION [$jvs] ###
   my (@columns);
      (@columns) = &ManySQL ($myODBC,
        "SELECT irItem FROM ICOLTABLE \
          WHERE irTBL = $tbl ORDER BY irOrder;",
        "get all columns");
  my($SQLstmt);
  $SQLstmt = ’’;   # initialise
  my (@rows);
  my($itmcnt);
  $itmcnt = 0;
  @CMDSTACK = (); # clear stack
```

7.2.2 [to fix]

[FIX ME: THIS IS STILL UNFIXED]

```
    &RunWholeScript ($myODBC, $tIni, $newW, -1, -1);
    @rows = @CMDSTACK;
    push (@rows, -1); # [3]
    $itmcnt = $#rows; # [should this be +1 ???]
```

The RunWholeScript invocation above is vulnerable if the script does nasty things to the stack.[34]

The line labelled `[3]` above illustrates a subtle flaw in the Ocelot database (it may be present elsewhere). Logically, a list is composed of zero or more items, and functions which operate on a list should accept this definition. In Ocelot, if we apply IN to lists of under 2 items, it balks. To permit identification of a single row, we push -1 onto the stack, so an IN statement will still succeed provided there is at least 1 row![35]

7.2.3 [to fix]

[FIX ME: THIS IS STILL UNFIXED]

```
# b. Get column params:
    my (@ivals);
    my ($iX, $iY, $iW, $iH, $irEnabled);
    # offset of individual item will be iX, iY, and width and height will be iW, iH:
```

[34] We should also check out the case where there is no initialising script invocation of SQL.

[35] There is no danger of this being mis-identified as a key, as we never use negative numbers as keys!

```
$iX = $tX;
$iH = $tH/$tLines; # for now, each row will have same height [& prob. forever]

my ($col);
my ($iv);
my ($ilimit); # limiting count (row count, or if monomorphic, item count)
$ilimit = $itmcnt;

my ($colw, $colcount);
```

7.2.4 [to fix]

[FIX ME: THIS IS STILL UNFIXED]

```
foreach $col (@columns)  # for each column component in the table
  {
    $iY = $tY;   # start at top of column
    $_ = &GetSQL ($myODBC,
              "SELECT irFraction, ’,’, \
               irName, ’,’, irEnabled FROM ICOLTABLE WHERE irTBL = $tbl AND irIte
              "get fractional width of item");
    /(.+?),(.*),(.)/; #pull out fraction, column name, enabled or not!
    $iW = $1 * $tW;  # convert from fractional to actual width
    $_ = $2;
    $irEnabled = $3;
    my($iText);
    $iText = ’’;
    if (/.+/) # if exists
       { $iText = $_;
       };

    my ($iType, $iList, $iInitial, $iResponse, $iScript);
    $_ = &GetSQL ($myODBC,
             "SELECT iType, ’@|’, iText, ’@|’, iList, \
               ’@|’, iInitial, ’@|’, iResponse, ’@|’, iScript \
               from ITEM WHERE iId = $col",
             "get attribs of COLUMN");
     /(.+?)\@\|(.*?)\@\|(.*?)\@\|(.*?)\@\|(.*?)\@\|(.*)/;
     # note use of @| rather than comma, as SQL statements may contain commas!!
     # cant use simple pipe, as we use this in iList statement!
     # IN FACT, WE WILL DISCARD iInitial et seq and just use ONE SQL STATEMENT
     # PER TABLE. INFINITELY PREFERABLE!
    $iType = $1;
    if (length $iText < 1) # if overriding column name exists already, preferenti
       { $iText = $2;      # buuut otherwise use default field name.
       };
    $iList = $3;
    $iInitial = $4;
```

```
        $iResponse = $5;
        $iScript = $6;
        $iResponse = $tResp; #force to standard table response if monomorphic.
```

7.2.5 [to fix]

[FIX ME: THIS IS STILL UNFIXED]

```
        # create several items:
        my ($icnt);
        $icnt = 0;
        while ($icnt < $tLines)
          { # LATER WILL SUPPLY A TEXT VALUE from database!!

            $iv = shift(@rows); # what if run out??  | HMM. FIX THIS ???
            if (! defined $iv)
               { $iv = ’’;
               };

            my($j);
            if ($ilimit > 0)
               { $j = CreateOneItem ($myODBC, $i, $newW,
                    $iX, $iY, $iW, $iH,
                    0, $iInitial, $iResponse, ’’,  # initial and final are null??? ??
                    $iType, $iv, $iList, 1, $irEnabled, 0, $iText, ’’);   #
                    #
                 if ($j)
                    { $GROUPS[$i] = 0; # default is NO group ???
                      $VVALS[$i] = $iv; # for later query: see: ˆV  18/5/2004
#                      &Print (" \n val1($i) = $iv"); # 18/5/2004
                                          # ??????????? CLUMSY HACK!
                      $i ++;        # bump overall item count (an index into @TKITEMS
                      $iY += $iH;    # move down to draw next item [??? DO THIS ANYWA
                      $ilimit --;
                    };
               };
            $icnt ++;     # bump count
          };

        $iX += $iW;  #move right to next column!  [COULD CHECK FOR OVERRUN???]
      };

  $i -= $idx;
  return ($i);  #return number of NEW items
}
```

8 Items

8.1 CreateOneItem

CreateOneItem is another one of those monster routines. We break it up, mainly into small sections each of which deals with a particular item type.

8.1.1 Startup

CreateOneItem accepts a whole host of parameters, and returns 1 or 0, depending on whether it succeeded or failed. As expected, we submit a database handle, the current total number of items (idx), a Tk window (newW), various coordinates, several scripts, a datum type, the item's text, an associated list (only really used for poplists), whether the item is enabled at startup, and a few other bits and bobs. The iPaper value is an ugly hack, at present only used for pushbuttons. The iScript value is used in highly experimental routines for communication between widgets in the same menu.

```
sub CreateOneItem
{ my($myODBC, $idx, $newW,
     $miX, $miY, $miW, $miH, $miGroup,
     $iInitial, $iResponse, $iScript,
    $iType, $iText, $iList, $iLines,
    $enabled, $uid, $fixedtext, $iPaper);
  ($myODBC, $idx, $newW,
    $miX, $miY, $miW, $miH, $miGroup,
    $iInitial, $iResponse, $iScript,
   $iType, $iText, $iList, $iLines,
   $enabled, $uid, $fixedtext, $iPaper) = @_;
   my ($rowparam);
   $rowparam = $iText;
   $INSCRIPT[$idx] = $iScript;
```

8.1.2 Minor initialisation

The following is largely concerned with paper and ink colours.

```
   my ($paper, $ink, $both);
   $paper = ’’;
   $ink = ’’;
   $both = ’’;
   if ($uid > 0)
      { $both = &GetSQL ($myODBC,
              "SELECT miPaper, ’,’, miInk \
                FROM MENUITEMS WHERE miUid = $uid;",
             "get item colour");
        if (length $both > 0)
           { $both =˜ /(.*),(.*)/;   # pull out colours
             $paper = $1;
             $ink = $2;
      };   };
  if ($BUG & 2)
    { &Print ("\n\n Type = $iType, \
              Text = ’$iText’, Lines = $iLines");
      &Print ("\n DEBUG: Item details (x,y,w,h) =\
         ($miX,$miY,$miW,$miH) group $miGroup");
      &Print ("\n    Initial: $iInitial");
      &Print ("\n    Response: $iResponse");
      &Print ("\n    Script: $iScript");
      &Print ("\n    List: $iList");
    };
  if ($iType == 4) #pushbutton
    { $iText = ’0’; # default to zero ?!
    };
  $TKVALUES[$idx] = $iText; # keep value!
```

In the above we obtain ink and paper values, but really should explore defaults for these a bit better! A lot of the above is taken up by the ugly debugging (BUG) section.

8.1.3 Create a label

We create and place a Tk label, and run the associated script.

```
  my($r);
  if ($iType == 1) # label. Poor inconstant Perl!
     { $TKITEMS[$idx] = $newW->Label(  );
       $TKITEMS[$idx]->place( -anchor => ’nw’,
                              -relx => $miX,
                              -rely => $miY);
       $r = &RunClearScript ($myODBC,
                $iInitial, $newW, $idx, $iText);
       $iText = pop(@CMDSTACK);
       if ($r <= 0)    # -1 = failed, 0 = HALT
          { $TKITEMS[$idx]->destroy; # delete
            return 0; # fail
          };
       $TKITEMS[$idx]->configure( -text => $iText );
       if ($BUG & 8)
          { &Print (
              "\n ITEM: label idx $idx<$iText>($miX,$miY)");
     };   }
```

If the script fails, we clean up the new widget and fail miserably.

8.1.4 Create a button

```
elsif ($iType == 2) # button
   { $TKITEMS[$idx] =
       $newW->Button(
         -command => [ \&DoButton2,
                       $myODBC, $iResponse, $newW, $idx]
                    );
     $TKVALUES[$idx] = $rowparam;
     $TKITEMS[$idx]->place( -anchor => 'nw',
                            -relx => $miX,
                            -rely => $miY,
                            -relheight => $miH,
                            -relwidth => $miW);
     if (length $ink > 0)
        { $TKITEMS[$idx]->configure (-foreground => $ink);
        };
     if (length $paper > 0)
        { $TKITEMS[$idx]->configure (-background => $paper);
        };
     $r = &RunClearScript ($myODBC,
             $iInitial, $newW, $idx, $iText);
     $iText = pop(@CMDSTACK);
     if ($r <= 0)
        { $TKITEMS[$idx]->destroy;
          return 0;
        };
     $TKITEMS[$idx]->configure( -text => $iText );
     if ($BUG & 8)
        { &Print (
           "\n ITEM: button idx $idx<$iText>($miX,$miY)");
   };   }
```

If the user now clicks on the button, DoButton2 will be invoked with the arguments supplied. See how the `idx` value will also be submitted to DoButton2. The initial iText value (now rowparam) is also retained (in TKVALUES). Here too we run an initialisation script, and pop the resulting value off the stack as iText, using the value to provide text on the button!

8.1.5 Create a checkbox

```
elsif ($iType == 3) # checkbox
   { $TKITEMS[$idx] =
        $newW->Checkbutton( -text => '',
          -command => [ \&DoCheckbox3, $myODBC, $iResponse, $newW, $idx]
                          );
     $TKITEMS[$idx]->place( -anchor => 'nw',
                            -relx => $miX,
                            -rely => $miY,
                            -relheight => $miH,
                            -relwidth => $miW);
     $TKITEMS[$idx]->configure (-variable => \$TKVALUES[$idx] );
     if ($enabled == 0)
        { $TKITEMS[$idx]->configure (-state => ’disabled’);
        };
     $r = &RunClearScript ($myODBC, $iInitial, $newW, $idx, $iText);
     $iText = pop(@CMDSTACK);
     if ($r <= 0)
        { $TKITEMS[$idx]->destroy;
          return 0;
        };
     if ($iText =˜ /ˆ1$/ )
        { $TKITEMS[$idx]->select; # value is 1
        };
     if ($BUG & 8)
        { &Print ("\n ITEM: checkbox idx $idx<$iText>($miX,$miY)");
   };   }
```

It’s important to note that, breaking with convention, a checkbox per se *never* has associated text. You have to create a separate label, and place it where you want![36]

See how we associate a variable (in TKVALUES) with the checkbox! Destruction of the widget if the script fails is similar to that in preceding widget creation code.

8.1.6 Create a pushbutton

There is no ‘intrinsic’ pushbutton in Perl, so we make our own. We also need the facility to mutually exclude (mutex) other pushbuttons in the same group.

```
elsif ($iType == 4)
   { my ($red);
     $red = $CONST{’RED’};   # ACTIVE colour
     if (length $iPaper > 0)
        { $red = $iPaper;
        };
     $TKITEMS[$idx] = $newW->Button(
         -text => $fixedtext,  # NOT: => $iText,
         -foreground => $red,
         -background => $CONST{’WHITE’},
         -activeforeground => $red,
```

[36] It makes some sense as you never then have to worry about how the system will jimmy things to fit the associated text in, or whatever. A checkbox is a checkbox, a label is a label!

```
       -activebackground => $CONST{’WHITE’},
       -command => [ \&FlipButton, $myODBC, $iResponse, $idx, $newW ]
                        );
   $TKITEMS[$idx]->place(
       -anchor => ’nw’,
       -relx => $miX,
       -rely => $miY,
       -relheight => $miH,
       -relwidth => $miW);
   $r = &RunClearScript ($myODBC,
             $iInitial, $newW, $idx, $iText);
   $iText = pop(@CMDSTACK);
   if ($r <= 0)
      { $TKITEMS[$idx]->destroy;
        return 0;
      };
   # here, enable button if 1 value in $iText!!
   if ( $iText =˜ /ˆ1$/ )
      { $TKITEMS[$idx]->configure(
          -background => $red,
          -foreground => $CONST{’GREY’});
        $TKITEMS[$idx]->configure(
          -activebackground => $red,
          -activeforeground => $CONST{’GREY’});
        $TKVALUES[$idx] = 1;
      };
   if ($BUG & 8)
      { &Print (
          "\n ITEM: pushbutton index $idx<$iText>($miX,$miY)");
};    }
```

The default active (‘red’) colour of the pushbutton is actually specified externally as a ‘constant’ so the smart user can alter this (a frill)! We can also override the colour value with a menu-defined one. Clicking on the button toggles it by invoking FlipButton. Other coding is very similar to that for the preceding widgets, with the exception of the check for a `1` value in iText, which, if it succeeds, flips the button into an ON state.[37]. See how, for visual appeal and to avert confusion, we also need to fiddle with the *active* fore and backgrounds!

8.1.7 Create a text field

We create a text field, and then run an initialisation script, as for the items above.

