PainForm: an Analgesia Database

The Perl Program

Version 0.95

J.M. van Schalkwyk

February 27, 2009

Contents

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 document 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]

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] We used version 5.6. 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!

1.2 A core user interface

The current main window created when the Perl program is run is sparse, simply providing the ability to enter a Perl simulation of the PDA interface, Synchronise the PDA with the ‘desktop’ database, and enter an administrative menu.

The administrative menu has the following functions:

1. Initial creation of the database and menu system;
2. Adding, activating and de-activating users of the system;
3. Primitive back-up and restore functions (in addition to whatever backup is being used with the chosen database);
4. A facility to export a copy of the PalmOS image of the program (and data) to a chosen directory.

1.3 Why not the iPhone?

The answer is here.

2 Initialisation

We describe initialisation under Windows; some modification will clearly be required on a Linux system. As we use mainly Windows 2000, we will describe use of this platform as an example. We have tried to avoid peculiarities of the Windoze operating system wherever possible, but sometimes the extreme functional limitations of Windows (in combination with Palm) obtrude!

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 `pain2.pl` file. We use the directory `\painform`. Within that directory, create a subdirectory called `data`, and populate it with the following files:

- constants.const
- MENUS.sql
- META.sql
- SETUP.sql
- XLOG.LOG
- XLOG.LST

One way of obtaining the files is to download them from our website,[8] and move them to the `data` subdirectory.

A better source of the above files is to extract them from the documentation of our program, in keeping with the 'DogWagger' principle! See the source files *AnalgesiaDBpart1.tex* and *AnalgesiaDB2.tex*, which are now the primary and preferential sources of the above files.

[6] Or you might wish to modify things youself to run under e.g. Linux/PostgreSQL, if you're a whiz kid! An even better idea under Windows or Linux might be MySQL.

[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

There is one apparent exception: the file *XLOG.LST* can be manually created from *XLOG.LOG*, which is generated on actual creation of the database. This should be done (manually, eugh) if you alter the structure of the database.[9] We provide both files in unmodified form in *PerlPgm.tex*.

2.1 Directory structure

Important directories are:

1. sync
2. data
3. log
4. Peculiar to development are sql3; numeric; scripting; console; osbox;
5. export
6. import

Now let's look at the actual source code for the Perl program.

2.2 Packages

The following code is far from elegant. Contemporary programmers will probably be more-or-less horrified at the plethora of global variables we use. My feeling is that there is a certain irreducible complexity and that, whatever way you try to hide our mask things, this complexity will out.[10] The nastiest part of the whole Perl/tk program is the first few pages. Note that the +OPTIONAL and -OPTIONAL commands are not part of the Perl, but of the DogWagger program that generates the final Perl from the .TEX source file![11]

We start the Perl program with some pretty trivial stuff:

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

[9] Experts only!

[10] I do however admit that most or all of the following code could withstand a lot of sprucing up!

[11] These are the only visible attributes of this approach within the PDF documentation.

```
require Tk::Font;
use Digest::MD5 'md5_hex';    # 2008-10-7

# use Date::Calc qw(:all);
use File::stat;
use Time::localtime;
use POSIX qw(floor);
use UNIVERSAL qw{isa}; # to test for object (ugh)
# if the following isn't installed, install or remove reference to gettimeofday la
# use Time::HiRes qw( gettimeofday tv_interval );

my ($ISPROGRESS) = 1;
# used in concert with OPTIONAL

+OPTIONAL
use Tk::WaitBox;
use Tk::ProgressBar;
-OPTIONAL
```

We use the Tk package (and specific components) extensively. In a past version we used the Date::Calc package within a few minor routines but this requirement has been removed. The `Time::localtime` overwrites the normal Perl `localtime` functionality, which can still be accessed using the `CORE::` pseudopackage. POSIX is included simply for the floor function used in date calculation.

2.3 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:

2.3.1 The log file(s)

```
my $TODAY = &GetLocalTime();
   print LOGFILE "\n TODAY: $TODAY\n"; # altered to a LOG print v 0.95 18/3/2008
```

```
 my $ENTRYTIME = $TODAY;
    $ENTRYTIME =~ s/-//g;  # get rid of dashes
    $ENTRYTIME =~ s/://g;  # and colons
    $ENTRYTIME =~ s/ /-/g; # replace space with dash.
  my $logfile;

if (! &ValidatePath('log'))
   { die "Directory 'log' does not exist";
   };
  $logfile= "log/EDLOG" . "$ENTRYTIME.LOG";
open LOGFILE, ">$logfile" or die
     "*CRASH* Could not open LOG $logfile :$!\n";
# v0.95 we've moved sync file write to log/ directory!

  my $SYNCFILE = "log/SYNC"."$ENTRYTIME.LOG";
open SYNCFILE, ">$SYNCFILE" or die
     "*CRASH* Could not open synchronisation file $SYNCFILE! \n$!\n";

  my $sqllog = "log/$ENTRYTIME.SQL"; # v0.95
  open SQLLOG, ">$sqllog" or die
     "\n Failed to open SQL log $sqllog";
```

Largely trivial. Open up a file to write errors and stuff to. Append `EDLOG.LOG` to its unique name. Then create a similar, uniquely named log file for synchronisation data within *sync*, a subdirectory of the current directory. The program will fail if this subdirectory doesn't exist, so it should be created beforehand.

2.3.2 Constants

```
my %CONST;
my %PTWARDS;  # only used in printing
my %WDLIST;   #   summary data (ugh)!

&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 in Section 4.1. You may wish to browse through this section and then return here.

2.3.3 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 $BIGFONT;
my $METASTORE = 0;
my @MARKERS;
my $MARK = 0;
$MARKERS[$MARK] = -1;
my $Stopped=0;
my $FAILURE = 0;
my $UIDBASE = 900_000_000;
my $UIDINCREMENT = 10000;
my $DebugRECNO = 0;
my %KEYHASH = ();
my @RETAINEDPROCS = ();
my @TABLEDATA = (); # clumsy global for LaTeX table generation!

$FRED = '1';
$SQLOK = 0;      # clumsy global hack
my $ISMENU = 0;  # has &GoMenu been used?
my $ISDB = -1;

my $ADMINMENU;
my $USERMENU;
my $MAKEBUT;
my $FORENAME;
my $ROLE;
my $USER2HIDE;
my $USER2FIND;
my $SURNAME;
my $USERLOGIN; # for setting user log-in name
my $PWDMENU;   # ugh. Alter password.
my $OLDPWD;
my $NEWPWD1;
my $NEWPWD2;

my $LOCALTIMEOFFSET = 12;
my $DAYLIGHTSAVING = 0;
my $EPSILON = 1e-6;
my $BACKGRND = '#8C8C8C';
```

```
my $SYNCBUTTONCOLOUR = '#F0C0C0';
my $TOGGLED = 0; # see usage
```

LOCALTIMEOFFSET depends on your time zone (12 is for New Zealand), and we assume that DAYLIGHTSAVING is correctly set to zero or one (unwise), something which isn't yet performed in our Perl program. Both affect Julian date calculations based on local time. EPSILON is used in fixing up integer conversions!

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. UIDBASE sets the startup value for temporary keys used in tables on the PDA (Section 16.4.7); UIDINCREMENT is the associated separation distance between table keyspaces, and DebugRECNO is used to count records imported. Use of the associative array KEYHASH is described in Section 17.2.1.

IAM, KEPTIAMS and INSCRIPT are very experimental, are used for communication between items in a menu, and 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 is a flag which determines, as we create tables, whether we record information about their creation in 'meta tables'![12]

MARK together with MARKERS keeps track of stack markings. We start off with the first marker at the bottom of the stack, and as we mark so the index $MARK increases, and the MARKERS stack grows commensurately. $Stopped is used to test whether a recent STOP has occurred.

$ADMINMENU ... $SURNAME are concerned with an administrative menu we later create.

2.3.4 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 )
   { $BIGENDIAN = 0;   # on WIN-DOwS system, should be zero.
     print "\n Little-endian";
   } else
   { $BIGENDIAN = 1;
```

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

```
    print "\n Big-endian";
  };
```

2.4 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 in the LoadConstants routine (4.1).

```
my $myODBC;
  use Win32::ODBC;
  $myODBC = new Win32::ODBC($CONST{'PAINDATABASE'});
  unless ($myODBC->Connection)
    { die "*CRASH* Failed to connect. Dearie me!\n";
    };
  print "\n ODBC: connection worked\n";
  $myODBC->SetMaxBufSize(500000); # set buffer size
```

2.5 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 = 0x18;  # 0x10 debugs stack..
   $BUG = 0x48;
  # we will use flag 64 (0x40) to dump @TKVALUES at the end of menu creation!

 $myODBC->Debug($DEBUG);
 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 a rather clumsy 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[13] Perl/tk feature that as we move the mouse around, so the widget focus follows it without any clicking! The initial window is on the right side of the screen, so that on the left we can pop up a replica of the PDA screen without any fiddling.

```
my $MAINW = new MainWindow;
   $MAINW->geometry('400x320');  # dimensions
   $MAINW->geometry('+350+20');   # screen offset!
   $MAINW->title('Pain Database 2005+');
   $MAINW->configure( -background => $BACKGRND );
   $MAINW->focusFollowsMouse;
   # ? $MAINW->focusForce;

$ISDB = &IsDbMade($myODBC); # also check if pain db made?
```

3.2 variables

We also check that the pain database is in existence (IsDbMade) as this information alters our subsequent display! A whole host of variables, arrays and so forth follows. First time readers may wish to skip to section 3.3.

```
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 $TKTOGGLE = 0; # toggles between 0 and 100
my $DEADWINDOW = 0;
my $ICOUNT = 0;    # topmost Tk menu item (TKITEMS)
#my $OLDICOUNT = 0;

my @VVALS;    # specific to V
my @POPVALUE; # for poptriggers
my @POPARRAYS; # array of hashes!
```

[13] And necessary. Try disabling this feature to see the problem!

TKITEMS is a record of all widgets created within Tk. We also retain associated values in TKVALUES. The table VVALS allow the V command (10.6.2) access to row values from within scripts.[14] 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). A recent modification (perhaps even more clumsy) is to store the initialisation array for the poptrigger in a separate 'array of hashes' called POPARRAYS.[15]

Next, let's look at several other important globals (eugh). The BURYSTACK is new, in keeping with our 'separate' use of a burial stack on the PDA.[16]

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

$XPARAM    = 0;
$NEWXPARAM = 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.

ROLLOFFSET is interesting — whenever we enter a menu (GoMenu), we push a zero to this 'stack'. Associated are $LOCALROLL (which counts the number of items displayed in a polymenu iff there are still more lines), and LINESLEFT, which is the count of remaining (undisplayed) lines. If during the construction of that menu, we encounter a polymorphic table[17] with more rows than can be displayed, we replace the most recent zero on ROLLOFFSET with the offset of the

[14] This coding is ugly and inefficient, and should be fixed!

[15] This allows us to store *pairs* of items; when a poptrigger value is selected, we can look up the corresponding number, and return this. Of immense value in identifying items within the database based on user selection!

[16] Previously we used to unshift/shift to head of CMDSTACK, but this is unwise.

[17] What is a 'polymorphic table'? See section 8.

first undisplayed row. If and only if we invoke the ROLLMENU function within this menu, then we reload a copy of the current menu on top of the current one, drawing the new polymorphic table rows starting at the specified offset!

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 MENU button which opens up our ‘PDA simulation’. We also toss in a new font (ROOTFONT). First, three frames:

```
my $MENUW;

my $topFrame =$MAINW->Frame();
my $midFrame =$MAINW->Frame();
my $bottomFrame = $MAINW->Frame();
```

Now, the patients’ menu button, to which we attach the GoMenu command (Section 6).

```
my $menuB = $topFrame->Button(
            -text => ’PATIENTS’,
            -command => [ \&GoMenu, $myODBC, ’MAIN’, $MAINW, 0]  );
   $menuB->configure(-width => 10);
   $menuB->pack(-side => ’left’,
                -expand => 1,
                -ipady => 10,
                -padx => 10,
                -pady => 3);
```

Administrative matters, explained in section 7 . . .

```
my $newBut = $topFrame->Button(
             -text => "Admin.",
             -command => [ \&DoAdmin, $myODBC] );

   $ROOTFONT = $newBut->fontCreate( ’fred’,
             -family => ’Helvetica’,
             -size =>7);

# next a button for the daily report:
   my $dailyRep = $topFrame->Button(
                  -text => ’Daily report’,
                  -command => [ \&PrintBasicData, $myODBC, $CURRENTUSER, $MAINW ]
   $dailyRep->pack (-side => ’right’,
                -ipady => 10,
                -padx => 10,
```

```
                -pady => 3);

# administration menu access:
   $newBut->configure(-font => $ROOTFONT);
   $newBut->pack(-side => 'right',
                 -padx => 10,
                 -expand => 1);

   $topFrame->pack(-side => 'top', -ipady => 10);
   $topFrame->configure( -background => $BACKGRND );
```

...and finally a button to synchronise the PDA with the desktop database (Section 17.2.1), as well as a button to exit.

```
my $ResynchBut = $midFrame-> Button(
              -text => 'SYNCHRONISE PDA',
              -command => [ \&ResynchFromPda, $myODBC ] );
   $ResynchBut->configure(-width => 20);
   $ResynchBut->configure(-background => $SYNCBUTTONCOLOUR);
   $ResynchBut->pack(-side => 'top', -ipady => 18, -pady => 20);


# check user:
my ($thsusr) = $ARGV[0];
my ($ucode) = &GetSQL($myODBC,
  "SELECT person FROM PERSON WHERE pUsername = '$thsusr'");
if ( ($ucode < 50)
   ||($ucode > 999)
   ) { &Alert($MAINW, "Warning. Unknown user '$thsusr'");
     } else
     { if ($ucode == 121) # sysadmin (me)
          { $ucode = 1;  # (keep)
          };
       $CURRENTUSER = $ucode;
       print LOGFILE "\n Logging on $thsusr ($ucode)";
     };


# ---------- testing a progress bar:

my $BARWIDTH = 200;
my $BARPROGRESS = 0;
my $PROGRESSBAR = $midFrame->ProgressBar(-width => 20,
                                  -from => 0,
                                  -to => $BARWIDTH,
                                  -blocks => 20,
                                  -foreground => 'red',
                                  -variable => \$BARPROGRESS
```

```
                                  )->pack(-side => 'bottom');
my $PROGTEXT = '';
my $ProgText = $midFrame->Label(-textvariable => \$PROGTEXT
                                )->pack(-side => 'bottom');
  $ProgText->configure( -background => $BACKGRND );
  # why does -background option not work??

  $midFrame->pack( -ipady => 10);
  $midFrame->configure( -background => $BACKGRND );

# -------------------
my $quitBut = $bottomFrame->Button(
            -text => 'Quit',
            -command => [ \&DoQuit, $myODBC ] );
   $quitBut->pack();

   $bottomFrame->pack(-side => 'bottom',
                      -ipady => 10,
                      -fill => 'both');
   $bottomFrame->configure( -background => $BACKGRND );

   if (! $ISDB) # if database doesn't exist
      { $menuB->configure (-state => 'disabled');
      };

MainLoop;

# for MainLoop in Perl/Tk see: http://oreilly.com/catalog/mastperltk/chapter/ch15.

# ---------- hook for progress bar:
sub TestProgressBar
{ if ($BARPROGRESS<$BARWIDTH)
     { $BARPROGRESS++;
+OPTIONAL
       $PROGRESSBAR->update();
-OPTIONAL
     } else
     {
+OPTIONAL
       $PROGRESSBAR->destroy if Tk::Exists($PROGRESSBAR);
-OPTIONAL
     };
}
```

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, most important of all is:

```
[ \&GoMenu, $myODBC, 'MAIN', $MAINW, 0]
```

See how we submit a handle to the database myODBC,[18] as well as the name of the first menu ('MAIN'), and of course, our current window which will be the parent of subsequent windows.

3.3.1 Exit

Here's the trivial exit routine:

```
sub DoQuit
{
   my ($myODBC);
      ($myODBC) = @_;

   if ($ISMENU)
      { my $i = &Confirm ($MAINW,
  "ABORT? Are you sure? \n(Changes will NOT be saved!)");
        if (! $i) { return; };
      };

  $myODBC->Close();
  close SQLLOG; # v0.95
  close LOGFILE;
  close SYNCFILE;
  print "\n Normal exit";
  exit;
}
```

[18] 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 2.3.2 above.

```
sub LoadConstants
{ my ($CONSTFILE, $ok, $key, $value);
  $ok = 1;
  $CONSTFILE = "data/constants.const";
  if (! &ValidatePath($CONSTFILE))
   { die "file $CONSTFILE does not exist";
   };

  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 (  ( ! /^\%/ ) # not a comment
               &&(length $_ > 2) # not tiny
              )
              { 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;
}
```

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 in section 4.2.1.

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.[19] We ignore lines which are very short (zero or one character) and those which start with a percentage sign, as this character is used to signal comments. The first line of the file is also not parsed.

We can now say e.g. `$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.

If you've jumped down to this section from above, click here to return. For details of the *constants.const* file and its contents, see Section 18.6.

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) = @_;
```

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

```
   print LOGFILE $fred; # redirect
}
```

We use this rather than the standard Perl `print`, as it permits us to flexibly redirect printing.[20] Here we direct 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 ++;
     };
}
```

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);
}
```

4.2.5 Ask

```
sub Ask
{ my ($win, $title, $default);
```

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

```
    ($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.

4.3 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. Even ConvDate is at present not used.

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).[21]

[21] Probably a silly decision!

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) = CORE::localtime(time);
  $year += 1900;      #fix y2k.
  $sec  = &DoubleDigit($sec );
  $min  = &DoubleDigit($min );
  $hour = &DoubleDigit($hour);
  $mday = &DoubleDigit($mday);
  $mon  = &DoubleDigit($mon );
  $mon ++;            #january is zero!
  return ("$year-$mon-$mday $hour:$min:$sec");
}
```

We use the Perl function ‘localtime’ to obtain these values, returning a text datestamp. Because the `Time::` package inconveniently overwrote this, we have to access it using the `CORE` facility.

4.4 Directory handling

It’s good to be able to check the validity of a path to a file. The ValidatePath routine does just that.

```
sub ValidatePath
{
  my @pth;
  my $wholep = "";
 ($_) = @_;

  if ( /^\/(.*)$/ ) # if starting slash
     { $wholep = "/";
       $_ = $1;
     };
  @pth = split /\//;
#  print "\n Debug: path components @pth";

  my $p;
  foreach $p (@pth)
    { $wholep = "$wholep$p";
      if (! (-e $wholep) )
         { # print "\n Debug: bad path $wholep";
           return 0; # fail
```

```
        };
      $wholep = "$wholep/";
    };
  return 1;
}
```

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. The DoSQL routine is a hack which splits up multipart insert statements, otherwise reverting to a former routine (Do2SQL).

```
sub DoSQL
 { my ($myODBC, $SQLstmt, $bugstmt);
      ($myODBC, $SQLstmt, $bugstmt) = @_;

  if (! ( $SQLstmt =~
  /(insert\s+into\s+\w+\s*\(.+\)\s*values\s*)(\(.+\)\s*,\s*\(.+\))/i ))
     { return (Do2SQL ($myODBC,$SQLstmt, $bugstmt));
     };
   my ($preamble, $rest);
   $preamble = $1;
   $rest = $2;
   # &Alert($MAINW, "Debugging: <$rest>"); # strictly debug
   while ( $rest =~ /(\(.+?\))\s*,\s*(\(.+\))/ )
     { $rest = $2;
       # &Alert($MAINW, "Debug executing: <$preamble $1>"); #debug
       Do2SQL ($myODBC, "$preamble $1", $bugstmt);
     };
   return (Do2SQL ($myODBC, "$preamble $rest", $bugstmt)); # the final stmt
 }

sub Do2SQL  # cf SQLRECENTFAILURE.DATA
 {
   $SQLOK=0; # default 'fail'
   my ($myODBC, $SQLstmt, $bugstmt);
      ($myODBC, $SQLstmt, $bugstmt) = @_;

  $_ = $bugstmt;
  if (/^\*/) # if annotation forces debug!
```

```
     { # print LOGFILE "\n\n Debug SQL $_:\n$SQLstmt\n" ;
     };
  if ($BUG & 4) # if debugging
    { print SQLLOG "\n $SQLstmt;";
    };

  my ($retcode);
  $retcode = ($myODBC->Sql($SQLstmt));
#  print LOGFILE "-";
  if ($retcode) # if problem
     { 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 = $myODBC->Error();
         if ($ISDB != -1) # if not busy kicking off..
            { &Warn ($myODBC, 4,
                     "SQL ret code 'retcode' \n<<$SQLstmt>>");
              if ( $sqlErrors !~ /\[911\].+\[1\].+\[0\]/ ) # NOT if 'no data' [911
                { print LOGFILE "Error message: \"$sqlErrors\"\n\n" ;
                  &Print ("\n WARNING: There was an SQL problem!\n\
                     \nSQL return code 'retcode' \n\
                     SQL STATEMENT\n:<<$SQLstmt>>\n");
                  &Alert($MAINW, "SQL error. See EDLOG.LOG");
                  # here try to keep just the most recent actual stmt:
                  my ($mostrecent) = 'SQLRECENTFAILURE.DATA';
                  open MOSTRECENT, ">$mostrecent"; # ??? is 'or die..' of any valu
                  print MOSTRECENT $SQLstmt;       # keep record
                  close MOSTRECENT;
                };
            };
       };
       $retcode = -1;
     } else  # no problem, $retcode zero
     { $SQLOK = 1;
     };
 $retcode; # 0=ok
 }
```

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.[22]

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.

Time for true confessions! In my first reading of the ODBC documentation, I not only failed to see that there are several different ways you can retrieve data, but I also didn't realise that you can retrieve an *array* with each `->Data` fetch! This embarrassing mistake resulted in a lot of turgid code, the legacy of which is with me to this day.

Okay, here's the slightly better code.[23] Given a database handle, SQL query and a comment (tag), return an array of data items. If there are n rows and m items in each row, we return one big array containing $n * m$ items. To extract each row, pull m items off either end of the returned array, and repeat.

```
sub SQLManySQL
 { my ($myODBC, $SQLstmt, $tag);
      ($myODBC, $SQLstmt, $tag) = @_;
  DoSQL ( $myODBC, $SQLstmt, $tag );
  my(@bigarray) = ();
  my(@big);

  while ( $myODBC->FetchRow() )     #
       { @big = $myODBC->Data();     #get data
         @bigarray = (@bigarray, @big);
       };
  if ($BUG & 4)
     { print SQLLOG "\n-- data {@bigarray}";
     };
  my($itmz);
  $itmz = $#bigarray;
  if ($itmz <= 0)
     { if ($itmz < 0)
          { $SQLOK = 0;   # signal 'problem'
          } else
          { if ((length $bigarray[0]) == 0)
               { $SQLOK = 0;
```

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

[23] Really cute would be to look into using associative arrays.

```
                @bigarray = ();
    };   };   };
  return( @bigarray);
 }
```

There is one remaining issue. ‘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.[24] 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); # 'not exit'
  $nxit = 1;
  while ($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
    };  };
 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 `--`.[25] A line without any characters in it is ignored and the next line is then fetched.

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

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

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 submitted *filename* lacks the '.SQL' suffix that the actual file has, and that the file itself 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";
  if (! &ValidatePath($SQLFILE))
   { &Alert($MAINW, "Error: file $SQLFILE does not exist");
     return 1; # one error
   };
  print "\n Running batch file: $SQLFILE";

  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!
        { $longline = "$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, we *should* be able to say something along the lines of
`$myODBC->Transact("SQL_COMMIT");`
. . . but this doesn't work as such[26], so we used 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.

In retrospect, there are several issues here. The first is that not all ODBC drivers support `Transact`. We might (but don't yet) check for this functionality along the lines of:

```
%Functions = $db->GetFunctions();
if( $Functions{$db->SQL_API_SQLTRANSACT()} )
{ print "\n Transact() supported";
}
```

Also note that the correct way to refer to constants is thus:
`$db->Transact( $db->SQL_COMMIT )`
because ODBC doesn't automatically import all of the hundreds of SQL constant values!

[26] My error. Fix me!

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, (PDA implementation details Part I) 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 access to the generator table without collisions.[27] Here's our key generator code, which uses the generator table UIDS...

```
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!
```

[27] 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.

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

  $METASTORE = 1;
  $ocm = BatchSQL ("specific", $myODBC); # ward/bed setup..
  $METASTORE = 0;

#  # --- START HACK --- #
#  my (@rooms);
#  (@rooms) = &SQLManySQL ($myODBC,
#     "SELECT room, Ward, srmText FROM ROOM",
#     "get list of all room codes and names");
#
#  my($id, $room, $ward);
#  while ($#rooms > 0)
#    {
#      $id = shift(@rooms);
#      $ward = shift(@rooms);
#      $room = shift(@rooms);
#      &DoSQL ($myODBC,
#        "INSERT INTO BED (bed, Room, sName) \
#           VALUES ($id" . "00, $id, '$ward/$room" . "--')",
#        "create a new bedspace");
#    };
#  # --- END HACK --- #

  &Commit($myODBC);

#  &ColdFields($myODBC); # removed
#  &Commit($myODBC);     #

  Alert( $ADMINMENU,  "\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. The pattern of the room creation is based on the following SQL ‘prototypes’:

```
  INSERT INTO ROOM (srmID, srmText)
    VALUES (3101, '31/1');
  INSERT INTO BED (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 was in the development phase, as for obvious reasons it is not advisable to have this functionality in a 'production' version. We have deactivated and removed this legacy.[28]

```
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";
}
```

5.2.3 Is database made?

We query the database to see whether [at present] the UIDS table yields a uKey value of 1. If it doesn't (or the query fails) then the database hasn't yet been created.

```
sub IsDbMade
{
  my ($myODBC);
     ($myODBC) = @_;
```

[28] In fact, we now just delete the whole directory contents from a DOS batch file, should we wish in development to kill the database!

```
  my ($ok) = &GetSQL ($myODBC,
              "SELECT uKey FROM UIDS WHERE uKey = 1",
              "or fail");
  if ((! $SQLOK) || ($ok != 1))
     { return 0;
     };
  return 1;
}
```

Note that by putting -1 into the global variable ISDB before invoking IsDbMade, one can suppress the otherwise inevitable warning that things didn't work.

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!
  $METASTORE = 1;
  $ocm =  BatchSQL ("MENUS", $myODBC);
  $METASTORE = 0;
  $ocm = "\n Menu creation: $ocm";
  print $ocm;

  &Commit($myODBC);
  print "\n Menus created!";
  Alert( $ADMINMENU, "\n Menus created.");
  $menuB->configure (-state => 'normal');
}
```

In fact, we now combine MakeMenus and MakeDB thus:

```
sub MakeDBandMenus
{
  my ($myODBC);
     ($myODBC) = @_;

  &MakeDB();
  &MakeMenus();
  &ReadAllCSV($myODBC);
```

```
  $MAKEBUT->configure (-state => 'disabled');
}
```

. . . and attach this routine to a single, clumsy, global button MAKEBUT. Once the menus are made, then the button is configured to the disabled state.

5.3.2 ClearMenus, MakePeople, ClearPeople, and KillPeople

The following legacy code has all been removed from the final Perl file:

```
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!!");
}


sub MakePeople
{ my ($ocm);         # outcome!
  $METASTORE = 1;
  $ocm =  BatchSQL ("PEOPLE", $myODBC);
  $METASTORE = 0;
  $ocm = "\n Patient creation: $ocm";
  print $ocm;
  &Commit($myODBC);
  print "\n People created!";
  Alert( $MAINW,  "\n People created");
}
```

The script CLEARPATIENTS is also a legacy. We formerly used it to 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.

5.4.1 QUERY

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

```
sub GetSQL
{ my ($myODBC, $SQLselect, $tagname);
      ($myODBC, $SQLselect, $tagname) = @_;
  $SQLOK = 1;
  if (&DoSQL ($myODBC, $SQLselect, $tagname))
     { return ""; # fail
     };
  my (@newrow);
  @newrow = ();

  if ( $myODBC->FetchRow() )
       { @newrow = $myODBC->Data(); #first data row
       } else
       { $SQLOK = 0;
       };
  if ($BUG & 4)
     { print SQLLOG "\n -- data {@newrow}";
     };
 return (@newrow);
 }
```

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

Initially I didn't properly read/understand the ODBC documentation, and therefore returned a concatenated string of values rather than an array, causing myself a lot of pain in the process!

5.4.2 QMANY

See the SQLManySQL section above (Section 5.1.2).

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 within 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+[^,]*),(.+)/ )
    { $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,[29] 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:

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

```
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 ( /^check\s+\((.+)\)/i )
          { $_ = $1;
            if ( /([^\s]+)\s+is not null\s*/i ) # ugly
               { $col = $1;
#                 print LOGFILE "\n   >CHECK !0: Name:$name Col:$col";
                  $chk = 'X';
               }
            elsif ( /([^\s]+)\s+is null\s*/i )
               { $col = $1;
#                 print LOGFILE "\n   >CHECK =0: Name:$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', $licol)",
               "record CHECK constraint"); # 'N'= is null, 'X' = not
               print ("+");
             } else
             { print ("\n ERR: Column not found A: <$col>");
             };
          } elsif (/^primary\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 = $licol",
                "adjust key type"); # our 1ary keys all type I (!!)
               print ("+");
             } else
             { print ("\n ERR: Column not found B: <$col>");
             };
          } elsif (/^foreign\s+key\s+\((.+)\)\s+references\s+(.+)/i)
          { $col = $1;
            $tbl = $2;
            if ( $tbl =~ /(\w+)\s+ON UPDATE CASCADE/ )
               { # if tablename + space + ON UPDATE CASCADE, then pull out table!
                 $tbl = $1;    # 21/5/2006 fix.
               };
#            print LOGFILE  "\n   >FOREIGN: Name:$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 = $licol",
                "adjust key type");
               print ("+");
             } else
             { print ("\n ERR: Column not found C: <$col>");
             };
          } else
          { print "\n UNKNOWN: <$_>";
          };
```

```
      } else
      { /\s*([^\s]+)\s+([^\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 =~ /^float$/i)
           { $len = 8;
             $prec = 0;
             $t = 'F';
           }  # havoc due to 'if' ipo 'elsif' in next line:
        elsif ($type =~ /^integer$/i)
           { $len = 4;
             $prec = 0;
             $t = 'I';  # added 2005/9/25
           }
        elsif ($type =~ /^date$/i)
           { $len = 8;
             $prec = 0;
             $t = 'D';
           }
        elsif ($type =~ /^time$/i)
           { $len = 6;   # [ NOT 12;at least for now]
             $prec = 6;              #
             $t = 'T';
           }
        elsif ($type =~ /^timestamp$/i)
           { $len = 14; # [NOT: $len = 20; at least for now]
             $prec = 6;             # ???
             $t = 'S';
           }
        elsif ($type =~ /^varchar$/i)  # ? CHARACTER VARYING?
           { $t = 'V';
           }
```

```
       elsif ($type =~ /^decimal$/i)
          { $t = 'N';
          } else
          { &Alert($MAINW, "PDB failure. ? type '$type': See EDLOG");
            print "\n BAD TYPE: '$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.

5.6 Cold cases

We need some method for distinguishing cold cases (no longer of interest to the person holding the PDA) from active cases. Initially I wrote routines to insert a 'cold' field in every table, simply not exporting these fields to the PDA, but the problem then becomes the routines which look at the wards for new cases on the desktop — either we have two desktop programs (which is hateful) or we need two sets of code, one looking at this field on the desktop, and the matching routine ignoring the field on the PDA, where the field doesn't exist.

Eventually I decided to simply have 'cold' as another database field everywhere, with no special privileges. We interrogate it on the PDA too (where we can't avoid this catch) but we can now easily check whether the cold flag is set, and if so not export a particular row to the PDA. See below.

Now we can also (on the desktop) check for a set cold flag on the desktop (in the BADOBS table) and simply not select these patients for display. Of course

our PDA program also checks for a null cold flag, but this is at present without function.

It might be wise to allow a patient who has been discharged to linger on the PDA for the next day, just in case they are needed.

5.6.1 Flagging all cold data

In the section after this one we discuss an alternative approach, where all closed processes are eliminated from subsequent PDA updates. The problem with such an approach is that we might also eliminate information on e.g. an epidural which has been removed but we still require on the PDA!

A better approach (perhaps!) is to identify all patients who still have active processes attached to them. We then flag all patients who do NOT have active processes, and then use this information to identify all processes which are inactive but not yet cold. Finally, we chill out all associated data.