```
elsif ($iType == 5) # textfield
   { $TKITEMS[$idx] = $newW->Entry();
```

[37] FlipButton shouldn’t be used *here*

```
      $TKITEMS[$idx]->place( -anchor => 'nw',
                             -relx => $miX,
                             -rely => $miY,
                             -relheight => $miH,
                             -relwidth => $miW);
      $TKITEMS[$idx]->configure (
                  -validatecommand => [ \&CheckEntry5,
                        $myODBC, $iResponse, $newW, $idx],
                  -validate => 'focusout',
                  -textvariable => \$TKVALUES[$idx] );
      $r = &RunClearScript ($myODBC,
                  $iInitial, $newW, $idx, $iText);
      $iText = pop(@CMDSTACK);
      if ($r <= 0)
         { $TKITEMS[$idx]->destroy;
           return 0;
         };
      $TKVALUES[$idx] = $iText;
      if ($BUG & 2)
         { &Print ("\n DEBUG: TEXT: value is <$iText>");
         };
      if ($BUG & 8)
         { &Print ("\n ITEM: txt idx $idx<$iText>($miX, $miY)");
   };    }
```

Validation of the text string entered by CheckEntry5 occurs on leaving the box (focusout). See how, as usual, we associate the text with an index into the array TKVALUES. We still need to explore (within Tk) altering the text field's -state (normal or disabled), and perhaps -font, colours and so forth.

8.1.8 Create a poptrigger

The poptrigger is a bit more exacting, because the poplist must be generated by a script if the submitted 'list' starts with `->`. Typically the script run by RunWholeScript will invoke SQLMANY. The following is patterned on the preceding code.

```
  elsif ($iType == 6) # poptrigger
     { $TKITEMS[$idx] = $newW->Optionmenu();
       my(@iv);
       $_ = $iList;
       if ( /^->(.+)/ ) # if -> run script!
          { $_ = $1;
            @CMDSTACK = ();
            &RunWholeScript ($myODBC,
               $_, $newW, $idx, -1);
            @iv = @CMDSTACK;
          } else
          { s/\|$//;   # rid of last pipe!
            @iv = split /\|/;
          };
       $TKITEMS[$idx]->addOptions(@iv);
       $TKITEMS[$idx]->place( -anchor => ’nw’,
                              -relx => $miX,
                              -rely => $miY,
                              -relheight => $miH,
                              -relwidth => $miW);
       $POPVALUE[$idx] = $iText;
       $TKITEMS[$idx]->configure(
               -textvariable => \$POPVALUE[$idx]);
       $r = &RunClearScript ($myODBC,
               $iInitial, $newW, $idx, $iText);
       $iText = pop(@CMDSTACK);
       if ($r <= 0)
          { $TKITEMS[$idx]->destroy;
            return 0;
          };
       $POPVALUE[$idx] = $iText;
       $TKITEMS[$idx]->configure (
          -command => [ \&Dopoptrigger6,
               $myODBC, $iResponse, $newW, $idx] );
       if ($BUG & 8)
          { &Print ("\n ITEM: poptrigger $idx <$iText> ($miX, $miY)");
     };   }
```

8.1.9 Create a scrollbar

This section is just a stub, at present.

8.1.10 Exit

If all of the above testing failed, we warn of the failure. Finally we exit, with a return code of 1 only if the routine succeeded.

```
 else
     { warn "Bad item type $iType";
       return 0; # fail
     };
return 1; #success.
}
```

8.2 Subsidiary ‘Item’ routines

The following routines are used by CreateOneItem above, or attached as responses to Tk items created by it.

8.2.1 RunClearScript

This routine completely clears the command stack before it runs the script provided. It accepts an ODBC connection, a script (iInitial), a Tk window (newW), as well as an index into TKITEMS and a text string. The text string is pushed to the stack *after* the stack has been cleared, and before the script is run.

```
sub RunClearScript
{ my ($myODBC, $iInitial, $newW, $idx, $iText);
     ($myODBC, $iInitial, $newW, $idx, $iText) = @_;
  @CMDSTACK = (); # clear it. clumsy.
  push (@CMDSTACK, $iText);
  if (length $iInitial > 1)
     { return &RunWholeScript ($myODBC,
                 $iInitial, $newW, $idx, -1);
     };
  return 1;   # ’success’.
}
```

Smarter would be to lock access to the command stack below the current level. We have actually already implemented such a mechanism in the PDA program, but this code lags behind.[38]

8.2.2 FlipButton

Toggle a pushbutton state between 1 and 0. This routine is invoked whenever a pushbutton is clicked upon — it is made more complex because it also has to ensure that other pushbuttons in the same group are now turned off, if the click turns the button *on*. Do *not* use FlipButton if the other buttons in the group haven’t yet been created or linked to one another! The routine which ensures the other buttons turn off is called Mutex2. Only after Mutex2 do we run the associated script (iResponse) which determines the button’s response.

The arguments for Flipbutton are minimal — the database handle, a Tk window (newW), the index of the item within TKITEMS, and the response script.

If the user clicks on a button that is already *on*, the click doesn’t turn the button off again, it simply does nothing![39]

[38] Needs work, what’s new?

[39] This seems strange, but is a consequence of our choice of focusFollowsMouse in the Tk options right at the start of the program.

```
sub FlipButton
{ my($myODBC, $iResponse, $i, $newW);
  ($myODBC, $iResponse, $i, $newW) = @_;
  my($btn);
  $btn = $TKITEMS[$i];
  my($bclr, $fclr);
  $bclr = &GreyGet( $btn->cget('-background') );
  $fclr = &GreyGet( $btn->cget('-foreground') );
  if ( $bclr ne $CONST{'GREY'} )
     { return;
     };
  $btn->configure( -background => $fclr,
                   -foreground => $CONST{'GREY'}); # on
  $btn->configure( -activebackground => $fclr,
                   -activeforeground => $CONST{'GREY'});
  $TKVALUES[$i] = 1;
  &Mutex2($i);
  if (length $iResponse < 1)
     { return;
     };
  @CMDSTACK = ();
  push (@CMDSTACK, $TKVALUES[$i]); #clumsy
  &RunWholeScript ($myODBC, $iResponse, $newW, $i, -1);
}
```

8.2.3 ClearButton

ClearButton is only called by the following routine: Mutex2. It simply clears the relevant pushbutton, using the index into TKITEMS (`i`) provided.

```
sub ClearButton
{ my ($i);
     ($i) = @_;
    my($fclr);
    $fclr = $TKITEMS[$i]->cget('-background');
    if ($fclr eq $CONST{'GREY'})
       { return;
       };  # return if already clear.
    if ($fclr eq $CONST{'WHITE'})
       { $fclr = $TKITEMS[$i]->cget('-foreground');
       }; # if white, get ink colour!
  $TKITEMS[$i]->configure (-background => $CONST{'GREY'},
                           -activebackground => $CONST{'GREY'},
                           -foreground => $fclr,
                           -activeforeground => $fclr
                           );
  $TKVALUES[$i] = 0;  #also clear *value*
  if ($BUG & 8) { &Print ("\n CLEAR: $i"); };
};
```

See how we don’t invoke another script when we clear the button! This is because our database should hold a record of *when* the former item and any current item was set, so storing information about the ‘clearing’ is redundant and time-consuming.

8.2.4 Mutex2

Here we go through the array GROUPS looking for items associated with the current pushbutton (in the same group). Being obsessive, we have a ‘belt and braces’ approach to ensure that we don’t get caught up in this routine. See how we *don’t* clear the *current* button.

```
sub Mutex2
{ my ($i);
  ($i) = @_;
  my ($belt, $todo);
  $belt = 100;
  $todo = $i;
  if ($BUG & 32) { &Print ("\n Debug mutex: entry is $i; "); };
  while ($belt > 0)
    { $i = $GROUPS[$i];
      if ($BUG & 32) {&Print (" ->$i"); };   # debug
      if ($i == $todo)   # back to DOH
         { return;
         };
      &ClearButton ($i); # else, clear
      $belt --;
    };
  Print ("\n ERROR: infinite loop in mutex. Index $todo");
}
```

8.2.5 GreyGet

As mentioned before, ‘constants’ such as RED, GREY and WHITE can be altered in the file `data\constants.const`. A pristine pushbutton is ‘white’, an active one is ‘red’, and an inactivated one is ‘grey’. In the following routine we provide a colour, and if its white, then we return grey.[40] And that’s it.

```
sub GreyGet
 { my ($clr);
      ($clr) = @_;
```

[40] By default, this grey is not the same as MS Windoze institutional grey.

```
      if ($clr eq $CONST{’WHITE’})
         { return ($CONST{’GREY’});
         };
   return ($clr);
 }
```

8.2.6 DoButton2

We respond to a button click. Necessary arguments are the ODBC handle, a response script, a Tk window, and the index of the clicked button.

```
sub DoButton2
{ my($myODBC, $iResponse, $newW, $i);
    ($myODBC, $iResponse, $newW, $i) = @_;
  if (length $iResponse < 1)
     { return;
     };
  @CMDSTACK = ();
  &RunWholeScript ($myODBC, $iResponse, $newW, $i, -1);
}
```

8.2.7 Dopoptrigger6

The poptrigger version of DoButton2.

```
sub Dopoptrigger6
{ my($myODBC, $iResponse, $newW, $i);
    ($myODBC, $iResponse, $newW, $i) = @_;
  my ($fred);
  $fred = $POPVALUE[$i];
  if (length $iResponse < 1)
     { return; # do nothing if undefined
     };
  @CMDSTACK = (); # clumsy as usual.
  push (@CMDSTACK, $fred);
  &RunWholeScript ($myODBC, $iResponse, $newW, $i, $i); # hmm.
}
```

Clumsy for several reasons. In this and several other areas, we should consider using RunClearScript!

8.2.8 DoCheckbox3

Similar to the above response routines is DoCheckbox3.

```
sub DoCheckbox3
{ my($myODBC, $iResponse, $newW, $i);
    ($myODBC, $iResponse, $newW, $i) = @_;
  if (length $iResponse < 1)
     { return;
     };
  @CMDSTACK = ();
  my ($valu);
  $valu = $TKVALUES[$i];
  push (@CMDSTACK, $valu);
  &RunWholeScript ($myODBC, $iResponse, $newW, $i, -1);
}
```

8.2.9 CheckEntry5

```
sub CheckEntry5
{ my ($myODBC, $iResponse, $newW, $i);
     ($myODBC, $iResponse, $newW, $i) =@_;
    if (length $iResponse < 1)
       { return; # do nothing
       };
  my($newval);
  $newval = $TKVALUES[$i];
  if (length $newval < 1)
     {return;
     };
  $newval =˜ s/’/’’/g;  # [4]
  $newval =˜ s/\|/\?/g; # [5]
  @CMDSTACK = ();
  push (@CMDSTACK, $newval);
  &RunWholeScript ($myODBC, $iResponse, $newW, $i, -1);
  return 1;
}
```

We won’t use other Perl/tk values available. There’s a problem in the above with clearing of a string [look at this]! The duplication of single quotes `[4]` and removal of pipes `[5]` limits the damage which might be done when arbitrary text strings and SQL interact. A return value of 1 signals success.

9 Scripting

Our scripting language is at present very primitive. We anticipate in the future being able to view scripts in a variety of ways, permitting macro-like abstraction, but without the limitations of macros in eg C++.[41]

Scripts are linear, each script *command* being separated from the next by the character sequence `->`, this representing an arrow, or if you wish, flow of control from left to right.

Many commands take things off the *stack*, or push values back onto the stack. We have a few slightly odd but satisfying conventions:

- Commands which take two items off the stack usually 'apply' the topmost argument to the next one down. For example, if we say `100->2->DIV`, this reads as "Put 100 on the stack, then two, and then divide 100 by two, placing the result back on the stack".[42]

- When we substitute values, moving them into a text string from the stack, the order we read them is the order of substitution. For example

  ```
  "Flopsy"->"Mopsy"->"Cottontail"->"We ate $[], $[] and $[]"
  ```

  becomes: `"We ate Flopsy, Mopsy and Cottontail"`

9.1 Complete script execution

In this section we discuss first RunWholeScript (which does just that, given a string containing the script), and the minor routine InterList which clumsily turns an array into a delimited string.[43]

9.1.1 RunWholeScript

This routine accepts the usual database and Tk window, as well as the script (iScrpt), the index of the widget responsible in TKVALUES, and the nasty little variable `pop`. This clumsy hack is used by DoCommand below for resetting the displayed value of a poplist.

[41] My idea is that it should be easy to toggle between a macro-like overview, and a full-text script.

[42] We end up with something resembling reverse Polish notation, that powerful but to most people rather confusing notation found on HP calculators. We envisage the future ability to view such scripts in 'infix-translated' form.

[43] There must be a better way in Perl.

```
sub RunWholeScript
{ my($myODBC, $iScrpt, $newW, $idx, $pop);
    ($myODBC, $iScrpt, $newW, $idx, $pop) = @_;
  &Print("\n Running script{<$iScrpt>}"); # debug
  if (  ( $iScrpt =˜ /-> *now/i )
      ||( $iScrpt =˜ /now *->/i )
     )
     { $TODAY = &GetLocalTime();
     };
```

Another inept hack is that at the start of the script we identify the existence of a reference to the command NOW.[44] If this is the case, we update the variable TODAY, which is really a timestamp. In any one script, NOW always has the same time — that of the start of the script!

9.1.2 Pull out commands

We split the script up into component commands, and execute these one after another. We'll regard each component command as a 'line'.

```
  my (@lines);
  $_ = $iScrpt;
  @lines = split /->/;
  my($l, $skip, $mark);
  $mark = -1; # none
  $skip = 1;  # normal
  foreach $l (@lines)
    { if ( $skip == 2)
         { $skip = 1; # skip line
           next;
         };
      $skip =  &DoCommand($myODBC, $newW, $l, $idx, $pop);
```

Here we introduce the skip and mark variables, used later. The normal value for `skip` is 1, and for mark, -1. If `skip` takes on a value of 2, then the subsequent command is, well, skipped. You can see that `skip` corresponds to the simple SKIP command! Each invocation of DoCommand results in an update to the value in `skip`.

9.1.3 Other skip values: MARK

There are other signals which can be sent in `skip`. Here are some negative ones, and appropriate responses:

[44] We permit leading and trailing spaces, which is perhaps silly.

```
if ($skip <= 0)
   { if ($skip < -10) # marked!
        { $skip += 100;
          $mark = $#CMDSTACK - ($skip);
          if ($BUG & 16)
             { &Print ("\n DEBUG: marked position is $mark");
             };
          $skip = 1;
        } # TO BE CONTINUED..
```

We indulge in some sneakiness here. If the value in `skip` is under -10, then we use this value (after modification) to *mark the stack*. This functionality corresponds to the MARK command. The value in `mark` changes from its normal -1 value to a positive value which indexes CMDSTACK, the stack.

To understand what we are doing here, we need to understand the MARK command. MARK accepts a positive number (call it N)and then codes this by subtracting 100, returning that value. We recover N by adding 100. We mark the stack N deep by *subtracting* N from the top of CMDSTACK.

9.1.4 skip -3: LIST

We have a LIST command which turns the rest of the stack (above the current mark) into a string. A value of -3 in `skip` forces this action:

```
elsif ($skip == -3) # LIST
   { my (@lst);
     if ($mark == -1)  # if unmarked
        { @lst = @CMDSTACK[0..$#CMDSTACK];
        } else
        { @lst = @CMDSTACK[$mark..$#CMDSTACK];
        };
          $#CMDSTACK = $mark; # trim stack
          my($sep) = pop(@lst); # separator
          $sep = &InterList($sep, @lst);
          push(@CMDSTACK, $sep);
     $skip = 1;       #continue
   }
```

The separator used by the InterList routine to delimit the items in the newly-created string is obtained from the top of the stack.

9.1.5 skip -4: URZN

This odd but useful command (‘Unmark and return if zero or null’) is invaluable in testing a value and then returning appropriately.[45]

[45] It was formerly URNZ but NZ is the common abbreviation for ’nonzero’.

```
          elsif ($skip == -4)   # [1-4-2005]
             {  $#CMDSTACK = $mark; # trim
                $mark = -1;     # ?
                $skip = 3;      # force return below
             }
```

As with many commands involving marks, this needs some work (The PDA program implements a hierarchy of marks, but the Perl one must still catch up). The trimming of the stack is because URNZ returns, not because it unmarks! [EXPLORE THIS, CF C++ PROGRAM]. You must also read the URZN section below (Section 9.5.13).

9.1.6 Other negative skip values

A value of -2 in `skip` turns off marking, and the important value -1 signals failure of a script. Such failure is deliberately induced by using the FAIL command!

```
          elsif ($skip == -2)   # unmark
             { $mark = -1;
               $skip = 1;       #continue
             }
          else  # -1 signals FAIL
             { last;
             };
        }; # END of -ve skip values..
```

The value(s) on the stack are irrelevant if we FAIL, as the stack isn't used subsequently! The Perl `last` command breaks us out of the current loop. Note that UNMARK does *not* haul stuff off the stack above the current mark, this can only be done using RETURN or URNZ. [EXPLORE THIS, CF PDA C++ PROGRAM].

9.1.7 Returning

Other, positive, `skip` values transmit yet other signals. The value 3 forces a RETURN (with relevant stack trim if marked [EXPLORE THIS?? FIX ME]). The return value from RunWholeScript has significance: -1 signals failure, 1 is 'normal', 0 is terminated (STOP).

```
    if ( $skip == 3 ) # RETURN
       { $skip =1; # after RETURN we continue!
         if ($mark != -1)  # NOOOOOO! [fix me]
           { $#CMDSTACK = $mark;
           };
         last;
       };
  };  # end of inner foreach $l
return $skip;
}
```

9.1.8 InterList

[This routine needs to be checked]. As mentioned above (Section 9.1.4), InterList takes an array and turns this into a text string, with the submitted delimiter separating each item in that final string. The final character of the string is always the separator!

```
sub InterList
{ my ($sep, @lst);
     ($sep, @lst) = @_;
  my($rslt);
  $rslt="";
  my($i);
  foreach $i (@lst)
    {$rslt = "$rslt$i$sep";
    };
  return ($rslt);
}
```

9.2 Command execution

Each script command is represented here. The command list is fairly well bedded-down, but there are still some minor differences between the PDA C++ program and the Perl. These problems must be addressed!

9.2.1 DoCommand

DoCommand is *the* central routine. It's basically just an enormous Perl if . . . elsif . . . statement, which is the exact opposite of elegant. There is undoubtedly enormous scope for optimising this little monster, for example using an associative array in some sneaky fashion.

DoCommand accepts a database handle, Tk window (newW), and three additional items — the command itself (`$l`), the index of the widget associated with the command, and a nasty little value (`pop`), mentioned above in section 9.1.1, used solely in altering the appearance of poplists. The value of `pop` is an index into the array POPVALUE, used by commands such as REFRESH.

```
sub DoCommand
{  my ($myODBC, $newW, $l, $idx, $pop);
      ($myODBC, $newW, $l, $idx, $pop) = @_;
   XPrint ("\n    stack: <@CMDSTACK>"); # debugging
   my ($i, $s);
   XPrint (": line: <$l>");  # debug
   $l =˜ / *(.+) */; # clip
   $_ = $1;
```

After a couple of debugging statements (for XPrint see section 9.15 far below), we clip leading and trailing spaces off the command.

9.2.2 Insert stack pops

```
   while ( /(.*)\$\[\](.*)/ )    # while $[] are present
     { $s = pop(@CMDSTACK);
       $_ = "$1$s$2";
     };
```

In the above we sequentially pop the stack until there are no more `$[]` character sequences within the command string. This allows us to insert stack items flexibly into strings.[46] We insert stack values starting on the *right*![47]

9.2.3 Parenthetic argument

```
   if ( /ˆ(\&*\w+) *\((.*)\)$/ )
      { push (@CMDSTACK, $2);
        $_ = $1;
      };
```

We will commonly specify arguments after an &routine thus:

&Fred(argument here)

Note that the argument in parenthesis is *as if* it's in "quotes" — so we simply push it to the stack. The routine name is made up of any character that's not 'whitespace', i.e. Perl regex `\w`: this includes a. . . Z, a. . . Z, 0. . . 9, and underscore.

[46] Explore the potential side effects of odd stack pops within commands themselves, rather than strings, as well as the potential problems where the inserted string contains this $[] sequence! Such quirks would currently *not* work on the PDA, anyway, so we should probably eliminate them.

[47] We must also consider what to do if stack pop fails because the stack is empty — FIX ME!

9.2.4 Invoke a routine

This section is cumbersome and might be amalgamated with the preceding one. Look for a routine name, and invoke it if present.

```
if ( /\&(\w+)$/ )
  { return &Invoke($myODBC, $newW, $1, $idx, ’’, $pop);
  };
```

Invoke can return success, failure, or even SKIP![48]

9.2.5 Implement REPEAT

```
if (  /ˆrepeat$/i  ) # repeat fx:
   { $_ = pop(@CMDSTACK);
     if (! /\&(.+)/ )
       { &Print ("\n Error: Bad repeat<$_>");
         return -1;  # fail
       };
     $l = $1; # ugly
     $i = 1;
     while ($i > 0)
       { $i = &Invoke($myODBC, $newW, $l, $idx, ’’, $pop)
       };
     if ($i > -1)
       { return 1; # 0=STOP terminates.
       };
     return -1;   # FAIL
   };
```

At present REPEAT must be handed a &routine name. It repetitively invokes that routine, until a STOP statement has been processed (Invoke returns zero).[49]

9.2.6 A quoted item

An item in double quotes is treated as a string and placed on the stack.

```
if ( /ˆ\"(.*)\"$/ )
   { push (@CMDSTACK, $1);
     return 1; # ok
   };
```

[48] Check that the PDA version implements the last-named.

[49] An idea would be to also have an internally defined limiting count, or a timeout!

9.3 SQL commands

There is a defect in our current SQL function QUERY, in Perl (but not on the PDA). [FIX ME: look at GetSQL] QUERY should retrieve a single *row*, and not just a single item, as it currently does.

9.3.1 QUERY

```
if (( /^SQL$/ ) || (/^QUERY$/i))
   { $i = pop(@CMDSTACK); # get sql
     $i = &GetSQL($myODBC, $i, "get one SQL [row] HMM?");
    if (length $i < 1)
       { $SQLOK = 0;
       };
    if ($SQLOK)
       { push (@CMDSTACK, $i);
       };
    return 1;
   }
```

The equivalent SQL command is a legacy which must be removed. As usual, SQLOK is used to signal the success/failure of the SQL statement.

9.3.2 DOSQL

Here we execute an SQL statement, and put nothing back on the stack.

```
elsif ( /^DOSQL$/i )
   { $i = pop(@CMDSTACK); # get instruction
     $i = &DoSQL($myODBC, $i, "perform SQL statement");
     return 1;
   }
```

Note that if we accidentally replace DOSQL with QUERY, then Perl will crash horribly in trying to retrieve a non-existent value.

9.3.3 SQLMANY

Here we retrieve multiple rows from SQL.

```
elsif ( /^SQLMANY$/i )
   { $i = pop(@CMDSTACK); # get query
      my (@mny);
      @mny = &SQLmanySQL ($myODBC, $i, "get SQL array");
       @CMDSTACK = (@CMDSTACK, @mny); # append
      return 1;
   }
```

9.3.4 KEY

Generate an auto-incrementing key, and push it to the stack. The variable kyn which is popped off the stack is used to refer to a column in the SQL table called UIDS. Consult the AutoKey function (Section 5.1.6) for details.

```
elsif (( /^auto$/i ) || ( /^KEY$/i ))
   { my($kyn);
     $kyn = pop(@CMDSTACK);
     $i = &AutoKey($myODBC, $kyn);
     push (@CMDSTACK, $i);
   }
```

An obsolete synonym for KEY is the AUTO command (don’t use it)!

9.3.5 OKSQL

OKSQL pushes a 1 to the stack if the *most recent* SQL statement succeeded, otherwise zero.

```
elsif ( /^OKSQL$/i )
   { push(@CMDSTACK, $SQLOK);
   }
```

9.3.6 COMMIT

See section 5.1.5.

```
elsif ( /^COMMIT$/i )
   { &Commit($myODBC);
   }
```

9.3.7 ROLLBACK

As for COMMIT, see section 5.1.5.

```
elsif ( /^ROLLBACK$/i ) #
   { &Rollback($myODBC);
   }
```

9.3.8 ME

Strictly speaking this command has little to do with SQL, but as we almost always use it in an SQL context, we include it here. It pushes the unique ID of the current user to the stack:

```
elsif ( /^me$/i )
   { push (@CMDSTACK, $CURRENTUSER);
   }
```

9.4 Arithmetic and Logical commands

In many ways, the arithmetic and logical commands in the Perl program are *less* advanced than their interpretation in the PDA program. We need to work on this. The main reason for this deficiency is that we only really conceived of having data types identical to the SQL ones when we started on the PDA version! In addition, there's the seductive typing of Perl, which is ultimately rather evil.

9.4.1 ISNULL

Check for a NULL (zero length string) on the stack, by popping the stack. Return 1 if present, 0 otherwise. This topmost item on the stack vanishes, of course, to be replaced by the answer.

```
elsif (/ˆisnull$/i)
   { $i = pop(@CMDSTACK);
     if (length $i < 1)
        { $i = "1";
        } else
        { $i = "0";
        };
     push (@CMDSTACK, $i);
   }
```

9.4.2 NEG

Negate a number on the stack.

```
elsif ( /ˆneg$/i )
   { $i = pop(@CMDSTACK);
     push (@CMDSTACK, -($i));
   }
```

9.4.3 NOT

If anything other than 0 on the stack, return 0. If zero, return 1.[50]

```
elsif (/ˆnot$/i)
   { $i = pop(@CMDSTACK);
     if ( $i =˜/ˆ *0+ *$/ )
        { push (@CMDSTACK, 1);
        } else
        { push (@CMDSTACK, 0);
        };
   }
```

[50] But what about F, NULL and perhaps even "FALSE"? Look into this, and make it compatible with the PDA representation!

9.4.4 ADD

Add two numbers, replacing them on the stack with the result.

```
elsif (/^add$/i)
   { $i = pop(@CMDSTACK);
     $i += pop(@CMDSTACK);
     push (@CMDSTACK, $i);
   }
```

9.4.5 SUB

Similar to ADD. Note that the topmost stack item (number) is subtracted from the one below, and the result is placed on the stack.

```
elsif (/^sub$/i)
   { $_ = pop(@CMDSTACK);
     $i = pop(@CMDSTACK);
     $i -= $_;
     push (@CMDSTACK, $i);
   }
```

9.4.6 DIV

Similar to SUB. Divide deeper number by topmost (more superficial) one.[51]

```
elsif (/^div$/i)
   { $i = pop(@CMDSTACK);
     $_ = pop(@CMDSTACK);
     $_ /= $i;
     push (@CMDSTACK, $_);
   }
```

9.4.7 MOD

Modulus (remainder) of two numbers. Similar to DIV but we keep the remainder and throw away the quotient. In Perl %= gives the modulus.

```
elsif (/^mod$/i)
   { $i = pop(@CMDSTACK);
     $_ = pop(@CMDSTACK);
     $_ %= $i;
     push (@CMDSTACK, $_);
   }
```

[51] We need to look carefully into typing, integer versus float, and IEEE 754r. Aagh!

9.4.8 MUL

Product of two numbers. As for the others above we need to look into more careful typing!

```
elsif (/ˆmul$/i)
   { $i = pop(@CMDSTACK);
     $i *= pop(@CMDSTACK);
     push (@CMDSTACK, $i);
   }
```

9.4.9 SAME

Check for *identical* items on the stack.[52] Of limited, well actually, no utility with floating point numbers.