```
sub MakeCold
{ my ($myODBC);
     ($myODBC) =@_;

  # wbd means 'warm but dead'
  my @coldtables;
  my @wbdprocs;
  my @wbdobs;

  (@coldtables) = &SQLManySQL ($myODBC,
     "SELECT xTaName FROM xTABLE, XCOLUMN
       WHERE xTaKey = xCoTable AND
       xCoName = 'Epoch'",
        "get tables referencing EPOCH");

  # 0. Determine all patient ids of wbd data:
  &DoSQL($myODBC, "UPDATE PERSON SET cold = 1 WHERE pStatus = 1 \
   AND cold is NULL AND Person NOT IN \
   (SELECT DISTINCT Person FROM PROCESS WHERE rEnd IS NULL AND Proctype = 3)",
   'chill unused patients');

  # 1. Flag all newly cold processes:
  &DoSQL($myODBC, "UPDATE PROCESS SET cold = 2 WHERE cold IS NULL \
   AND Person > 1000 \
   AND Person NOT IN \
 (SELECT DISTINCT Person FROM PROCESS WHERE rEnd IS NULL AND ProcType =3)",
   'chill wbd processes');
   # amended 2007-12-28: can turn off all by turning off Proctype = 3
   # allows for other 'orphan processes'. Hmm.
```

```
  # 2. render all wbd PROCESSes (rEnd is not null, cold is clear) cold:
  (@wbdprocs) = &SQLManySQL($myODBC,
   "SELECT process FROM PROCESS WHERE cold = 2", "get wbd procs");
   &DoSQL($myODBC,
   "UPDATE PROCESS SET cold = 1 WHERE cold = 2", "clumsy: make cold");

  #2. for each process thus chilled, chill the children (epoch)
  my $p;
  foreach $p (@wbdprocs)
    { (@wbdobs) = &SQLManySQL($myODBC,
      "SELECT epoch FROM EPOCH WHERE EPOCH.Process = $p",
      "get relevant observations");
      &DoSQL($myODBC,
      "UPDATE EPOCH SET cold = 1 WHERE EPOCH.Process = $p",
      "chill epoch");
      my $o;
      foreach $o (@wbdobs)
        { &ChillObsChildren ($myODBC, $o, @coldtables); # hideous
        };
      # the alternative for chilling EPOCH is to update epoch set cold = 1
      # where epoch in (<insert wbdobs here, comma delimited>). More
      # fancy and quicker options might be possible.

      #3. In a similar fashion, sort out Rx (using same $p):
       &DoSQL($myODBC,
      "UPDATE RX SET cold = 1 WHERE RX.Process = $p",
      "chill rx");
    };

  # Finally, if a cold process has no rEnd, set an artificial value:

  &DoSQL($myODBC,
    "UPDATE PROCESS SET rEnd = TIMESTAMP '1899-01-01 00:00:00' WHERE \
      rEnd IS NULL and cold IS NOT NULL",
    'hack bad proc ends');

  #4. commit:
   &Commit($myODBC);
}
```

Here's the subsidiary routine.

```
sub ChillObsChildren
{
  my ($myODBC, $o, @coldtables);
     ($myODBC, $o, @coldtables)=@_;

  my $t;
```

```
  foreach $t (@coldtables)
    {
      &DoSQL($myODBC, "UPDATE $t set cold = 1 WHERE Epoch = $o",
             'make row cold');
    };
  return;
}
```

5.6.2 Finding and flagging cold cases

The following approach is a simple one, which we've now *abandoned*! We do *not* use this process and it is included simply for interest's sake![30]

We first identify all processes which are *not yet* cold, but are terminated. We remember the process IDs, and then make these processes cold. We next find all tables referencing these processes (EPOCH and RX), and similarly cool these. The next level is to identify the welter of tables referencing EPOCH (ACTOR2, PERSDATA, MEDSCORE, SURGTYPEOB, NONEVENT, MEASURE, PAINSCORE, RXOBS, INFUSIONOBS, PCASETTINGS, PCA, RGNOBS, COMMENT, ISPROBLEM and BADOBS), and ensure that these tables in turn are cooled appropriately! It is possible that more such tables might exist in future editions, so we use our wonderful x-tables to identify references to *all* such tables.

A final requirement is that if a person (other than a staff member) isn't referenced by an active process, then they too should be made cold and not exported to clutter up the PDA.[31]

```
sub OldMakeCold
{ my ($myODBC);
     ($myODBC) =@_;

  # wbd means 'warm but dead'
  my @coldtables;
  my @wbdprocs;
  my @wbdobs;

  # 0. Determine all tables referencing EPOCH:
  (@coldtables) = &SQLManySQL ($myODBC,
     "SELECT xTaName FROM xTABLE, XCOLUMN
       WHERE xTaKey = xCoTable AND
       xCoName = 'Epoch'",
        "get tables referencing EPOCH");
```

[30] See the preceding section for the actual routine we use.

[31] A PERSON might also be represented in the PERSDATA table, but this is a deliberate denormalisation of the same reference in the PROCESS table.

```
  # 1. render all wbd PROCESSes (rEnd is not null, cold is clear) cold:
  (@wbdprocs) = &SQLManySQL($myODBC,
   "SELECT process FROM PROCESS WHERE cold IS NULL AND \
     rEnd IS NOT NULL", "get wbd procs");
   &DoSQL($myODBC,
   "UPDATE PROCESS SET cold = 1 WHERE cold IS NULL AND \
     rEnd IS NOT NULL", "clumsy: make cold");

  #2. for each process thus chilled, chill the children (epoch)
  my $p;
  foreach $p (@wbdprocs)
    { (@wbdobs) = &SQLManySQL($myODBC,
      "SELECT epoch FROM EPOCH WHERE EPOCH.Process = $p",
      "get relevant observations");
      &DoSQL($myODBC,
      "UPDATE EPOCH SET cold = 1 WHERE EPOCH.Process = $p",
      "chill epoch");
      my $o;
      foreach $o (@wbdobs)
        { &ChillObsChildren ($myODBC, $o, @coldtables); # hideous
        };

      #3. In a similar fashion, sort out Rx (using same $p):
       &DoSQL($myODBC,
      "UPDATE RX SET cold = 1 WHERE RX.Process = $p",
      "chill rx");
    };

  #3. remove cold people, then commit:
  &DoSQL($myODBC, "UPDATE PERSON SET cold = 1 WHERE pStatus = 1 AND person NOT IN
   (SELECT DISTINCT PERSON FROM PROCESS WHERE rEnd IS NULL)",
   'chill unused patients');
   &Commit($myODBC);
}
```

Because of the above selection from XCOLUMN (which is case-sensitive) it is mandatory that in our table definitions, all columns called 'Epoch' have just the first letter capitalised.

A problem is in identifying which people we should export to the database. It is possible that a process on a person has been terminated, but another process has been created. We need to be able to identify everyone with an active process and then render everyone *else* cold, apart from active staff members! The 'UPDATE PERSON' statement selects all patients (pStatus is 1) and chills those with no current active process.[32]

[32]Note that if we don't close all processes on a discharged patient, then the PERSON entry will still be passed to the PDA, eventually clogging things up!

Why don’t we use the above? The problem is that we may well want stuff (eg comments, observations) attached to a particular process which has been closed, with a patient who is still active.

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, $roll);
     ($myODBC, $menu1, $newW, $roll)=@_;

  $ISMENU = 1;
  $menuB->configure (-state => 'disabled');

  if ($BUG & 8) { &Print ("\n\n ===NEW MENU=== <$menu1> "); };
  # !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  if ($BUG & 0x40) { &DumpTkValues };
  @TKVALUES = (); # clear the old values : try 10/7/2008 to fix CheckEntry5 proble
  # [? problem is that invoking item is attached to TKVALUES entry! nasty]
  # !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

  my ($menuname, $rolloff);
  $menuname = $menu1;
  $rolloff = 0;
  @CMDSTACK = ();  # destroy stack!!
  @BURYSTACK = (); # likewise for burial
  @MARKERS = ();
  $MARK = 0;
  $MARKERS[$MARK] = -1; # clumsy.
```

The above introductory code allows for debugging (using a bit flag in BUG). We used to make MAINW really tiny, but have abandoned this approach. See how we delete everything on CMDSTACK, which constrains us to really tight, modular menu coding! We must also clear all stack marks.

The principal (right hand) menu has a MENU button (menuB) which is disabled when we enter this routine. The variable $rolloff is used below in handling

of ROLLOFFSET. $LOCALROLL is made afresh every time we *draw* a new menu.

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.[33] 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!

```
  $_ = $menuname;
  if ( /^0$/ )  # MENU(0) reloads!
     { $menuname = pop (@MENUS);
       $NEWXPARAM = pop (@X);
       $rolloff = pop (@ROLLOFFSET);
       if ($roll) # if rolling menu
          { push (@MENUS, $menuname);
            push (@X, $NEWXPARAM);
            push (@ROLLOFFSET, $rolloff);
            $rolloff += $LOCALROLL;
          };
     }
  elsif ( /^-*(\d+)$/ )  # if numeric, usually 1
     { my ($mc);
       $mc = $1;      # trim off -ve
       $mc ++;
       while ($mc > 0)
         { $menuname = pop (@MENUS);
           $NEWXPARAM = pop (@X);
           $rolloff = pop (@ROLLOFFSET);
           $mc --;
         };
     };
$LOCALROLL = 0;
$LINESLEFT = 0;
```

If we submitted a non-zero value in $roll, this implies that we are reloading a *copy* of the current menu on top of this instance, so we re-push the three relevant values. The value on the top of @ROLLOFFSET at the end of GoMenu will be further updated by the MakeTable routine!

[33] 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 (although at present tolerated). If you want to go back one menu, you submit just one, not negative one.

We clear LOCALROLL after (perhaps) using it, so that a fresh value can be established in it when we draw a polymorphic table.[34]

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.[35] And that's the whole of this tiny section.

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

6.1.4 Prepare to create

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

```
  my($mTitle);
  my($mCode);
  &ClearGroups();
  ($mCode, $mTitle) = &GetSQL($myODBC, "SELECT iID, iText FROM ITEM \
    WHERE iType = 20 AND iName = '$menuname'",
    "Get menu ID");

  my ($mX, $mY, $mW, $mH);
  ($mX, $mY, $mW, $mH) = &GetSQL($myODBC,
      "SELECT miX, miY, miW, miH FROM MENUITEMS \
        WHERE miMenu = $mCode AND miItem = $mCode",
      "get menu parameters");

$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!

[34] For definition of a polymorphic table, see section 8.

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

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.

6.1.5 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.[36]

```
  if ($TKTOGGLE)
     { $TKTOGGLE = 0;
     } else
     { $TKTOGGLE = 100;
     };

#  while ($OLDICOUNT > $TKTOGGLE)
#    { $OLDICOUNT --;
#      my ($itm) = $TKITEMS[$OLDICOUNT];
#      if (IsObject($itm)) # hmm?
#         {$itm->destroy;
#          $TKITEMS[$OLDICOUNT] = '';
#         };
#    };
  if ($DEADWINDOW)
     { &ExitWindow($DEADWINDOW);
     };
```

The above laboriously addresses a rather simple problem (We've remmed out the code that individually destroys items as it's not necessary — they vanish when their parent $DEADWINDOW does.

Originally, before we introduced the trick of *only later on* deleting all items in a menu (as embodied above) Perl would intermittently crash (fatally) as the garbage collector (sometimes) caught up with the fact that clicking on e.g. a button had triggered its own demise.

6.1.6 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.

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

```
$MENUW = $MAINW->Toplevel( );  # create new menu

$MENUW->geometry("+$mX+$mY");
$MENUW->geometry("$mW" . "x$mH");
$MENUW->title($mTitle);
$MENUW->bind('<Key-F1>' => [\&AlertMain]); # debugging
```

6.1.7 Run associated script

If the menu has an associated initialisation script, now's the time to run it. Formerly we ran this ini script *before* the menu creation, but this was incompatible with out Palm version. It also messed things around with TITLE in the ini script..

```
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, $_, $newW, -1, -1, 0x01);
     pop(@CMDSTACK);
     if ($r < 0) # if script execution failed
        {  # $menuname = pop (@MENUS);
           # push (@MENUS, $menuname);
           # $XPARAM = pop (@X);
           # push (@X, $XPARAM);
           # $NEWXPARAM = $XPARAM;
           # &RestoreLocalNames();
           # return; #fail!
        };
   };
$XPARAM = $NEWXPARAM;
```

Oops! If the above fails and we restore, then we have a problem where we rolled the menu — we must pop three more stacks.

We obtain the iInitial script if it exists[37] 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

[37] 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.

script.[38] After the script has run, we pop the stack.[39]

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!

6.1.8 Create menu components

Thus ...

```
  push (@MENUS, $menuname);
  push (@X, $XPARAM);
  push (@ROLLOFFSET, $rolloff);
  if ($XPARAM !~ /^\d+$/)
     { &Alert($MAINW,
        "Warning: non-numeric X <$XPARAM> won't work on PDA");
     };

#  $OLDICOUNT = $ICOUNT;
  $ICOUNT = &SubMenu($myODBC, $mCode, $MENUW, 0, 0, $TKTOGGLE);
  &FixGroups($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). Note that as per our PDA usage, X *must be an integer*! This constraint limits our ability to mess with modularity by passing arbitrary strings between menus. We don't actually forbid this usage in Perl, merely issuing the dire warning, which should be heeded.

6.1.9 Close prior window

```
 if (! ($oldW eq $MAINW) )
        { $DEADWINDOW = $oldW;
          $oldW->withdraw();  # hide window
        };
  $MENUW->focus; # set focus to this window

  if ($BUG & 0x40) { &DumpTkValues };
  if ($BUG & 8) { &Print ("\n (end NEW MENU) "); };
}
```

[38] Pushing the name allows the script to know which menu it's in!

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

Not only do we schedule destruction of the old window, we also force the focus to the new one. This concludes GoMenu.

Here's a tiny diagnostic routine:

```
sub DumpTkValues
{
  &Print ("\n TKVALUES: ");
  my($i) = 0;
  foreach (@TKVALUES)
   { &Print ("$i($_) ");
     $i += 1;
   };
}
```

Here's the nasty Perl test for an object:

```
sub IsObject
{ my ($x) = @_;
  return(isa ($x, 'UNIVERSAL'))
}
```

6.1.10 A frill: AlertMain

If we hit F1 in the Perl program, it's like clicking on the menu bar on the PDA — we invoke the associated iResponse script!

```
sub AlertMain
{ my $menuname;
  $menuname = pop(@MENUS);
  push (@MENUS, $menuname);

  my $ir;
  ($ir) = &GetSQL($myODBC, "SELECT iResponse FROM ITEM \
    WHERE iType = 20 AND iName = '$menuname'",
    "Get menu response!");
   &RunWholeScript ($myODBC, $ir, $MENUW, -1, -1, 0x02);
}
```

[LOOK INTO what happens if this script loads a new menu]!

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
```

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.[40]

```
  my(@items);
    (@items) = &SQLManySQL ($myODBC,
             "SELECT miItem, miUid from MENUITEMS \
               WHERE miMenu = $mCode \
               AND miItem <> $mCode \
               ORDER BY miOrder",
             "get items for this menu");
  my($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.

[40] Hey, let's read this line again :-)

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:

```
while ($#items > 0)
  {
    $item = shift(@items);
    $iu   = shift(@items);
    ($miX, $miY, $miW, $miH, $miGroup, $miEnabled, $miInitial) =
       &GetSQL ($myODBC,
          "SELECT miX, miY, miW, miH, \
            miGroup, miEnabled, miInitial \
            from MENUITEMS \
            WHERE miMenu = $mCode AND miUid = $iu",
          "get local item attributes");
    ($iType, $iText, $iList, $iLines, $iInitial, $iResponse, $iScript) =
      &GetSQL ($myODBC,
      "SELECT iType,  iText,  iList,  iLines, \
           iInitial,  iResponse,  iScript \
          from ITEM WHERE iId = $item",
      "get general attribs of this item");

    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.

Identify a monomorphic table

For definition of a monomorphic table, see section 8. Here we invoke Make1Table (Section 8.2) 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); # no iResponse for now.
       }
```

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?[41] See how we make (x, y) relative to X0 and Y0 —

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

this is a theme throughout the following code. Should we also allow for creation of these tables row-by-row as well as the current column-by-column?

Sub-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. (Particularly if this script were unwise enough to say MENU).

```
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, $newW,
                                -1, -1, 0x03);
          pop(@CMDSTACK);
        };
     if ($r != 0)  # [unless stopped] ???jvs
        { $i = &SubMenu($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.[42] 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.[43]

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

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

Identify a polymorphic table

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

```
      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).

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.

There is a problem — if widget i has just invoked a script that has called GoMenu, and the new menu creates a *new* widget (i') with the same number (i.e. numeric index into TKITEMS and TKVALUES), on completion of GoMenu, the destruction of the menu containing widget i will force removal of this widget by Perl. *But* when this happens, strange things seem to happen to the variable *bound* to i, which is now also most unfortunately bound to i'. To avoid these side effects,

we initially tried to skip use of index i within the menu. But this caused major crashes, so now we use two different sections of TKITEMS and alternate between these using $TKTOGGLE, avoiding the collision.

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!

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) = @_;
   if ($thisW)
      {  print LOGFILE "\n Destroying window <$thisW>";
         $thisW->destroy(); #
      };
}
```

6.3 Grouping

We introduced the idea of grouping above in section 6.2.1. 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.[44] 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($maxgrp);
  ($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)
```

[44] Helped by prior sorting by group.

```
                {&Print (")");
          };    };
        $g ++;
      };
    return;
}
```

7 Administrative functions

We need an administrative menu. This will permit the following tasks:

1. Creation of new menus (a one-off, unless we're designing);
2. Export of data for use on the PDA or a simulator (a debugging function);
3. Addition of new users;
4. Querying of the database (to design later).

Make a menu with a quit button:

```
sub DoAdmin
{
  my ($myODBC);
     ($myODBC) =@_;

  $ISDB = -1;
  $ISDB = &IsDbMade($myODBC); # again check if pain db made

  $ADMINMENU = $MAINW->Toplevel( );  # create new menu

  $ADMINMENU->geometry("+100+50");
  $ADMINMENU->geometry("800" . "x600");
  $ADMINMENU->title("Administrative Matters");

my $quitBut = $ADMINMENU->Button(
            -text => 'Done',
            -command => [ \&EndAdmin ] );
   $quitBut->place( -x => 260, -y => 540, -width => 270, -height => 30 );
```

We kick off with a button which allows access to (and editing of) user details, and the (experimental, as of 5/08) button for detailed, long reports.

```
my $userBut = $ADMINMENU->Button(
            -text => 'Edit User details',
            -command => [ \&DoUser, $myODBC ] );
   $userBut->place( -x => 240, -y => 50, -width => 350, -height => 30 );


my $longReport = $ADMINMENU->Button(
                  -text => 'Report between two intervals',
                  -command => [ \&PrintMonthlyData2, $myODBC, $CURRENTUSER ] );
   $longReport->place ( -x => 240, -y => 100, -width => 350, -height => 30);
```

We also need a button to back things up, and one for restoring such a backup:

```
my $backupBut = $ADMINMENU->Button(
                  -text => 'Backup',
                  -command => [ \&BackupAll, $myODBC ] );
   $backupBut->place ( -x => 650, -y => 300, -width => 120, -height => 30);

my $restoreBut = $ADMINMENU->Button(
                  -text => 'Restore',
                  -command => [ \&RestoreAll, $myODBC ] );
   $restoreBut->place ( -x => 650, -y => 350, -width => 120, -height => 30);

# the following installs the whole program (PRC+PDB files) to the PDA:
my $testBut = $ADMINMENU->Button(
                  -text => 'Install to PDA',
                  -command => [ \&PdaInstall, $myODBC ] );
   $testBut->place ( -x => 650, -y => 400, -width => 120, -height => 30);
   $testBut->configure (-background => 'red');


   $MAKEBUT = $ADMINMENU-> Button(
            -text => 'Make DB');
   $MAKEBUT->configure ( -command => [ \&MakeDBandMenus, $myODBC] );
   if ($ISDB) # if database exists
      { $MAKEBUT->configure (-state => 'disabled');
      };
   $MAKEBUT->place( -x => 650, -y => 450, -width => 120, -height => 30 );
   $MAKEBUT->configure (-background => 'yellow');

my $MakePDB = $ADMINMENU-> Button(
            -text => '(export PDB/PRC)',
            -command => [ \&ExportAllPDBsAndPRCs, $myODBC ] );
   $MakePDB->configure(-font => $ROOTFONT);
   $MakePDB->place( -x => 650, -y => 500, -width => 120, -height => 30 );



# on the other side we have an IDAS database query button:

my $idasBut = $ADMINMENU->Button(
                  -text => 'Import from SaferSleep',
                  -command => [ \&ExtDbImport, $myODBC, 0 ] );
   $idasBut->place ( -x => 260, -y => 470, -width => 270, -height => 50);
   $idasBut->configure (-background => 'green');

# and a print button too:

my $printBut = $ADMINMENU->Button(
                  -text => 'Print Discharges',
```

```
                  -command => [ \&PrintAllDischarges, $myODBC, $CURRENTUSER ] );
   $printBut->place ( -x => 35, -y => 300, -width => 150, -height => 30);
   $printBut->configure (-background => 'cyan');
# temporarily disable this:
# $printBut->configure (-state => 'disabled');

my $print1But = $ADMINMENU->Button(
                  -text => 'Print One (D)',
                  -command => [ \&PrintOneDischarge, $myODBC, $CURRENTUSER ] );
   $print1But->place ( -x => 35, -y => 350, -width => 150, -height => 30);
# temporarily disable this:
# $print1But->configure (-state => 'disabled');



# my $tsBut = $ADMINMENU->Button(
#                   -text => 'Do NOT press me',
#                   -command => [ \&Test_Fixup, $myODBC ] );
#   $tsBut->place ( -x => 35, -y => 350, -width => 180, -height => 30);
## $tsBut->configure (-state => 'disabled');


#my $coldtest = $ADMINMENU->Button(
#                  -text => '(Test only)',
#                  -command => [ \&FindWidows, $myODBC] );
#   $coldtest->place ( -x => 35, -y => 370, -width => 150, -height => 30);

# formerly
##                   -command => [ \&FixPERSONkeys, $myODBC] );
#                  -command => [ \&MakeCold, $myODBC] );
#
#my $coldtest = $ADMINMENU->Button(
#                  -text => '(Test only)',
#                   -command => [ \&SimpleBackup] );
#  $coldtest->place ( -x => 35, -y => 370, -width => 150, -height => 30);


my $coldtest = $ADMINMENU->Button(
                 -text => '(Data Tree)',
                 -command => [ \&PrintPatientDataTree, $myODBC] );
   $coldtest->place ( -x => 35, -y => 420, -width => 150, -height => 30);


#my $coldtest2 = $ADMINMENU->Button(
#                  -text => '(Fix Orphans)',
#                  -command => [ \&FindAndFixOrphans, $myODBC] );
#   $coldtest2->place ( -x => 35, -y => 470, -width => 150, -height => 30);
```

```
my $dailyReport = $ADMINMENU->Button(
                  -text => 'Daily report',
                  -command => [ \&PrintBasicData, $myODBC, $CURRENTUSER, $ADMINMEN
   $dailyReport->place ( -x => 240, -y => 300, -width => 100, -height => 30);


# a test routine for fixing up errors in PCA sequencing:

# my $tstbut = $ADMINMENU->Button(
#                  -text => 'Test Fix',
#                  -command => [ \&TestFixMany, $myODBC] );
#   $tstbut->place ( -x => 390, -y => 300, -width => 100, -height => 30);
## code 110 is epidural, code 390 is PCA.

# my $tstbut = $ADMINMENU->Button(
#                  -text => 'Weekend data: test',
#                  -command => [ \&WeekendTimes, $myODBC] );
#   $tstbut->place ( -x => 390, -y => 300, -width => 100, -height => 30);

# my $tstbut = $ADMINMENU->Button(
#                  -text => 'Test only',
#                  -command => [ \&SetUsernames, $myODBC] );
#   $tstbut->place ( -x => 390, -y => 300, -width => 100, -height => 30);


## 2008-09-21: See CheckDates routine. A hack.
## my $tstbut = $ADMINMENU->Button(
##                   -text => 'Test (NO!)',
##                   -command => [ \&CheckDates, $myODBC] );
##   $tstbut->place ( -x => 430, -y => 300, -width => 100, -height => 30);


# my $tstbut = $ADMINMENU->Button(
#                  -text => '!Upgrade',
#                  -command => [ \&BatchSQL, 'UPGRADE20080901', $myODBC] );
#   $tstbut->place ( -x => 430, -y => 300, -width => 100, -height => 30);



# the following imports PDB files from \perlpgm\import WITHOUT a PDA fetch!
# it does NOT commit, but also does not rollback.
# A TEST ROUTINE TO BE USED WITH EXTREME CAUTION.
 my $tstbut = $ADMINMENU->Button(
                  -text => 'DEBUG IMPORT',
                  -command => [ \&ImportPdbData, $myODBC, $CONST{IMPORTDIR}] );
   $tstbut->place ( -x => 430, -y => 300, -width => 100, -height => 30);
```

```
 my $tst2but = $ADMINMENU->Button(
                  -text => '!Fix Type',
                  -command => [ \&FixSurgeryType, $myODBC, $ADMINMENU] );
   $tst2but->place ( -x => 430, -y => 370, -width => 100, -height => 30);


} # the end of the routine.
```

Here's the routine to destroy the administrative menu, called by its Quit button.

```
sub EndAdmin
{
  $ADMINMENU->destroy();
}
```

7.1 A user-related submenu

To speed things up and 'unclutter' the administrative menu, we define a separate user menu, added on 3/5/2008.

```
sub DoUser
{
  my ($myODBC);
     ($myODBC) =@_;

  $USERMENU = $ADMINMENU->Toplevel( );  # only from within admin

  $USERMENU->geometry("+10+30");
  $USERMENU->geometry("900" . "x600");
  $USERMENU->title("Edit User Details");
  $USERMENU->configure( -background => $CONST{'USERMENUBACKGROUNDCOLOUR'} );

my $quitBut = $USERMENU->Button(
            -text => 'Done',
            -command => [ \&EndUser ] );
   $quitBut->place( -x => 300, -y => 450, -width => 200, -height => 30 );
```

Allow insertion of a new user:

```
my $topLabel = $USERMENU->Label( -text => 'Add a new User:');
  $topLabel->configure( -background => $CONST{'USERMENUBACKGROUNDCOLOUR'} );
  if (! $BIGFONT)
     { $BIGFONT = $newBut->fontCreate( 'bigfont',
              -family => 'Helvetica',
              -size =>14);
```

```
    };
   $topLabel->configure(-font => $BIGFONT);
   $topLabel->place(    -x => 100, -y => 15, -width => 300, -height => 20 );

my $foreLabel   = $USERMENU->Label( -text => 'Forename' );
my $surLabel    = $USERMENU->Label( -text => 'Surname' );
my $ROLELabel = $USERMENU->Label( -text => 'Designation' );
my $LogNameLabel = $USERMENU->Label( -text => 'User login' );
   $foreLabel->configure( -background => $CONST{'USERMENUBACKGROUNDCOLOUR'} );
   $surLabel->configure( -background => $CONST{'USERMENUBACKGROUNDCOLOUR'} );
   $ROLELabel->configure( -background => $CONST{'USERMENUBACKGROUNDCOLOUR'} );
   $LogNameLabel->configure( -background => $CONST{'USERMENUBACKGROUNDCOLOUR'} );

   $foreLabel->place(   -x => 20, -y => 50, -width => 100, -height => 30 );
   $surLabel->place(    -x => 20, -y => 100, -width => 100, -height => 30);
   $ROLELabel->place( -x => 20, -y => 150, -width => 100, -height => 30);
   $LogNameLabel->place( -x => 20, -y => 200, -width => 100, -height => 30);

my $foreBox = $USERMENU->Entry();
my $surBox  = $USERMENU->Entry();
my $logBox  = $USERMENU->Entry();

   $foreBox->place ( -x => 150, -y => 50, -width => 200, -height => 30);
   $surBox->place ( -x => 150, -y => 100, -width => 200, -height => 30);
   $logBox->place ( -x => 150, -y => 200, -width => 200, -height => 30);

   $foreBox->configure ( -textvariable => \$FORENAME );
   $surBox->configure  ( -textvariable => \$SURNAME );
   $logBox->configure ( -textvariable => \$USERLOGIN );

my $ROLEOptions = $USERMENU->Optionmenu();
my @statu = ("Pain team nurse", "Paint team Consultant", "Registrar/fellow");
   $ROLEOptions->addOptions(@statu);
   $ROLEOptions->place ( -x => 150, -y => 150, -width => 200, -height => 30);
   $ROLEOptions->configure  ( -textvariable => \$ROLE );

my $addUserBut = $USERMENU->Button(
                  -text => 'Add User',
                  -command => [ \&AddUser, $myODBC ] );
   $addUserBut->place ( -x => 150, -y => 250, -width => 200, -height => 30);

my $addUserPwd = $USERMENU->Button(
                  -text => 'Alter/add user password',
                  -command => [ \&AlterPwd, $myODBC ] );
   $addUserPwd->place ( -x => 150, -y => 325, -width => 200, -height => 30);
```