```
elsif (/ˆsame$/i )  # if two strings are identical
   { $i = pop(@CMDSTACK);
     $_ = pop(@CMDSTACK);
     if ($i eq $_)
        { push (@CMDSTACK, ’1’);
        } else
        { push (@CMDSTACK, ’0’);
   };   }
```

The routine could be made far less cumbersome.

9.4.10 GREATER

As with SUB and other binary (dyadic) commands, check whether the deeper number is larger than the number more superficially placed on the stack.

```
elsif (/ˆgreater$/i )
   { $_ = pop(@CMDSTACK);
     $i = pop(@CMDSTACK);
     if ($i > $_)
        { push (@CMDSTACK, ’1’);
        } else
        { push (@CMDSTACK, ’0’);
   };   }
```

The code is clumsy.

[52] We will need to be careful here. What about, for example, NULL.

9.4.11 LESS

As for GREATER above.

```
elsif (/ˆless$/i )
   { $_ = pop(@CMDSTACK);
     $i = pop(@CMDSTACK);
     if ($i < $_)
        { push (@CMDSTACK, ’1’);
        } else
        { push (@CMDSTACK, ’0’);
   };   }
```

9.4.12 AND

If two ones on the stack, return 1, otherwise 0. Typical Boolean logic.[53]

```
elsif (/ˆand$/i )
   { $_ = pop(@CMDSTACK);
     $i = pop(@CMDSTACK);
     if (($i == 1) && ($_ == 1))
        { push (@CMDSTACK, ’1’);
        } else
        { push (@CMDSTACK, ’0’);
        };
   }
```

9.4.13 OR

Similar Boolean logic to AND. Return 1 if either value is one.[54]

```
elsif (/ˆor$/i )  # logic: either must be 1
   { $_ = pop(@CMDSTACK);  #
     $i = pop(@CMDSTACK);  #
     if (($i == 1) || ($_ == 1))
        { push (@CMDSTACK, ’1’);
        } else
        { push (@CMDSTACK, ’0’);
   };   }
```

We should probably also define an XOR command.

[53] Agonise over other possible values and interpretations. What about NULL?

[54] No? Perhaps better to return 1 unless both are zero??

9.4.14 ISNUMBER

One of the liabilities of lacking strong typing is the lengths we have to go to for simple questions like “Is it a number?”. Our C++ PDA program has less trouble!

```
elsif (/ˆisnumber$/i)
   { $_ = pop(@CMDSTACK);
     if (/ˆ-?\d+$/)  # if integer
       { push (@CMDSTACK, 1);
       } else
       { push (@CMDSTACK, 0);
   };  }
```

Should the regex allow other leading or trailing characters? Surely not — we thus amended the code on 31/7/2005.

9.4.15 INTEGER

Convert a number (force, coerce it!) into an integer.[55]

```
elsif (/ˆinteger$/i)
   { $i = int pop(@CMDSTACK);
     push (@CMDSTACK, $i);
   }
```

9.4.16 BOOLEAN

A general purpose function to coerce anything into 0 or 1. Makes a lot of our agonising above less important, as we can BOOLEAN almost anything and then use it with logical commands.

Null, zero, ‘false’ or ‘F’ all become zero, regardless of case; others default to one.

```
elsif (/ˆboolean$/i)
   { $i = pop(@CMDSTACK);
     if (   ( $i =˜ /ˆ[0|f|false]$/i )
         || (length $i == 0)
        ) # ugly test
        { $i = ’0’;
        } else
        { $i = ’1’;
        };
     push (@CMDSTACK, $i);
   }
```

[55] Look carefully at what we are actually doing, especially as AFAIK Perl has different options for how *int* actually works!? Make this consistent with PDA!

9.4.17 NULL

NULL shouldn't be confused with ISNULL. NULL puts a null value onto the stack, while ISNULL tests for one!

```
elsif (/^NULL$/i )
   { push (@CMDSTACK, "");
   }
```

9.5 Flow of control and stack commands

9.5.1 RETURN

The magic value of 3 is discussed in Section 9.1.7 above.

```
elsif (/^return$/i)
   { return (3);  # force return
   }
```

9.5.2 STOP

STOP forces termination (without prejudice) in a variety of settings.[56] Particularly useful with REPEAT (See section 9.2.5).

```
elsif ( /^stop$/i )
   { return 0;
   }
```

9.5.3 FAIL

The failure value of -1 causes havoc and destruction (well, script termination anyway) when returned:

```
elsif ( /^fail$/i )  # abort!
   { return -1;
   }
```

9.5.4 SKIP

As discussed above (Section 9.1.2), a return value of 2 forces skipping of the following command. If the command is in the form '&Fred(datum)', then the whole schmeer is skipped, not just poor &Fred. SKIP only skips if there is a '1' on the stack.

[56] Check this, make sure it's identical in Perl and on PDA.

```
elsif ( /^skip$/i )
   { $i = pop(@CMDSTACK);
     if ( $i !~/^1$/ )
        { return 1;   # do NOT skip if zero
        };
     return 2; # force skip
   }
```

9.5.5 COPY

Make an exact copy of the top of the stack.[57]

```
elsif ( /^copy$/i )
   { $i = pop(@CMDSTACK);
     push (@CMDSTACK, $i);
     push (@CMDSTACK, $i);
   }
```

9.5.6 DISCARD

Discard the top item on the stack!

```
elsif (/^discard$/i)
   { $i = pop(@CMDSTACK);
   }
```

9.5.7 LIST

This command turns the whole stack (above the current mark) into a string! So if we say:

```
"A"->"B"->"C"->LIST(,)
```

Then we will obtain: `"A,B,C"`
For details of how this works, see above (Section 9.1.4).

```
elsif (/^list$/i )
   { return -3;
   }
```

[57] As usual, explore what happens if there is nothing on the stack??

9.5.8 SWOP

A singularly useful little command! All it does is interchange the top two items on the stack.

```
elsif (/^swop$/i)
   { $i = pop(@CMDSTACK);
     $_ = pop(@CMDSTACK);
     push (@CMDSTACK, $i);
     push (@CMDSTACK, $_);
   }
```

9.5.9 BURY

Note that the implementation of BURY (and DIGUP) in Perl is convenient (using unshift and shift) but has the drawback that if we unshift enough things, they appear on the stack top again. We must re-implement these functions using the method we employ on the PDA, which is more secure.

Peculiar little commands, but most useful.

```
elsif (/^bury$/i)
   { $i = pop(@CMDSTACK);
     unshift(@CMDSTACK, $i); # good ole Perl.
   }
```

9.5.10 DIGUP

See the note under BURY above (Section 9.5.9).

```
elsif (/^digup$/i)
   { $i = shift(@CMDSTACK);
     push(@CMDSTACK, $i);
   }
```

9.5.11 MARK

The MARK and UNMARK commands are far more primitive (at present) than the PDA implementation. We need to address this problem. At present here, we only have one mark here, rather than having multiple levels of marking, a significant limitation. We've briefly discussed MARK above (Section 9.1.3). We need to do a *lot* of work to 'synchronise' these functions with those on the PDA.

```
elsif ( /^mark/i )
   { $i = pop(@CMDSTACK); # mark index
             if ($BUG & 16)
```

```
                 { &Print ("\n DEBUG: mark index is $i");
                 };
        if (($i > 16) || ($i < -16))
           { &Print ("\n ERROR: Bad mark param: $i");
             return -1;
           };
          if ($i < 0) { $i = -$i; }; # [6]
        return (-100 + $i);
      }
```

The flag [6] illustrates that values submitted to MARK are always *positive* — they say how many items we intend to clip off the current stack.[58] Do NOT submit negative values to MARK, despite the fact that it will accept them!

9.5.12 UNMARK

We *must* make MARK and particularly UNMARK identical in function to the corresponding commands on the PDA. This injunction particularly pertains to what happens to marked items when we unmark (as opposed to RETURNing), as well as multiple levels of marking (only present on the PDA, for now)!

```
   elsif ( /^unmark/i )
      { return -2;
      }
```

Also look at Section 9.1.3.

9.5.13 URZN

As noted above (Section 9.1.5), we here unmark and return *only if* there is a zero *or* null value on the stack! An odd but extremely useful command. Note in particular that the item tested on the stack is *restored* if URZN fails!!

```
   elsif (  ( /^urnz/i )
          ||(/^urzn/i )
         )
      { $i = pop(@CMDSTACK);
        if (((length $i) < 1) || ($i == 0))
           { return -4; # signal urzn
           };
        push(@CMDSTACK, $i); # restore!
      }
```

[58] Previously we had only negative values, but for several reasons, mainly PDA ones, we changed the convention.

There is a problem in that we should probably still succeed if the stack is just plain empty![59]

9.5.14 RUN

This powerful command takes a string off the stack and runs it as a script.[60]

```
elsif ( /^run/i )
   { $i = pop(@CMDSTACK);
     return &Invoke($myODBC, $newW, $i, $idx, '', $pop);
   }
```

The return value depends on the success or failure of Invoke (Section 9.15.2).

9.6 Single letter commands and their friends

9.6.1 X

The subject of a menu, X, is retrieved. Recall that X is passed by default as the subject of the next menu, and that a stack exists to preserve the current X value when a new menu is loaded.[61]

```
elsif (/^X$/)
   { push (@CMDSTACK, $XPARAM);
   }
```

Also have a look at SetX.

9.6.2 V

V refers to the value associated with a particular row (in a polymorphic table), or a particular element in a monomorphic table. A special VVALS array stores this value.[62]

```
elsif (/^V$/)
   { if ($idx < 0)
       { &Alert($newW,
           "Woops! failed to get local [V]ariable");
         return -1;
       };
     $i = $VVALS[$idx];
     push (@CMDSTACK, $i);
   }
```

[59] CHECK this vs the PDA?

[60] Check out the potential for abuse!

[61] Fine print: in the past we rendered this $[X], which is really cumbersome.

[62] We need to look in some detail at error handling here! The current Alert is rather clumsy.

9.6.3 SETX

This command does *not* immediately set a new value for X. It places a new value in NEWXPARAM, and then, when we move to the next menu, *that* new value becomes the new X for that menu. For more immediate and brutal coercion of X, see FORCEX (but be careful).

```
elsif ( /^setX$/i )
    { $i = pop(@CMDSTACK);
      $NEWXPARAM = $i;
    }
```

9.7 General purpose/text commands

9.7.1 IN

Check for a string within a string. The usual rule applies (as for GREATER and so forth) — we check for the topmost string within the deeper string, returning 1 or 0.

```
elsif (/^in$/i )
   { $i = pop(@CMDSTACK);
     $_ = pop(@CMDSTACK);
     if ( /$i/ )
        { push (@CMDSTACK, "1");
        } else
        { push (@CMDSTACK, "0");
   };   }
```

The above is *unsafe* as regex is involved, so we need to look for backticks and so on [FIX ME]! The code is also clumsy.

9.7.2 SPLIT

We split a string into several strings, using a ‘string to split on’ obtained from the top of the stack.

```
elsif (/^split$/i )
   { my(@spl);
     $i = pop(@CMDSTACK); # to split on
     $_ = pop(@CMDSTACK); # string to split
     @spl = split /$i/;
     push (@CMDSTACK, @spl);
   }
```

I love Perl in such circumstances. We need to look at `/$i/` in terms of hacks.

9.7.3 LENGTH

Determine the length of a string.[63]

```
elsif (/^length$/i )
   { $i = pop(@CMDSTACK);
     $i = length $i;
     push (@CMDSTACK, $i);
   }
```

9.7.4 UPPERCASE

This command and the next one should probably be replaced by a more generic text-alteration command based on regex.

```
elsif (/^uppercase$/i)
   { $i = pop(@CMDSTACK);
     $i =~tr/a-z/A-Z/; # good ole Perl!
     push (@CMDSTACK, $i);
   }
```

9.7.5 LOWERCASE

Similar to UPPERCASE.

```
elsif (/^lowercase$/i)
   { $i = pop(@CMDSTACK);
     $i =~tr/A-Z/a-z/;
     push (@CMDSTACK, $i);
   }
```

9.8 Date and time

This section needs reworking in the light of our PDA functions, which are somewhat more mature!

9.8.1 NOW

A simple timestamp — now! Well, not entirely simple, if you examine Section 9.1.1. The timestamp in TODAY remains fixed for the duration of the executing script![64]

```
elsif (/^now$/i)
   { push (@CMDSTACK, $TODAY);
   }
```

[63] Explore the implications of other ‘data types’ in the PDA environment?!

[64] Explore the wisdom of this choice!

9.9 Menu-related commands

9.9.1 ALERT

Consult the Alert routine (Section 4.2.1) for details.

```
elsif (( /^SAY$/i ) || ( /^ALERT$/i ))
    { $i = pop(@CMDSTACK);
      &Alert ($newW, $i);
    }
```

SAY is an older, deprecated variant.

9.9.2 ASK

See the relevant routine in section 4.2.5.

```
 elsif ( /^ASK$/i )
    { $i = pop(@CMDSTACK); # default text
      $_ = pop(@CMDSTACK); # dialog title
      $i = &Ask ($newW, $_, $i);
      push(@CMDSTACK, $i);
    }
```

9.9.3 CONFIRM

This command is discussed under section 4.2.4.

```
 elsif ( /^CONFIRM$/i )
    { $i = pop(@CMDSTACK);
      $i = &Confirm ($newW, $i);  # returns 0 or 1
      push(@CMDSTACK, $i);
    }
```

9.9.4 QUIT

This somewhat dangerous routine terminates everything. Consider having confirmation, as we do on the PDA.

```
 elsif ( /^QUIT$/i )
    { exit;  # DANGER: terminate Perl!
    }
```

9.9.5 MENU

Given a menu name, we go to that menu; given a number, we move back that number of menus, discarding the current menu and intervening ones too! Menu handling is extensively discussed in section 6.

```
    elsif ( /^MENU$/i )
       { $i = pop(@CMDSTACK); # menu name/No.
         &GoMenu ($myODBC, $i, $newW);
         return -1;  # force "fail" (MUST do d/t recursion!)
       }
```

9.9.6 ENABLED

We use this simple command with its single stack argument of either 1 or zero[65] to enable or disable a widget. The widget is indexed into TKITEMS using `idx`.

```
   elsif ( /^enabled$/i )
      { $i = pop(@CMDSTACK);
        if ( /^1$/ )  # if one..
           { $TKITEMS[$idx]->configure (-state => 'enabled');
           } else
           { $TKITEMS[$idx]->configure (-state => 'disabled');
           };
       }
```

9.9.7 POPMENU

This is an interesting command — it clips out the *preceding* menu from the menu stack. the current menu is *preserved*. In addition, the menu and X value we popped are placed on the stack!! Occasionally useful.

```
   elsif ( /^POPMENU$/i )
       { my($jom,$jox);
         $jom = pop(@MENUS);
         $jox = pop(@X); # preserve me!
         $i = pop (@MENUS);
         $_ = pop (@X);
         push(@MENUS,$jom);
         push(@X,$jox);
         push (@CMDSTACK, $_); # push X
         push (@CMDSTACK, $i); # push menu name!
       }
```

We don't, but probably should, check that there is a preceding menu to clip out!

[65] Actually at present, 1 or something else. We should probably check for a zero! FIX and check vs PDA!

9.9.8 PUSHMENU

Even more odd (and ugly) than POPMENU is PUSHMENU. It can occasionally be useful, taking a menu name off the very top of the stack, followed by a value for X, and then storing these below the current menu so that we eventually return to the PUSHed menu with its X (subject) value.

```
elsif ( /ˆPUSHMENU$/i )
    { my($jum,$jux);
      $jum = pop(@MENUS);
      $jux = pop(@X); # keep current
      $i = pop(@CMDSTACK); # menu name
      $_ = pop(@CMDSTACK); # X
      push(@X, $_);        # push these
      push(@MENUS, $i);    #
      push(@X, $jux);
      push(@MENUS, $jum);  # restore current
    }
```

9.10 Local variables

To make our scripting powerful, we need to be able to create local variables. This cumbersome necessity is implemented in the next few sections. We refer to a local variable fred as:

```
$[fred]
```

This is all very well for accessing the value, but how do we make a local variable, test for the existence of a name, and set the value? Let’s explore . . .

9.10.1 NAME

We make a name using NAME.

```
elsif ( /ˆname$/i )
   { $i = pop(@CMDSTACK);
     &CreateLocalName ($i);
   }
```

9.10.2 ISNAME

Here, we test whether a name exists:

```
elsif ( /ˆisname$/i )
   { $i = pop(@CMDSTACK);
```

```
          $i = &IsLocal($i);  # get 1/0
          push(@CMDSTACK,$i);
        }
```

9.10.3 $[name]

We devote a whole section below (9.16) to functions such as FetchLocal, which deal with local variables.

```
   elsif ( /^\$\[(.+)\]$/ )
      { $i = &FetchLocal($1);
        push(@CMDSTACK, $i);
      }
```

9.10.4 SET

SET sets the value of a local variable. See the discussion of `$[name]` above, and section 9.16.5.

```
   elsif ( /^set$/i )
      { $i = pop(@CMDSTACK);   # get name
        $_ = pop(@CMDSTACK);   # and value
        &SetLocal($i, $_);
      }
```

The above should work as normal whether we say set(fred) or `fred->set`. Deeper is the actual value to set ‘fred’ to!

9.11 Graphical

All of the following routines still need to be effectively implemented on the PDA. They are frilly. Later we must include bitmap handling here too!

9.11.1 PAPER

Set the paper colour (background).

```
   elsif ( /^paper/i )
      { $i = pop(@CMDSTACK);
        $TKITEMS[$idx]->configure( -background => $i);
      }
```

9.11.2 INK

Set the ink colour (foreground).

```
elsif ( /ˆink/i )
   { $i = pop(@CMDSTACK);
     $TKITEMS[$idx]->configure( -foreground => $i);
   }
```

9.11.3 LABEL

[The following needs fixing]

```
elsif ( /ˆlabel/i )
   {
     $i = pop(@CMDSTACK);
     $_ = $TKITEMS[$idx]->PathName;
     /.+\.(\D+)\d*/;
     $_ = $1;

     # 1: label
     if ( /ˆlabel$/i )
        {
          $TKITEMS[$idx]->configure( -text => $i);
        }

     # 2: button (and 4?!)
     if ( /ˆbutton$/i )
        {
          $TKITEMS[$idx]->configure( -text => $i);
        }

     # 3: checkbutton
     if ( /ˆcheckbutton$/i )
        {
         # not: $TKITEMS[$idx]->configure( -text => $i);
         # (we don’t allow this sort of attached label)
         # ? provide WARNING message! (silly)
        }

     # 5: entry
     if ( /ˆentry$/i )
        {  # likewise, has no label
        }

     # 6: optionmenu
     if ( /ˆoptionmenu$/i )
        {
```

```
#            print ("\n pop value ($idx) is now $i");
            $POPVALUE[$idx] = $i;  # ugly??
           }

#        # 7:
#        if ( /^button$/ )
#           {
#           }
      }
   # still need to do:
   #  xco, yco
   #  width, height
   #  enabled, hidden
   #  images (pictures)
   #  (sound?)
   #  ...
   #  border? style? font? text?
```

9.12 Communication (experimental)

The following commands are highly tentative. Avoid them!

It would be easy to simply force a property or action upon say a button (for example make the button turn green, in order to signal to the user some sort of alert – or 'it is ok to press me'). To decrease the tightness of coupling between widgets, we might use a more friendly system, as follows:

1. A widget script 'declares' itself with a name (let's call it 'fred'), usually at initialisation, but this can be at any time;

2. A second script (call her 'jane', but she can be nameless) passes a message to fred. The message is stored in an 'inbox' (or 'blackboard') data area that belongs to fred.

3. Fred is made aware of the incoming datum, by invocation of an 'inbox' script. (In our case, what the inbox script usually does is simply *run* the message in the inbox).The inbox script is contained in an item's database definition as a varchar field. The inbox sees its single argument as a string on the stack.

As things stand, this looks very complex for little gain. In the longer term, we believe this approach is sound.

To get the above to work, we require:

1. a register of names (like 'fred'). Each name is associated with its declaring widget by an index into TKITEMS, and the inbox script, so that the inbox script can see its own widget and act upon it.

2. In addition, the name is associated with a blackboard data area.
3. the SEND command looks up the name, finds the associated blackboard, writes to it, and then invokes the inbox script.

We populate INSCRIPT with the relevant script WHEN WE CREATE THE WIDGET, but entries in the associative array IAM are dynamic. Note that in our current implementation, a BLACKBOARD array is NOT necessary, as we directly (ugly!) pass the parameter on the stack.

There's an issue with table items, as they are 'clones'. The solution is to use the V value for the table item to create its *name*; using this convention, an item on the same row KNOWS your name and can talk to you! (But what about communication between rows?)

NB, We could use the 'option' database in perl to achieve a lot of the above. for now, KISS.

9.12.1 IAM

```
elsif ( /ˆiam/i )
   { $i = pop(@CMDSTACK);
     if (exists $IAM{$i})   # not 'defined' but 'exists' !!
        { Print ("\n ERROR: '$i' widget iam exists \
           [value " . $IAM{$i} . "], cannot set to $idx");
          return -1;        #ˆfail
        };
     $IAM{$i} = $idx;      #record the index
   }
```

9.12.2 SEND

```
elsif ( /ˆsend/i )
   { my($nm);
     $nm = pop(@CMDSTACK);  # get destination name
     # on stack is "BLACKBOARD" value. We just send it thus, don't pop it!
     # NO! HMMMMMMMMMM MUST SAVE, CLEAR AND RESTORE STACK, NOT PASS IT ???????????
     # !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
     if (! exists $IAM{$nm})  #if name not defined, fail
        { Print ("\n FAILURE: widget iam '$nm' doesn't exist");
          return -1;
        };
     $i = $IAM{$nm};        # get index
     # -------> first here get INSCRIPT associated with item:
     my ($rs);
     $rs = $INSCRIPT[$i];
     if ( (! defined $INSCRIPT[$i])
         ||(length $rs < 2)
```

```
           )
           { &Print ("\n ERROR: null script for iam ’$nm’");
             return -1;
           };
        return &RunWholeScript ($myODBC, $rs, $newW, $i, -1);
      }
```

9.13 Experimental, obsolete and debugging routines

Avoid using these, or use them with extreme caution!

9.13.1 PRINT

Print to the console, or if redirected, to a file (usually the LOGFILE).

```
   elsif (/ˆPRINT$/i) # print to console!
      { $i = pop(@CMDSTACK);
        &Print ($i);
      }
```

9.13.2 DEBUG

Alter the debugging flags by writing directly to BUG.

```
   elsif ( /ˆDEBUG$/i )
      { $i = pop(@CMDSTACK);  # get code value eg 16
        $BUG = $i;
        &Print ("\n DEBUG CHANGED TO $i");
      }
```

9.13.3 REDRAW

[This command must be fixed].

```
   elsif ( /ˆredraw$/i )
      {  $i = pop(@CMDSTACK);
      }
```

A stub at present, we simply discard the submitted value. Do NOT use this instead of DISCARD.

9.13.4 REGEX

Unfortunately we still haven't implemented regex on the PDA, so this command should be regarded as experimental.

```
elsif ( /^regex$/i )
   { $i = pop(@CMDSTACK);
     $_ = pop(@CMDSTACK);
     if ( /$i/ )
        { push (@CMDSTACK, "1");
        } else
        { push (@CMDSTACK, "0");  # clumsy.
        };
   }
```

If we finally implement the above in all its glory (and this is desirable) then we must be careful of backticks and so forth. In the above we get the regex off the stack, and then apply it to the next (deeper) stack item, returning 1 or 0.

When we do implement this regex on the PDA, we will probably have certain limitations (no back references).

9.13.5 CHOOSE

Choose is discussed in section 4.2.6.

```
 elsif ( /^CHOOSE$/i )
    { $i = pop(@CMDSTACK);
      $_ = pop(@CMDSTACK);
      $i = &Choose ($newW, $_, $i, $myODBC, $idx);
      push(@CMDSTACK, $i);
    }
```

9.13.6 MAP

A clumsy routine to map an option value into a set of option:result pairs, returning the required result. Deprecated. The top argument was the option:result string, with pairs separated by pipes.

```
elsif (/^map$/i )
   { $_ = pop(@CMDSTACK);
     $i = pop(@CMDSTACK);
     my(@sela);
     @sela = split /\|/;
     foreach (@sela)
       { /(.+):(.*)/;
         if ($1 eq $i)
```

```
            { last;
            };
       };
     if (! defined $2)
        { $i = ’’;
        } else
        { $i = $2;
        };
     push (@CMDSTACK, $i);  # THE DEFAULT IS THE LAST VALUE!!
   }
```

9.13.7 RECIPROCAL

We should probably just get rid of this completely, although it logically fits under arithmetic routines!

```
elsif (/ˆreciprocal$/i)
   { $i = pop(@CMDSTACK);
     $i = 1/$i;
     push (@CMDSTACK, $i);
   }
```

9.13.8 FORCEX

This brutal forcing of the X value should probably never be used. If we decide to keep it, it belongs near SETX.

```
elsif ( /ˆforceX$/i )
    { $i = pop(@CMDSTACK);
      $XPARAM = $i;
      pop(@X);  # throw away old X value
      push (@X, $XPARAM); # push new value
      $NEWXPARAM = $XPARAM;
    }
```

The above ‘special pleading’, if used, will generally imply that bad programming is being employed. Beware!

9.13.9 SETZ

This odd command takes a value off the stack, and uses it to set the *variable* value of the current item (referenced by the index `idx` into TKVALUES). There are few if any indications to use it, and there are potential problems with poptriggers![66]

[66] LOOK into the advisability of retaining it, and the implications for poptriggers.

```
elsif ( /ˆsetZ$/i )
    { $i = pop(@CMDSTACK);
      $TKVALUES[$idx] = $i;
    }
```

9.13.10 TEXTBEFORE

Don’t use this text-related command. it’s obsolete. Use SPLIT.

```
elsif (/ˆtextbefore$/i)
   { $i = pop(@CMDSTACK);
     $_ = pop(@CMDSTACK);
      if ( $i =˜ /\./ )
         { $i =˜ s/\./\\\./; }; # ugly
      if ( /(.*?)$i/ )
        { $i = $1;
        }; # if fails, simply return $i unchanged
     push (@CMDSTACK, $i);
   }
```

TEXTAFTER is also obsolete. We won’t even define it.

9.13.11 PAD

This apparently worthless command, which was used to left pad a number with zeroes, should almost certainly be removed! Use string interpolation — `$[]`.

```
elsif (/ˆpad$/i)
   { $i = pop(@CMDSTACK);
     if ( ($i < 0) || ($i > 31) ) # arbitrary max 31
       { &Print ("\n ERROR: bad padding length <$i>");
         return 0; # hmm?
       };
     $_ = pop(@CMDSTACK);  # get string to pad
     while (length $_ < $i) # hmm
       { $_ = "0$_";
       };
     push (@CMDSTACK, $_);
   }
```

9.13.12 JOIN

Don’t use this either. Use string interpolation, that is `"$[]$[]"`

```
elsif (/ˆjoin$/i)
   { $i = pop(@CMDSTACK); # ? order
```

```
        $_ = pop(@CMDSTACK);
        $i = "$i$_"; #clumsy
        push (@CMDSTACK, $i);
      }
```

9.13.13 SECONDS

Look at the logic and value of this nasty command. Replace with e.g. INTEGER variant! The idea is that we convert a date to ugly local system ’seconds’. Hmm. We pop a fully fledged timestamp, and mutilate it. Also see section 4.3.1.