Next we wish to *inactivate* a user. The idea is that if somebody is no longer

active on the pain service, their name clutters up menus, so we hide things. All we do is place a non-null value (todays date) into the rEnd field of the process describing that user, and ignore such users.

```
my $iaLabel = $USERMENU->Label( -text => 'Inactivate a user:');
   $iaLabel->configure(-font => $BIGFONT);
   $iaLabel->configure( -background => $CONST{'USERMENUBACKGROUNDCOLOUR'} );
   $iaLabel->place(    -x => 400, -y => 15, -width => 200, -height => 20 );

my $iUserOption = $USERMENU->Optionmenu();
my @namelist;

if ($ISDB)
   { (@namelist) = &FindUserList ($myODBC,
      "SELECT person FROM PERSON WHERE cold IS NULL AND
        PERSON.pStatus <> 1 AND person > 50");
   $iUserOption->addOptions(@namelist);
   };

   $iUserOption->place ( -x => 400, -y => 40, -width => 250, -height => 30);
   $iUserOption->configure  ( -textvariable => \$USER2HIDE );

my $hideUserBut = $USERMENU->Button(
                  -text => 'Hide User',
                  -command => [ \&HideUser, $myODBC ] );
   $hideUserBut->place ( -x => 400, -y => 80, -width => 250, -height => 30);
```

Very similar are the controls to reactivate a user:

```
my $raLabel = $USERMENU->Label( -text => 'Reactivate user:');
   $raLabel->configure( -background => $CONST{'USERMENUBACKGROUNDCOLOUR'} );
   $raLabel->configure(-font => $BIGFONT);
   $raLabel->place(    -x => 400, -y => 130, -width => 200, -height => 30 );

my $rUserOption = $USERMENU->Optionmenu();
if ($ISDB)
   { (@namelist) = &FindUserList ($myODBC,
      "SELECT person FROM PERSON WHERE cold IS NOT NULL AND
        PERSON.pStatus <> 1 AND person > 50");
     $rUserOption->addOptions(@namelist);
   };

   $rUserOption->place ( -x => 400, -y => 160, -width => 250, -height => 30);
   $rUserOption->configure  ( -textvariable => \$USER2FIND );

my $findUserBut = $USERMENU->Button(
                  -text => 'Reveal User',
```

```
                    -command => [ \&RevealUser, $myODBC ] );
   $findUserBut->place ( -x => 400, -y => 200, -width => 250, -height => 30);

} # the end of the routine.
```

Let's destroy the USER menu:

```
sub EndUser
{
  $USERMENU->destroy();
}
```

7.1.1 Add a user

Next, the more ambitious routine to add a user. Here we have to do several things:

1. Confirm that all fields are occupied and contain vaguely reasonable data;
2. If users already exist with the same surname (or the forename entry the same as somebody's surname!) then confirm that we should proceed;
3. Generate a new ID for that user, and insert them into the database.

```
sub AddUser
{
  my ($myODBC);
     ($myODBC) =@_;

  my $i;
  my $rolecode;

  $i = &Confirm($USERMENU,
    "Name is <$FORENAME $SURNAME>, \n Role is <$ROLE>.\n \n Is this correct?");
  if (! $i)
     { &Alert ($USERMENU, "Aborted");
       return;
     };

  if ( $ROLE =~ /consultant/i )
     { $rolecode = 8;
     }
  elsif ($ROLE =~ /nurse/i )
     { $rolecode = 4;
     }
  elsif ($ROLE =~ /registrar/i )
     { $rolecode =2;
     } else
```

```
     { &Alert($USERMENU, "Please select person's ROLE");
       return;
     };

  if (! $FORENAME =~ /\w+/ )
     { &Alert($USERMENU, "Please enter FORENAME or initial");
       return;
     };

  if (! $SURNAME =~ /\w+/ )
     { &Alert($USERMENU, "Please enter SURNAME");
       return;
     };

  if (! $USERLOGIN =~ /\w+/ )
     { &Alert($USERMENU, "Please enter user log-in name");
       return; # 2008-01-07
     };
  $USERLOGIN =~ tr/A-Z/a-z/; # force lower case

my ($oldlogin) = &GetSQL ($myODBC,
  "SELECT person FROM PERSON WHERE pUserName = '$USERLOGIN'",
  'ensure no duplicate name');

  if (length $oldlogin > 0)
     { &Alert($USERMENU, "Sorry. Log-in '$USERLOGIN' is taken.");
       return;
     };
  # &Altert($USERMENU, "Debug: login name is $USERLOGIN");
```

After the above basic checks, we pull out all matches between either the entered surname *or* the entered forename and identical *surnames* in the database![45] But first we fix up single quotes within the surname:[46]

```
 $SURNAME = &SpitnPolish($SURNAME);
 $FORENAME = &SpitnPolish($FORENAME);

 my @matches;
 (@matches) = &SQLManySQL ($myODBC,
    "SELECT PERSON.person FROM PERSDATA,EPOCH,PROCESS,PERSON \
    WHERE PERSDATA.Epoch = EPOCH.epoch AND \
    EPOCH.Process = PROCESS.process AND \
    PROCESS.person = PERSON.person AND \
    PERSON.pStatus > 1 AND \
```

[45] This is to catch the case where the user mixes up the surname and forename!

[46] It might be smart to get rid of backticks and other embarrassing characters too!

```
    (PERSDATA.pdoSurname = '$SURNAME' OR PERSDATA.pdoSurname = '$FORENAME')",
       "get matching names");

 my $mat = $#matches;
 my $m;
 my $mlist = "";
 my $fore = "";
 my $sur = "";

  if ($mat > -1)
     { foreach $m (@matches)
          {
            ($fore) = &SQLManySQL ($myODBC,
              "SELECT PERSDATA.pdoForename FROM PERSDATA,EPOCH,PROCESS \
               WHERE PERSDATA.Epoch = EPOCH.epoch AND \
               EPOCH.Process = PROCESS.process AND \
               PERSDATA.pdoForename IS NOT NULL AND \
               PROCESS.Person = $m",
            "get surname(s)");

            ($sur) = &SQLManySQL ($myODBC,
              "SELECT PERSDATA.pdoSurname FROM PERSDATA,EPOCH,PROCESS \
               WHERE PERSDATA.Epoch = EPOCH.epoch AND \
               EPOCH.Process = PROCESS.process AND \
               PERSDATA.pdoSurname IS NOT NULL AND \
               PROCESS.Person = $m",
            "get forename(s)");

            $mlist = "$mlist \n $fore : $sur";
          };

       $i = &Confirm($USERMENU, "There is at least one matching surname! \n \
         $mlist \n \n Are you SURE you wish to continue?");
       if (! $i)
          { &Alert ($USERMENU, "Aborted");
            return;
          };
     };
```

Finally we insert the name into the database, and commit. This is trickier than it sounds. We have to:

1. Insert a new person, with the relevant role;
2. Create a type 1 process related to that person;
3. Create an observation on the process;

4. Create a PERSDATA entry linked to the observation!

```
  my ($LASTWORKER) = &GetSQL($myODBC,
    "SELECT MAX(person) FROM PERSON WHERE person < 1000",
    'get last dr/nurse'); # ugh
  $LASTWORKER ++;

  $TODAY = &GetLocalTime();
  my $perskey =  $LASTWORKER;  # WAS: &AutoKey($myODBC, "PERSON");
  my $prockey = &AutoKey($myODBC, "PROCESS");
  my $obskey  = &AutoKey($myODBC, "EPOCH");
  my $datkey  = &AutoKey($myODBC, "PERSDATA");

  &Do2SQL($myODBC, "INSERT INTO PERSON (person, pMade, pStatus, pUserName) \
          VALUES ($perskey, TIMESTAMP '$TODAY', $rolecode, '$USERLOGIN')",
      "new person");
  &Do2SQL($myODBC, "INSERT INTO PROCESS
          (process, Person, ProcType, rCreated, rStart, rPlanner) \
          VALUES ($prockey, $perskey, 1, TIMESTAMP '$TODAY', \
                  TIMESTAMP '$TODAY', 1)",
      "new type 1 proc");
  &Do2SQL($myODBC, "INSERT INTO EPOCH (epoch, oMade, Person, Process) \
          VALUES ($obskey, TIMESTAMP '$TODAY', 1, $prockey)",
      "epoch on proc");
  &Do2SQL($myODBC, "INSERT INTO PERSDATA
          (persdata, Epoch, pdoSurname, pdoForename, pdoPerson) \
          VALUES ($datkey, $obskey, '$SURNAME', '$FORENAME', $perskey)", \
          "person data");
  # what about pdoGender? To fix!

  &Commit($myODBC);

  &Alert($USERMENU, "User $FORENAME $SURNAME inserted into database \n
  This user's unique key is: $perskey. User name '$USERLOGIN'");
  $SURNAME = "";
  $FORENAME = "";
  $ROLE = "";

  &EndUser(); # reload!
  &DoUser($myODBC);
}
```

We might amend the above to include gender. Here's the trivial SpitnPolish routine:

```
sub SpitnPolish
{
```

```
  my ($dat);
     ($dat) = @_;

  $dat =~ s/''''/g; # single quotes
  $dat =~ s/  / /g; # double spaces
  $dat =~ s/\d//g;  # numerics
  $dat =~ s/[^ -\w'.]//g; # others

  return ($dat);
}
```

Okay, there are several other characters we might get rid of.

7.1.2 Add/alter user password

The AlterPwd routine simply makes the relevant menu.

```
sub AlterPwd
{
  my ($myODBC);
     ($myODBC) =@_;

  $PWDMENU = $ADMINMENU->Toplevel( );  # only from within admin

  $PWDMENU->geometry("+10+30");
  $PWDMENU->geometry('900x450');
  $PWDMENU->title("Alter User Password");

my $quitBut = $PWDMENU->Button(
            -text => 'Done',
            -command => [ \&EndPwd ] );
   $quitBut->place( -x => 300, -y => 400, -width => 200, -height => 30 );


my $UsernameLabel   = $PWDMENU->Label( -text => 'User name:' );
   $UsernameLabel->place(-x => 20, -y => 50, -width => 100, -height => 30 );
my $UsernameBox = $PWDMENU->Entry();
   $UsernameBox->configure ( -textvariable => \$USERLOGIN );
   $UsernameBox->place  (-x => 150, -y => 50, -width => 200, -height => 30);

my $oldPwdLabel   = $PWDMENU->Label( -text => 'Old Password' );
   $oldPwdLabel->place(-x => 20, -y => 100, -width => 100, -height => 30 );
my $oldPwdBox = $PWDMENU->Entry( -show => '*');
   $oldPwdBox->configure ( -textvariable => \$OLDPWD );
   $oldPwdBox->place  (-x => 150, -y => 100, -width => 200, -height => 30);

my $newPwd1Label   = $PWDMENU->Label( -text => 'New Password' );
   $newPwd1Label->place(-x => 20, -y => 150, -width => 100, -height => 30 );
```

```
my $newPwd1Box = $PWDMENU->Entry( -show => '*');
   $newPwd1Box->configure ( -textvariable => \$NEWPWD1 );
   $newPwd1Box->place  (-x => 150, -y => 150, -width => 200, -height => 30);

my $newPwd2Label   = $PWDMENU->Label( -text => 'CONFIRM new password' );
   $newPwd2Label->place(-x => 20, -y => 200, -width => 100, -height => 30 );
my $newPwd2Box = $PWDMENU->Entry( -show => '*');
   $newPwd2Box->configure ( -textvariable => \$NEWPWD2 );
   $newPwd2Box->place  (-x => 150, -y => 200, -width => 200, -height => 30);

my $quitBut = $PWDMENU->Button(
            -text => 'ALTER password!',
            -command => [ \&DoAlterPwd ] );
   $quitBut->place( -x => 300, -y => 250, -width => 200, -height => 30 );
}
```

A trivial routine to destroy the menu:

```
sub EndPwd
{
  $PWDMENU->destroy();
}
```

Here we actually alter the password:

```
sub DoAlterPwd
{

  if (! $USERLOGIN =~ /\w+/ )
     { &Alert($PWDMENU, "Please enter user name");
       return;
     };
#  if (! $OLDPWD =~ /\w+/ )
#     { &Alert($PWDMENU, "Please enter old password");
#       return;
#     };
  if (! $NEWPWD1 =~ /\w+/ )
     { &Alert($PWDMENU, "Please enter new password");
       return;
     };
  if (! $NEWPWD1 =~ /\w+/ )
     { &Alert($PWDMENU, "Please re-enter new password (confirmation)");
       return;
     };
  if ($NEWPWD1 ne $NEWPWD2)
     { &Alert($PWDMENU, "New password and confirmation don't match");
       return;
     };
```

```
  my ($m5new) = md5_hex($NEWPWD1);
##  &Alert($PWDMENU, "DEBUG: Value is '$m5new'");

  # validate user name:
  $USERLOGIN =~ tr/A-Z/a-z/; # force lower case
  my ($usr) = &GetSQL ($myODBC,
    "SELECT person FROM PERSON WHERE pUserName = '$USERLOGIN'", 'get user');
  if (length $usr < 1)
     { &Alert($PWDMENU, "User '$USERLOGIN' not found");
       return;
     }

  #   # first, check that entry exists:
  my ($ispw) = &GetSQL($myODBC,
    "SELECT Person FROM USERPWD WHERE Person = $usr",
    'confirm entry');
  if ($ispw)
     {  # here check old password (if present):
        my ($oldp) = &GetSQL ($myODBC,
         "SELECT pwd FROM USERPWD WHERE Person = $usr", 'get old pwd');
        if (length $oldp > 0)
           {  my ($m5old) = md5_hex($OLDPWD);
              if ($m5old ne $oldp)
                 { &Alert($PWDMENU, "Bad OLD password!");
                   return;
                 };
           };
        # overwrite with new (md5 encoded):
        &DoSQL ($myODBC,
          "UPDATE USERPWD SET pwd = '$m5new' WHERE Person = $usr",
          'set new password');
     } else
     { &DoSQL ($myODBC,
         "INSERT INTO USERPWD (Person, pwd)
           VALUES ($usr, '$m5new')",
         'insert new password entry');
     };
  &Commit($myODBC);
  &Alert($PWDMENU, "Password altered");
  $PWDMENU->destroy();
}
```

7.1.3 Hiding a user

Here's the simpler routine which hides users:

```
sub HideUser
{
```

```
  my ($myODBC);
     ($myODBC) =@_;
  my $h2;

  $USER2HIDE =~ /:(\d+)$/ ;  # terminal number after colon

  if (! defined $1)
     { &Alert ($USERMENU, "Choose a user to hide!");
       return;
     };
  $h2 = $1;

  if (! &Confirm($USERMENU,
    "Hide process for user <$USER2HIDE>? \n\n Are you sure?") )
     { &Alert ($USERMENU, "Not hidden");
       return;
     };

  &Do2SQL($myODBC, "UPDATE PROCESS SET rEnd = TIMESTAMP '$TODAY' \
          WHERE PROCESS.Person = $h2 AND PROCESS.ProcType <= 2",
          "End process for that person");
  # [might check for success]

  &Do2SQL($myODBC, "UPDATE PERSON SET cold = 1 WHERE person = $h2",
         'convenient flag of hidden person speeds things');
  # should also cascade COLD flags [hmm 20080524: FIX ME!]

  &ChillUser($myODBC, $h2); # chill relevant cold flags.

  &Commit($myODBC);
  &Alert ($USERMENU, "Hidden");
  $USER2HIDE = "";

  &EndUser();
  &DoUser($myODBC); # reload ??
}
```

7.1.4 'Revealing' a user

Given the details of a hidden user, re-activate them by setting the rEnd value for the relevant process to NULL.

```
sub RevealUser
{
  my ($myODBC);
     ($myODBC) =@_;
  my $h2;
```

```
  $USER2FIND =~ /:(\d+)$/ ;  # terminal number after colon

  if (! defined $1)
     { &Alert ($USERMENU, "Choose a user to re-activate!");
       return;
     };
  $h2 = $1;

  if (! &Confirm($USERMENU,
         "Reactivate user <$USER2FIND>? \n\n Are you sure?") )
     { &Alert ($USERMENU, "Nothing done");
       return;
     };

  &Do2SQL($myODBC, "UPDATE PROCESS SET rEnd = NULL \
          WHERE PROCESS.Person = $h2 AND PROCESS.ProcType < 3",
          "Restart process for that person");

  &Do2SQL($myODBC, "UPDATE PERSON SET cold = NULL WHERE person = $h2",
         'reactivate PERSON');
  &WarmUser($myODBC, $h2); # warm up relevant cold flags

  &Commit($myODBC);
  &Alert ($USERMENU, "$USER2FIND re-activated!");
  $USER2FIND = "";

  &EndUser();
  &DoUser($myODBC);
}
```

7.1.5 Chill user

If a user has been removed from the active list, they will still have active entries in the PROCESS, EPOCH and PERSDATA tables. These data do *not* have to be exported to the PDA, so let's cool them off. We pass the ODBC handle and the user (person) id.

```
sub ChillUser
{ my ($myODBC, $user);
     ($myODBC, $user) =@_;

  # wbd means 'warm but dead'
  my @wbdprocs;
  my @wbdepochs;

  &Do2SQL ($myODBC,
   "UPDATE EPOCH SET cold=3 WHERE epoch IN
```

```
     (SELECT epoch FROM EPOCH,PROCESS WHERE
     PROCESS.Person = $user AND
     EPOCH.Process = PROCESS.process)",
   'chill user epochs');
  # [might check outcome]
  # consider the interesting case of user being admitted [check me!]

  &Do2SQL ($myODBC,
    "UPDATE PROCESS SET cold=3 WHERE Person = $user",
    'chill user processes');

  &Do2SQL ($myODBC,
    "UPDATE PERSDATA SET cold=3 WHERE pdoPerson = $user",
    'chill user personal data');

  # don't COMMIT here
}
```

7.1.6 Warm users

If a user has been reactivated, we must warm up the most recent persdata entry, and the associated type 1/2 process, as well as the relevant epoch.

```
sub WarmUser
{ my ($myODBC, $user);
     ($myODBC, $user) =@_;

  my ($upersdata) = &GetSQL($myODBC,
    "SELECT MAX(persdata) FROM PERSDATA WHERE
      pdoPerson = $user", 'get recent persdata');
    # [hmm consider merits of GetSQL vs SQLManySQL ??];
    # also consider filling in NULL fields for this row from
    #   previous values, as elsewhere [examine this]
  #  &Alert($MAINW, "DEBUG: Max PERSDATA is $upersdata");

  my ($uepoch) = &GetSQL($myODBC,
    "SELECT epoch FROM PERSDATA WHERE persdata = $upersdata",
    'get corresponding epoch');

  my ($uproc) = &GetSQL($myODBC,
    "SELECT process FROM EPOCH WHERE epoch = $uepoch",
    'get the process');

  &Do2SQL($myODBC, "UPDATE PERSDATA SET cold = NULL WHERE
    persdata = $upersdata", 'warm PERSDATA');
  &Do2SQL($myODBC, "UPDATE EPOCH SET cold = NULL WHERE
    epoch = $uepoch", 'warm EPOCH');
  &Do2SQL($myODBC, "UPDATE PROCESS SET cold = NULL WHERE
```

```
    process = $uproc", 'warm PROCESS');

  # don't COMMIT here
}
```

7.1.7 An array of user names

A (formerly cumbersome) routine which takes an sql statement and produces a list of names (with the associated code) in the format "forename surname :code".

```
sub FindUserList
{
  my ($myODBC, $cmd);
     ($myODBC, $cmd) =@_;

  my (@namelist) = &SQLManySQL ($myODBC,
   "SELECT pdoForename || ' ' ||
      pdoSurname || ' :' || cast(pdoPerson as varchar(8))
     FROM PERSDATA WHERE pdoPerson in ($cmd)",
     'get user list');

  return (@namelist);
}
```

Note the CAST of pdoPerson: the length should be sufficient for most implementations.

7.2 Installation on the PDA

Formerly we relied on the HotSync capabilities of the usual Palm PDA software to transport files between the PC and the PDA. This is a rather hit-and-miss business, for two reasons:

1. Palm made the rather silly decision that *conduits* were a good idea. A conduit is a customised program which only deals with a particular Palm program. So far, so good . . . but in order to make your own conduit, you would formerly have had to pay MS for the privilege of using their Visual C++ suite.[47] I keeping with our freeware orientation, we eschewed this option.

2. When you first hotsync the Palm PDA, everything new is moved over, which may take some time. Even worse, although you would think that Palm would ensure that backup copies are always up-to-date, and on superficial

[47] Now that .NET is a bloated 500M monstrosity, it's available free for download, with some restrictions.

testing this would seem to be the case, with repeated backups you would seem to have no guarantee that all information will be preserved, *unless* you have your own conduit to ensure the backup![48] So you get the double whammy of a whole lot of stuff you don't want being moved over, and your vital files not being backed up!

Our solution is as follows (and is actually faster than using a conduit, albeit a little more cumbersome):

1. After you've ensured that the usual Palm hotsync process installed and working on yor PDA, obtain the following two programs:

   (a) Mathias Lüdtke's impressive little PRC program filepc2pda.[49]

   (b) The Windows program Palm File Browser (PFB)[50] You may need to register with MyTreo.

2. Install filepc2pda on the PDA, connect up your USB cable,[51] and run filepc2pda on the PDA. Ignore the initial error and *set the baud rate to 57600*. Click on 'Connect'.

3. Install *PFB.exe* on the desktop. This involves copying this executable to the */painform* directory, together with the files *sertransplg.dll* (which came in the zip with PFB), and the elusive file *USBPort.dll*. The latter file should have come with your Palm software; otherwise get a recent version off the 'Net.[52] Both DLL files *must* be in the same *painform* directory.

4. It's okay to have the normal HotSync program active on the Windows toolbar, but *do not run* HotSync while you are doing any of the following or conflict will likely result.

5. Run the pain application (pain.bat)[53] as usual, go to the Administration Menu, and click on 'Install to PDA'. This brings us to the following routine, which will *only* really work if you have already created the desktop database. What this routine does is:

[48] We initially made the mistake of assuming data integrity and simply using the backups of our PDB files in the *Backup* directory on the PC, but then discovered to our horror that sometimes the changed files are simply not transferred.

[49] Obtainable from http://www.ghisler.com/serial.htm

[50] Try http://en.pdassi.com/13639 or http://mytreo.net/downloads/details-220.html

[51] As most modern Palm PDAs are USB-connected, we have concentrated on this. You might try playing on the serial port or even Bluetooth, however

[52] There may be a copy elsewhere, e.g. under Win2K in the directory */WINNT/system32*.

[53] Which invokes pain2.pl

(a) Create a new image of the database PDB files in the directory */painform/export*;

(b) Use the Perl 'system' command to invoke the DOS batch file *install2pda.bat*.

The reason why we use this cumbersome approach is to allow Linux users to insert their own system command (e.g. to invoke the powerful ColdSync) to do what we next do. The DOS batch file in turn invokes an AutoIt version 3 script, which we'll look at in a moment. The following is very similar to ExportAllPDBsAndPRCs (Section 16.4.2).

```
sub PdaInstall
{
  my ($myODBC);
     ($myODBC)=@_;

  if (! &Confirm($ADMINMENU,
       "Are you SURE? \n(Files on PDA will be **OVERWRITTEN**)" ) )
     {  &Alert($ADMINMENU, "Nothing done. (Whew!)");
        return;
     };

  my $expdir = $CONST{EXPORTDIR}; # usually /painform/export
  if (! &ValidatePath($expdir))
   { &Alert($MAINW, "Error: directory $expdir does not exist");
     return;
   };

  $MAINW->focus(); # even focusForce() seems to do nil if focusFollowsMouse ???
  $ADMINMENU->iconify;
  &MakeAllPDBs ($myODBC, $expdir);
  $ADMINMENU->deiconify;
  $ADMINMENU->focus();

  my $pbat;
  $pbat = $CONST{PDAINSTALLBATFILE};
    # batch file accepts %1 as the first argument
  $_ = $expdir; # translate / to \ for Windoze
  tr /\//\\/;
  $pbat = "$pbat $_";
  my $r;
 #$r = system ( $pbat );
 # Noo. let's try backticks:
 $r = `$pbat`;
 # We then look for the string 'Something went wrong'
 if ($r !~ /Something went wrong/im)
    { $r = "Success! On PDA please click 'Abort+Disconnect', then the 'Home' key.'
    } else
    { $r = "Ooops. An error occurred. Files NOT transferred";
```

```
    };
   # In Linux how ABOUT CAPTURING THE ERROR along lines of:
   # $r = ‘$pbat 2>&1 1>/dev/null‘;

 &Alert ($ADMINMENU, $r);
}
```

The DOS program first copies over all relevant PRC files to the same */painform/export* directory, and then calls upon an AutoIt program which sets about moving files to the PDA. This and the AutoIt program is discussed below in section 17.7.1. The use of backticks is rather special, providing all of the output. We set up the batch file to write ‘Something went wrong’ (See section 17.7.1) so we can pick this up and signal the problem. Inelegant but the information transfer works!

7.3 Fixing an error (1)

In our initial implementation, we failed to test for null processes in the KillDated function (see *AnalgesiaDB2.tex*) and therefore overwrote e.g. PCA termination dates. Here’s the fix which fixes up PCA if code is 390, epidural if 110.

```
sub TestFixItem
{ my ($myODBC, $icode);
     ($myODBC, $icode) = @_;

  # get all patients with eg. PCA (proctype=390) problem:
  my ($COUNT) = 0;

  my (@pts) = &SQLManySQL($myODBC,
   "select former.person \
    from process as former, process as latter \
     where former.person = latter.person \
      and former.rend = latter.rend \
      and former.process in \
        (select min(process) from process \
           where proctype = $icode group by person) \
      and former.process <> latter.process \
      and former.proctype=latter.proctype \
      and former.rend IS NOT NULL \
      and former.rend > timestamp ’2007-01-01 00:00:00’ \
    group by former.person", ’get dup end pcas’);

  my ($pt, $p0, $end0, $start0, $p1, $end1, $start1, $stmt);
  foreach $pt (@pts)
    {
      my (@procdat) = &SQLManySQL($myODBC,
       "select process, rEnd, rStart FROM PROCESS \
```

```
        WHERE Person = $pt \
         AND ProcType = $icode \
         AND rEnd IS NOT NULL \
         AND rEnd > TIMESTAMP '2007-01-01 00:00:00' \
        ORDER BY rEnd, rStart", 'get times');
      $p0 = shift(@procdat);
      $end0 = shift(@procdat);
      $start0 = shift(@procdat);
      while ($#procdat > -1)
        {
         $p1 = shift(@procdat);
         $end1 = shift(@procdat);
         $start1 = shift(@procdat);
         if ($end0 ne $end1)
            { # print LOGFILE "\n Terminated match: <@procdat> <p1=$p1, end1=$end
              # we will manually examine and fix such cases!
              @procdat = ();
            } else
            { $stmt = "UPDATE PROCESS SET rEnd = TIMESTAMP '$start1' WHERE proces
              # print LOGFILE "\n EXECUTING: <$stmt>";
               &DoSQL($myODBC, $stmt, 'fix end timestamp');
               $COUNT ++;
              $p0 = $p1;
              $end0 = $end1;
              $start0 = $start1;
            };
        };
    };
  &Commit($myODBC);
  &Alert($MAINW, "Fixed (code $icode): Count was $COUNT");
}
```

Here's the 'bigger fix':

```
sub TestFixMany
{ my ($myODBC);
     ($myODBC) = @_;

  TestFixItem($myODBC, 110);
  TestFixItem($myODBC, 210);
  TestFixItem($myODBC, 310);
  TestFixItem($myODBC, 390);

  TestFixItem($myODBC, 102);
  TestFixItem($myODBC, 103);
  TestFixItem($myODBC, 104);
  TestFixItem($myODBC, 105);
  TestFixItem($myODBC, 106);
```

```
  TestFixItem($myODBC, 107);
  TestFixItem($myODBC, 108);
  TestFixItem($myODBC, 109);
  TestFixItem($myODBC, 120);
  TestFixItem($myODBC, 125);
  TestFixItem($myODBC, 130);
  TestFixItem($myODBC, 135);
  TestFixItem($myODBC, 140);
  TestFixItem($myODBC, 145);
  TestFixItem($myODBC, 150);
  TestFixItem($myODBC, 220);
  TestFixItem($myODBC, 225);
  TestFixItem($myODBC, 230);
  TestFixItem($myODBC, 235);
  TestFixItem($myODBC, 240);
  TestFixItem($myODBC, 245);
  TestFixItem($myODBC, 250);
  TestFixItem($myODBC, 320);
  TestFixItem($myODBC, 325);
  TestFixItem($myODBC, 330);
  TestFixItem($myODBC, 335);
  TestFixItem($myODBC, 340);
  TestFixItem($myODBC, 345);
  TestFixItem($myODBC, 350);

}
```

7.3.1 Pulling out weekend data

A brief section for TC:

```
sub WeekendTimes
{ my ($myODBC);
     ($myODBC) = @_;

  my $weekfile= "data/WEEKEND.TXT";
open WEEKFILE, ">$weekfile" or die
     "*CRASH* Could not open $weekfile :$!\n";

 # initially hard code the days we want:
  my @DATES = ('2008-01-05', '2008-01-06', , '2008-01-12',  '2008-01-13',  '2008-0
     '2008-01-26' , '2008-01-27', , '2008-02-02', '2008-02-03', '2008-02-09', '200
     '2008-02-17', '2008-02-23', '2008-02-24', '2008-03-01', '2008-03-02', '2008-0

  my ($d);

foreach $d (@DATES)
  {
```

```
    my @DY = ();
    my ($q) ="SELECT oMade as mytime FROM EPOCH
         WHERE oLength IS NOT NULL
            AND cast(oMade as varchar(19)) > '$d 08:00:00'
            AND cast(oMade as varchar(19)) < '$d 17:00:00'
         ORDER BY mytime";
    @DY = &SQLManySQL ($myODBC, $q, 'get all times');
        #  print WEEKFILE "\n\n DEBUG All times: @DY";
        #  print WEEKFILE "\n\n";
    my ($dlen) = $#DY;
    my ($mintime) = $DY[0];
    my ($maxtime) = $DY[$dlen]; # last item

   # here should also calculate difference!
   $mintime =~  / (\d{2}):(\d{2}):(\d{2})/ ;
   my ($mint) = ($1*3600 + $2*60 + $3)/60;  # minutes
   $maxtime =~  / (\d{2}):(\d{2}):(\d{2})/ ;
   my ($maxt) = ($1*3600 + $2*60 + $3)/60;  # minutes
   my ($delta) = $maxt - $mint;

    print WEEKFILE "\n $d, $mintime, $maxtime, $delta, $dlen";
  }
 close WEEKFILE;
}
```

7.3.2 User name updates

A simple routine (v 0.95) that derives user names (pUsername in the PERSON table) from the forename and surname. Ideally use once only!

```
sub SetUsernames
{
 my ($myODBC);
    ($myODBC) = @_;

  my (@ppl) = &SQLManySQL($myODBC, "SELECT person FROM PERSON WHERE \
     person > 50 AND person < 1000", 'get all users');
  my ($p, $unam);
  foreach $p (@ppl)
    {
      ($unam) = &GetSQL ($myODBC, "select lower(pdoforename || cast(pdosurname as
      from persdata where pdoperson = $p",  'get user name');
      &DoSQL ($myODBC, "UPDATE PERSON SET pUsername = '$unam' WHERE person = $p",
      print LOGFILE "\n username is '$unam', code is $p";
    };
  &Commit($myODBC);
}
```

7.4 Fixing an error(2)

Ultimately functions in this section should also result in logging of all changes in a FORENSIC table.