```
   elsif (/^seconds$/i)
      { $i = pop(@CMDSTACK);
        $i = &ConvDate($i); #convert to seconds
        push (@CMDSTACK, $i);
      }
```

9.14 End of a long run

We finally reach the end of the mammoth DoCommand routine. If each of the above tests failed, we have our final `else`:

```
   else
      { Alert ($MAINW, "SCRIPT ERROR: \
        I don’t understand <$_> \n(omitted ampersand?)");
        &Print ("\n ERROR: I don’t understand <$_>");
      };
return 1; # continue..
}
```

The default is to return 1 (and continue processing). Other routines which return different values have already done so!

9.15 Subsidiary routines

Here are some of the routines referred to above.

9.15.1 XPrint

```
sub XPrint
{ my($d);
    ($d) = @_;
    if ($BUG & 16)
       { &Print ($d) ;
       };
}
```

XPrint will only print if the relevant flag in BUG is set. Otherwise it does nothing.

9.15.2 Invoke

Given a database handle, Tk window (newW, as usual), a routine name, and a few other arguments, Invoke retrieves the routine from the FUN database table, and executes the script by calling on RunWholeScript. The submitted argument `i` is the index of the associated widget. For the use of the clumsy argument `pop`, see the documentation on this routine (Section 9.1.1).

```
sub Invoke
{ my ($myODBC, $newW, $fxname, $i, $noargs, $pop);
     ($myODBC, $newW, $fxname, $i, $noargs, $pop) = @_;
  my ($scrpt);
  $scrpt = &GetSQL( $myODBC,
    "SELECT fBody FROM FUN WHERE fName = '$fxname'",
    "get function body");
  if (length $scrpt < 2)
     { &Alert($newW, "BAD SUBROUTINE NAME: <$fxname>");
       return (-1);
     };
  return &RunWholeScript ($myODBC, $scrpt, $newW, $i, $pop);
}
```

A script cannot just be one character long, which isn't much of a limitation.

9.16 Local variables

We've briefly discussed these above (Section 9.10). Here we flesh things out. We have limited the number of local variables to just sixteen per menu (although the PDA program allows 32, which is probably a bit more reasonable).[67] We need to be able to create and clear local names, as well as testing for the existence of a variable, and retrieving its value. We've explored the scripting commands above, now let's look at the routines they call:

9.16.1 ClearLocalNames

This routine simply destroys all local names. It is invoked every time we enter a new menu, as local variables are not passed to a new menu, except via the subject (X).

[67] In terms of complexity, I believe it's silly to allow any more than 32.

```
sub ClearLocalNames
{  %LOCALNAMES = ();
   $LOCAL = 0;
   %IAM = ();
 if ($BUG == 2) { &Print ("\n debug: cleared local names"); };
}
```

In the current clumsy implementation, LOCAL always points to the first empty variable. The clearing of the IAM array is part of the experimental communication routines discussed above (Section 9.12).

9.16.2 KeepLocalNames

We have the ability to transiently store the local names, and restore them if a new menu load didn't work out![68]

```
sub KeepLocalNames
{  %KEPTLOCALNAMES = %LOCALNAMES;
   %KEPTIAMS = %IAM;
   $KEPTLOCAL = $LOCAL;
}
```

9.16.3 RestoreLocalNames

As noted (See KeepLocalNames above).

```
sub RestoreLocalNames #
{  %LOCALNAMES = %KEPTLOCALNAMES;
   %IAM = %KEPTIAMS;
   $LOCAL = $KEPTLOCAL;
}
```

9.16.4 CreateLocalName

Here we make a local variable name, using the associative array LOCALNAMES. The corresponding NAME command is discussed in section 9.10.1.

```
sub CreateLocalName
{ my ($name);
     ($name) = @_;

 if ($BUG == 2)
    { &Print ("\n debug: creating local variable '$name'");
```

[68] See usage.

```
    };
 if ($LOCAL > 15)
    { &Alert ($MAINW, "Too many locals: ’$name’ FAILED!");
      return;
    };
 $LOCALNAMES{$name} = $LOCAL;
 $LOCALARRAY[$LOCAL] = ’’;
 $LOCAL ++;
}
```

LOCALARRAY contains the value (here cleared to a null string), and LOCAL is used to provide the index into LOCALARRAY.

9.16.5 SetLocal

Given the name of a local variable, we set the value.

```
sub SetLocal
{ my ($name, $value);
     ($name, $value)= @_;
  if ($BUG == 2)
     { &Print ("\n debug: set ’$name’ to ’$value’");
     };
  my ($i);
  $i = $LOCALNAMES{$name};
  if (! ($i =˜ /\d/) )
    { &Alert ($MAINW,
       "Oh dear. local var set ’$name’=’$value’ FAILED!");
      return;
    };
  $LOCALARRAY[$i] = $value;
}
```

We look up the index of the name, and access LOCALARRAY using this index.

9.16.6 IsLocal

Check whether a local name exists!

```
sub IsLocal
{ my ($name);
     ($name)=@_;
     my ($i);
  $i = $LOCALNAMES{$name};
  if (  (! defined($i))
      ||(! ($i =˜ /\d/) ) # not numeric = doesn’t exist
     )
    { return (0);
    };
  return (1); # is ok
}
```

9.16.7 FetchLocal

We fetch the value of a local name, given the name. We look up the index into LOCALARRAY, using the associative array LOCALNAMES.

```
sub FetchLocal
{ my ($name);
     ($name) = @_;
  if ($BUG == 2)
     { &Print ("\n debug: accessing name ’$name’");
     };
  my ($i);
  $i = $LOCALNAMES{$name};
  if (! ($i =˜ /\d/) )
    { &Alert ($MAINW, "Dear me! ’$name’ WAS NOT FOUND!\
         \n (It is CaSE sENSITIVE)");
      return (’’);
    };
 $name = $LOCALARRAY[$i];
 if ($BUG == 2)
    { &Print ("->$name");
    };
  return ( $name );
}
```

The coding is clumsy.

10 PDB Creation

We have to be able to export all of our database files in a format which can be read on a Palm PDA. The 'PDB' format of the PalmOS files is moderately complex. Let's look at it, and the format we create to represent SQL data.[69]

10.1 PDB file format

Whatever the platform, the PDB file format specifies only *big-endian* numbers. There are three sections to a PDB, the header, a *recordList* which has its own internal header, and finally, the records themselves (if there are any). The records (data) follow immediately after the recordList, which specifies their *absolute* offsets from the very start of the PDB file.

The header fields are contained in table 2.

Offset	Name	Contents
+0h	name (asciiz)	eg. "xTABLE" padded with hex zeroes
+20h	attrib+version	2 bytes each
+24h	creationDate	'seconds after 12am 1 Jan 1970'
+28h	modificationDate	this, above must be NONZERO
+2Ch	lastBackupDate	this may be zero.
+30h	modificationNumber	0 is ok
+34h	appInfoID	0
+38h	sortInfoID	0 (avoid using these 2; keep = 0)
+3Ch	type	we will use 'DATA'
+40h	creator	'JoVS'
+44h	uniqueIDSeed	Used to generate uids! ZERO!
+48h	recordList	has its own HEADER, then record entries!

Table 2: PDB header fields

The recordList has a header that is usually only six bytes long. The first four bytes are almost always zero,[70] and the last two bytes are an unsigned integer containing the number of records. If there are no records, then it is customary to 'pad' the recordList with two bytes.[71]

[69] The following documentation is abstracted from some of my previous documentation, and may need a little work.

[70] Okay, there's a lot of stuff about optional multiple recordLists, application info blocks and sort info blocks, but these should all be assiduously avoided!

[71] We shouldn't encounter this, as we will always have at least one record in the file — our own header — but in Palm programming, especially with file import and export, you'll find that some programmers don't always stick to the rules. Be cautious.

After its header, the recordList has a section that describes each record in the database. Eight bytes are set aside here for each and every record entry, in the following format:

Offset	Name	Contents
+0	localChunkID	four byte offset of record *from start of PDB*
+4	attributes	keep this single byte zero
+5	uniqueID	3 bytes = 0; PalmOS will alter appropriately

Table 3: PDB Record Entry format

Although the above looks superficially adequate, a major liability is that one can only determine the size of a record by subtracting its offset from the offset of the next item. You have a particular problem when you want the size of the final record, as now you have to subtract the offset of this item from the file size, which is just plain silly!

10.2 Our own header

The first record in each PDB database we create will be strictly formatted according to my own specification. The 'final' format is displayed in table 4. The idea is that each column descriptor is 'its own person', with all text names etc contained within the scope of the column. We can then move column headers around with alacrity.

Offset	Size	Contents
+0	4	CRC32 (extends over rest of header); or 0 if NO CRC.
+4	2	Flags. Bit 0 of low order byte set to 1 iff no CRC.
+6	2	byte length of header, including flags and CRC
+8	4	To accommodate sorting, this 32 bit number MUST always be < 0
+C	2	all zeroes (at present)
+E	2	Number of columns=n
+10h	$2 * (n + 1)$	offsets of column descriptors, relative to start of CRC32 above
2*n+10h	(varies)	Actual column descriptors

Table 4: Our header format

We have $n + 1$ offsets so that the width of the final column descriptor can also be calculated with ease. We include a final ‘phantom’ column which merely contains the offset of the first byte after the top of the last column! The length of each column descriptor varies, but the format is constant, described in table 5.

Offset	Size	Contents
+0	2	offset of column name from *here* i.e. 10h
+2	2	k = length of column name, max 15 chars
+4	2	max width of column data
+6	1	Type of column
+7	1	scale, or zero.
+8	2	relative offset (from +0) of name of table depended on
+A	2	n = length of name of table depended on, max 15 chars
+C	4	all zeroes
+10h	k+1	name of column, with added 0x0 *
+11h+k	n+1	name of table depended on, added 0x0 *
* = ASCIIZ redundancy		

Table 5: Our column format

Although the first entry in Table 5 (offset of column) is redundant, we retain it, and *mandate* its use![72]

We will later [DO THIS!] use the bytes at offset 0xC to reference database tables and columns that refer to this table, using an internal linked list of table names and binary column numbers.[73]

10.2.1 Column data types

The column data type is specified by a single character, mnemonic for the minimal set of types we have chosen (V=varchar, I=integer, N=numeric[74] with precision and scale, D=date, T=time, S=timeStamp, F=float). We have deliberately *not* implemented the full SQL range of data types. Table 6 describes the types.

[72] The above has been changed from a preceding format, where we used offsets that were absolute, and lumped names together after all of the column descriptors.

[73] Implying that any alteration to table structure will mandate mutual updates.

[74] The only reason why we prefer ‘numeric’ over ‘decimal’ is because it begins with an ‘N’, not because we’re into Oracle, or anything!

Code	Type	Description
V	Varchar	Character varying
I	Integer	0–999999999. For table keys
N	Numeric	Fixed point, with scale and precision
D	Date	YYYY-MM-DD Internal format YYYYMMDD
T	Time	HH:MM:SS. Internally HHMMSSffffff
S	TimeStamp	Date followed by a space, then time
F	Float	Floating point. IEEE 754 standard

Table 6: Our seven SQL data types

We have *none* of the following types: smallint, real, double precision, bit, bit varying, character, national character and its variants, CLOB, variants that incorporate time zone, interval and boolean.[75] For decimal, use numeric. We eschew use of a fixed length character format, as this is inefficient.[76]

We also use 'Integer' in a very specific context — in our database, only integer keys are allowed, and only single primary keys are permitted! (Almost any database can be converted to such a format, which has many merits). We discourage the use of integers in other contexts — rather use a Numeric with a scale of zero. Smallint is not provided owing to the inconsistency of the length across machines, and its lack of utility.[77]

The bit type is not implemented because the information can be stored in numerics/integers, in fact, on most systems usually are, as well as the unfortunate errors contained in their SQL definition. As regards all the other types, well, KISS.

Floating point variables are in 'standard' IEEE 754 format (64 bits = 53+1+10). We thus avoid vague terms like 'double precision'.

10.3 Data row format

There's one final format we need to know, and that is the internal format of the data rows. This is fairly simple, with just two tiny wrinkles. The format is shown in table 7.

[75] We may eventually implement a BLOB type.

[76] If desired, user side routines could be used to pad a varchar up to the appropriate length!

[77] Note that integer column variables (type I) should have a dependency on another table, but there are two exceptions — primary keys, and components of the 'generator' table for such keys! [PERHAPS DISCUSS IN MORE DETAIL]

Offset	Size	Contents
+0	4	CRC32. If not used, clear to zero. (NB) Extends over rest.
+4	2	flags. Default is all zero. (0x0000). Bit 0 set to 1 if NO CRC.
+6	2	LENGTH OF THIS ROW, INCLUDING flags, CRC32.
+8	4	Key. not stored in row data itself! value of this 32 bit signed index is NEVER < 0 or $>= 10^9$
+C	4	all zeroes.
+10h	2+2*n	offsets of n items. Final entry is offset of last byte of last item PLUS ONE

Table 7: Our data row format

The wrinkles are as follows:

1. As noted, the key is not stored in the data component (n items), but the offset points to offset +8 from the start of the data row, where the key is stored as a 32 bit unsigned integer (This facilitates sorting within PalmOS). The first item in the row is always the key item.[78]

2. We store the offset of the ‘record that isn’t’ at the top of the data items, so it’s easy to work out the size of the item occupying the last column.

10.4 PDB creation routines

Here we discuss the routines used in translating to PDB format. Each SQL table is converted into a single PDB file. The PalmOS header of the PDB reflects the number of records in our data ‘file’.

10.4.1 MakeAllPDBs

We query our meta table xTABLE, finding all of the tables, and translating each table into a PDB file using MakeOnePDB. All that is needed is the ODBC handle, as we brutally use the global Tk window MAINW for display.

[78] Much agonizing about this one. Probably a bad idea, but the temptation is just too great to have the key as a component of the first 10h bytes of the data row, and thus the potential for quick sorting/searching of rows based on keys without using another offset jump. The apparent advantage is probably illusory.