7.4.1 Updating type of surgery

As we refine the ability of our program to identify different types of surgery, it's reasonable to apply these changes retrospectively to existing data where the classification is 'unknown', in other words, entries in the SURGTYPEOB table that have a SurgType field with a value of zero. Here's a copy of entries in the surgtype table:

Code	*Meaning*
0, unspecified	
1	general
49	endoscopy
99	plastics
149	orthopaedic
199	neurosurgery
249	ophthalmic
299	dental
399	ORL
449	cardiac
499	thoracic
599	vascular
649	hepatobiliary
699	colorectal
799	urology/renal
899	gynaecology
999	obstetrics

Table 2: Coding of types of surgery

In the following section, we read in a file that matches new SQL 'match' strings to the above codes. The format of the table (default name: *data/surgtype.lst*) is a header line that is ignored, followed by entries in the format

```
'matchstring':code
```

Where **code** is one of the codes from above, and the match string starts with a backtick and ends with a quote followed by a colon. The last line (after the last line of valid pairs) must start with a star (*) or an error might result.

```
sub FixSurgeryType
{
 my ($myODBC, $MNU);
    ($myODBC, $MNU) = @_;
 my ($fail) = 0;
 my ($line) = 0;

 my ($q) = "select count(surgtypeob) from SURGTYPEOB where surgtype = 0";
 my ($count) = &GetSQL($myODBC, $q, 'get initial dud count');

 my ($surgtypefile) = 'data/surgtype.txt';
  $surgtypefile = &Ask ($MNU, "Count $count. Enter surgery type file", $surgtypefi
  if (length $surgtypefile < 1)
     { return 0;
     };
 open (SURGTYPEFILE, $surgtypefile) || ($fail = 1) ;
 if ($fail)
    { &Alert($MNU, "File <$surgtypefile> not found!");
      return;
    };
  <SURGTYPEFILE>;  # discard header line
  while (($_ = <SURGTYPEFILE>) && ( ! /^\*/ ))  # while not last line
    { print $_; # debug
      $line ++;
      if (! /\`(.+)':(\d+)/ ) # if no pair match
         { &Alert($MNU, "Bad pair, line $line: <$_>. Aborting!");
           return;
         };
      my $q = "UPDATE SURGTYPEOB SET surgtype = $2 where surgtypeob in \
       (SELECT surgtypeob from SURGTYPEOB,EPOCH,COMMENT where \
        SURGTYPEOB.epoch = EPOCH.epoch and EPOCH.epoch = COMMENT.epoch and \
        UPPER(COMMENT.ctext) like '%$1%' and SurgType = 0)";
      &DoSQL($myODBC, $q, 'fix one operation type match');
    };
  &Commit($myODBC); # commit changes
  my ($count) = &GetSQL($myODBC, $q, 'get final count');
  &Alert($MNU, "Completed. Final count is $count");

  # debug:
  my(@duds) = &SQLManySQL($myODBC,
   "SELECT cText FROM COMMENT,EPOCH,SURGTYPEOB WHERE
     COMMENT.Epoch = EPOCH.epoch AND EPOCH.epoch = SURGTYPEOB.Epoch AND
     SURGTYPEOB.SurgType = 0", 'get duds');
  foreach (@duds) { print "\n$_" ; };
```

```
}
```

7.5 Fixing an error(3)

The problem was that we issued a global UPDATE on terminating processes when we discharged a patient. This error was current until 2008-09-18. Dates were overwritten for the rEnd of many processes (ugh). A partial fix is to look at the timestamp of the most recent EPOCH on that PROCESS, and replace the erroneous (too late!) date with the date from the most recent epoch. First, a convenient subroutine.

```
sub CheckFix
{
  my ($myODBC, @prs);
     ($myODBC, @prs) = @_;

  print LOGFILE "\n type / process: old -> new (difference)";

  my ($p, $maxTs, $thisTs, $pt);
  foreach $p (@prs)  # clumsy, better to do in SQL?!
    {
      ($thisTs, $pt) = &GetSQL($myODBC,
        "SELECT rEnd, ProcType FROM PROCESS WHERE process = $p",
        'get current end timestamp, and process type');
      ($maxTs)  = &GetSQL($myODBC,
        "SELECT oMade FROM EPOCH WHERE epoch = (SELECT max(epoch) FROM EPOCH WHERE
        'get last epoch timestamp');

      if (length $maxTs > 1)
         { my ($jdiff) = &JD($thisTs) - &JD($maxTs);
           # print LOGFILE "\n $pt, $thisTs, $maxTs, $p, $jdiff";
         # HERE'S THE FIX:
           if ($jdiff > 0.5)
              { &DoSQL ($myODBC,
                "UPDATE PROCESS SET rEnd = TIMESTAMP '$maxTs' WHERE process = $p",
                  print LOGFILE "\n $pt / $p: $thisTs -> $maxTs ($jdiff)";
              };
         } else
         { ## print LOGFILE "\n no EPOCH match for process $p";
         }
    };

}
```

Here's the main routine.

```
 sub CheckDates
 {
  my ($myODBC);
  ($myODBC) = @_;

  print LOGFILE "\n\n DEBUGGING ALL PROCESS END TIMESTAMPS";
  my (@prs) = &SQLManySQL($myODBC,
    "SELECT process FROM PROCESS WHERE ProcType > 108 AND
     PROCTYPE < 501 AND
     rEnd IS NOT NULL AND
     CAST(rEnd AS VARCHAR(4)) > '2000' AND
     CAST(rStart AS VARCHAR(4)) > '2000' ",
    'get all procs, eliminating unfinished/duds');
  CheckFix($myODBC, @prs);
  print LOGFILE "\n ***END DEBUG ALL PROCESSES*** ";
  &Commit($myODBC);

#  print LOGFILE "\n\n DEBUGGING EPIDURAL INFUSIONS END TIMESTAMPS";
#  my (@prs) = &SQLManySQL($myODBC,
#    "SELECT process FROM PROCESS WHERE ProcType = 210 AND
#     rEnd IS NOT NULL AND
#     CAST(rEnd AS VARCHAR(4)) > '2000' AND
#     CAST(rStart AS VARCHAR(4)) > '2000' ",
#    'get epi procs as above');
#  CheckFix($myODBC, @prs);
#  print LOGFILE "\n ***END DEBUG EPIDURAL INFUSIONS*** ";

 }
```

8 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 the same, distinct database ID, for example, an ID of a person.

8.1 Polymorphic tables

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

8.1.1 MakeTable

The MakeTable routine is cumbersome. It accepts an ODBC handle, `idx` which is the current number of items already created[54], a Tk window newW, the ID of the table item itself (tbl), the maximum number of table rows displayed (tLines),[55] 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
   my ($rolloff);
   $rolloff = pop(@ROLLOFFSET);
   push(@ROLLOFFSET, $rolloff);
```

$rolloff is used to ‘continue where we left off’ if a copy of a menu has been loaded using the ROLLMENU routine. This routine is a simple, PDA-friendly alternative to having a scroll bar.[56] We *must* always push a value back onto ROLLOFFSET!

[54] This value is returned after being incremented by MakeTable.

[55] This number includes the header line!

[56] Have you noticed how painful scroll bars are to use on PDAs?

Identify columns

First we find the columns.

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

Get row V values

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

```
my (@rows);
my ($rowcount);
  @CMDSTACK = (); # hmm. No need to mark.
  &RunWholeScript ($myODBC, $tIni, $newW, -1, -1, 0x04);
  @rows = @CMDSTACK; # whole!

  if ($rolloff > 0)
     { splice(@rows, 0, $rolloff); # clip used leading elements
     };
  $rowcount = 1 + $#rows;
  if ($BUG & 2)
     { &Print ("\n DEBUG: items <@rows>, row count $rowcount");
     };
  if ($rowcount > ($tLines-1))  # if more than can fit..
     { $LOCALROLL = $tLines-1;
       $LINESLEFT = $rowcount - $LOCALROLL;
       # &Alert($MAINW, "Roll offset is now $LOCALROLL");
       $rowcount = ($tLines-1);

       # here we might also splice @rows to trim off all above
       # first tLines-1 elements. [CHECK ME!]
     };
  @CMDSTACK = (); # clear it
```

If we have already displayed previous lines in a menu (ROLLOFFSET stack contains a non-zero number on the top). 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.

Minor preparation

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.

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, $oText);
      ($iType, $iText, $iList, $iInitial, $iResponse, $iScript) =
        &GetSQL ($myODBC,
            "SELECT iType,  iText,  iList, \
               iInitial,  iResponse, iScript \
              from ITEM WHERE iId = $col",
            "get attribs of COLUMN");
      $oText = $iText;

      my($irPaper);
      my ($irFrac, $irN);
      ($irFrac, $irN, $irEnabled, $irPaper) = &GetSQL ($myODBC,
```

```
   "SELECT irFraction, irName, irEnabled, irPaper \
     FROM ICOLTABLE \
     WHERE irTBL = $tbl AND irItem = $col",
   "get fractional width of item");
/(.+?),(.*),(.),(.*)/;
$iW = $irFrac * $tW;  # pixel width
$_ = $irN;  # irName. clumsy.
if (/.+/) # if column name exists
   { $iText = $_; # overwrite
   };
```

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.

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).

Make the rest of the column

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 ++;
```

```
                    my ($iv); # HIDEOUS!
                    $iv = "";

                    $VVALS[$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
                  } else
                  { @rowcopy = (); #clear
                  };
            };
          $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 row 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.[57]

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`).

8.2 Monomorphic table (revised)

Here we create a 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

[57] What about pushbuttons? Explore.

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!

Make1Table accepts parameters similar to, but simpler than, MakeTable (Section 8.1.1). We similarly have an ODBC handle, item count (idx, an index into TKITEMS), Tk window (newW), the table ID, a line count (`tLines`) and various coordinates and widths, as well as the initialisation script `tIni`.

At the end of Make1Table we return the *number of items created*. We thus keep track of the increments to the total item count (`idx`) in the variable `newi`.

```
sub Make1Table
{  my ($myODBC, $idx, $newW, $tblid, $tLines,
       $tX, $tY, $tW, $tH,
        $tIni);
      ($myODBC, $idx, $newW, $tblid, $tLines,
       $tX, $tY, $tW, $tH,
       $tIni) = @_;
  my ($newi);
  $newi = $idx;
 # &Alert($MAINW, "Ini Script is $tIni");
```

First, we determine the item to be replicated, by looking for it in ICOLTABLE. At the same time we obtain whether it's enabled, and the fractional width of each column.

```
  my ($iX, $iY, $iW, $iH); # i-variables
  my ($irItem, $irFraction, $irEnabled);
  ($irItem, $irFraction, $irEnabled) = &GetSQL ($myODBC,
  "SELECT ICOLTABLE.irItem, ICOLTABLE.irFraction, \
     ICOLTABLE.irEnabled FROM ICOLTABLE \
     WHERE ICOLTABLE.irTBL = $tblid AND ICOLTABLE.irOrder = 1",
    "get column properties");

  $iW = $irFraction * $tW;
  $iH = $tH / $tLines;
  $iX = $tX;
  $iY = $tY;
```

The item width `iW` is a portion of `tW` the table width, and the height `iH` depends on the line count.

Next we find parameters for the item we are replicating: the type of item, any associated list, and a response script (to be used when somebody clicks on a particular item). We don't (at present) use an initialisation script (`iInitial`) or the script-response variable `iScript`.

```
 my ($iType, $iList, $iResponse, $iInitial, $iScript);
($iInitial, $iType, $iList, $iResponse) = &GetSQL ($myODBC,
              "SELECT iInitial, iType, iList, iResponse \
                from ITEM WHERE iId = $irItem",
              "get attribs of column item");
      # might check for success, or fail

      $iScript = '';
```

Next we run the initialising script. This is a bit kludgey, as it must provide two datums for each item, the V value and the text value (in order). Ultimately we should fix things so that we mark the stack before running this script, and then pop off only the row data.

```
 my(@rowdata);
 my($itmcnt);
 &RunWholeScript ($myODBC, $tIni, $newW, -1, -1, 0x05);
 @rowdata = @CMDSTACK; # really should MARK!
 $itmcnt = ($#rowdata)/2; # [?]
```

The RunWholeScript invocation above is vulnerable if the script does nasty things to the stack.[58]

Finally, we go through and create each item, supplying the relevant intialisation parameters to CreateOneItem. We loop around until we run out of items, or space to put them. Once we've completed a column, we move right to the next one.

In the following we should at some stage address the irritating duplication of `itmtext`.

```
 my($itmid, $itmtext);
 my($j, $rowcount);
 $rowcount = $tLines;

while ($itmcnt > 0)
  { $itmtext = pop(@rowdata);
    $itmid   = pop(@rowdata);
    # &Alert($MAINW, "Code is $itmid, text is '$itmtext'"); # DEBUGGING
    $itmcnt --;
      $j = &CreateOneItem ($myODBC, $newi, $newW,
                           $iX, $iY, $iW, $iH,
                           0, $iInitial, $iResponse, '',
                           $iType, $itmtext, $iList, 1,
                           $irEnabled, 0, $itmtext, '');
```

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

```
        $rowcount --;
        if ($j) { $GROUPS[$newi] = 0;
                  $VVALS[$newi] = $itmid;
                }; # might warn on failure ??
        $newi ++; # bump overall item count
        $iY += $iH;
        if ($rowcount < 1)
           { $rowcount = $tLines;
             $iX += $iW;   # [??? ADD SOME WHITE SPACE]
             if (($iX+$iW) > ($tX+$tW))  # over right margin of table
                { $itmcnt = 0; # force end
                };
             $iY = $tY;    # back up to top of row.
           };
    };
   $newi -= $idx;
   return ($newi);  # number of NEW items
}
```

As an aside, note 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. (In a previous incarnation of the above routine, this mattered).

9 Items

9.1 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.

9.1.1 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;
```

9.1.2 Minor initialisation

The following is largely concerned with paper and ink colours.

```
   my ($paper, $ink, $both);
   $paper = '';
   $ink = '';
   if ($uid > 0)
      { ($paper, $ink) = &GetSQL ($myODBC,
             "SELECT miPaper, miInk \
               FROM MENUITEMS WHERE miUid = $uid",
             "get item colour");
```

```
      };
  if ($BUG & 2)
    { &Print ("\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] = $iText; # keep value!
  &Print ( "\n 1. TKVALUES($idx) <- '$iText'" );
```

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.

9.1.3 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)||$FAILURE)    # -1 = failed, 0 = HALT
          { $FAILURE = 0;
#            print LOGFILE "\n Destroying item B <$idx>";
            $TKITEMS[$idx]->destroy; # delete
            return 0; # fail
          };
       if ((!$r) || $Stopped) # == zero
          {# print LOGFILE "\n Stopped 1";
            $Stopped = 0; # reset once more (used)
          } else
          { # print LOGFILE "\n Text value <$iText>";
          };
       $TKITEMS[$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.

9.1.4 Create a button

```
  elsif ($iType == 2) # button
     { $TKITEMS[$idx] =
         $newW->Button(
           -command => [ \&DoButton2,
                         $myODBC, $iResponse, $newW, $idx]
                     );
       $TKVALUES[$idx] = $rowparam;
  &Print ( "\n 2. TKVALUES($idx) <- '$iText'" );
       $TKITEMS[$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, $iText); # ???
       if ($TOGGLED)
          { # here might GET paper and ink colours:
            $ink = $TKITEMS[$idx]->cget( '-foreground');
            $paper = $TKITEMS[$idx]->cget ('-background');
            $TKITEMS[$idx]->configure (-foreground => $paper);
            $TKITEMS[$idx]->configure (-background => $ink);
          };
       $TOGGLED = 0;
       $iText = pop(@CMDSTACK);
       if (($r < 0) || $FAILURE )
          { $FAILURE = 0;
#            print LOGFILE "\n Destroying item C <$idx>";
            $TKITEMS[$idx]->destroy;
            return 0;
          };
       if ($Stopped)
          { $Stopped = 0; # reset once more (used)
            $iText = ''; # ??? [check me]
          }
```

```
#       print LOGFILE "\n Text value <$iText>";
        $TKITEMS[$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!

9.1.5 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[$idx] );
        if ($BUG & 0x40){&Print ("\n A. TKVALUES bound($idx)"); };

        if ($enabled == 0)
           { $TKITEMS[$idx]->configure (-state => 'disabled');
           };
        $r = &RunClearScript ($myODBC, $iInitial, $newW, $idx, $iText);
#       print LOGFILE "Debug checkbox: Return value was $r";
        $iText = pop(@CMDSTACK);
        if ( $FAILURE ||($r < 0)) # FAIL was invoked, forcing end
           { $FAILURE = 0;
#             print LOGFILE "\n FAIL invoked: destroying <$idx>";
             $TKITEMS[$idx]->destroy;
             return 0;
           };
        if ((!$r) || $Stopped) # == zero
           {# print LOGFILE "\n Stopped 4";
             $iText = ""; # null
             $Stopped = 0; # reset once more (used)
           } else
           { # print LOGFILE "\n Text value <$iText>";
           };
```

```
      if ($iText =~ /^1$/ )
         { $TKITEMS[$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![59] 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.

9.1.6 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,
         -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)||$FAILURE)
        { $FAILURE = 0;
          print LOGFILE "\n *SCRIPT FAILED <$iInitial>";
```

[59] 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!

```
            $TKITEMS[$idx]->destroy;
            return 0;
         };
      if ((!$r) || $Stopped) # == zero
         { # print LOGFILE "\n Stopped 5";
           $iText = ""; # null
           $Stopped = 0; # reset once more (used)
         } else
         { # print LOGFILE "\n Text value <$iText>";
         };
```

Here, enable the button if *non-zero, non-null* value present in $iText!!

```
      if ( (length $iText > 0) && ($iText !~ /^0$/ ) )
         { $TKITEMS[$idx]->configure(
             -background => $red,
             -foreground => $CONST{'GREY'});
           $TKITEMS[$idx]->configure(
             -activebackground => $red,
             -activeforeground => $CONST{'GREY'});
           $TKVALUES[$idx] = $iText; # 29-1-2006
&Print ( "\n 3. TKVALUES($idx) <- '$iText'" );
         };
      if ($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.[60] See how, for visual appeal and to avert confusion, we also need to fiddle with the *active* fore and backgrounds!

9.1.7 Create a text field

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

```
elsif (   ($iType >= 10)
       && ($iType < 18)
      ) # textfield
   { # print LOGFILE "\n Making new text field: idx is $idx, value " . $TKVALUES
```

[60] FlipButton shouldn't be used *here*.

```
       $TKITEMS[$idx] = $newW->Entry();
       $TKITEMS[$idx]->place( -anchor => 'nw',
                              -relx => $miX,
                              -rely => $miY,
                              -relheight => $miH,
                              -relwidth => $miW);
       $TKITEMS[$idx]->configure (
                  -textvariable => \$TKVALUES[$idx]
                                   );
       if ($BUG & 0x40){&Print ("\n B. TKVALUES bound($idx)"); };
       $r = &RunClearScript ($myODBC,
                  $iInitial, $newW, $idx, $iText);
#       print LOGFILE "Debug: Return value for ($iInitial) was $r";
       $iText = pop(@CMDSTACK);
       if (($FAILURE)||($r < 0))
          { $FAILURE = 0;
            print LOGFILE "\n FAIL: destroying <$idx>";
            $TKITEMS[$idx]->destroy;
            return 0;
          };
       if (!$r) # == zero
          { $iText = ""; # null
          }
       $TKVALUES[$idx] = $iText;  # the problem is here 10/7/2008
  &Print ( "\n 4. TKVALUES($idx) <- '$iText'" );
       if ($BUG & 2)
          { &Print ("\n DEBUG: TEXT: value is <$iText>");
          };
       if ($BUG & 8)
          { &Print ("\n ITEM: txt idx $idx<$iText>($miX, $miY)");
          }
       $TKITEMS[$idx]->configure (
                  -validatecommand => [ \&CheckEntry5,
                        $myODBC, $iResponse, $newW, $idx],
                  -validate => 'focusout'
                                   );
     }
```

Validation of the text string entered by CheckEntry5 occurs on leaving the box (focusout). 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.

There is a *problem* in the above, because when we start fiddling with the Entry, then the validatecommand is invoked!

9.1.8 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 QMANY. The following is patterned on the preceding code.

```
elsif ($iType == 6) # poptrigger
   { $TKITEMS[$idx] = $newW->Optionmenu();
     my(@iv);
     print LOGFILE "\n DEBUG: making poptrigger ($idx)";
     $_ = $iList;
     if ( /^->(.+)/ ) # if -> run script!
        { $_ = $1;
          @CMDSTACK = ();
          &RunWholeScript ($myODBC,$_, $newW, $idx, -1, 0x06);
          @iv = @CMDSTACK;
        } else
        { s/\|$//;   # rid of last pipe!
          @iv = split /\|/;
        };
```

In the following code we create an associative array (hash) which we store for later reference in the 'array of hashes' called POPARRAYS. When we create a poplist (OptionMenu) we submit *pairs* of items, so that when the user selects a text item from the poplist, our program puts the corresponding (unique) ID onto the stack for use by the response script. The Perl hash is ideal for this sort of application, provided each text item is unique.[61] The MakeHash routine (Section 9.1.11) does all of the hard work.

```
my $hsh;
my $ary;

($hsh, $ary) = &MakeHash(@iv);
@iv = @$ary; # dereference
$POPARRAYS[$idx] = $hsh; #store reference to hash!
```

The above is thanks to some inspired Perl documentation.[62]

```
$TKITEMS[$idx]->addOptions(@iv);
```

[61] Need to look into the implications of non-unique items, smartest would be to check for this and disambiguate identical items by concatenating the ID itself onto the end of the string, perhaps in parenthesis!

[62] Ta! http://www.sunsite.ualberta.ca/Documentation/Misc/perl-5.6.1/pod/perldsc.html

```
      $TKITEMS[$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)
         { # print LOGFILE "\n Destroying item F <$idx>";
           $TKITEMS[$idx]->destroy;
           return 0;
         };
      if (!$r) # == zero
         { $iText = ""; # null
         }
      $POPVALUE[$idx] = $iText;
      $TKITEMS[$idx]->configure (
         -command => [ \&Dopoptrigger6,
              $myODBC, $iResponse, $newW, $idx] );
      if ($BUG & 8)
         { &Print ("\n ITEM: poptrigger $idx <$iText> ($miX, $miY)");
           &Print ("\n *** response is '$iResponse'");
    };   }
```

9.1.9 Create a scrollbar

This section is just a stub, at present. It will probably remain a stub forever, as we've gone off scrollbars, especially on PDAs — they're just too fiddly, and there are almost always other options.

9.1.10 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.
}
```

9.1.11 Turning an array into a hash

The MakeHash routine is a little tricky! Our poplists (OptionMenus) are populated using *pairs* of items. The first item in the pair is usually a number (often referring to a primary key in a table within the SQL database)[63] and the second is usually a text string associated with this number.

For example, we might have the unique identifier of a particular person as the first item, and a string containing their forename and surname as the second. What MakeHash does is to return *two things*:

1. A reference to an associative array of identifiers and text items;
2. A reference to an array of the text items!

We will use the convenience of Perl and turn our list of pairs of items into the logical Perl equivalent — a hash (associative array). Given an array of pairs, we create the associative array. Note that the first (leftmost) item in each pair will usually be numeric with the second being an associated text value.

```
sub MakeHash
{
  my(@elems);
  (@elems) = @_;

  my $hsh = {}; # actually a reference to a hash!
  my @ary;

  my $value;
  my $key;

#  my $d;
#  $d = $#elems;
#  &Alert($MAINW, "DEBUG: Length is $d");

  while ($#elems > 0)
    {
      $value = shift(@elems);
      $key = shift(@elems); # 2nd element is 'key' (!)
      $hsh->{$key} = $value;
      unshift(@ary, $key);
    };

  if ($#elems > -1)
     { &Alert($MAINW, "Bad list: odd length! <@elems>");
```

[63] Although this is not mandatory, and no check is made!

```
    };
  return ($hsh, \@ary); # \@ references the array
}
```

See how we ‘turn things around’ so that we can look up the numeric value, given the text string!

9.2 Subsidiary ‘Item’ routines

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

9.2.1 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, 0x07);
    };
  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.[64]

9.2.2 FlipButton

This ugly routine is a misnomer. It has been modified since inception. What it should do now is turn on a button, and turn off all other grouped buttons. It must run the script associated with the button whenever clicked, even if the button is already on! (Previously we used to toggle the button if it was clicked again, now we should retain the state!)

[64] Needs work, what’s new?

FlipButton is invoked when a pushbutton is clicked.[65]

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, we should *not* toggle the other buttons or turn the button off, but we *should* still respond to the click — this gives the user more flexibility, especially in clicking on a button with an associated menu!

```
sub FlipButton
{ my($myODBC, $iResponse, $i, $newW);
  ($myODBC, $iResponse, $i, $newW) = @_;

  my($btn, $hits);
  $btn = $TKITEMS[$i];
  my($bclr, $fclr);
  $fclr = $CONST{'GREY'}; # &GreyGet( $btn->cget('-background') );
  $bclr = $CONST{'RED'}; # &GreyGet( $btn->cget('-foreground') );

  if ( $TKVALUES[$i] == 1) # amendment: 31-5-2006
     { print LOGFILE "\n *Pushbutton <$i> already on";
     } else # not yet on:
     { $TKVALUES[$i] = 1;
  &Print ( "\n 5. TKVALUES($i) <- 1" );
       $btn->configure( -background => $bclr,
                   -foreground => $fclr); # on
       $btn->configure( -activebackground => $bclr,
                   -activeforeground => $fclr);
#       print LOGFILE "\n Item <$i>[flip] set to 1";
       $hits = &Mutex2($i);
#       print LOGFILE "\n (Hit count <$i> = $hits)";
     };

  if (length $iResponse < 1)
     { return;
     };

  @CMDSTACK = ();
  push (@CMDSTACK, $TKVALUES[$i]); #clumsy
  &RunWholeScript ($myODBC, $iResponse, $newW, $i, -1, 0x08);
  if ($BUG & 0x40){&Print ("\n (END button push($i))"); };
```

[65] See how the Perl focusFollowsMouse makes things ugly if you hold the cursor over a button!

```
  if ($hits < 1)
     { $TKVALUES[$i] = -1;
       &Print ( "\n 6. TKVALUES($i) <- -1" );
       print LOGFILE "\n Item <$i>[flip2] set to -1";
     };
  # the above accommodates the case where no mutual button
  # exists, so must use own discretion. Here, we DO run
  # the script again if we re-click!?
  # (the -1 value is not == 1 in test above)!
}
```

[NOTE: We need to examine the above in more detail, especially the fiddles regarding a value of -1, which at present is of no value on the PDA].

9.2.3 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*
  &Print ( "\n 1. TKVALUES($i) <- 0" );
#  print LOGFILE "\n Item <$i>[G] cleared";
  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.

9.2.4 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. The number of items encountered and cleared (hits) is returned.

```
sub Mutex2
{ my ($i, $hits);
  ($i) = @_;
  my ($belt, $todo);
  $belt = 100;
  $hits = 0;

  $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 $hits;
         };
      &ClearButton ($i); # else, clear
      $hits ++;
      $belt --;
    };
  Print ("\n ERROR: infinite loop in mutex. Index $todo");
  return $hits;
}
```

9.2.5 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.[66] And that's it.

```
sub GreyGet
 { my ($clr);
      ($clr) = @_;
      if ($clr eq $CONST{'WHITE'})
         { return ($CONST{'GREY'});
         };
   return ($clr);
 }
```

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

9.2.6 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, 0x09);
}
```

9.2.7 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. [? fix]



  # 25-6-2006: here must replace $fred by the matching index value!!

  my %hsh; # associative array

#  my $pi;
#  $pi = $POPARRAYS[$i];
#  %hsh = %$pi; # defereference
#  # better is simply:

  %hsh = %{$POPARRAYS[$i]};

  my$f;
  $f = $hsh{$fred}; # get associated value!
  if ($f !~ /^-?\d+$/)  # if not integer
     { &Alert($MAINW, "Warning: non-numeric key <$f> WILL NOT WORK on PDA");
     };
```

```
  push (@CMDSTACK, $f);
  &RunWholeScript ($myODBC, $iResponse, $newW, $i, $i, 0x0a); # hmm.
}
```

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

9.2.8 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, 0x0b);
}
```

9.2.9 CheckEntry5

```
sub CheckEntry5
{ my ($myODBC, $iResponse, $newW, $i);
     ($myODBC, $iResponse, $newW, $i) =@_;
    if (length $iResponse < 1)
       { return; # do nothing
       };
  if ($BUG & 0x40) { &DumpTkValues }; # 2008-07-10 debug
  my($newval);
  $newval = $TKVALUES[$i];
  if (length $newval < 1)
     {return;
     };
#  $newval =~ s/'/''/g;  # [4] removed! Do in script, NOT here!
  $newval =~ s/\|/\?/g; # [5]
  @CMDSTACK = ();
  push (@CMDSTACK, $newval);
  &RunWholeScript ($myODBC, $iResponse, $newW, $i, -1, 0x0c);
  return 1; # always OK. We might modify this??
}
```

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.