```
sub MakeAllPDBs
{ my ($myODBC);
     ($myODBC) =@_;
  my(@tablenames);
  @tablenames = &ManySQL ($myODBC,
     "SELECT xTaKey, ’|’, xTaName FROM xTABLE;",
     "fetch table codes/names");
  my ($tbl);
  foreach $tbl (@tablenames)
    { MakeOnePDB($tbl);
    };
  Alert( $MAINW,  "\n PDBs created.");
}
```

10.4.2 MakeOnePDB

PDB creation is rather complex, so we’ve broken the following routine into digestible chunks. An important point in all of the following is that each PDB *must* be written with the records sorted in ascending order by primary key.

First we obtain the name and ID of the table, by splitting the single argument of MakeOnePDB on the separating pipe (\|) character. We open a PDB file to write to on the local machine.

```
sub MakeOnePDB
{ my ($tbl);
     ($tbl) = @_;
  my($tname, $tcode);
  $tbl =˜ /(.+)\|(.+)/;
  $tname = $2;
  $tcode = $1;
  print LOGFILE "\n $tname -------->";  # debug
  my ($colcount, $myhdr, $myQUERY, $dTYPE);
  open PDBFILE, ">pdb/$tname.PDB" or
      die "*CRASH* Could not open PDB pdb/$tname \
      \n(Does ’pdb’ subdirectory exist?) :$!\n";
  binmode PDBFILE;
```

The assumption is made that the file will be written in an existing subdirectory of the current one, called `pdb`.

It’s extremely important to write to the file in binmode, as otherwise in DOS line feed characters are converted to carriage return + line feed, with disastrous results.

Next we create our own header. A peculiarity is that if the table is called UIDS, it is made up entirely of integer key references, so we test for this!

```
  my($isuids); # peculiar!
  $isuids = ($tname =˜ /ˆUIDS$/i);
  ($colcount, $myhdr, $myQUERY, $dTYPE) =
     &MakeOurHeader($myODBC, $tcode, $isuids);
```

We now create and format all records. FetchAllRecs internally formats each record according to our own format.

```
  my (@myrecs);
  my($ph);
  @myrecs = &FetchAllRecs ($myODBC,
                $colcount, $tname, $myQUERY, $dTYPE);
```

Finally, we create a PalmOS header, and write everything to the PDB file we opened above.

```
 $ph = &MakePalmDBHeader ($myhdr, $tname,  @myrecs);
  print PDBFILE $ph;     # write PalmOS header
  print PDBFILE $myhdr;  # write our header
  my ($rec);
  foreach $rec (@myrecs) # write all records
    { print PDBFILE $rec;
    };
  close PDBFILE;
}
```

The following sections detail the routines we've briefly referred to above.

10.4.3 MakeOurHeader

There are a few tricky features here. The first column *must* always be the primary key. There is a good rationale for being able to 'clip out' each column with all of its details, to allow for easy moving around of columns as we create temporary tables and so forth.

MakeOurHeader accepts a database handle and the unique key code of the table (in the meta-table structure), as well as the ugly flag (`isuids`) which says whether we are dealing with the peculiar UIDS table. We create a header descriptor, and return four items: the number of columns, the header text string itself, as well as a comma-delimited text 'list' of the column names, and a similar list of the column types.

The column name list allows us to later on retrieve actual column data, the latter list to format the data according to our peculiar requirements. We've broken up the following code into four sections.

```
sub MakeOurHeader
{ my ($myODBC, $tcode, $isuids);
     ($myODBC, $tcode, $isuids)=@_;
  my (@colkeys);
  (@colkeys) = &FetchAllColumns($myODBC, $tcode);
  if (! defined @colkeys) # [check me]
     { print ("\n No columns (table code : $tcode)");
       return (0, "No columns found", "", "");
     };
  my ($colcount);
  $colcount = 1+ $#colkeys; # usual Perl
```

We fetch column data into an array (See FetchAllColumns). Next, we create column descriptors and concatenate them:

```
  my ($COFF, $CH);
  my($COLNAMECOMMAS, $COLTYPECOMMAS);
  $COFF = '';           # string of offsets
  $CH = '';             # string made up of column descriptors
  $COLNAMECOMMAS = '';  # string (list) of names
  $COLTYPECOMMAS = '';  # string (list) of types
  my ($ck, $chead, $coffset, $cname, $ctype);
  $coffset = 0x12 + 2*$colcount;

  foreach $ck (@colkeys)
    { ($chead, $cname, $ctype) =
         &MakeColDescriptor ($myODBC, $ck, $isuids);
      $CH = $CH . $chead;
      $COFF = $COFF . &Print2($coffset);
      $coffset += length $chead; # move to next offset
      $COLNAMECOMMAS = $COLNAMECOMMAS . $cname . ',';
      $COLTYPECOMMAS = $COLTYPECOMMAS . $ctype . ',';
    };
  $COFF = $COFF . &Print2($coffset); # keep top, too!
```

The initial offset in `coffset` is two times the number of columns plus hex 12 — this is two bytes per column reference, plus 0x10 for the header, plus two for an extra reference which points to *after* the last column.

We then use a foreach loop to make a descriptor for each column. Three items are returned by MakeColDescriptor: a head, a name and a type. Each of these items is concatenated into a different string, the head going to the header string CH, the name into the comma-delimited list COLNAMECOMMAS, and the type into the similar COLTYPECOMMAS. Both of the comma lists end up with a *terminal* comma too.

We next make a header according to our format. This header is only 0x10 bytes long. We've left comments in the following to make allocation clear.

```
  # 3. create overall header (0x10 bytes):
  my ($myh);
#    Offset    Size     Description
#    +0         4       CRC32
    $myh = &Print4 (0);
#    +4         2       Flags.
    $myh = $myh .  &Print2 (1);
#    +6         2       Total header length
    $myh = $myh .  &Print2 ( 0x10 +
                             length($COFF) +
                             length($CH) );
#    +8         4       This number must be < 0:
    $myh = $myh .  &Print4 (0x80000000);
#    +C         2       all zeroes:
    $myh = $myh .  &Print2 (0);
#    +E         2       Number of columns
    $myh = $myh .  &Print2 ($colcount);
```

The CRC32 field is at present unused, and the flag field will for now always contain just 1. The value at offset 0xC is zero, for now. Finally, we return results, chopping off the terminal commas.

```
  # 4. return results:
  chop($COLNAMECOMMAS);
  chop($COLTYPECOMMAS);
  return ($colcount,
          $myh . $COFF . $CH,
          $COLNAMECOMMAS,
          $COLTYPECOMMAS);
}
```

10.4.4 FetchAllColumns

FetchAllColumns is called by the preceding routine. It accepts a database connection and the key of the table, and returns an array of column keys. Using our meta-data, we first identify the primary key (as this must be first in the list of column keys).

```
sub FetchAllColumns
{
  my ($myODBC, $tcode);
     ($myODBC, $tcode) = @_;
  my($key1);
  $key1 = &GetSQL ($myODBC,
     "SELECT xLIMIT.xLiColumn FROM xLIMIT, xCOLUMN \
     WHERE xLIMIT.xLiColumn = xCOLUMN.xCoKey \
     AND xLIMIT.xLiType = ’P’ AND xCOLUMN.xCoTable = $tcode",
    "get primary key");
  if (! $key1)
     { print ("\n No primary key (table code : $tcode)");
       return ’’;  # [check me]
     };
  my (@colkeys);
  @colkeys = &ManySQL ($myODBC,
        "SELECT xCoKey FROM xCOLUMN where xCoTable = $tcode \
          AND xCoKey <> $key1 \
          ORDER BY xCoKey;",
        "fetch columns for this table");
  unshift(@colkeys, $key1);
  return (@colkeys);
}
```

After retrieving the remaining keys using a second SQL query, we put the primary key (key1) up front, and return the list of keys.

10.4.5 MakeColDescriptor

This routine returns information about a column — its header, its name, and its type. The header is in a format which facilitates easy moving around of column headers.[79]

MakeColDescriptor accepts an ODBC connection, the key of the column in our meta-tables, and that nasty flag `isuids`. We’ve chopped the code up into four sections:

```
sub MakeColDescriptor
{ my ($myODBC, $ck, $isuids);
    ($myODBC, $ck, $isuids) = @_;

 # 1. Get basic column information.
 my ($cwidth, $cscale, $ctype, $colname);
 $cwidth = &GetSQL ($myODBC,
             "SELECT xCoSize FROM xCOLUMN WHERE xCoKey = $ck;",
             "get column width");
 $cscale = &GetSQL ($myODBC,
             "SELECT xCoScale FROM xCOLUMN WHERE xCoKey = $ck;",
             "get column scale");
 $ctype = &GetSQL ($myODBC,
             "SELECT xCoType FROM xCOLUMN WHERE xCoKey = $ck;",
             "get column type");
      if ($isuids) { $ctype = ’I’; }; # force 32 bit integer
```

[79] Despite the fact that at present we don’t have e.g. correlated subqueries on the PDA, this approach will make things a lot easier should we need temporary data tables, subqueries, views and so on!

```
$colname = &GetSQL ($myODBC,
           "SELECT xCoName FROM xCOLUMN WHERE xCoKey = $ck;",
           "get name of this column");
if ($ctype eq ’I’)
   { $cwidth = 4;
   };
```

First we obtain basic information about the column — its width, scale and type, as well as the name. We fiddle a little to ensure that integers have a width of 4, and that UIDS values are all integers (eugh)!

Things which we could (but don’t) do include ensuring that the column type is a single byte, and other checks on width and scale. The SQL code is atrociously clumsy. Next, let’s resolve foreign keys, get table name for the foreign key . . .

```
my($ctable, $cnlen, $tnlen);
$cnlen = length $colname;
$ctable = ’’;
$tnlen = 0;
my ($xtbl);
$xtbl = &GetSQL ($myODBC,
       "SELECT xLiTable FROM xLIMIT \
         WHERE xLiColumn = $ck AND xLiType = ’F’;",
       "get foreign key table");
if ($xtbl)
        { $ctable = &GetSQL ($myODBC,
            "SELECT xTaName FROM xTABLE \
               WHERE xTaKey = $xtbl;",
            "get table name");
          $tnlen = length $ctable;
        };
     print LOGFILE ("\n     $colname($ctype) :\
          $cwidth($cscale) -> $ctable");
```

In the above we might check that key reference is type I, and fail (or coerce) if not![80] The LOGFILE printing is simply debugging. Next we create our ‘column descriptor’. Again, we’ve left in extensive comments to explain what we’re doing:

```
 my ($descrip);
 $descrip = ’’;
 my($asciiz);
 $asciiz = 1;  # terminal zero=yes
     #  +0  2  Offset of column name, from here!
     $descrip = $descrip . &Print2 (0x10);
     #  +2  2  Length of column name
     $descrip = $descrip . &Print2 ($cnlen);
```

[80] In our restricted environment, this is appropriate.

```
      #  +4  2  Max width of column data (n bytes)
      $descrip = $descrip . &Print2 ($cwidth);
      #  +6  1  Type of column
      $descrip = $descrip . $ctype;
      #  +7  1  Scale, or zero.
      $descrip = $descrip . sprintf ("%c", $cscale);
      #  +8  2  Offset of name of table
      $descrip = $descrip . &Print2 (0x10 + $cnlen + $asciiz);
      #  +A  2  Length of name of table depended on
      $descrip = $descrip . &Print2 ($tnlen);
      #  +C  4  all zeroes
      $descrip = $descrip . &Print4 (0);
```

Even though we don't use ASCIIZ (zero-terminated) strings, we slip in a terminal zero in the column name.[81] Another redundancy is that at offset +2 we have the offset of the start of the name, despite then name 'always' starting at offset 0x10. Routines accessing our column descriptor are well advised to read this offset rather than assuming 0x10.

There are several cautions:

- We limit the length of a column name to 15 characters.
- The name length written at offset +4 (`cnlen`) does NOT include the terminal zero
- Scale and type are single characters (We might check this!)
- The offset of the table name referenced is relative to the start of this descriptor.[82]

See how we liberally use the routines Print2 and Print4 to 'print' exactly two or four hexadecimal characters, concatenating these strings onto `descrip`.

The final four bytes will point to the DEFAULT value for a column, something which we have not yet implemented.[83] Finally, we concatenate all strings into a descriptor:

```
   # 4. append the string(s):
   $descrip = $descrip . $colname;
   if ($asciiz)
      { $descrip = $descrip . sprintf ("%c", 0x0);
      };
```

[81] The variable 'asciiz' is clumsy but explicit.

[82] A number is present here, even if there is no table name — but then the length of the name in the next entry will be zero.

[83] Incredulous gasps! There's room for a pointer and length.

```
   if ($tnlen) # if table dependency
      { $descrip = $descrip . $ctable;
        if ($asciiz)
           { $descrip = $descrip . sprintf ("%c", 0x0);
      };   };
  return ($descrip, $colname, $ctype);
}
```

The standard Perl character printer, sprintf is used for single character printing.

10.4.6 FetchAllRecs

Given a table and the column names, we retrieve corresponding data, returning an array of *all* rows! Data are formatted appropriately. We break up the routine into bite-sized chunks.

The routine accepts a database connection, the number of columns, the name of the table, the list of columns created by MakeOurHeader (myQUERY), and the corresponding list of datum types (dTYPE).

```
sub FetchAllRecs
{ my ($myODBC, $colcount, $tname, $myQUERY, $dTYPE);
     ($myODBC, $colcount, $tname, $myQUERY, $dTYPE) = @_;
  if ($colcount < 1)
     { return ""; # fail [check me]
     };
  my (@dtypes); # column ’data types’
  $_ = $dTYPE;
  @dtypes = split /,/; # array of column data types
  my ($keyname);
     $myQUERY =~ /([^,]+),/; # get key column
     $keyname = $1;
     $myQUERY =~ s/,/, ’\@\|’, /g;
```

The first column is *always* the primary key. When we process myQUERY above, we replace all of the delimiting commas with the character sequence `@|` implying that this particular sequence cannot be used in any string.[84] Next, we select the columns (with the peculiar delimiter):

```
  my ($thisSQL);
  $thisSQL = "SELECT $myQUERY FROM $tname ORDER BY $keyname";
  my($reccnt);  # total number of rows
  my(@myrecs);
  (@myrecs) = &ManySQL ($myODBC,
                       $thisSQL, "get all values");
```

[84] Not perhaps a major limitation, but one well worth noting!