10 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++.[67]

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 the integer 100 on the stack, then two, and then divide 100 by two, placing the result back on the stack".[68]

  Another way of looking at this is as `100->DIV(2)`, which is more explicit and will in fact work. Note that we distinguish between use of integers (which are preceded by a hash) as in `#100`, and plain old numbers (100). In general it's advisable to *always* use the formal integer designation, so the preceding example should actually have read: `#100->DIV(#2)`, and this is what you should do!

- 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"`

Scripting is described in more detail in the document *AnalgesiaDB2.tex*.

10.1 Complete script execution

In this section we discuss RunWholeScript (which does just that, given a string containing the script).

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

[68] 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.

10.1.1 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.

```
sub RunWholeScript
{ my($myODBC, $iScrpt, $newW, $idx, $pop, $dbg); # dbg is debug param
    ($myODBC, $iScrpt, $newW, $idx, $pop, $dbg) = @_;
  &Print("\n Running script ($dbg) index is $idx: {<$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.[69] 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!

10.1.2 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 = -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`.

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

10.1.3 Other skip values: MARK

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

```
if ($skip <= 0)
   { if ($skip < -10) # marked!
        { $skip += 100;
          $MARK++;
          $MARKERS[$MARK] = $#CMDSTACK - ($skip);
          if ($MARKERS[$MARK] < -1)
             { &Alert($MAINW, "Error: Stack under-marked");
               &Print ("\n UnderMARK?($skip): $MARK [@MARKERS]");
               $MARKERS[$MARK] = -1;
             };
          if ($BUG & 16)
             { &Print ("\n DEBUG: marked $MARK [@MARKERS]");
             };
          $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. We mark the stack at an offset obtained by subtracting the stated value from the current top of the stack, storing this position in the MARKERS array. [70]

Formerly we used -3 as a signal for turning the stack into a list, but we have removed this function, unwanted at least for now!

10.1.4 skip -4: URZN

This odd but useful command (‘Unmark and return if zero or null’) is sometimes useful in testing a value and then returning appropriately.[71] It’s now a deprecated function, but still supported.

```
        elsif ($skip == -4)   # [1-4-2005]
           {
              &DecMarker();
              if ($BUG & 16)
                 { &Print ("\n DEBUG: URZN $MARK [@MARKERS]");
```

[70] 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 thus *subtract* N items from the top of CMDSTACK. We check that the submitted value makes sense, otherwise complaining! Another problem might be large marks, necessitating values over 100 for the trickery.

[71] In earlier prototypes it was URNZ but NZ is a common abbreviation for ’nonzero’.

```
            };
         $skip = 0;      # force return below
         last;           # actually STOP! [14/11/2005]
      }
```

As with many commands involving marks, this needs some work. You must also read the URZN section below (Section 10.5.16).

10.1.5 Other negative skip values

A value of -2 in `skip` turns off marking *and trims the stack back to the marked position*,[72] and the important value -1 signals failure of a script. Such failure is deliberately induced by using the FAIL command!

```
   elsif ($skip == -2)   # unmark
      { &DecMarker();
        $skip = 1;       #continue
      }
   else  # -1 signals FAIL, 0 signals STOP
      { 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.[73]

10.1.6 Returning

Other, positive, `skip` values transmit yet other signals. The value 3 forces a RETURN (with relevant stack trim if marked). The return value from RunWhole-Script has significance: -1 signals failure, 1 is 'normal', 0 is terminated (STOP).

```
    if ( $skip == 3 ) # RETURN
       { $skip =1; # after RETURN we continue!
         last;
       };
  };  # end of inner foreach $l
return $skip;
}
```

[72] What should we do with UNMARK if $MARK is already zero?

[73] In our initial incarnation, UNMARK did *not* haul stuff off the stack above the current mark but we trashed this idea on the PDA, forcing it to do so, and have now carried the amendment back to Perl.

10.1.7 Decreasing the marker

We 'commonly' have to decrease the stack marker index $MARK, that we write a separate routine to do just this:

```
sub DecMarker
{
  &Print("\n Unmarking: $MARK [@MARKERS]");
  $#CMDSTACK = $MARKERS[$MARK];
  if ($MARK > 0)
     { $MARK --;
       # should we not "$#MARKERS = $MARK" ???
     } else
     { $MARKERS[$MARK] = -1;
     };
  return;
}
```

Trivial, really, but should we warn if $MARK is already -1?

10.2 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.

10.2.1 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 10.1.1, 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, $pop) = @_;
      my($outcome); # see usage

#   print LOGFILE '||'; # debugging for flow
```

```
XPrint ("\n    stack: <@CMDSTACK> \n"); # debugging
my ($i, $s);
XPrint (": line: <$l>");  # debug
$l =~ / *(.+) */; # clip
$_ = $1;
```

After a couple of debugging statements (for XPrint see section 10.14 far below), we clip leading and trailing spaces off the command.

10.2.2 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.[74] We insert stack values starting on the *right*![75]

10.2.3 Parenthetic argument

In the following regex we look for a routine name followed by something in parenthesis, and if found, strip out the argument in parenthesis and push it to the stack. The routine name begins with either an ampersand (&) or an equals sign.

```
if ( /^([=\&]*\w+) *\(#*(.*)\)$/ )
   { push (@CMDSTACK, $2);
#     print LOGFILE "\n Parenthetic<$2><@CMDSTACK>";
      $_ = $1;
   };
```

See how, in the Perl program, we ignore a leading hash in the parenthetic value. This is because we don't (yet) use the strong typing of the PDA program within the Perl, so we don't need to identify an integer value. [HMM].[76]

We will commonly specify arguments after an &routine thus:

&Fred(argument here)

[74] 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.

[75] We must also consider what to do if stack pop fails because the stack is empty!

[76] Actually, we might here even have multiple hashes as per the regex; this is a silliness.

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.

10.2.4 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+)$/ )
  { $outcome = &Invoke($myODBC, $newW, $1, $idx, '', $pop);
    if ($outcome != 0) # specific handling of SKIP condition ???
       { return $outcome;
       };
    return 1; #hmm??
  }
elsif ( /=(\w+)$/ )
  { $outcome = &Invoke($myODBC, $newW, $1, $idx, '', $pop);
    return (3);  # force return, WHATEVER (!)
  }
elsif ( /#(.+)/ )  # if inline integer
  {
     push (@CMDSTACK, int($1));
     return 1; # ok
  }
elsif ( /%(.+)/ ) # if float
  {  push (@CMDSTACK, $1); # Perl default is float!
  }
```

Invoke can return success, failure, or even SKIP![77] We intercept STOP, preventing complete cessation of a script, otherwise the STOP generated by URZN (which is useful in the context of REPEAT) halts scripts in an undesirable fashion!

If the character preceding the routine name is an equal sign, then we *force* a RETURN after the invocation, terminating the current script![78]

We've added the ability to unequivocally enter integers by preceding them with a hash (pound) character.[79]

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

[78] We MUST explore the consequences of failure in the invoked script under these circumstances. At present, whatever happens,we simply return, well, RETURN. It might be wise to have the ability to both return and then STOP, or whatever??

[79] The above is a clumsy hack and baad things may happen if we enter floating point numbers and expect these to be converted to integers on the PDA. Only say #123 *never* #123.4. To convert, do so directly or use INTEGER(123.4).

```
elsif ( /@.{4}/ )
  { return 1; # ok
  }
```

In the above, we check for a string in the format @nnnn, where n is any character. A stub at present, this permits later extension (script optimisation on the PDA, mainly).

10.2.5 Implement REPEAT

```
if (  /^repeat$/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, '', $pop)
       };
     if ($BUG & 16)
        { &Print ("\n Exit REPEAT, code $i"); # jvs
        };
     $Stopped = 0; # know is 1, no extra info thus clear!!
#       print LOGFILE "\n STOPped 6";
     if ($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).[80]

10.2.6 A quoted item

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

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

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

10.3 SQL commands

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

10.3.1 QUERY

```
if (/^QUERY$/i)
   { my (@sq);
     $i = pop(@CMDSTACK); # get SQL
     (@sq) = &GetSQL($myODBC, $i, "get one SQL [row] HMM?");
    if ($SQLOK)
       { push (@CMDSTACK, @sq); # can push a null!
       }; # amended 2007-10-20 to retrieve _row_ [hooray!]
    return 1;
   }
```

As usual, SQLOK is used to signal the success/failure of the SQL statement. There is a subtle problem here — there's a difference between returning a value of NULL and failing! The QUERY function might still *succeed* despite the length being zero if null was returned. As GetSQL sets SQLOK on failure, we surely don't need another test here, and have therefore remmed out the erroneous code!

10.3.2 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.[81]

10.3.3 QMANY

Here we retrieve multiple rows from SQL.

[81] It might be wise to check for the presence of SELECT before we even submit the statement, but we don't do so at present!

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

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

10.3.5 QOK

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

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

10.3.6 COMMIT

See section 5.1.5.

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

10.3.7 ROLLBACK

As for COMMIT, see section 5.1.5.

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

10.3.8 ME and SETME

Strictly speaking ME 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);
   }
```

To alter the value of ME ($CURRENTUSER), use SETME:

```
elsif ( /^setme$/i )
   { $i = pop(@CMDSTACK);
     if ( $i =~ /^\d+$/ ) # if numeric
        { $CURRENTUSER = $i;
        } else
        { &Alert($MAINW, "Bad user: <$i>");
        };
   }
```

10.4 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.

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

10.4.2 NEG

Negate a number on the stack.

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

10.4.3 NOT

If anything other than 0 on the stack, return 0. If zero, return 1.[82]

```
elsif (/^not$/i)
   { $i = pop(@CMDSTACK);
     if ( $i =~/^ *0+ *$/ )
        { push (@CMDSTACK, 1);
        } else
        { push (@CMDSTACK, 0);
        };
   }
```

10.4.4 ADD

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

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

10.4.5 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);
   }
```

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

10.4.6 DIV

Similar to SUB. Divide deeper number by topmost (more superficial) one.[83]

```
elsif (/^div$/i)
   { $i = pop(@CMDSTACK);
     $_ = pop(@CMDSTACK);
     if ( ( $i =~ /[\.e]/ )
        ||( $_ =~ /[\.e]/ )
        )               # crude test for floating point
        { $_ /= $i;     # floating division
        } else
        { $_ /= $i;
          $_ = int($_); # force integer, stop helpful Perl [ugh]
        };
     push (@CMDSTACK, $_);
   }
```

10.4.7 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, $_);
   }
```

10.4.8 MUL

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

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

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

10.4.9 SAME

Check for *identical* items on the stack.[84] Of limited, well actually, no utility with floating point numbers.

```
elsif (/^same$/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.

10.4.10 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 (/^greater$/i )
   { $_ = pop(@CMDSTACK);
     $i = pop(@CMDSTACK);

     # hmm what if string rather than number?
     if ( (/^-?\d+$/)
        &&($i =~ (/^-?\d+$/))
        )
        { if ($i > $_)
            { push (@CMDSTACK, '1');
            } else
            { push (@CMDSTACK, '0');
            }
        } else
        { if ($i gt $_)
            { push (@CMDSTACK, '1');
            } else
            { push (@CMDSTACK, '0');
            };
        };
   }
```

The code is clumsy.

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

10.4.11 LESS

As for GREATER above.

```
elsif (/^less$/i )
   { $_ = pop(@CMDSTACK);
     $i = pop(@CMDSTACK);

     # hmm what if string rather than number?
     if ( (/^-?\d+$/)
        &&($i =~ (/^-?\d+$/))
        )
        { if ($i < $_)
            { push (@CMDSTACK, '1');
            } else
            { push (@CMDSTACK, '0');
            }
        } else
        { if ($i lt $_)
            { push (@CMDSTACK, '1');
            } else
            { push (@CMDSTACK, '0');
            };
        };
   }
```

10.4.12 AND

If two ones on the stack, return 1, otherwise 0. Typical Boolean logic.[85]

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

10.4.13 OR

Similar Boolean logic to AND. Return 1 if either value is one.[86]

[85] Agonise over other possible values and interpretations. What about NULL?
[86] No? Perhaps better to return 1 unless both are zero??

```
elsif (/^or$/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.

10.4.14 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 (/^isnumber$/i)
   { $_ = pop(@CMDSTACK);
     if (/^-?\d+$/)  # 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.

10.4.15 INTEGER

See 10.8.2.

FLOAT

See 10.8.5.

10.4.16 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 (/^boolean$/i)
   { $i = pop(@CMDSTACK);
     if (   ( $i =~ /^[0|f|false]$/i )
         || (length $i == 0)
        ) # ugly test
        { $i = '0';
        } else
        { $i = '1';
        };
     push (@CMDSTACK, $i);
   }
```

10.4.17 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, "");
   }
```

10.5 Flow of control and stack commands

10.5.1 RETURN

The magic value of 3 is discussed in Section 10.1.6 above.

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

10.5.2 STOP

STOP forces termination (without prejudice) in a variety of settings.[87] Particularly useful with REPEAT (See section 10.2.5).

```
elsif ( /^stop$/i )
   { $Stopped=1;
     return 0;
   }
```

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

10.5.3 FAIL

FAIL is more severe — if invoked, creation of an individual widget ceases immediately. We have a global FAILURE variable which we set.

```
elsif ( /^fail$/i )
   {$FAILURE = 1;
     return 0;
   }
```

10.5.4 SKIP

As discussed above (Section 10.1.2), 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.

```
elsif ( /^skip$/i )
   { $i = pop(@CMDSTACK);
     if ( $i !~/^1$/ )
        { return 1;   # do NOT skip if zero
        };
     if ($BUG & 16)
        { &Print ("[SKIPPED]");
        };
     return 2; # force skip
   }
```

10.5.5 CACHE

This experimental routine has no function on the desktop, as it's confined to the PDA. Here, all it does is discard one value from the stack.

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

We also have an uncaching function, which we likewise ignore:

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

10.5.6 TEST

This also does nothing.

```
elsif ( /^test$/i )
   { # do nil.
   }
```

10.5.7 COPY

Make an exact copy of the top of the stack.[88]

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

10.5.8 DISCARD

Discard the top item on the stack!

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

10.5.9 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, $_);
   }
```

10.5.10 REPLACE

Replace the item below the topmost one, with the top 1. It's really discarding the next but topmost item, but will usually be stated along the lines of:

```
->REPLACE(#123)->
```

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

(Often with a skip before the REPLACE). Here's the Perl code:

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

10.5.11 BURY

The previous implementation of BURY (and DIGUP) in Perl was convenient (using unshift and shift) but had the drawback that if we unshifted enough things, they appear on the stack top again. We re-implemented these functions using a method similar to that we employ on the PDA, which is more secure.

Peculiar little commands, but most useful.

```
elsif (/^bury$/i)
   { $i = pop(@CMDSTACK);
     push(@BURYSTACK, $i);
     XPrint ("\n      bury: <@BURYSTACK>"); # debugging
   }
```

10.5.12 DIGUP

See the note under BURY above (Section 10.5.11).

```
elsif (/^digup$/i)
   { $i = pop(@BURYSTACK);
     push(@CMDSTACK, $i);
     XPrint ("\n      bury: <@BURYSTACK>"); # debugging
   }
```

10.5.13 MARK

Until recently (11/05) the MARK and UNMARK commands were far more primitive (at present) than the PDA implementation. WE've now addressed this. We've briefly discussed MARK above (Section 10.1.3).

```
elsif ( /^mark/i )
   { $i = pop(@CMDSTACK); # mark index
            if ($BUG & 16)
               { &Print ("\n DEBUG: mark index is $i");
               };
     if (($i > 16) || ($i < -16)) # [hmm?]
        { &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.[89] Do NOT submit negative values to MARK, despite the fact that it will accept them!

10.5.14 UNMARK

Return -2 as a control code.

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

Also look at Section 10.1.3.

10.5.15 DEPTH

```
elsif ( /^depth/i )
   { $i = $#CMDSTACK - $MARKERS[$MARK];
     if ($i < 0)
        { $i = 0;
        };
     push(@CMDSTACK, $i); # store count
     # &Alert($MAINW, "Debug: depth is $i"); # ???
   }
```

10.5.16 URZN

As noted above (Section 10.1.4), 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 ( /^urzn/i )

   {  if ($#CMDSTACK <= $MARKERS[$MARK]) # nothing on stack
         { return -4;
           # ?? we might signal error if '<'
         };
     $i = pop(@CMDSTACK);
```

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

```
        if (    ((length $i) < 1)    # null on stack
             || ($i == 0)            # OR zero on stack
             || (! $SQLOK)           # OR SQL failed
           )
           { return -4;
           };
        push(@CMDSTACK, $i); # restore!
      }
```

Should we still succeed if the stack is just plain empty??[90] Should SQLOK cascade or be reset by URZN? [CHECK ME?]

10.5.17 RUN

This powerful command takes a string off the stack and runs it as a script.[91]

```
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 10.14.2).

10.6 Single letter commands and their friends

10.6.1 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.[92]

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

Also have a look at SetX.

[90] CHECK this vs the PDA?
[91] Check out the potential for abuse!
[92] Fine print: in the past we rendered this $[X], which is really cumbersome.

10.6.2 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.[93]

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

10.6.3 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.

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

10.7 General purpose/text commands

10.7.1 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. The code is also clumsy.

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

10.7.2 SPLIT and JOIN

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
        $i =~ s/\./\\\./g;
        $_ = pop(@CMDSTACK); # string to split
        @spl = split /$i/;
#        print LOGFILE "\n Debug: split <@spl>";
        if ($#spl < 0)
           { push (@CMDSTACK, '');
           } else
           { push (@CMDSTACK, @spl);
           };
      }
```

I love Perl in such circumstances. We need to look at `/$i/` in terms of hacks. Now let’s join:

```
   elsif (/^join$/i )
      {
        $i = pop(@CMDSTACK); # join sequence
        my (@j);
        my ($mkr, $stklen, $topi);
        $mkr = $MARKERS[$MARK];
        $stklen = $#CMDSTACK;

        # get number of items on stack
        if ($stklen - $mkr < 1)
          { &Alert($newW, "Join failed: too few arguments");
            return -1;
          };

        # pull off that many items
        @j = @CMDSTACK[$mkr+1..$stklen];
        @CMDSTACK=@CMDSTACK[0..$mkr];
        $topi = pop(@j);

        # concatenate them, interposing the join string
        $s = '';
        foreach (@j)
            { $s = "$s$_$i" ;
            };
        $s = "$s$topi"; # last item on end!
        push (@CMDSTACK, $s); # push result.
      }
```

10.7.3 LENGTH

Determine the length of a string.[94]

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

10.7.4 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);
   }
```

10.7.5 LOWERCASE

Similar to UPPERCASE.

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

10.7.6 CUT

Cut a string into two at a stated character offset:

```
elsif (/^cut$/i)
   { $i = pop(@CMDSTACK);
     if ( $i !~ /^\d+$/)
        { Alert("Bad CUT integer: <$i>");
          return -1;
        };
     $s = pop(@CMDSTACK);
     if ($s !~ /^(.{$i})(.*)$/ ) # match $i chars
        { push (@CMDSTACK, $i);
          push (@CMDSTACK, ""); # push null
```

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

```
        } else
        { push (@CMDSTACK, $1);
          push (@CMDSTACK, $2);
        };
    }
```

If there are insufficient characters, simply push the original string to the stack, and on top of this, a NULL.

10.8 Date and time, etc.

This section now needs reworking on the PDA.

10.8.1 NOW

A simple timestamp — now! Well, not entirely simple, if you examine Section 10.1.1. The timestamp in TODAY remains fixed for the duration of the executing script![95]

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

To set the time, we should use SETTIME but on the desktop we have disabled this facility, simply returning false (failed):

```
elsif (/^settime$/i)
   { $i = pop(@CMDSTACK); # discard (for now)
     &Alert($MAINW, "Cannot yet set date on desktop! Use Windows.");
     push (@CMDSTACK, 0); # 20080518
   }
```

We also need a whole lot of conversion utilities. We will be able to represent dates as (Julian) floating point numbers, integers (Julian days), and of course as a text string in the YYYY-MM-DD format.

We will be inconsistent with translating times, as we will translate to seconds and not to part of a day (as float or integer)! To convert from time as seconds to part of a day, the user must divide by 86400.

We must carefully consider the implications of fractional seconds (with eg 6 digits after the decimal point) — on the PDA this is very wasteful of storage and generally undesirable.

[95] Explore the wisdom of this choice!

There is another issue in Perl. As Perl has the odd idea of representing things as floats by default, and no strong typing, we must be careful. We need to identify TIME, DATE and TIMESTAMP using their formatting, and convert as required.[96]

10.8.2 INTEGER

Convert a number (force, coerce it!) into an integer.[97]

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

    if ($i =~ /^[+-]?\d+$/ )
        { # integer: do nothing
        }
    else
        { $i = &FixFloat($i);
          if (length $i > 0)
             {$i = int($i+$EPSILON);
             };
        };
    push (@CMDSTACK, $i);
  }
```

10.8.3 DATE

Given a Julian date (float or integer) we render a Gregorian date as YYYY-MM-DD. Similar to TIMESTAMP. Providing a TIME is meaningless.

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

    my ($dgy, $dgm, $dgd, $dok);
    $dok = 0;

    if ( $i =~ /^(\d{4})-(\d{1,2})-(\d{1,2})$/ )
       { # date alone, NOT timestamp!
         $dgy = $1;
         $dgm = $2;
         $dgd = $3;
         $dok =1;
       }
    elsif ( $i =~ /^([+-]?)(?=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+))?$/ )
       { # floating point number (or integer)
```

[96] At some stage we should probably add on some typing, e.g. with a unique prefix!

[97] 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!

```
          # here convert TO Gregorian date string:
             ($dgy, $dgm, $dgd) = &Gregorian($i);
          $dgd = int($dgd);
          $dok =1;
        }
      else
        { $i = ""; # force null
        };
      if ($dok)
        {
          $dgm = &DoubleDigit($dgm);
          $dgd = &DoubleDigit($dgd);
          $i = "$dgy-$dgm-$dgd";
        };
      push (@CMDSTACK, $i);
    }
```

10.8.4 TIME

Given a float or integer, we convert from *seconds* to HH:MM:SS! Given a timestamp we pull out the time portion. Providing a DATE is meaningless so we return NULL.

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

       my ($th, $tm, $ts, $tok);
       $tok = 0;

       if ( $i =~ /(\d{2}):(\d{2}):(\d{2})(\.\d+)?$/ )
          { # time: ignore fractional seconds
            # also picks up time part of timestamp [ugly]
            $th = $1;
            $tm = $2;
            $ts = $3;
            $tok = 1;
          }
       elsif ( $i =~ /^([+-]?)(?=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+))?$/ )
          { # floating point number (or integer) as SECONDS
            # here convert TO time:
            ($th, $tm, $ts) = &HrMinSec ($i/86400);
            $ts = int($ts);
            $th %= 24; # modulo 24! [CAUTION: ? warn]
            $tok = 1;
          }
       else
          { $i = ""; # force null
          };
```

```
    if ($tok)
       {
         $th = &DoubleDigit($th);
         $tm = &DoubleDigit($tm);
         $ts = &DoubleDigit($ts);
         $i = "$th:$tm:$ts";
       };
    push (@CMDSTACK, $i);
  }
```

Note the float inconsistency with our Julian functions, where HH:MM:SS are regarded as a fraction of a day.

10.8.5 FLOAT

FLOAT is the default type of Perl (eugh). When 'converting' from 'text' to float, we replace the value with NULL if the function fails! At present we permit a C-style float! An integer is simply seen as a subset of floats, and at present our Perl makes no distinction between float and fixed point[98]

In addition, if we encounter something which is clearly a TIMESTAMP, we convert this to a (Julian) floating point number. A DATE we convert to a Julian date. Rather than becoming a fraction of a day, a time becomes a number of seconds (our idiosyncrasy).

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

One idea is to allow some leeway in terms of converting e.g. dd-mm-YYYY into YYYY-mm-dd, and perhaps some other conversions/loosening; better would be to relegate this functionality to user-defined code.

10.8.6 TIMESTAMP

Given a floating point (Julian) date, we create the equivalent Gregorian timestamp.[99] An integer is regarded as a Julian day number; if we submit just a DATE then this is converted to a TIMESTAMP, but what about a TIME? It seems peculiarly meaningless to convert a time alone to a timestamp, so this operation gives NULL. A string timestamp is validated as a 'true' timestamp. We allow some leeway in the digits used for month, day, and time components (1 or 2).

[98] This must be fixed as per 754r.

[99] Gregorian dates start on 15-Oct-1582, earlier extensions are termed 'proleptic'.

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

    my ($gy, $gm, $gd);
    my($gh, $gmi, $gs, $gok);
    $gok = 0;

    if ($i =~
    /^(\d{4})-(\d{1,2})-(\d{1,2}) (\d{1,2}):(\d{1,2}):(\d{1,2})(\.\d+)?$/ )
       { $gy = $1;
         $gm = $2;
         $gd = $3;

         $gh = $4;
         $gmi = $5;
         $gs = $6; # ignore fractional seconds [?]
         $gok = 1;
       }
    elsif ( $i =~ /^(\d{4})-(\d{1,2})-(\d{1,2})$/ ) # date
       { $gy = $1;
         $gm = $2;
         $gd = $3;
         $gh = 0;
         $gmi = 0;
         $gs = 0;
         $gok = 1;
       }
    elsif ( $i =~ /^([+-]?)(?=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+))?$/ )
       { # floating point number (or integer)
         # here convert TO Gregorian date string:
            ($gy, $gm, $gd) = &Gregorian($i);

         $gh = $gd;
         $gd = int($gd);
         $gh -= $gd;
         ($gh, $gmi, $gs) = &HrMinSec($gh);
         $gs = int($gs); # get rid of decimal part! [?]
         $gok = 1;
       }
    else
       { $i = ""; # force null
       };

    if ($gok)
       { $gm = &DoubleDigit($gm);
         $gd = &DoubleDigit($gd);
         $gh = &DoubleDigit($gh);
         $gmi = &DoubleDigit($gmi);
```

```
        $gs = &DoubleDigit($gs);
        $i = "$gy-$gm-$gd $gh:$gmi:$gs";
    };
    push (@CMDSTACK, $i);
  }
```

The above will render years under 1000 AD with under 4 digits in the YYYY position.

10.8.7 TICKS

The following must be fixed: should be used for relatively high-resolution timing (get current timer tick). We want resolution of 10ms, noting that with DPCs in the Win2K(+) HAL, the resolution is often far, far worse than this.

```
elsif (/^TICKS$/i)
       { # here must get clock ticks
         # my $tck = gettimeofday(); # (package Time-HiRes)
         my $tck = 0; # Time-HiRes not installed.
         push (@CMDSTACK, $tck);
         #
       }
```

10.9 Menu-related commands

10.9.1 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.

10.9.2 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);
    }
```

10.9.3 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);
   }
```

10.9.4 EXIT

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

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

10.9.5 PRINT

Print a string to the console. Note that Perl isn't fussy but PDA at present requires a *string*, nothing else will do!

```
elsif (/^PRINT$/i)
       { $i = pop(@CMDSTACK);
         $i =~ s/\\n/\n/g;
         print ($i); # direct console print
         # might disable this using ${CONST} rather than here!
       }
```

10.9.6 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, 0);
     return -1;  # force "fail" (MUST do d/t recursion!)
   }
```

10.9.7 ENABLED

We use this simple command with its single stack argument of either 1 or zero[100] to enable or disable a widget. The widget is indexed into TKITEMS using `idx`.

```
   elsif ( /^enabled$/i )
      { $i = pop(@CMDSTACK);
        if ( $i )  # if nonzero
           { $TKITEMS[$idx]->configure (-state => 'normal');
#             print LOGFILE "\n <$idx> _ENABLED_";
           } else
           { $TKITEMS[$idx]->configure (-state => 'disabled');
#             print LOGFILE "\n <$idx> _DISABLED_";
           };
      }
```

10.9.8 POPMENU

In a previous incarnation, this command clipped the preceding menu from the menu stack. Such an approach was inflexible. The new version [should] take an integer value off the stack, and clip the relevant menu out, or return an error if the operation is impossible. Be extremely careful if you clip out the current menu (by submitting an integer value of zero). Integer values should otherwise be positive.

The menu name and X value are placed on the stack, with X deep to the menu name.

```
   elsif ( /^POPMENU$/i )
       { my($jom,$jox,$joi);
         $i = pop(@CMDSTACK);
         # might check here $i is numeric!
         $joi = $#MENUS;  # number of items
         if ($i > $joi)
            { &Alert($MAINW, "POPMENU too shallow.");
            } else
            { $i = -(abs $i);  # force -ve
              $i += $joi;
              $jom = $MENUS[$i]; # get menu
              $jox = $X[$i]; # likewise for Xfer variable
              @MENUS = @MENUS[0..$i-1, $i+1..$joi];
              @X = @X[0..$i-1, $i+1..$joi];
              push (@CMDSTACK, $jox); # push X
              push (@CMDSTACK, $jom); # push menu name!
            };
      }
```

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

We index from the top of MENUS and X, clipping out that menu and X value.

10.9.9 PUSHMENU

Even more odd than POPMENU is PUSHMENU, which given an index (on the stack top) as well as a menu name, and deep to this an X value, uses the index to insert the other values into the MENUS and X arrays! It too has been ‘upgraded’ from the original primitive command.

[DEBUG THE FOLLOWING!]

```
elsif ( /^PUSHMENU$/i )
    { my($kom,$kox,$koi);
      $i = pop(@CMDSTACK);
      # might check here $i is numeric, others exist:
      $kom = pop(@CMDSTACK);
      $kox = pop(@CMDSTACK);
      $koi = $#MENUS;  # number of items
      if ($i > $koi)
         { &Alert($MAINW, "PUSHMENU too shallow.");
         } else
         { $i = -(abs $i);  # force -ve
           $i += $koi;
           @MENUS = @MENUS[0..$i, $kom, $i+1..$koi];
           @X = @X[0..$i, $kox, $i+1..$koi];
         };
    }
```

10.9.10 ROLLMENU & LINESLEFT

An interesting wrinkle! If the value in the LOCALROLL is nonzero, then we GoMenu forcing a reload of a copy of the current menu (above this one) with the current offset.

```
elsif ( /^ROLLMENU$/i )
   { if ($LOCALROLL)
        { &GoMenu ($myODBC, 0, $newW, 1);
          return -1;  # force "fail" (MUST do d/t recursion!)
        };
     # otherwise simply ignore!
   }
```

Submitting a final ‘1’ parameter to GoMenu forces a rolling of the current menu. Here’s the related LINESLEFT:

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

10.9.11 TITLE

TITLE simply sets the title of the current menu.

```
elsif ( /^TITLE$/i )
  { $i = pop(@CMDSTACK);
    $MENUW->title($i);
  }
```

10.10 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 ...

10.10.1 NAME

We make a name using NAME.

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

10.10.2 $[name]

We devote a whole section below (10.15) to functions such as FetchLocal, which deal with local variables.

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

10.10.3 SET

SET sets the value of a local variable. See the discussion of `$[name]` above, and section 10.15.5.

```
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!

10.11 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!

10.11.1 PAPER

Set the paper colour (background).

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

10.11.2 INK

Set the ink colour (foreground).

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

10.11.3 TOGGLE

Swop the paper and ink colours.

```
elsif ( /^toggle/i )
   { $TOGGLED = 1;
   }
```

10.12 Experimental, obsolete and debugging routines

Avoid using these, or use them with extreme caution!

10.12.1 DEBUG

Alter the debugging flags by writing directly to BUG.

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

10.13 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/parenthesis?)");
        &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!

10.14 Subsidiary routines

Here are some of the routines referred to above.

10.14.1 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.

10.14.2 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 10.1.1).

```
sub Invoke
{ my ($myODBC, $newW, $fxname, $i, $noargs, $pop);
     ($myODBC, $newW, $fxname, $i, $noargs, $pop) = @_;
  $Stopped = 0; # clear flag (see usage)
```

```
#  print LOGFILE "\n stopPED 7";

  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, 0x0d);
}
```

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

10.14.3 Julian day calculations

For all AD *Gregorian* dates, the Julian date (created by Joseph Scaliger in 1583) is:

$$J = 367Y - \text{int}\left(\frac{7(Y + \text{int}(\frac{M+9}{12}))}{4}\right) - \text{int}\left(\frac{3(\text{int}(\frac{Y+\frac{M-9}{7}}{100}) + 1)}{4}\right) + \text{int}\left(\frac{275M}{9}\right) + D + 1721028.5 + \frac{UT}{24}$$

This number measures time from *noon* on January 1, 4713 BC. Note that it's from noon, not midnight. In the following we submit year, month, day, hours, minutes and seconds, and finally the digits after the decimal point, or zero if there are none.

We were tempted to use Peter Meyer's *Chronological Julian Day* which is the Julian day number *plus* 0.5, and relates to the *local* time, not GMT. We didn't, however.

In order to convert to UTC, we have to add or subtract a LOCALTIMEOFFSET constant. For New Zealand, local time is UTC+12 hours, so to obtain UTC from the system, ie local time, we must *subtract* 12 hours! (Our timestamp provided by the NOW function has added in 12 hours).

We also must adjust for local daylight saving! For each country which uses DST, similar calculations are required. The following assumes that DST has been calculated and set correctly! This is a pain, as rules for DST vary from country to country. For example, in New Zealand, DST starts at 2am on the first Sunday in October, and ends at 3am on the 3rd Sunday in March. During DST, you should set the variable DAYLIGHTSAVING to a value of 1. It must otherwise be zero.

```
sub Julian
{ my ($fy, $fm, $fd, $fh, $fmi, $fs, $ff);
     ($fy, $fm, $fd, $fh, $fmi, $fs, $ff)=@_;

  my ($f);
  $f= 367*$fy - int(7*($fy+int(($fm+9)/12))/4)
              - int(3*(int(($fy+($fm-9)/7)/100)+1)/4)
              + int(275*$fm/9)+$fd+1721028.5
              - ($LOCALTIMEOFFSET+$DAYLIGHTSAVING)/24
              + ($fh + ($fmi + ($fs+ "0.$ff")/60)/60)/24;
  return $f;
}

# similarly, given date string, return julian value:
sub JD
{ my ($d);
  ($d) = @_;
  if ($d !~ /(\d{4})-(\d{2})-(\d{2}) (\d{2}):(\d{2}):(\d{2})/ )
     { return (0);
     };
  return (&Julian ($1, $2, $3, $4, $5, $6, 0));
}
```

We return the floating point Julian date.

10.14.4 Gregorian date

Baum's algorithm follows. In his definition, FIX is the number without its fraction, and INT is the floor function (as in C++ or C).

```
Z = INT(JD - 1721118.5)
R = JD - 1721118.5 - Z
G = Z - .25
A = INT(G / 36524.25)
B = A - INT(A / 4)
year = INT((B+G) / 365.25)
C = B + Z - INT(365.25 * year)
month = FIX((5 * C + 456) / 153)
day = C - FIX((153 * month - 457) / 5) + R
IF month > 12 THEN
       year = year + 1
       month = month - 12
END IF
```

In Perl the INT function simply removes the decimal value, returning the integer portion of a number. Version 5 includes a POSIX module defining the standard

C functions floor() and ceil().
We simply say `use POSIX qw(floor)`

Here's our function. See how we adjust for local time and daylight saving *before* we convert:

```
sub Gregorian
{ my ($jd);
     ($jd)=@_;

 $jd += ($LOCALTIMEOFFSET+$DAYLIGHTSAVING)/24;
 $jd += $EPSILON; # 1 != 0.99999999...99

  my ($Z, $R, $G, $A, $B, $C);
  my ($year, $month, $day);

$Z = floor($jd - 1721118.5);
$R = $jd - 1721118.5 - $Z;
$G = $Z - 0.25;
$A = floor($G / 36524.25);
$B = $A - floor($A / 4);
$year = floor(($B+$G) / 365.25);
$C = $B + $Z - floor(365.25 * $year);
$month = int((5 * $C + 456) / 153);
$day = $C - int((153 * $month - 457) / 5) + $R;
if ($month > 12)
   { $year = $year + 1;
     $month = $month - 12;
   };
return ($year, $month, int($day));
}
```

We return the year, month and day. Day is a float, including a part of the day. Recall that a Julian date is relative to midday.

10.14.5 Hours, minutes and seconds

Given a fractional date, convert to hours, minutes and seconds. Seconds is a float.

```
sub HrMinSec
{
   my ($gd);
      ($gd) = @_;

   my($gh, $gmi, $gs);
            $gh = $gd; # clumsy
            $gh *= 24;
            $gmi = $gh;
```

```
            $gh = int($gh);
            $gmi -= $gh;
            $gmi *= 60;
            $gs = $gmi;
            $gmi = int($gmi);
            $gs -= $gmi;
            $gs *= 60;
   return ($gh, $gmi, $gs);
}
```

10.14.6 DoubleDigit and date fixing

If a number n is single digit i.e. under 10, convert to the string ”0n”! We don’t check for negative numbers, which will screw things up.

```
sub DoubleDigit
{
   my ($i);
      ($i) = @_;

   if (length $i > 1)
      { return $i;
      };
   return "0$i"; # concatenate.
}
```

We use the above to fix a date where a single digit has been used e.g. the digits 2007, 1 and 3 become ‘2007-01-03’:

```
sub FixDate
{
   my ($y, $m, $d);
      ($y, $m, $d) = @_;

   $m = &DoubleDigit($m);
   $d = &DoubleDigit($d);

   return ("$y-$m-$d");
}
```

10.14.7 Fixing up a Float

Given arguments in various formats, return a float, or the float equivalent of a date, timestamp, integer, or time.

```
sub FixFloat
{
  my ($i);
  ($i) = @_;

  my ($fy, $fm, $fd, $fh, $fmi, $fs, $ff, $okf);
     $okf = 0;

       if ($i =~ /^([+-]?)(?=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+))?$/ )
          { # do nothing!
#            print LOGFILE "\n Debug: <$i> ==> float";
          }
       elsif ($i =~
  /^(\d{4})-(\d{1,2})-(\d{1,2}) (\d{1,2}):(\d{1,2}):(\d{1,2})(\.\d+)?$/ )
          { $okf = 1; # timestamp
            $fy = $1;
            $fm = $2;
            $fd = $3;
            $fh = $4;
            $fmi = $5;
            $fs = $6;
            $ff = $7;
#            print LOGFILE "\n Debug: <$i> ==> timestamp";
          }
       elsif ($i =~ /^(\d{2}):(\d{2}):(\d{2})(\.\d+)?$/ )
          { $okf = 2; # time: signal
            $fy = 0;
            $fm = 0;
            $fd = 0;
            $fh = $1;
            $fmi = $2;
            $fs = $3;
            $ff = $4;
#            print LOGFILE "\n Debug: <$i> ==> time";
          }
       elsif ($i =~ /^\d{4}-\d{2}-\d{2}$/ )
          { $okf = 1; # date
            $fy = $1;
            $fm = $2;
            $fd = $3;
            $fh = 0;
            $fmi = 0;
            $fs = 0;
            $ff = 0;
#            print LOGFILE "\n Debug: <$i> ==> date";
          }
       else
          { print LOGFILE "\n Debug: <$i> ==> FLOAT ERROR";
```

```
            $i = "";
          };
       if ($okf)
          { # here calculate Julian [numeric] date
            if (length $ff < 1)
               { $ff = 0;
               } else
               { $ff =~ /.(.+)/;
                 $ff = $1;
               };
            $i= &Julian($fy, $fm, $fd, $fh, $fmi, $fs, $ff);
            if ($okf > 1) # clumsy: time
               { $i *= 86400; # seconds?! [ugly]
               };
          };
  return $i;
}
```

10.15 Local variables

We've briefly discussed these above (Section 10.10). 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).[101] 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:

10.15.1 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).

```
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.

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

10.15.2 KeepLocalNames

We have the ability to transiently store the local names, and restore them if a new menu load didn't work out![102]

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

10.15.3 RestoreLocalNames

As noted (See KeepLocalNames above).

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

10.15.4 CreateLocalName

Here we make a local variable name, using the associative array LOCALNAMES. The corresponding NAME command is discussed in section 10.10.1.

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

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

[102] See usage.

10.15.5 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,
       "Local var set '$name'='$value' FAILED! (Does it exist?)");
      return;
    };
  $LOCALARRAY[$i] = $value;
}
```

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

10.15.6 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
}
```

10.15.7 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.

11 Printing

There are various circumstances when we might wish to print reports based on the data in our database. The three most important reports are perhaps:

1. Printing a summary for a patient discharged from the Pain Service;
2. Printing feedback information for a particular doctor about their performance;
3. Printing summary statistics for the Pain Service for a given time interval.

We will eventually detail software for all of the above; we will start with the first.

11.1 Printing a discharge summary

Our strategy is to:

1. Identify admission processes (type 3 PROCESS) culminating in a recent discharge. There are several possible approaches here, the two most obvious being to print all process reports not yet generated (perhaps storing the date of the last printing), or to print reports on a 'daily' basis.

   We will choose the latter, but will allow periods longer than one day, and also record processes which have been printed, using the PRINTED table described in *AnalgesiaDBpart1.pdf*. In addition the program will give the user a separate option to print an individual discharge summary (recording this fact in the same table).

2. Retrieve all relevant details from the database, and organise these details, printing each report to a LaTeX file (with a .TEX suffix);
3. Externally invoke the program **pdflatex** to convert the .TEX file to a PDF file;
4. Batch print all the generated PDF files;
5. Transfer the generated files in an archive directory. Each PDF file will contain the time of printing. We should also archive a list of batch-printed files for each individual session.

Let's examine each of these processes in detail, but first the main driving routine:

```
sub PrintAllDischarges
{
  my ($myODBC, $userid);
     ($myODBC, $userid)=@_;

  if (! &Confirm($ADMINMENU,
       "Have you synchronised all the PDAs?" ) )
     {  &Alert($ADMINMENU, "Nothing printed.");
        return;
     };

  my($now, @plist);
    ($now, @plist) = FetchPrintList($myODBC);
  print LOGFILE "\n\n PRINTING at $now: \n list is (@plist) ";
  my $numfiles = 1+$#plist;
  if ($numfiles < 1)
     { &Alert($MAINW, "Nothing to print");
       return(0);
     };

  my (@outlist) = ();
  my ($id, $prid);
  $MAINW->focus();
  $ADMINMENU->iconify;
  $PROGTEXT = 'Preparing files..';
  $BARPROGRESS = 0;
+OPTIONAL
  $PROGRESSBAR->update();
-OPTIONAL
  my $barstep = $BARWIDTH/$numfiles;
  foreach $id (@plist)
    { $prid = FillUpLatexTemplate ($myODBC, $id, $now, 1); # many files -> COMMIT
      if ($prid > 0)
         { push (@outlist, $prid);
         };
      $BARPROGRESS += $barstep;
+OPTIONAL
      $PROGRESSBAR->update();
-OPTIONAL
    };
#  print LOGFILE "\n Output list is (@outlist) ";

  my $numfiles = 1+$#outlist;
  if ($numfiles < 1)
     { &Alert($MAINW, "Nothing to print");
       return(0);
     };
```

```
  $MAINW->focus();
  $ADMINMENU->iconify;
  $PROGTEXT = 'Printing files..';
  $BARPROGRESS = 0;
+OPTIONAL
  $PROGRESSBAR->update();
-OPTIONAL
  my $barstep = $BARWIDTH/$numfiles;

    foreach $id (@outlist)
      { &LatexToPdf ($id);
        $BARPROGRESS += $barstep;
+OPTIONAL
        $PROGRESSBAR->update();
-OPTIONAL
        &PrintPdf ($id);
      };
  $ADMINMENU->deiconify;
  $ADMINMENU->focus();

  # finally, write LASTPRINTED.DATA
  my $ok = 1;
  open LP, ">LASTPRINTED.DATA" or $ok = 0;
  if (! $ok)
     { &Alert($MAINW, "Could not store print time (LASTPRINTED.DATA)");
     } else
     { print LP $now;
       close LP;
     };

  # and commit!
  # &Commit($myODBC);
  $PROGTEXT = '';
  $BARPROGRESS = 0;
+OPTIONAL
  $PROGRESSBAR->update();
-OPTIONAL

  &Alert($ADMINMENU, "Printing completed!\n ($numfiles documents)");
  return(1);
}
```

11.1.1 Printing one discharge summary

The following subsections describe the subsidiary routines, but first, printing of *one* discharge:

```
sub PrintOneDischarge
```

```
{
  my ($myODBC, $userid);
     ($myODBC, $userid)=@_;

  my ($PATIENT);
  my ($nhi);
  $nhi = &Ask ($ADMINMENU,
     "Enter NHI",
     $nhi);
  if (length $nhi < 7)
     { if ( $nhi =~ /^\d+$/ )  # a key number!!
           { $PATIENT = $nhi;
           } else
           { &Alert($MAINW, "What?? <$nhi>");
             return;
           };
     } else
     { $nhi =~tr/a-z/A-Z/; # uppercase
       my (@ptids) = &GetSQL ($myODBC,
        "SELECT DISTINCT pdoPerson from PERSDATA WHERE pdoHospNo = '$nhi'",
        'get all hosp Nos');
       if ($#ptids < 0)
          {&Alert($MAINW, "No patients found for Hosp. No. <$nhi>");
           return;
          };
       if ($#ptids > 0)
          {&Alert($MAINW, "ERROR! Duplicates (@ptids) for Hosp. No. <$nhi>. Termin
           return;
          };
       $PATIENT = pop(@ptids);
     };

  # the following takes the most recent. LATER provide user with a list! ???
  my ($ptproc) = &GetSQL ($myODBC,
     "SELECT MAX(process) FROM PROCESS where \
      cold IS NOT NULL AND ProcType = 3 \
      AND process > -1 \
      AND Person = $PATIENT",
        "get most recent valid, terminated admission processes");
  if (length $ptproc < 1)
     { &Alert($MAINW, "No terminated admission for this person! Can't print!");
       return;
     };

  my($now) = &GetLocalTime();
  my ($prid) = FillUpLatexTemplate ($myODBC, $ptproc, $now, 0); # single file!
  if ($prid > 0)
     { &LatexToPdf ($prid);
```

```
      &PrintPdf ($prid);
    } else
    { &Alert($MAINW, "Failed to print! (Template error?)");
    };
}
```

11.1.2 Identify recent, unprinted discharges

There are two unfortunate problems with retrieving *all* unprinted records. If for some reason, however obscure, somebody doesn't want to print a particular record, that record will always pop up (and presumably have to be cancelled by an irate user). Another disadvantage is the SQL join syntax required to retrieve such data. Because we're looking for nonexistent items in the PRINTED table, we have to use an outer join along the lines of:

```
SELECT process FROM PRINTED RIGHT OUTER JOIN PROCESS USING(process)
  WHERE PRINTED.printed IS NULL
```

Outer joins are slow and the syntax varies depending on the SQL vendor. Mainly because of the last reason, we will simply fetch all discharges since the time of last printing, (stored by us in a file called *LASTPRINTED.DATA*) and then print only those not recorded as having been printed![103] Although even slower than an outer join, this approach has advantages in terms of minimising irritation. Another test we will perform is that if over three days have elapsed since the last printing, we will truncate the time to three days and warn the user that records prior to this have been ignored — if they wish to print these records, they need to do so separately.

Here's the code. We only look at admission processes (ProcType = 3).

```
sub FetchPrintList
{
  my($myODBC);
    ($myODBC) = @_;

  my (@plist);
  my ($lastok, $lastdate, $now, $nowjd);
  $lastok = 1;

  $now = &GetLocalTime();
  $now =~ /^(\d{4})-(\d{1,2})-(\d{1,2})/;
  $nowjd = &Julian($1, $2, $3, 0, 0, 0, 0);
```

[103] We will record this fact, and alert the user that 'N' records have not been printed as they are already recorded as having been printed off.

```
  open LASTPRIN, 'LASTPRINTED.DATA' or $lastok = 0;

  if ($lastok)
     { $lastdate = <LASTPRIN>;  # YYYY-MM-DD HH:MM:SS
       close LASTPRIN;
       $lastdate =~ /^(\d{4})-(\d{1,2})-(\d{1,2})/;
       my $lastjd = &Julian($1, $2, $3, 0, 0, 0, 0);
       if ($nowjd - $lastjd > $CONST{THREEDAYS})
          { $lastok = 0; # if over 3 days, force error
          };
     };

   if (! $lastok)
     { $nowjd -= $CONST{THREEDAYS}; # go back 3 days..
       my ($yyyy, $mm, $dd) = &Gregorian($nowjd);
       $lastdate = &FixDate($yyyy, $mm, $dd);
       $lastdate = "$lastdate 00:00:00";
       &Alert($MAINW, "Warning. No recent print date! Using $lastdate. \n \
  Prior records will NOT be printed!");
     };

  # Hmm. We should exclude those already printed, perhaps by:
  # AND PROCESS NOT IN (SELECT distinct process from PRINTED)
  # try this 2008-10-23:

  (@plist) = &SQLManySQL ($myODBC,
     "SELECT process FROM PROCESS where cold IS NOT NULL \
      AND ProcType = 3 \
      AND process > -1 \
      AND cast(rEnd as varchar(19)) > '$lastdate' \
      AND process NOT IN (SELECT process FROM PRINTED) \
      ORDER BY process desc",
        "get recent terminated admission (type 3) processes");  # ??: check me!
  # the cast is for Ocelot which thinks November comes before October (with timest


  return($now, @plist);
}
```

The returned 'plist' (which might conceivably be null) will also contain any processes which have actually been printed. We relegate checking for this to a subsequent routine.

We do not take the risk that printing will be cancelled at this stage, rather returning the current timestamp at the head of our returned list. Later, the caller will need to rewrite this value to LASTPRINTED.DATA.

11.1.3 Generate LaTeX file

The template sample file is contained in *template_summary.tex*. Note the requirement for the *fancyhdr* LaTeX package. Many substitutions must be made within this file before it can be printed. A full list of substitution variables is present as a LaTeX comment at the start of the file. Each substitution variable has the following format:

```
$[varname]
```

In other words, we have the name of a variable encased in square brackets and preceded by a dollar sign.

We here load the file, substitute in all the necessary data, and save the altered file under a unique name in the *latex* subdirectory of the *PainForm* directory. The unique name is simply the print key in the PRINTED table, where we create a new entry! We return this value to the calling routine, which has the responsibility of actually generating a list of items to be turned to PDF documents and then batch-printed, responsibilities *not* served by this routine.

We submit the ODBC handle, the ID of the type 3 admission process, the 'current' time, and the ID of the user requesting the printing. We also submit (30/1/2008) $MANYFILES, which allows us to print a single file if this variable is cleared to zero — in such a case, the printing is always to a file called '1.tex'. If MANYFILES is set, then we COMMIT the update to fix the PRINTED table.

```
sub FillUpLatexTemplate
{
  my($myODBC, $procid, $now, $MANYFILES);
    ($myODBC, $procid, $now, $MANYFILES) = @_;

  my ($userid) = &GetLastUser($myODBC, $procid);

#  print LOGFILE ("\n\n Printing LaTeX for Admission code: $procid, user: $userid'
```

First, we slurp in the template file *template_summary.tex*. We simply invoke the ancillary routine **SlurpFile**.

```
  my $template = &SlurpFile('template_summary.tex');
  if (length $template < 1)
     { return 0;
     };
```

Next we derive and substitute in the relevant values. First, we obtain basic patient demographics (etc). We obtain, in order, surname and forename (deriving

their first initial), gender, birth date (calculating age), NHI, weight, ASA rating, and date of death (if applicable). The nested select statements are required because our architecture records many of these data as observations (Surnames can change ...).

```
my ($patient, $STARTDATE, $ENDDATE) = &GetSQL($myODBC,
  "SELECT Person, rStart, rEnd FROM PROCESS WHERE process = $procid",
  "get patient");
my ($obsproc) = &GetSQL($myODBC,
  "SELECT MAX(process) FROM PROCESS WHERE \
    Proctype < 3 \
    AND cast (rEnd as varchar(19)) > '1910-01-01 00:00:00' \
    AND Person = $patient",
  "get most recent observation process for this patient");
  # hmm. might want to get other recent data!
if ($ENDDATE =~ /^(.*) / )
   { $ENDDATE = $1; # date alone
   };

my ($SURNAME) = &GetSQL($myODBC,
  "SELECT pdoSurname FROM PERSDATA WHERE \
    persdata = (SELECT MAX(persdata) FROM PERSDATA \
      WHERE pdoPerson = $patient AND pdoSurname IS NOT NULL)",
    "get most recent surname");
$SURNAME = &FixName($SURNAME);

my ($FORENAME) = &GetSQL($myODBC,
  "SELECT pdoForename FROM PERSDATA WHERE \
    persdata = (SELECT MAX(persdata) FROM PERSDATA \
      WHERE pdoPerson = $patient AND pdoForename IS NOT NULL)",
    "get most recent forename");
# what if null? [check me]
$FORENAME = &FixName($FORENAME);

my $INI ="-";
if ($FORENAME =~ /^(.+)/ )
   { $INI = $1;
   };

my ($GENDER) = &GetSQL($myODBC,
  "SELECT pdoGender FROM PERSDATA WHERE \
    persdata = (SELECT MAX(persdata) FROM PERSDATA \
      WHERE pdoPerson = $patient AND pdoGender IS NOT NULL)",
    "get most recent gender");
if ($GENDER == 1)
   { $GENDER = 'F';
   }
elsif ($GENDER == 2)
```

```
     { $GENDER = 'M';
     } else
     { $GENDER = '?';
     };

   $now =~ /(\d{4})-(\d{1,2})-(\d{1,2})/;
   my ($TODAY) = "$1-$2-$3";
   my $jnow = &Julian($1, $2, $3, 0, 0, 0, 0); # julian day
   my $AGE = "";

   my ($DOB) = &GetSQL($myODBC,
  "SELECT pBorn FROM PERSON WHERE person = $patient",
  "get date of birth");
  if (length $DOB > 0)
     { $DOB =~ /(\d{4})-(\d{1,2})-(\d{1,2})/;
       $DOB = "$1-$2-$3";
       my $jborn = &Julian ($1, $2, $3, 0, 0, 0, 0);
       $AGE = int($EPSILON + ($jnow - $jborn)/365.25); #ugly
       # [thought. What if they died and we're printing *now* ?]
       $AGE = "$AGE yr";
     };

  my ($NHI) = &GetSQL($myODBC,
    "SELECT pdoHospno FROM PERSDATA WHERE \
      persdata = (SELECT MIN(persdata) FROM PERSDATA \
        WHERE pdoPerson = $patient AND pdoHospno IS NOT NULL)",
      "get most recent hospital number");
  # as first hospital number is the major one, use this!

my ($WT) = &GetSQL($myODBC,
   "SELECT meWt FROM MEASURE,EPOCH \
     WHERE MEASURE.Epoch = EPOCH.epoch AND \
     EPOCH.Process = $obsproc",
   "get most recent weight");
  if (length $WT < 1)
     { $WT = "-"; # Hmm. ? OR check for *recent* prior process!
     } else
     { $WT = $WT/1000; # TO kilograms from grams
       $WT = "$WT kg";
     };

my ($ASAE) = &GetSQL($myODBC,
   "SELECT msoValue FROM MEDSCORE,EPOCH \
     WHERE MEDSCORE.Epoch = EPOCH.epoch AND \
     EPOCH.Process = $obsproc AND msoNature = 1",
   "get recent ASA E score");
   if ($ASAE == 0)
      { $ASAE = "";
```

```
      } else
      { $ASAE = 'E';
      };
my ($ASA) = &GetSQL($myODBC,
   "SELECT msoValue FROM MEDSCORE,EPOCH \
     WHERE MEDSCORE.Epoch = EPOCH.epoch AND \
     EPOCH.Process = $obsproc AND msoNature = 2",
   "get recent ASA 1--5");
$ASA = "$ASA" . "$ASAE";

my ($DOD) = &GetSQL($myODBC,
   "SELECT pDied FROM PERSON \
     WHERE person = $patient",
   "date of death?");
my ($ALIVE) = 'Yes';
if ($DOD) { $ALIVE = 'No';
          };
```

Next we wish to document all visits from the pain team. We will get all EPOCHs attached to the type 1 (observation) process. The following code is rather cumbersome.

```
  my ($TEMPLATEVISIT, $isvisit);
  $isvisit = 0;
  $TEMPLATEVISIT = '';

  my (@visits) = &SQLManySQL ($myODBC,
    "SELECT epoch FROM EPOCH WHERE Process = $obsproc",
    "get all observation epochs");
  my $PAINVISITTABLE = '';
  # technically might not be actual 'visits'. hmm.
#  print LOGFILE "\n List of visits (@visits)";

  my ($v);
  foreach $v (@visits)
    {
      my $ONEVISITLINE = '';
      my $ignore = 1;

      # Added in $whosaw (2008-09-23)
      my ($made, $duratn, $whosaw) = &GetSQL($myODBC,
       "SELECT oMade, oLength, Person FROM EPOCH \
          WHERE epoch = $v", "get epoch data");
      # [here we might skip very brief 'visits']
      my (@comments) = &SQLManySQL($myODBC,
       "SELECT cText FROM COMMENT WHERE Epoch = $v",
       "get GENERAL comments for this epoch");
```

```
#      print LOGFILE "\n Comments: <@comments>";

      $made =~ /(.+):\d\d\./; # trim seconds and fractional seconds
      $made = $1;
      my $c = '-';

      if ($#comments >= 0)
         { $ignore = 0;
           $c = pop(@comments);
           $c = &FixLatex($c);
         };
      $duratn /= 60000; # convert to minutes
      if ($duratn > 1)  # if over 1 min
         { $ignore = 0; # what happens if say "$ ignore = 0;" ??
         };

      if (! $ignore)
         { # round up duration to nearest 5 min:
#           print LOGFILE "\n Rounding duration up: $duratn";
           $isvisit = 1; # success
           $duratn += 4.99;
           $duratn /= 5;
           $duratn = int($duratn);
           $duratn *= 5;

           # also add initials of person who documented this:
           my ($whoINI) = '';
           if ($whosaw > 50) # exclude entries for e.g. IDAS [2009-09-23]
              {             # [only if comment added without entering menu]
               # hack: as JvS is sysadmin, has no separate PERSDATA representation
               if ($whosaw == 121)
                  { $whosaw = 1; # [ugh]
                  };
                my ($whoFore, $whoSur) = &GetSQL ($myODBC,
                  "SELECT pdoForename, pdoSurname FROM PERSDATA WHERE
                  pdoPerson = $whosaw", 'get documenting person');
                $whoFore =~ /^(.?)/; # zero or one match
                $whoINI = $1;
                $whoSur =~ /^(.?)/;
                $whoINI .= $1;
                while ($whoSur =~ /\w+\s+(.)(\w+)/ ) # if word+space+word..
                   { $whoINI .= $1;
                     $whoSur = $2;
                   };
              };

           my $line = "";
           $ONEVISITLINE = "$made & $duratn & $c & $whoINI \\\\ ";
```

```
          while ($#comments >= 0)
            { $c = pop(@comments);
              $c = &FixLatex($c); # 2007-12-02
              $ONEVISITLINE = "$ONEVISITLINE & & $c & \\\\ ";
            };
          $PAINVISITTABLE .= "$ONEVISITLINE \\hline \n";
        };  # immense pain from omitted . in ".="
    };

  if ($isvisit)
     { $TEMPLATEVISIT = &SlurpFile('template_visit.tex');
       $TEMPLATEVISIT =~ s/\$\[PAINVISITTABLE\]/$PAINVISITTABLE/mg;
     };
```

Regarding operations, at present we retrieve the lot. Later we might consider only listing recent ones!

In the following SELECT statement we use the ugly fact that 'retired' (duplicate) processes have a timestamp before 1910:[104]

```
  my $TEMPLATEOP= '';

  my (@ops) = &SQLManySQL($myODBC,
    "SELECT process FROM PROCESS WHERE Person = $patient \
      AND process > -1 \
      AND Proctype = 500 AND cast(rEnd AS varchar(19)) > '2007-01-01 00:00:00' \
     ORDER BY process DESC",
    "get all surgery, most recent first");

  my $op;
  my $OPERATIONTABLE='';

  if ($#ops < 0)
     {
       $TEMPLATEOP = 'No operation is recorded in the Pain Database.';
     } else
     {
       $TEMPLATEOP = &SlurpFile('template_op.tex');
       foreach $op (@ops)
         { my ($opmade) = &GetSQL($myODBC,
           "SELECT rStart FROM PROCESS WHERE process = $op",
           "get op date"); # [fix up importing...hmm]
           $opmade =~ /^(.+) /;
           $opmade = $1;  # trim off time.

           # at present we only associate one type of surgery, but
           # later we might enhance this. So allow for more:
```

[104] This ugly hack will probably cause us a lot of grief in the end!