```
$reccnt = 1 + $#myrecs;
print LOGFILE (" .. records=$reccnt ");
my(@c);
my($i, $ci);
my($offs);
my($dat);
my($formrec);
my($hdr);
$i = 0;
```

The variables defined above are `c`, an array of column values for a row; `i` and `ci`, row and column counts; `offs`, the offset of an item in a generated row; `dat`, a concatenated string of data; `formrec`, the output string for a row which has been formatted; and `hdr`, our header section for the row.[85]

Let's now create an outer 'while' loop which iterates over all rows:

```
while ($i < $reccnt)
  { $_ = $myrecs[$i];
    print LOGFILE "\n Record($i) <$_>";
    @c = split /\@\|/;
    $offs = 0x10 + 2 + 2*$colcount; # offset of 1st datum
    my ($prim);
    $prim = $c[0];  # primary key (clumsy but explicit)
    $formrec = &Print2(8);
    $dat = '';
    $ci = 1;
```

At the start of this outer loop, we first pull out the primary key, that is, c[0]; we then print its offset, which is always simply 0008, to formrec. We also initialise the dat string to null, and the column count (ci) for the inner loop.[86]

```
    while ($ci < $colcount)
      { $formrec = $formrec . &Print2($offs); # write offset
        my ($formd);
        if (! defined $c[$ci])
           { $formd = "";
           } else
           { $formd = &FormatDatum($dtypes[$ci], $c[$ci]);
             print LOGFILE (
                "\n   $c[$ci]: $dtypes[$ci] -> $formd");
             if (defined $formd)
                { $offs += length $formd;
                  $dat = $dat . sprintf ("%s", $formd);
                } else
```

[85] Every row has its own tiny header.

[86] The value of ci is 1 because the first item, the primary key, has already been processed.

```
                { print LOGFILE
    "\n Bad formatting: $dtypes[$ci]:$c[$ci] (@dtypes)";
                };
            };
          $ci ++;
        };
      $formrec = $formrec . &Print2 ($offs); #[7]
      $formrec = $formrec . $dat; # append actual strings
```

In the inner loop, we write the offset of each item to formrec (just as we wrote 0008 for the offset of the primary key), and either write nothing or a formatted version of the datum.[87] Null strings are therefore represented by a pointer to a datum of null length (The datum length can always be worked out by finding the difference between the pointer to the current datum and the next one — this works because we also have a pointer to just after the end of the last datum, as if there was yet another datum at the end of the row of data).

The marker [7] indicates where we write that final offset pointing to the first byte after the top of the last datum. Finally we make the header:

```
      # finally, make the HEADER:
      #     +0       4        CRC32:
      $hdr = &Print4 (0);
      #     +4       2        flags:
      $hdr = $hdr . &Print2 (1);
      #     +6       2        Row length
      $hdr = $hdr . &Print2 ($offs);
      #     +8       4        Key:
      $hdr = $hdr . &Print4 ($prim);
      #     +C       4        all zeroes:
      $hdr = $hdr . &Print4 (0);

      $formrec = $hdr . $formrec; # [8]
      $myrecs[$i] = $formrec;
      $i ++;  # BUMP record count
    };
  return (@myrecs);
}
```

The row length includes the flags and CRC32 (or the four zero bytes in its place). As usual, a flag value of 1 signals ‘no CRC’. See how we put the primary key value at offset +8.[88] In line [8] we complete the line by prepending the header.[89]

[87] We also keep a record of what we’ve done in LOGFILE, for debugging purposes.

[88] This convention, about which I agonized a lot — it may well be inappropriate — allows us to sort rows and search on them only accessing the header of each row.

[89] The subsequent line of code is rather clumsy.

10.4.7 FormatDatum

Here we format all data according to our own peculiar internal SQL conventions on the PDA. We initially check the datum type (dtype) as well as the second argument which we load straight into $_.

```
sub FormatDatum
{ my ($dtype);
     ($dtype, $_) = @_;
     if (! defined $_)
        { return "";
        };
     if (length $_ < 1)
        { return "";
        };
```

Now let’s examine the response to each datum type. First numeric:

```
  if ($dtype eq ’N’)

     { /(\d+)\.?(\d*)/;
       if (! defined $2)
          { $_ = $1;
          } else
          { $_ = "$1$2";
          };
     }
```

For a fixed point number we require the precision and scale. We pull out the parts before and after the period. The above routine assumes that the number is already formatted according to the required precision and scale, so that there is a correct number of digits after the point.[90] The number is right aligned, with no leading zeroes. Next, a date:

```
  elsif ($dtype eq ’D’)      # date is YYYY-MM-DD --> YYYYMMDD
     { $_ = &SqueezeDate($_);
     }
```

All we do with a date is clip the hyphens out of the assumed YYYY-MM-DD to create YYYYMMDD. A time is similarly squeezed:

```
  elsif ($dtype eq ’T’)
     { $_ = &SqueezeTime($_);
     }
```

[90] Check this for the various databases, one or more commercial databases may screw this up?

The squeezed time will always have twelve digits, the last six for the fraction after the period. See SqueezeTime for details. Next a timestamp:

```
elsif ($dtype eq ’S’)
   { my ($dt, $tm);
     /(.+) (.+)/;
     $tm = $2;
     $dt = &SqueezeDate($1);
     $_ = &SqueezeTime($tm);
     $_ = "$dt$_";
   }
```

The timestamp ends up with eighteen ‘squeezed’ digits.[91] The varchar (character varying) field is easy as we do precisely nothing:

```
elsif ($dtype eq ’V’)
   {
   }
```

You might think we need to verify the length, but long strings will have already been truncated or otherwise abused by the database! Penultimately we have floating point numbers:

```
elsif ($dtype eq ’F’)
   {
      $_ = pack ("d", $_);
      my($i0, $i1);  # [9]
      ($i0, $i1) = unpack( "L2", $_);
      if ($BIGENDIAN)
         { $_ = pack ("N2", $i0, $i1);
         } else
         { $_ = pack ("N2", $i1, $i0);
         };   # [check me!?]
   }
```

In the above the Perl L2 unpack instruction unpacks two unsigned longs

Perl uses double precision, so we seem safe if we pack using the Perl pack command. Remember that certain databases may not necessarily conform to IEEE 754 floating point double precision, however.

There is another wrinkle. If our machine is 8086-based, then Perl will store the float as little-endian. We want big-endian, which is our invariant convention, so we use the trickery in line [9] onwards.

The last option (for now) is a (nearly) 4-byte integer:[92]

[91] This storage is far from economical, as we could easily use BCD, for example, but we don’t.

[92] We limited integers to the range 0–999 999 999, remember?

```
  elsif ($dtype eq 'I')
     { $_ = &Print4($_);
     }
```

In our restricted database, we limit keys to integers in the range zero to a billion minus one. If all else fails, we print a log message; we then exit, returning the value in $_.

```
  else
     { print LOGFILE
             "\n Bad datum type: $dtype ($_)";
     };
return $_;
}
```

10.4.8 SqueezeDate

Here's the first of the squeezing routines. We trim the dashes out of a date:[93]

```
sub SqueezeDate
{ ($_) = @_; # redundant.
  my ($y,$m,$d);
  if (! /(\d{4})-(\d{1,2})-(\d{1,2})/ )
          { print LOGFILE "\n Bad date <$_>";
            return "";
          };
       $y = $1;
       $m = $2;
       $d = $3;
       if (length $m < 2)
          { $m = "0" . $m;
          };
       if (length $d < 2)
          { $m = "0" . $m;
          };
       $_ = "$y$m$d";
  return $_; # redundant.
}
```

10.4.9 SqueezeTime

Similarly for time:

[93] Hideous code, I'm afraid!

```
sub SqueezeTime
{ ($_) = @_;
  my ($h, $m, $s, $f);
  if (! /(\d{1,2}):(\d{1,2}):(\d{1,2}).?(\d{0,6})/ )
        { print LOGFILE "Bad time <$_>";
          return "";
        };
  $h = $1;
  $m = $2;
  $s = $3;
  $f = $4;
  if (length $h < 2)
     { $h = "0" . $h;
     };
  if (length $m < 2)
     { $m = "0" . $m;
     };
  if (length $s < 2)
     { $s = "0" . $s;
     };
  while (length $f < 6)
     { $f = "$f" . "0";
     };
  $_ = "$h$m$s$f";
  return ($_);
}
```

10.4.10 MakePalmDBHeader

This routine rather carefully writes a PDB file header. Note that the format is that specified by PalmOS for PDBs on the desktop. The internal format used on the Palm PDA itself is their proprietary format, and may differ![94]

The PalmOS header must not only contain a whole lot of Palm stuff, but also refer to the offsets of our SQL 'header' line, as well as specify the offset of each of the records in our SQL file. We therefore submit three parameters: our header, the name of the table, and an array of records. The data lines are already sorted by primary key. We've broken the code into three:

First we determine the number of records (clumsily) and create the PalmOS header:

```
sub MakePalmDBHeader
{ my ($myhdr, $tname, @myrecs);
     ($myhdr, $tname, @myrecs) = @_;
  my($reccount);
```

[94] But we don't have to worry about this!

```
$reccount = 1 + $#myrecs;
$reccount ++; # +1 for header column
my ($ph, $strlen);
  $strlen = 32 - (length $tname);
  $ph = sprintf "$tname";        # a. name:
  $ph = $ph . &Print0 ($strlen); # pad with hex zeroes
  #    b. +20h attrib+version (2 bytes each)
  $ph = $ph . &Print4 (0);
  #    c. +24h creationDate
  $ph = $ph . &Print4 (1);
  #    d. +28h modificationDate
  $ph = $ph . &Print4 (1);
  #    e. +2Ch lastBackupDate:
  $ph = $ph . &Print4 (0);
  #    f. +30h modificationNumber
  $ph = $ph . &Print4 (0);
  #    g. +34h appInfoID
  $ph = $ph . &Print4 (0);
  #    h. +38h sortInfoID
  $ph = $ph . &Print4 (0);
  #    i. +3Ch type
  $ph = $ph . sprintf ("%s", ’DATA’);
  #    j. +40h creator
  $ph = $ph . sprintf ("%s", ’JoVS’);
  #    k. +44h uniqueIDSeed
  $ph = $ph . &Print4 (0);
  #       +48h normally zero
  $ph = $ph . &Print4 (0);
  #       +4Ch number of records (UInt16)
  $ph = $ph . &Print2 ($reccount);
  #       +4Eh         +6
  # RECORD ENTRIES GO HERE...
```

Notes:

- We should probably check that the length of tname is under 32 bytes, and otherwise truncate;
- All dates are ‘seconds after 12am 1 Jan 1970’;
- Only lastBackupDate is allowed to be zero (on pain of pain)!
- Avoid using appInfoID and sortInfoID
- ‘JoVS’ is our arbitrary code (We should register this with PalmOS);
- uniqueIDseed should be cleared to zero

The number of records at +4C is followed by record entries of the 'RecordEntryType' format. This format comprises a four byte (dword) offset of the raw record data, measured from the start of the PDB file, and four bytes all of which we reset to zero.[95] The first record entry is slightly special, referring to our 'own' SQL header:

```
  my ($uptotop);  # offset of 1st record from start of PDB
  $uptotop = 0x4E + (8*$reccount);  # 8 bytes per record

  # first record is distinct, OUR header:
  $ph = $ph . &Print4 ( $uptotop );
  $ph = $ph . &Print4 (0); # 0 attrib + uniqueID
  $uptotop += length $myhdr; # next offset..
```

Next, fill in pointers to all of the remaining records:

```
  my($c);
  $c = 0;
  print LOGFILE ("\n Creating $reccount-1 records: ");
  while ($c < ($reccount-1)) # -1 as header already done
    { $ph = $ph . &Print4 ( $uptotop );
      print LOGFILE (" $c:$uptotop "); # debug
      $ph = $ph . &Print4 (0); # as before
      $uptotop += (length $myrecs[$c]);
      $c ++;
    };
return $ph;
}
```

We decrease the count by one in the above because reccount includes the header. At the end, we simply return the completed header.

10.4.11 Print0

This trivial routine creates a string made up of the required number of hexadecimal zeros.

```
sub Print0
{ my ($len, $s);
     ($len) = @_;
     $s = "";
     while ($len > 0)
       { $s = "$s" . sprintf ("%c", 0);  # clumsy
         $len --;
```

[95] PalmOS fills these in.

```
       };
 return $s;
}
```

10.4.12 Print4

An inelegant routine which prints an integer as a 4-byte hexadecimal number. Formatting is always big-endian.

```
sub Print4
{ my ($i, $s);
     ($i) = @_;

  my ($n1, $n2, $n3, $n4);
  $n1 = $i % 256; #modulo
  $i /= 256;
  $n2 =  $i % 256;
  $i /= 256;
  $n3 =  $i % 256;
  $i /= 256;
  $n4 =  $i % 256;

  $s = sprintf ("%c%c%c%c", $n4, $n3, $n2, $n1);
  return $s;
}
```

10.4.13 Print2

Similar to Print4, but prints two bytes, also in big-endian format.

```
sub Print2
{ my ($i, $s);
     ($i) = @_;
  my ($n1, $n2);
  $n1 = $i % 256;
  $i /= 256;
  $n2 =  $i % 256;
  $s = sprintf ("%c%c", $n2, $n1);
  return $s;
}
```

We might simply say “`$n2 = $i`”, if we were certain the submitted number was in range.