```
        my (@types) = &SQLManySQL($myODBC,
          "SELECT SURGTYPE.ctText FROM SURGTYPE, SURGTYPEOB,
            EPOCH WHERE \
            SURGTYPEOB.Surgtype = SURGTYPE.surgtype AND \
            SURGTYPEOB.Epoch = EPOCH.epoch AND \
            EPOCH.Process = $op", "get type(s) of surgery");

        my (@scomments) = &SQLManySQL($myODBC,
          "SELECT cText FROM COMMENT, EPOCH WHERE \
            COMMENT.Epoch = Epoch.epoch AND \
            EPOCH.Process = $op", "get surgical comments");

        my $st = '-';
        if ($#types >= 0)
           {$st = pop(@types);
           };

        my $sc = '-';
        if ($#scomments >= 0)
           { $sc = pop(@scomments);
             $sc = &FixLatex($sc);
           };

        my $sline = "$opmade & $st & $sc";
        $OPERATIONTABLE = "$OPERATIONTABLE $sline \\\\ ";

        while (  ($#scomments >= 0)
               ||($#types >= 0)
              )
              { $st = '';
                if ($#types >= 0)
                   {$st = pop(@types);
                   };
                $sc = '';
                if ($#scomments >= 0)
                   { $sc = pop(@scomments);
                   };
                $sline = " & $st & $sc";
                $OPERATIONTABLE = "$OPERATIONTABLE $sline \\\\ ";
              };
        $OPERATIONTABLE = "$OPERATIONTABLE \\hline \n";
      };
    $TEMPLATEOP =~ s/\$\[OPERATIONTABLE\]/$OPERATIONTABLE/mg;
  };
```

The ‘Alerts’ section contains pain-related information. This includes the flags on the final page of the PDA program, signalling nausea, sedation and hypotension, and whether a ‘problem’ was identified. Rest and movement pain scores (VRS) are also reported. We use visits (from above) which contains the epochs relevant to the pain data,

```
my $TEMPLATEALERT ='';
my $isalert = 0;

my $PAINALERTTABLE='';
my $pnline;
my $useme;

foreach $v (@visits)
  { $useme = 0;
    my ($pdate, $prest, $pmov, $pcough) = &GetSQL($myODBC,
     "SELECT oMade, psoRest, psoMovement, psoCough FROM \
      PAINSCORE, EPOCH WHERE PAINSCORE.Epoch = EPOCH.epoch \
       AND EPOCH.epoch = $v", 'get pain scores etc');
      # slicker would be to maintain ordered epoch timestamps in
      # a separate array.
      # [PROBLEM!? ORDERING OF EPOCHS ... ORDER BY]
      # [THE ABOVE PROBLEM APPLIES THROUGHOUT!]
    $pdate =~ /(.+):\d\d\./;
    $pdate = $1; # trim seconds
    if ($pcough == 1)
       { $pcough = '{\\footnotesize Y}';
         $useme = 1;
       }
    elsif (length $pcough > 0)
       { $pcough = '{\\footnotesize N}';
         $useme = 1;
       } else
       { $pcough = ''; # in case?
       };

    my ($isprob) = &GetSQL($myODBC,
      "SELECT prIsOrNot FROM ISPROBLEM \
       WHERE Epoch = $v", 'fetch problem status');
    if ($isprob == 1)
       { $isprob = '+';
         $useme = 1;
       }
    elsif (length $isprob > 0)
       { $isprob = '-';
         $useme = 1;
       } else
       { $isprob = '';
```

```
        };

     if (  (length $prest > 0)
         ||(length $pmov > 0)
        )
        { $useme = 1;
        };

     if ($useme)
        { $pnline = "$pdate & $isprob & $prest & $pmov & $pcough \\\\";
          $PAINALERTTABLE = "$PAINALERTTABLE $pnline \\hline \n";
          $isalert = 1;
        };
   };
```

In our database we record hypotension, sedation, nausea and bowel opening as separate processes (with process type codes 1120, 1110, 1130 and 1140 respectively). In the following table we will only record one 'Yes' for each day for each process, to simplify things.

```
my $PAINPROBLEMTBL='';

# first, hypotension:
my %bpassoc = &GetAssocs ($myODBC, $patient, $STARTDATE, $now, 1120);
my %sedassoc = &GetAssocs ($myODBC, $patient, $STARTDATE, $now, 1110);
my %nassoc = &GetAssocs ($myODBC, $patient, $STARTDATE, $now, 1130);
my %pooassoc = &GetAssocs ($myODBC, $patient, $STARTDATE, $now, 1140);

# we must now ORDER the dates, eliminate duplicates, and then work through
# the dates in order, writing out table columns!
my %bighash;

foreach (keys %bpassoc)
    { $bighash{$_} = $bpassoc{$_} . '!A';
    };
foreach (keys %sedassoc)
    { $bighash{$_} =  $bighash{$_} . $sedassoc{$_} . '!B';
    };
foreach (keys %nassoc)
    { $bighash{$_} =  $bighash{$_} . $nassoc{$_} . '!C';
    };
foreach (keys %pooassoc)
    { $bighash{$_} =  $bighash{$_} . $pooassoc{$_} . '!D';
    };

my ($f, $k);

foreach $k (sort keys %bighash) # also sort!
```

```
    { $f = $bighash{$k};
      if ($f !~ /!A/ )
         { $f = " !A$f";
         };
      if ($f !~ /!D/)
         { $f = "$f !D";
         };
      if ($f !~ /!B/)
         { $f =~ /^(.*!A)(.*)$/;
           $f = "$1 !B$2";
         };
      if ($f !~ /!C/)
         { $f =~ /^(.*!B)(.*)$/;
           $f = "$1 !C$2";
         };
      $f =~ s/!D//; # get rid of terminal !3
      $f =~ s/!./ & /g; # put in table delimiters
      $f =~ s/0/-/g;    # replace 0 with -
      $f =~ s/1/\{\\footnotesize Y\}/g;    # replace 1 with Y
      $PAINPROBLEMTBL = "$PAINPROBLEMTBL $k & $f \\\\ \\hline \n";
    };

    if ($isalert)
       { $TEMPLATEALERT = &SlurpFile('template_alert.tex');
         $TEMPLATEALERT =~ s/\$\[PAINALERTTABLE\]/$PAINALERTTABLE/mg;
         $TEMPLATEALERT =~ s/\$\[PAINPROBLEMTBL\]/$PAINPROBLEMTBL/mg;
       } else
       { $TEMPLATEALERT = 'No pain alerts or problems were recorded in the pain \
       database for this visit.';
       }
```

We determine whether IV PCA was used. If it was, we determine all IV PCA processes, and order these. We use an auxiliary PCA template. The process code for IV PCA is 390.

```
my $TEMPLATEIVPCA='';

my @pcaprocs = &SQLManySQL($myODBC,
 "SELECT process FROM process WHERE \
   Person = $patient AND \
      process > -1 AND \
     ProcType = 390 AND \
     cast(rEnd as varchar(19)) > '$STARTDATE' \
   ORDER BY process",
   'get all PCA processes'); # ??: check me!
   # cast is for Ocelot, don't use TIMESTAMP
```

```
if ($#pcaprocs < 0)
   { $TEMPLATEIVPCA='';
   } else
   { my ($ivpca_template, $pcaid);
     my $STARTIVPCA='';
     my $STOPIVPCA='';
     my $WHYSTOPIVPCA='';

     foreach $pcaid (@pcaprocs)
      {
        $ivpca_template = &SlurpFile('template_ivpca.tex');
        if (length $ivpca_template < 1)
           { &Alert($MAINW, "Missing template: <template_ivpca.tex>");
             return 0; #fail.
           };
        #get start & stop dates:
        ($STARTIVPCA, $STOPIVPCA) = &GetSQL($myODBC,
          "SELECT rStart, rEnd FROM PROCESS WHERE \
           process = $pcaid", 'get pca start,end');
        if ($STARTIVPCA =~ /(.+) /)
           {$STARTIVPCA = $1; # discard time
           };
        if ($STOPIVPCA =~ /(.+) /)
           {$STOPIVPCA = $1; # discard time
           };

        # check whether a reason exists:
        ($WHYSTOPIVPCA) = &GetSQL($myODBC,
          "SELECT wText FROM STOPPROC, WHYSTOP WHERE \
            STOPPROC.Whystop = WHYSTOP.whystop AND \
            Process = $pcaid", 'why stopped?');
        if (length $WHYSTOPIVPCA < 1)
           { $WHYSTOPIVPCA = '-';
           };

        # next, get PCA details to IVPCARXTABLE:
        # We want date/time, mix, doses(good), tries, total, bolus & lock-out!
        # bolus and lockout are in the PCASETTINGS table
        # RXOBS contains the total ('doses' here is unused for now).
        # the PCA table contains tries and number good.
        # the date/time comes from the epoch, and
        # the nature of the infusion we get from the RX table:

        my ($IVPCAMIX) = &GetSQL($myODBC,
         "SELECT DRUG.dTrade FROM RX,DRUG \
          WHERE RX.Drug = DRUG.drug AND \
          RX.Process = $pcaid", 'get mix');
```

```
my(@pcaepochs) = &SQLManySQL($myODBC,
 "SELECT epoch FROM EPOCH WHERE \
  Process = $pcaid ORDER BY epoch", 'get pca epochs');

my($e, $edate, $bolus, $lockout, $ptotal, $pgood, $ptries);
my $IVPCARXTABLE='';
my $isIPRX = 0;

foreach $e (@pcaepochs)
  { ($edate) = &GetSQL($myODBC,
     "SELECT oMade FROM EPOCH WHERE epoch = $e",
     'get pca epoch timestamp');
    if($edate =~ /(.+):\d\d\./ )
      { $edate = $1; # discard seconds
      };
    ($bolus, $lockout) = &GetSQL($myODBC,
     "SELECT pseDose, pseLockout FROM PCASETTINGS \
      WHERE Epoch = $e", 'get bolus, lockout');
    if (length $bolus > 0)
       { $bolus /= 1000; # convert to mg.
       };
    if (length $lockout > 0)
       { $lockout /= 60; # convert to min.
       };

    ($ptotal) = &GetSQL($myODBC,
     "SELECT rxoTotal FROM RXOBS WHERE \
       Epoch = $e", 'get total pca');
     if (length $ptotal > 0)
        { $ptotal /= 1000; # to mg
        };

     ($ptries, $pgood) = &GetSQL($myODBC,
      "SELECT pcoTries, pcoGood FROM PCA \
       WHERE Epoch = $e", 'get tries/good');

    # We want date/time, doses(good), tries, total, bolus & lock-out!
    if (  (length $edate > 0)
        ||(length $pgood > 0)
        ||(length $ptries > 0)
        ||(length $ptotal > 0)
        ||(length $bolus > 0)
        ||(length $lockout > 0)
       )
       { $IVPCARXTABLE = $IVPCARXTABLE .
         "$edate & $pgood & $ptries & $ptotal & $bolus & $lockout \\\\ ";
         $isIPRX = 1;
```

```
            };
        };
      $IVPCARXTABLE = $IVPCARXTABLE . " \\hline \n";

      my ($TEMPLATEIVPCARX) = '';
      if ($isIPRX)
         { $TEMPLATEIVPCARX = &SlurpFile('template_ivpca_rx.tex');
           $TEMPLATEIVPCARX =~ s/\$\[IVPCARXTABLE\]/$IVPCARXTABLE/mg;
         };


      # must still sort out BASALINFUDATA
      my $BASALINFUDATA='';

      $ivpca_template =~ s/\$\[STARTIVPCA\]/$STARTIVPCA/mg;
      $ivpca_template =~ s/\$\[IVPCAMIX\]/$IVPCAMIX/mg;
      $ivpca_template =~ s/\$\[STOPIVPCA\]/$STOPIVPCA/mg;
      $ivpca_template =~ s/\$\[WHYSTOPIVPCA\]/$WHYSTOPIVPCA/mg;
      $ivpca_template =~ s/\$\[TEMPLATEIVPCARX\]/$TEMPLATEIVPCARX/mg;
      $ivpca_template =~ s/\$\[BASALINFUDATA\]/$BASALINFUDATA/mg;
      $TEMPLATEIVPCA = "$TEMPLATEIVPCA $ivpca_template";
    };
  };
```

Similar to the IV PCA section above is the regional section which follows. A problem in this code is that at present we do not report on spinals.

We select out all regional catheters (process type code 100–159) but do NOT at this stage identify regional infusions (codes 200–259) or PCA with regional infusions (codes 300-399).

```
my $TEMPLATERGN='';

# in the following we CANNOT use rStart as catheters are usually inserted
# BEFORE they are referred to the pain service!

my (@rgnprocs) = &SQLManySQL($myODBC,
 "SELECT process FROM PROCESS WHERE \
   Person = $patient \
      AND process > -1 \
    AND ProcType BETWEEN 100 AND 159 \
    AND cast(rEnd as varchar(19)) > '$STARTDATE' \
   ORDER BY process",
   'get all RGN caths'); # ??: check me!

if ($#rgnprocs < 0)
   { $TEMPLATERGN='';
   } else
   { my ($rgnid);
```

```
    foreach $rgnid (@rgnprocs) # ???
      {
        $TEMPLATERGN .= &FillRegionalData ($myODBC, $rgnid, $patient);
      };
  };
```

We're now interested in processes with type codes between 1000 and 1099 (inclusive). These codes are used to represent major organ dysfunction, or other substantial concerns, meanings of which can be obtained from the PROCTYPE table. We list these as LaTeX items.

```
my $PROBLEMITEMLIST='';

  my (@probs) = SQLManySQL($myODBC,
  "SELECT process FROM PROCESS WHERE \
    PROCESS.Person = $patient \
      AND process > -1 \
      AND PROCESS.ProcType BETWEEN 1000 AND 1099 \
      AND cast(rEnd as varchar(19)) >= '$STARTDATE' \
    ORDER BY ProcType, process", 'get problems'); # ??: check me!
    # cast is for ocelot.

  if ($#probs < 0)
     { $PROBLEMITEMLIST = 'No systemic problems were flagged by the pain team duri
     } else
     { $PROBLEMITEMLIST = 'The following problems were identified:\begin{itemize}'
       my ($prb, $prstart, $prend, $prbtxt);
       my $o_prbtxt = '';
       my $o_prstart = '';
       my $o_prend = '';

       foreach $prb (@probs)
         { ($prbtxt, $prstart, $prend) = &GetSQL($myODBC,
           "SELECT rptNature, rStart, rEnd FROM PROCESS,PROCTYPE
             WHERE PROCESS.ProcType = PROCTYPE.proctype AND
             process = $prb", 'get problem type, times');
           if ($prstart =~ /(.+) /)
              { $prstart = $1; # drop time
              };
           if ($prend =~ /(.+) /)
              { $prend = $1; # drop time
              };
           if ($prbtxt ne $o_prbtxt) # if no flow into next problem..
              { if (length $o_prbtxt > 0)
                   {
                     $PROBLEMITEMLIST .= "\\item $o_prbtxt \n";
                     ###################################
                     # dates removed: 2008-09-23:
```

```
                    ###################################
                    # if ( ($prbtxt =~ /chronic/i )
                    #   ||($prbtxt =~ /allerg/i ) # don't specify time range!
                    #   )
                    #   {$PROBLEMITEMLIST .= "\\item $o_prbtxt \n";
                    #   } else
                    #   {$PROBLEMITEMLIST .= "\\item $o_prbtxt: ($o_prstart -- $o
                    #   };
                    ###################################

                  }; # NOT on first time around!
                $o_prbtxt = $prbtxt;
                $o_prstart = $prstart;
              };
            $o_prend = $prend; #
          };

        $PROBLEMITEMLIST .= "\\item $o_prbtxt \n";
        ###################################
        # dates removed: 2008-09-23:
        ###################################
        # $PROBLEMITEMLIST .= "\\item $o_prbtxt: ($o_prstart -- $o_prend) \n";

        $PROBLEMITEMLIST .= '\end{itemize}';
      };
```

In obtaining a list of oral therapy, we look for enteral processes (type 50 processes). At present we simply pull out the name of the drug, and also record when it was first and last documented as given. Later we might consider stating the maximum dose, and possibly even detailing opiate doses.

```
my $ORALRXLIST='';

my (@oprocs) = &SQLManySQL($myODBC,
 "SELECT DISTINCT dTrade, \
   cast (rStart as varchar(11)) as mystart,
   cast (rEnd as varchar(11)) \
   FROM PROCESS, RX, DRUG WHERE \
     DRUG.drug = RX.Drug \
     AND RX.Process = PROCESS.process \
       AND Person=$patient \
       AND cast(rEnd as varchar(19)) >= '$STARTDATE' \
       AND ProcType = 50 \
       AND PROCESS.process > -1 \
     order by dTrade, mystart", 'get enteral rx');
  # the DISTINCT will get rid of identical (duplicate) processes [hmm]
  # the cast removes time after space (leave the blank at the end!)
```

```
  if ($#oprocs < 0)
     { $ORALRXLIST='';
     } else
     { $ORALRXLIST='\subsection*{Oral therapy} \
     The following summary list of oral analgesia-related drugs should be read \
  together with the patient\'s treatment chart. \begin{itemize} ';

       my ($otrade, $ostrt, $oend);
       while ($#oprocs >= 0)
         { $otrade = shift(@oprocs);
           $ostrt = shift(@oprocs);
           $oend = shift(@oprocs); # ugly
           if ($ostrt =~ /^(.+) /)
              {$ostrt = $1;
              };
           if ($oend =~ /^(.+) /)
              {$oend = $1;
              };
           if ($ostrt eq $oend) # altered 2007-12-02
              { $ORALRXLIST .= "\\item $otrade: ($ostrt) \n";
              } else
              { $ORALRXLIST .= "\\item $otrade: $ostrt -- $oend \n";
              }
         };
       $ORALRXLIST .= ' \end{itemize} ';
    };
```

Similar are other modalities, but they have process codes which (rather unfortunately) lie between 260 and 299 inclusive:

```
my $OTHERAGENTLIST='';

my (@xprocs) = &SQLManySQL($myODBC,
 "SELECT DISTINCT dTrade, rStart, rEnd \
   FROM PROCESS, RX, DRUG WHERE \
     DRUG.drug = RX.Drug AND \
     RX.Process = PROCESS.process AND \
     Person=$patient AND \
      PROCESS.process > -1 AND \
     ProcType BETWEEN 260 AND 299 \
     order by dTrade, rStart", 'get other rx');

  if ($#xprocs < 0)
     { $OTHERAGENTLIST='';
     } else
     { # &Alert($MAINW, "DEBUG: patient for OTHER RX is $patient");
       $OTHERAGENTLIST='\subsection*{Other agents} \
```

```
        Other therapy included: \begin{itemize} ';

       my ($xtrade, $xstrt, $xend);
       while ($#xprocs >= 0)
         { $xtrade = shift(@xprocs);
           $xstrt = shift(@xprocs);
           $xend = shift(@xprocs); # ugly
           if ($xstrt =~ /^(.+) /)
              {$xstrt = $1;
              };
           if ($xend =~ /^(.+) /)
              {$xend = $1;
              };
           $OTHERAGENTLIST .= "\\item $xtrade: $xstrt -- $xend \n";
         };
       $OTHERAGENTLIST .= ' \end{itemize} ';

    };

    if (  (length $OTHERAGENTLIST < 2)
           ||(length $TEMPLATEIVPCA < 2)
           ||(length $TEMPLATERGN < 2)
           ||(length $ORALRXLIST < 2)
       )
          { $OTHERAGENTLIST .= ' No other therapy is documented in the database. \
            Please consult the Patient record for further details.';
          };


 my $GENPAINNOTE='';
```

The final comment is not used (at present), however we do print the user's initial and surname.

```
my $FINALCOMMENT= '';
  my ($USERNAME) = &GetUserName($myODBC, $userid);
```

We also determine the most recent ward for this admission:

```
my $FINALCOMMENT= '';
  my ($LASTWARD) = &GetLastWardName($myODBC, $procid);
#  print LOGFILE "\n Debug last ward was <$LASTWARD>";
```

Finally, substitute in all values:

```
$template =~ s/\$\[LASTWARD\]/$LASTWARD/mg;
$template =~ s/\$\[INI\]/$INI/mg;
$template =~ s/\$\[SURNAME\]/$SURNAME/mg;
```

```
$template =~ s/\$\[NHI\]/$NHI/mg;
$template =~ s/\$\[FORENAME\]/$FORENAME/mg;
$template =~ s/\$\[DOB\]/$DOB/mg;
$template =~ s/\$\[WT\]/$WT/mg;
$template =~ s/\$\[GENDER\]/$GENDER/mg;
$template =~ s/\$\[AGE\]/$AGE/mg;
$template =~ s/\$\[ASA\]/$ASA/mg;
$template =~ s/\$\[ALIVE\]/$ALIVE/mg;
$template =~ s/\$\[TODAY\]/$TODAY/mg;
$template =~ s/\$\[PROBLEMITEMLIST\]/$PROBLEMITEMLIST/mg;
$template =~ s/\$\[ORALRXLIST\]/$ORALRXLIST/mg;
$template =~ s/\$\[OTHERAGENTLIST\]/$OTHERAGENTLIST/mg;
$template =~ s/\$\[GENPAINNOTE\]/$GENPAINNOTE/mg;
$template =~ s/\$\[ENDDATE\]/$ENDDATE/mg;

$template =~ s/\$\[TEMPLATEIVPCA\]/$TEMPLATEIVPCA/mg;
$template =~ s/\$\[TEMPLATERGN\]/$TEMPLATERGN/mg;
$template =~ s/\$\[TEMPLATEOP\]/$TEMPLATEOP/mg;
$template =~ s/\$\[TEMPLATEVISIT\]/$TEMPLATEVISIT/mg;
$template =~ s/\$\[TEMPLATEALERT\]/$TEMPLATEALERT/mg;

$template =~ s/\$\[FINALCOMMENT\]/$FINALCOMMENT/mg;
$template =~ s/\$\[USERNAME\]/$USERNAME/mg;
```

We must also check for any ‘orphan’ variables, which haven’t been filled in, and warn the user of the presence of such problem variables. Rather than flunking out horribly, we replace all such variables with three hard spaces, which in LaTeXare represented by tildes. The alert should be sufficiently irritating to ensure that the user gets me to fix the problem!

```
while ($template =~ /\$\[(.+)\]/)
      { my $dud = $1;
        &Alert($MAINW,
  "Irritating error! Unrecognised variable <$dud> in template");
        $template =~ s/\$\[$dud\]/~~~/;
      };
```

Next, we accommodate local date variations. At present, this consists of altering a date in the format YYYY-MM-DD to DD-MM-YYYY, as per New Zealand format:

```
 $template =~ s/(\d{4})-(\d{2})-(\d{2})/\3\/\2\/\1/g;
```

Finally we write the file to the *latex* subdirectory and return its file number (name). To do so we must create an entry in the PRINTED table.

```
  my $printid = 1;
  if ($MANYFILES)
     { $printid = &AutoKey($myODBC, 'printed'); # ??? [fix up, commit]
       &DoSQL ($myODBC, "INSERT INTO PRINTED (printed, Process, prTime, Person) \
       VALUES ($printid, $procid, TIMESTAMP '$now', $userid)", "record printing");
       &Commit($myODBC); # ensure record is kept in PRINTED table.
     };

  # next, create the file:  ## [???]
  my $prok = 1;
  my $pfile="latex/$printid.tex";
  open PFILE, ">$pfile" or $prok = 0;
  if (! $prok)
     { &Alert($MAINW, "Failed to create print file <$pfile>. Aborted!");
       &DoSQL ($myODBC,
 "UPDATE PRINTED SET prTime = NULL WHERE printed = $printid",
 'signal print failure');   # this clumsily records failure!
       return 0;
     };

  binmode(PFILE); # otherwise WinDOS messes up CR/LF
  print PFILE $template;
  close PFILE;
  return($printid);
}
```

On failure, a value of zero is returned, as a print process cannot have this value.

11.1.4 Inserting regional data

Here we fill in regional data:

```
sub FillRegionalData
{ my ($myODBC, $rgnid, $patient);
  my ($myODBC, $rgnid, $patient)=@_;

  my($TEMPLATERGN) = '';
  my ($rgn_template);

  my $STARTRGN='';
  my $STOPRGN='';
  my $WHYSTOPRGN='';
  my $isTRO = 0;

  $rgn_template = &SlurpFile('template_rgn.tex');
  if (length $rgn_template < 1)
     { &Alert($MAINW, "Missing template: <template_rgn.tex>");
       return ''; #fail.
```

```
    };
 # get type of regional:
 my ($REGIONALTYPE) = &GetSQL($myODBC,
  "SELECT rptNature FROM PROCTYPE, PROCESS \
  WHERE PROCESS.ProcType = PROCTYPE.proctype \
  AND PROCESS.process = $rgnid", 'get rgn type');

 #get start & stop dates:
 ($STARTRGN, $STOPRGN) = &GetSQL($myODBC,
   "SELECT rStart, rEnd FROM PROCESS WHERE \
    process = $rgnid", 'get rgn start,end');

 # check whether a reason exists:
 ($WHYSTOPRGN) = &GetSQL($myODBC,
   "SELECT wText FROM STOPPROC, WHYSTOP WHERE \
     STOPPROC.Whystop = WHYSTOP.whystop AND \
     Process = $rgnid", 'why stopped?');
 if (length $WHYSTOPRGN < 1)
    { $WHYSTOPRGN = '-';
    };

# next, get observations on the regional to \$REGIONOBSTABLE.
# We want: pressure areas, motor fx, block level, site and mobility flags ---
# these are all recorded in association with the catheter, and NOT the infusion.
# The table is RGNOBS.

 my $REGIONOBSTABLE='';
 my (@rgne) = &SQLManySQL ($myODBC,
  "SELECT epoch FROM EPOCH WHERE \
    Process = $rgnid ORDER BY epoch",
  'get regional epoch list');

 my $re;
 my ($dt, $motor, $press, $level, $site, $mob);
 if ($#rgne < 0)
    { $REGIONOBSTABLE .= " &  & & & & \\\\ \n";
    };
 foreach $re (@rgne) # fix swop of site/level in following 2008-09-23:
   { ($dt, $motor, $press, $site, $level, $mob)=&GetSQL($myODBC,
    "SELECT oMade, rgoMotor,rgoPressure,rgoSite,rgoLevel,rgoMobile \
     FROM RGNOBS, EPOCH WHERE RGNOBS.Epoch = EPOCH.epoch \
     AND EPOCH.epoch = $re", 'get rgn obs');
     if ($dt =~ /^(.+):\d\d\./)
        { $dt = $1; # trim seconds
        };
     $motor = &Abbreviate($motor);
     $press = &Abbreviate($press);
     $level = &Abbreviate($level);
```

```
      $site = &Abbreviate($site);
      $mob = &Abbreviate($mob);
      $REGIONOBSTABLE .= "$dt & $ press & $motor & $level & $site & $mob \\\\ \n";
      if (  (length $motor > 0)
          ||(length $press > 0)
          ||(length $level > 0)
          ||(length $site > 0)
          ||(length $mob > 0)
         )
         { $isTRO = 1;
         };
    };

   my $TEMPLATERGNOBS = '';
   if ($isTRO)
      { $TEMPLATERGNOBS = &SlurpFile('template_rgn_obs.tex');
        $TEMPLATERGNOBS =~ s/\$\[REGIONOBSTABLE\]/$REGIONOBSTABLE/mg;
      };

 # finally get details of infusions and usage. This is more tricky
 # as we must find associated regional infusions within the time of
 # the regional. We ASSUME that only one regional is running at one time
 # (if this convention were changed, then the following code would have to
 # be altered; the database itself will tolerate the change).

 # We first get those processes: ???

 my (@COMMAS) = &SQLManySQL ($myODBC,
  "SELECT process FROM PROCESS WHERE \
    Person = $patient AND \
    process > -1 AND \
    ((ProcType BETWEEN 200 AND 259) \
       OR(ProcType BETWEEN 300 AND 359)) \
      AND cast(PROCESS.rEnd as varchar(19)) >= '$STARTRGN' \
      AND cast(PROCESS.rEnd as varchar(19)) <= '$STOPRGN'",
  'get regional processes pcra/not');
  my ($COMPROCS) = &CommaList (@COMMAS);
# another Ocelot failing: BETWEEN fails in the above!

# print LOGFILE "\n\n SPECIAL DEBUG($rgnid) for $STARTRGN--$STOPRGN: <$COMPROCS>";

  my (@ergninfus) = &SQLManySQL ($myODBC,
   "SELECT epoch FROM EPOCH where \
     Process IN ($COMPROCS) AND \
     cast (oMade AS varchar(19)) >= '$STARTRGN' AND \
     cast (oMade AS varchar(19)) <= '$STOPRGN' \
   ORDER BY epoch",
   'get reg infus epochs');
```

```
# print LOGFILE "\n DEBUG epochs: <@ergninfus>";
```

The above assumes that the rEnd of the processes is never after the removal of the regional catheter, a reasonable assumption provided our coding is correct!

```
 # then for each epoch (in order) we obtain:
 #  date, mix, rate, top-ups, pca doses and attempts (if relevant)
 # to get the mix we look at the RX attached to the process;
 # to get top-ups we look for an RXOBS
 # to get rate, we look at INFUSIONOBS
 # PCA values are as for the IV PCA (same table)

  my($RGNRXTABLE) = '';
  my($isTRX) = 0;

  if ($#ergninfus < 0)
    { $RGNRXTABLE .= " & & & & \\\\ \n";
    };
  my($er, $rdat, $rmix, $rrat, $rtop, $rhit, $rtry);
  foreach $er (@ergninfus)
    {
      ($rdat) = &GetSQL($myODBC,
       "SELECT oMade FROM EPOCH WHERE epoch = $er",
       'get epoch stamp');
       if ($rdat =~ /^(.+):\d\d\./)
          {$rdat = $1; # trim seconds
          };

      ($rmix) = &GetSQL($myODBC,
        "SELECT DRUG.dTrade FROM RX,DRUG,EPOCH \
        WHERE RX.Drug = DRUG.drug AND \
        EPOCH.Process = RX.Process AND \
        EPOCH.epoch = $er", 'get rgn mix');
        $rmix = &FixLatex($rmix); # in case of % !

      ($rrat) = &GetSQL($myODBC,
        "SELECT inoRate FROM INFUSIONOBS WHERE \
         Epoch = $er", 'get infus rate');
         if (length $rrat > 0)
            { $rrat /= 1000; # convert micro to ml.
              # NOTE: check the units (per hour?)
            };

      ($rtop) = &GetSQL($myODBC,
        "SELECT rxoDoses FROM RXOBS WHERE \
         Epoch = $er", 'get rgn topups');
       ($rtry, $rhit) = &GetSQL($myODBC,
```

```
        "SELECT pcoTries, pcoGood FROM PCA \
        WHERE Epoch = $er", 'get rgn tries/good');

      if (  (length $rmix > 0)
          ||(length $rrat > 0)
          ||(length $rtop > 0)
          ||(length $rhit > 0)
          ||(length $rtry > 0)
         )
         { $RGNRXTABLE .= "$rdat & $rmix & $rrat & $rtop & $rhit & $rtry \\\\";
           $isTRX = 1;
         };
    };

  $RGNRXTABLE .= " \\hline \n";

  my $TEMPLATERGNRX = '';
  if ($isTRX)
     { $TEMPLATERGNRX = &SlurpFile('template_rgn_rx.tex');
       $TEMPLATERGNRX =~ s/\$\[RGNRXTABLE\]/$RGNRXTABLE/mg;
     };

  if ( (! $isTRX) &&
       (! $isTRO)
      )
      { $TEMPLATERGNRX = "\\hline \\multicolumn{6}{|l|}\
        {No abnormalities or observations documented.} \\\\ \\hline";
      };
  # if neither, then simple 'negative' message.


  if ($STARTRGN =~ /(.+) /)
     {$STARTRGN = $1; # discard time
     };
  if ($STOPRGN =~ /(.+) /)
     {$STOPRGN = $1; # discard time
     };
  $rgn_template =~ s/\$\[REGIONALTYPE\]/$REGIONALTYPE/mg;
  $rgn_template =~ s/\$\[STARTRGN\]/$STARTRGN/mg;
  $rgn_template =~ s/\$\[STOPRGN\]/$STOPRGN/mg;
  $rgn_template =~ s/\$\[WHYSTOPRGN\]/$WHYSTOPRGN/mg;
  $rgn_template =~ s/\$\[TEMPLATERGNOBS\]/$TEMPLATERGNOBS/mg;
  $rgn_template =~ s/\$\[TEMPLATERGNRX\]/$TEMPLATERGNRX/mg;

  $TEMPLATERGN .= $rgn_template;

  return ($TEMPLATERGN);
}
```

11.1.5 Get last ward

Obtain the most recent ward for the relevant admission. We look at the most recent value in the BADOBS table. We submit an ODBC handle and the ID of the type 3 process.

```
sub GetLastWardName
{ my ($myODBC, $procid);
     ($myODBC, $procid) = @_;

  my ($q) = "SELECT Bed FROM BADOBS WHERE badobs =
  (SELECT MAX(badobs) FROM BADOBS, EPOCH WHERE
    BADOBS.Epoch = EPOCH.epoch AND EPOCH.Process = $procid)";
  my ($wd) = &GetSQL($myODBC, $q, 'get last bed');
  if (  (length $wd < 1)
      ||($wd < 1_00_00)
     )
     { return ('?');
     };
  $wd = int($wd/100_00); # bed code to ward code
  $q = "SELECT swrdText FROM WARD WHERE ward = $wd";
  ($wd) = &GetSQL($myODBC, $q, 'get last ward');
  return ($wd);
}
```

11.1.6 Get most recent user

Given the ODBC handle and the ID of the type 3 process, find the last person who saw the patient prior to discharge, that is the person attached to the most recent epoch for the interval under consideration (the time between the start and end of the type 3 process). We will print a discharge summary for them.

```
sub GetLastUser($myODBC, $procid)
{
   my ($myODBC, $procid);
      ($myODBC, $procid) = @_;

my($start, $end, $patient) = &GetSQL($myODBC,
  "SELECT rStart, rEnd, Person FROM PROCESS WHERE process = $procid",
  'get start/end');

my($q) = "SELECT Person FROM EPOCH WHERE epoch =
(SELECT MAX(epoch) FROM EPOCH, PROCESS WHERE EPOCH.Process = PROCESS.process
  AND PROCESS.person = $patient
  AND cast(EPOCH.oMade as varchar(19)) >= '$start'
  AND cast(EPOCH.oMade as varchar(19)) <= '$end'
  AND EPOCH.person > 50)";  # cast fixes Ocelot's little problem
```

```
my($user) = &GetSQL($myODBC, $q, 'get most recent user');

 if ((length $user < 1) || ($user == 121)) # 121 is hack for sysop
    { $user = 1; # nasty default to sysop (me)
    };
   return($user);
}
```

11.1.7 Get user details

```
sub GetUserName
{ my ($myODBC, $userid);
     ($myODBC, $userid) = @_;

  my ($uSURNAME) = &GetSQL($myODBC,
    "SELECT pdoSurname FROM PERSDATA WHERE \
      persdata = (SELECT MAX(persdata) FROM PERSDATA \
        WHERE pdoPerson = $userid AND pdoSurname IS NOT NULL)",
      "get user surname");

  my ($uFORENAME) = &GetSQL($myODBC,
    "SELECT pdoForename FROM PERSDATA WHERE \
      persdata = (SELECT MAX(persdata) FROM PERSDATA \
        WHERE pdoPerson = $userid AND pdoForename IS NOT NULL)",
      "get user forename");
    if ($uFORENAME =~ /^(.)/)
       {$uFORENAME = $1; # initial
       };

 my $USERNAME = "$uFORENAME $uSURNAME";
 return ($USERNAME);

}
```

11.1.8 FixLatex

If we import a comment from the database and then write it as LaTeX, we must fix all reserved characters or we'll encounter pain. FixLatex does just this:

```
sub FixLatex
{ ($_) = @_;

  s/\\/ /g; # SIMPLY ALLOW NO BACKSLASHES, FIRST UP!
  s/\$/\\\$/g; # else, too messy!
  s/#/\\#/g;
  s/%/\\%/g;
```

```
  s/&/\\&/g;
  s/_/\\_/g;
  s/\{/\\\{/g;
  s/\}/\\\}/g;
  s/</\$<\$/g;
  s/>/\$>\$/g;
  s/\^/\\\^\{\}/g; # these last
  s/\~/\\\~\{\}/g;

  return ($_);
}
```

11.1.9 FixName

It's so easy when entering a name to forget to capitalise the first letter, especially on a PDA. This makes for ugly unprofessional printing of names. Let's fix it (clumsily):

```
sub FixName
{ my($n, $i);
    ($n) = @_;

  if ($n =~ /(.)(.*)/)
     { $i = $1;
       $n = $2;
       $i =~ tr/a-z/A-Z/;
       $n = "$i$n";
     };
  return ($n);
}
```

11.1.10 Abbreviate

With regional observations (only these, at present) we translate a 0 into an 'x' signalling a problem or concern, and a 1 into '-' (for no concerns). Any other value produces null, at least for now.

```
sub Abbreviate
{ ($_) = @_;
  if (/^1$/)
     { return '-';
     };
  if (/^0$/)
     { return 'x';
     };
  return '';
}
```

11.1.11 GetAssocs

Given the database and the process code for a particular class of process, as well as the correct PERSON, find all associated epochs between two dates, and then create an associative array of dates (only, not times) with a value of 0,1 or null attached to each date, depending on the value of the particular ISPROBLEM entry. If *any* ISPROBLEM (prIsOrNot) is set for a particular day, then we return 1 as the association; otherwise, if any value is zero, we return zero; otherwise we return null for that association!

```
sub GetAssocs
{
  my ($myODBC, $person, $STARTDATE, $now, $proctype);
     ($myODBC, $person, $STARTDATE, $now, $proctype)=@_;

# even the following fails in Ocelot:
#  my (@gepochs) = &SQLManySQL($myODBC,
#    "SELECT epoch FROM EPOCH, PROCESS WHERE \
#      EPOCH.Process = PROCESS.process AND \
#      PROCESS.Person = $person AND \
#      PROCESS.ProcType = $proctype AND \
#      cast(rStart as varchar(19)) BETWEEN '$STARTDATE' AND \
#      '$now' ORDER BY epoch",
#      "get relevant epochs"); # ??: check me!
# so we use ...

  my (@gepochs) = &SQLManySQL($myODBC,
    "SELECT epoch FROM EPOCH WHERE \
      Process IN (SELECT process FROM PROCESS WHERE \
              PROCESS.Person = $person AND \
              process > -1 AND \
              PROCESS.ProcType = $proctype AND \
              cast(rStart as varchar(19)) BETWEEN \
              '$STARTDATE' AND '$now') \
      ORDER BY epoch",
      "get relevant epochs");

  my ($e, $g, $bdate);
  my %gassoc = (); # associative array
  foreach $e (@gepochs)
    {
      ($bdate, $g) = &GetSQL($myODBC,
        "SELECT oMade, prIsOrNot FROM ISPROBLEM, EPOCH \
         WHERE ISPROBLEM.Epoch = EPOCH.epoch AND
         EPOCH.epoch = $e", 'problem?');
      if ($bdate =~ /^(.+) /) # discard time
         { $bdate = $1;
```

```
         };
      if (length $bdate > 0)
         { if (exists $gassoc{$bdate})
              { if ($gassoc{$bdate} != 1)
                   { if ($g == 1)
                        { $gassoc{$bdate} = 1;
                        }
                     elsif ($gassoc{$bdate} != 0)
                        { $gassoc{$bdate} = $g;
                        };
                   };
              } else
              { $gassoc{$bdate} = $g;
              };
         };
    };
  return (%gassoc); # return associative array
}
```

11.1.12 SlurpFile

We simply slurp in a complete file and return it as a string. On failure we return a null string.

```
sub SlurpFile
{
  my ($filename);
     ($filename)=@_;
  my $okF = 1;

  my ($hF);
  open ($hF, $filename) or $okF = 0; #hF is handle
  if (! $okF)
     { &Alert ($MAINW, "File <$filename> not found. Fatal!");
       return "";
     };

  binmode ($hF);
       my ($keepterm) = $/; # retain
       undef $/;            # undefine
       my ($wholefile) = <$hF>; # slurp
       $/ = $keepterm;      # restore
       close ($hF);

  return ($wholefile);
}
```

11.1.13 Create PDF files

This section is simple. We merely invoke the external **pdflatex** program on each of the files in turn, moving the resulting PDF files to the *pdf* subdirectory of the *PainForm* directory. PdfLatex seems to make the PDF file locally, whatever, so we execute a little batch file in DOS which does the necessary moving around of files from the PainForm directory. This batch file (*pdfify.bat*) is described later in this document. (Section 18.2).

```
sub LatexToPdf
{
  my($procid);
    ($procid) = @_;

  #  &Alert($MAINW, "Printing Latex, id: $procid");
  # external invocation:
  print '>';

#  print LOGFILE "\n Creating PDF ($procid) ";

  my $pbat = "pdfify.bat $procid";
  my $r;
  $r = `$pbat`;
  # here might check outcome?
  # &Alert($MAINW, "Outcome: <$r>");

  return(1);
}
```

11.1.14 Batch print and archive

Given the name of a PDF file (with the .PDF suffix), we invoke an external printing utility to print it. We should set things up so that the default printer is used without bothering the user. Our external file is here a DOS batch file, which in turn invokes e.g. Acrobat Reader using unsupported command line instructions. We do *not* include the path of the PDF file as it's assumed by the batch file to be in the *pdf* directory!

```
sub PrintPdf
{
  my($id);
    ($id) = @_;

  my $pbat = "batprint.bat $id.PDF";
  my $r;
  $r = `$pbat`;
```

```
  # here might check outcome?

  return(1);
}
```

12 Data extraction

12.1 Basic data: Daily report

Here we pull out very simple daily statistics. These include:

1. Number of active cases in the database for the stated day;
2. Number of these cases seen;
3. Number not seen;
4. Epidural information: number with epidurals on that day (at any time), and number of epidurals inserted and removed;
5. Similar data for PCA.
6. Discharges and new admissions for this day;
7. Number of cases with both epidural and PCA, and number with neither.

We will print these data to a single page using the same LaTeX tricks we used above.

The following represents a revised version of the basic data printing, which breaks down patient numbers by ward. Our rather clumsy strategy is to retrieve the list of patients, and then identify the current ward (on the date desired) for that patient. If the patient is listed as being in several wards on that date, the most recent ward is retrieved for that date. We then create an associative array between patient and ward, so on subsequent retrieval of the patient ID we can easily substitute the ward.

12.1.1 WhichWard

Given a patient ID and a date ($thisdate), we determine where the patient was on that day. The ward retrieved is a text string, not the corresponding integer.

Note that we wish to identify a BADOBS table entry which refers back to a type 3 PROCESS (via EPOCH table), where the rStart field of the type 3 process predates or equals $thisdate, and the rEnd field is at or after $thisdate, or is NULL.

The following code is cumbersome owing to limitations of the complexity of the SQL expressions in Ocelot.

```
sub WhichWard
{ my($myODBC, $ptID, $thisdate)=@_;

 # obtain the relevant type 3 process:
 my($q) = "SELECT MAX(process) FROM PROCESS WHERE Person = $ptID AND ProcType = 3
        cast(PROCESS.rStart as varchar(19)) <= '$thisdate 23:59:59' AND
        ( cast(PROCESS.rEnd as varchar(19)) >= '$thisdate 00:00:00'
          OR cast(PROCESS.rEnd as varchar(19)) IS NULL)";
   # (might check that there aren't several such procs??)
 my($ptproc) = &GetSQL($myODBC, $q, 'get pt type 3 proc');
 if (length $ptproc < 1)
    { return '?';
    };

# find most recent, relevant epoch:
$q = "SELECT MAX(epoch) FROM EPOCH WHERE Process = $ptproc AND
     cast(oMade as varchar(19)) <= '$thisdate 23:59:59'";
my($ptepo) = &GetSQL($myODBC, $q, 'get last epoch');
 if (length $ptepo < 1)
    { return '?';
    };

# finally get bed and thus, ward:
$q = "SELECT swrdText FROM WARD WHERE ward = (SELECT Bed/10000 FROM BADOBS WHERE E
my($wd) = &GetSQL($myODBC, $q, "get patient's ward");
 if (length $wd < 1)
    { return '?';
    };
return ($wd);
}
```

Technically we might also constrain rStart to be greater than rEnd.

12.1.2 Ward Names and Counts

We assume that we have an associative array of ward names called %WDLIST. Here's a routine to generate a list of the number of patients by ward, given the patient IDs; the associative array which provides the ward, given the patient; and %WDLIST. PTWARDS is an associative array linking each patient to their ward via the internal database ID of the patient.

```
sub FindWardCounts
{ my(@ptids)=@_;

  # must here CLEAR counts in %WDLIST to zero:
  my ($w);
```

```
  foreach $w (keys %WDLIST)
    {
      $WDLIST{$w} = 0;
    };

  my ($p);
  foreach $p (@ptids)
    {
      $w = $PTWARDS{$p}; # get ward
      $WDLIST{$w} += 1;
    };
}
```

Let's now create an array of these counts. We use trickery to order the returned data according to an alphabetical sort of the ward names:

```
sub ListWardCounts
{
  my ($c, $w);
  my (@srt) = ();

  foreach $w (keys %WDLIST)
    { $c = $WDLIST{$w}; # get corresponding value
      push (@srt, "$w|$c"); # ward name must not contain pipe |
    };
  @srt = sort { $a cmp $b } @srt;
  # now pull out the ordered values!
  my (@opt) = ();
  foreach (@srt)
    {
      /^(.+)\|(.+)$/;
      push(@opt, $2); # push numeric, discard ward
    };
  return (@opt);
}
```

Similarly, we might wish to get a list of wards from the associative array %WDLIST. We will return an array of ward names. We must make sure that the order in which we generate the items is the same for the preceding and following routines. We do this by sorting the ward names ASCII-betically, and performing a similar trick on the counts, as described above.

```
sub ListWardNames
{
  my ($w);
  my (@srt) = ();
```

```
  foreach $w (keys %WDLIST)
    { push (@srt, $w);
    };
  @srt = sort { $a cmp $b } @srt;
  return (@srt);
}
```

We also wish to have a convenient method for turning an array into LaTeXcolumn data:

```
sub MakeLatexColumn
{ my(@ptids)=@_;
  # first, make a comma-string
  $_ = &CommaList(@ptids);
  # next, replace , with &
  s/,/\&/g;
  return($_);  # return the string
}
```

12.1.3 PrintBasicData

Finally, the actual printing routine:

```
 sub PrintBasicData
{
  my ($myODBC, $userid, $MNU);
     ($myODBC, $userid, $MNU) = @_;

  # get today, subtract 1 day (default day is yesterday)
  my ($now) = &GetLocalTime();
  $now =~ /^(\d{4})-(\d{2})-(\d{2})/;
  my ($jd) = &Julian($1, $2, $3, 0, 0, 0, 0);
  $jd --; # previous day!
  my ($yyyy, $mm, $dd) = &Gregorian($jd);
  my($thisday) = &FixDate($yyyy, $mm, $dd);

  $thisday = &Ask ($MNU, "Enter date as YYYY-MM-DD", $thisday);
  if (length $thisday < 1)
     { return 0;
     };

  $thisday =~ /^(\d{4})-(\d{1,2})-(\d{1,2})/;
  $thisday = &FixDate($1, $2, $3);
  my ($endstamp) = "$thisday 23:59:59";
  my ($startstamp) = "$thisday 00:00:00";

  my @CURRENTCASES = (); # useful list
  # note that if NO case is retrieved then SQL might fail!
```

```
  my ($ACTIVEPATIENTS, $SEEN, $NOTSEEN,
      $EPITOTAL, $EPISTART, $EPIEND,
      $PCATOTAL, $PCASTART, $PCAEND,
      $BOTH, $NEITHER,
      $ADMIT, $DISCHARGE);

  my (@L_SEEN,
      @L_EPITOTAL, @L_EPISTART, @L_EPIEND,
      @L_PCATOTAL, @L_PCASTART, @L_PCAEND,
      @L_BOTH,
      @L_ADMIT, @L_DISCHARGE);

  (@CURRENTCASES) = &SQLManySQL ($myODBC,
   "select distinct person from process where \
    proctype = 3 and \
      process > -1 AND \
   cast (rStart as varchar(19)) <=  '$endstamp' and \
   (rEnd IS NULL OR cast(rEnd as varchar(19)) >= '$startstamp')",
   "Get active patients");

   print LOGFILE "\n\n Current case list is <@CURRENTCASES>";
   # now associate cases with wards:
   # my %PTWARDS;  # these are global
   # my %WDLIST;   # """

   %PTWARDS = (); # clear
   %WDLIST = ();
   my ($pt, $wd);
   foreach $pt (@CURRENTCASES)
     { $wd = &WhichWard($myODBC, $pt, $thisday);
       $PTWARDS{$pt} = $wd;
       $WDLIST{$wd} = 0; # initialise counter!
     };
   my $COMMACASES = &CommaList(@CURRENTCASES);

#   print LOGFILE "\n Comma list is <$COMMACASES>";

   $ACTIVEPATIENTS = 1+$#CURRENTCASES;

  (@L_SEEN) = &SQLManySQL ($myODBC,
  "select distinct person from process \
   where person IN ($COMMACASES) AND \
     process IN \
   (SELECT process from epoch where \
   cast(oMade as varchar(19)) > '$startstamp' and \
   cast(oMade as varchar(19)) < '$endstamp')",
  "Patients seen");
```

```
$SEEN = 1 + $#L_SEEN;

$NOTSEEN = $ACTIVEPATIENTS - $SEEN;
# might have check [see later]

(@L_EPITOTAL) = &SQLManySQL($myODBC,
 "select distinct person from process \
 where person IN ($COMMACASES) AND \
  proctype = 110 and \
  process > -1 AND \
  cast(rStart as varchar(19)) <= '$endstamp' \
  and (rEnd IS NULL OR
        cast(rEnd as varchar(19)) >= '$startstamp')",
  "Total epidurals");
 $EPITOTAL = 1+ $#L_EPITOTAL;

(@L_EPISTART) = &SQLManySQL($myODBC,
 "select distinct person from process \
 where person IN ($COMMACASES) AND \
  proctype = 110 and \
  cast(rStart as varchar(19)) >= '$startstamp' \
  and cast(rStart as varchar(19)) <= '$endstamp'",
 "Epidurals started");
 $EPISTART = 1 + $#L_EPISTART;

(@L_EPIEND) = &SQLManySQL($myODBC,
 "select distinct person from process where \
  person IN ($COMMACASES) AND \
  proctype = 110 and \
  process > -1 AND \
  cast(rEnd as varchar(19)) >= '$startstamp' and \
  cast(rEnd as varchar(19)) <= '$endstamp'",
 "Epidurals removed");
 $EPIEND = 1 + $#L_EPIEND;

(@L_PCATOTAL) = &SQLManySQL($myODBC,
 "select distinct person from process where \
 person IN ($COMMACASES) AND \
 proctype = 390 and \
 process > -1 AND \
 cast(rStart as varchar(19)) <= '$endstamp' and\
 (rEnd IS NULL OR cast(rEnd as varchar(19))>= '$startstamp')",
"total PCAs");
$PCATOTAL = 1 + $#L_PCATOTAL;

(@L_PCASTART) = &SQLManySQL($myODBC,
 "select distinct person from process where \
 person IN ($COMMACASES) AND \
```

```
 proctype = 390 and \
 process > -1 AND \
 cast(rStart as varchar(19)) >= '$startstamp' and \
 cast(rStart as varchar(19)) <= '$endstamp'",
 "PCAs started");
 $PCASTART = 1 + $#L_PCASTART;

(@L_PCAEND) = &SQLManySQL($myODBC,
 "select distinct person from process where \
 person IN ($COMMACASES) AND \
 proctype = 390 and \
 process > -1 AND \
 cast(rEnd as varchar(19)) >= '$startstamp' and \
 cast(rEnd as varchar(19)) <= '$endstamp'",
"PCAs removed");
$PCAEND = 1 + $#L_PCAEND;

(@L_BOTH) = &SQLManySQL($myODBC,
 "select distinct person from process where \
  person IN ($COMMACASES) AND \
   proctype = 110 and person in
    (select distinct person from process where \
     proctype = 390 and \
      cast(rStart as varchar(19)) <= '$endstamp' and \
      (rEnd IS NULL \
        OR cast(rEnd as varchar(19))>= '$startstamp') ) \
   and cast(rStart as varchar(19)) <= '$endstamp' \
   and (rEnd IS NULL OR \
         cast(rEnd as varchar(19)) >= '$startstamp')",
 "Combined PCA+epidural");
 $BOTH = 1 + $#L_BOTH;


$NEITHER = $ACTIVEPATIENTS - ($EPITOTAL + $PCATOTAL -$BOTH);

(@L_ADMIT) = &SQLManySQL($myODBC,
 "select distinct person from process where \
 proctype = 3 and \
 process > -1 AND \
 cast(rStart as varchar(19)) >= '$startstamp' and \
 cast(rStart as varchar(19)) <= '$endstamp'",
 "Admissions");
 $ADMIT = 1 + $#L_ADMIT;

(@L_DISCHARGE) = &SQLManySQL($myODBC,
 "select distinct person from process where \
 proctype = 3 and \
  process > -1 AND \
```

```
   cast(rEnd as varchar(19)) >= '$startstamp' and \
   cast(rEnd as varchar(19)) <= '$endstamp'",
   "Discharges");
   $DISCHARGE = 1 + $#L_DISCHARGE;

  # finally slurp in template, fill in values and print PDF:

  my $template = &SlurpFile('daily_report.tex');
  if (length $template < 1)
     { &Alert($MAINW, "Daily reporting template not found");
       return 0;
     };



 # [here let's examine the cases not seen in more detail:

  my (@WERESEEN) = &SQLManySQL ($myODBC,
  "select distinct person from process \
   where person IN ($COMMACASES) AND \
   process > -1 AND \
     process IN \
   (SELECT process from epoch where \
   cast(oMade as varchar(19)) > '$startstamp' and \
   cast(oMade as varchar(19)) < '$endstamp')",
  "Unseen");

  my $uns;
  my ($UNSEENTABLE) =  '';
  my @UNSEEN = ();
  foreach $uns (@CURRENTCASES)
    {
      if (! &InArray ($uns, @WERESEEN))
         { push (@UNSEEN, $uns);
         };
    };

  my ($UNTEXT) = '';
  if ($#UNSEEN > -1)
     {
       my ($UNLIST) = &CommaList (@UNSEEN);
#       print LOGFILE "\n\n Unseen list is <$UNLIST>";

       my (@UNROWS) =
        &SQLManySQL ($myODBC,
        "SELECT DISTINCT pdoForename, pdoSurname, pdoHospNo, swrdText \
          FROM PERSDATA, BADOBS, WARD, ROOM, BED \
          WHERE pdoPerson IN ($UNLIST) AND \
```

```
           BADOBS.Bed = BED.bed AND \
           BED.Room = ROOM.room AND \
           ROOM.Ward = WARD.ward \
           AND BADOBS.Person = PERSDATA.pdoPerson \
           AND BADOBS.boInactive IS NULL \
         ORDER BY swrdText, pdoSurname",
         'get details, hmm');

         my ($fore, $sur, $hn, $wd);
         while ($#UNROWS > -1)
           {
             $fore = shift(@UNROWS);
             $sur  = shift(@UNROWS);
             $hn  = shift(@UNROWS);
             $wd = shift(@UNROWS);
             $UNTEXT .= "$wd & $sur & $fore & $hn \\\\";
           };
         $UNSEENTABLE = &SlurpFile('daily_unseen.tex');
         $UNSEENTABLE =~ s/\$\[UNTEXT\]/$UNTEXT/mg;
     };
# ??? CONSIDER THE POSSIBLITY THAT WARD FAILS ???
# end of examination]

# some frills:
 $now =~ /(^.* \d\d:\d\d)/;
 my ($NOW) = $1;
 my ($USERNAME) = &GetUserName($myODBC, $userid);


 # ward names:
 my (@WN) = &ListWardNames();
 my ($WARDNAMES) = &MakeLatexColumn(@WN); # no leading ampersand

 # refine data by ward:
 &FindWardCounts(@CURRENTCASES);
 $_ = &MakeLatexColumn(&ListWardCounts());
 $ACTIVEPATIENTS = "$ACTIVEPATIENTS & $_";

 &FindWardCounts(@L_SEEN);
 $_ = &MakeLatexColumn(&ListWardCounts());
 $SEEN = "$SEEN & $_";

 &FindWardCounts(@L_EPITOTAL);
 $_ = &MakeLatexColumn(&ListWardCounts());
 $EPITOTAL = "$EPITOTAL & $_";

 &FindWardCounts(@L_EPISTART);
 $_ = &MakeLatexColumn(&ListWardCounts());
```

```
$EPISTART = "$EPISTART & $_";

&FindWardCounts(@L_EPIEND);
$_ = &MakeLatexColumn(&ListWardCounts());
$EPIEND = "$EPIEND & $_";

&FindWardCounts(@L_PCATOTAL);
$_ = &MakeLatexColumn(&ListWardCounts());
$PCATOTAL = "$PCATOTAL & $_";

&FindWardCounts(@L_PCASTART);
$_ = &MakeLatexColumn(&ListWardCounts());
$PCASTART = "$PCASTART & $_";

&FindWardCounts(@L_PCAEND);
$_ = &MakeLatexColumn(&ListWardCounts());
$PCAEND = "$PCAEND & $_";

&FindWardCounts(@L_BOTH);
$_ = &MakeLatexColumn(&ListWardCounts());
$BOTH = "$BOTH & $_";

&FindWardCounts(@L_ADMIT);
$_ = &MakeLatexColumn(&ListWardCounts());
$ADMIT = "$ADMIT & $_";

&FindWardCounts(@L_DISCHARGE);
$_ = &MakeLatexColumn(&ListWardCounts());
$DISCHARGE = "$DISCHARGE & $_";


# fix up LaTeX table tabs:
my ($FIXTABS) = '';
my ($tabcount) = 1 + $#WN; # number of wards
while ($tabcount > 0)
  { $tabcount --;
    $FIXTABS .= 'c|'; # LaTeX-style tab. See usage.
  };

# substitute in ward tabs and names:
$template =~ s/\$\[FIXTABS\]/$FIXTABS/mg;
$template =~ s/\$\[WARDNAMES\]/$WARDNAMES/mg;

# substitute values into template:
$template =~ s/\$\[ACTIVEPATIENTS\]/$ACTIVEPATIENTS/mg;
$template =~ s/\$\[SEEN\]/$SEEN/mg;
$template =~ s/\$\[NOTSEEN\]/$NOTSEEN/mg;
$template =~ s/\$\[EPITOTAL\]/$EPITOTAL/mg;
```

```
  $template =~ s/\$\[EPISTART\]/$EPISTART/mg;
  $template =~ s/\$\[EPIEND\]/$EPIEND/mg;
  $template =~ s/\$\[PCATOTAL\]/$PCATOTAL/mg;
  $template =~ s/\$\[PCASTART\]/$PCASTART/mg;
  $template =~ s/\$\[PCAEND\]/$PCAEND/mg;
  $template =~ s/\$\[BOTH\]/$BOTH/mg;
  $template =~ s/\$\[NEITHER\]/$NEITHER/mg;
  $template =~ s/\$\[ADMIT\]/$ADMIT/mg;
  $template =~ s/\$\[DISCHARGE\]/$DISCHARGE/mg;
  $template =~ s/\$\[TODAY\]/$thisday/mg;
  #  $template =~ s/\$\[\]/$/mg;
  $template =~ s/\$\[NOW\]/$NOW/mg;
  $template =~ s/\$\[USERNAME\]/$USERNAME/mg;

  $template =~ s/\$\[UNSEENTABLE\]/$UNSEENTABLE/mg;

while ($template =~ /\$\[(.+)\]/)
      { my $dud = $1;
        &Alert($MAINW,
  "Error (grr)! Unknown variable <$dud> in daily stats template");
        $template =~ s/\$\[$dud\]/~~~/;
      };

  my $prok = 1;
  my $pfile="latex/$thisday.tex"; # write as DATE.tex to 'latex' subdirectory!
  open PFILE, ">$pfile" or $prok = 0; # hmm. Can overwrite!
  if (! $prok)
     { &Alert($MAINW, "Failed to create <$pfile>. Aborted!");
       return 0;
     };

  binmode(PFILE); # otherwise WinDOS messes up CR/LF
  print PFILE $template;
  close PFILE;

  # next convert LaTeX to PDF:
  &LatexToPdf ($thisday);  # written to pdf subdirectory

  # finally, print it:
  &PrintPdf ($thisday);
}
```

12.2 Monthly data

Useful routines:

12.2.1 Making a comma list

This clumsy routine takes an array and turns it into a list of items separated by commas:

```
sub CommaList
{ my (@commas);
     (@commas) =@_;

  my ($c);
  my ($opt) = '';

  foreach $c (@commas)
    {
      $opt .= "$c,";
    };

  if ($#commas < 1)
     { $ opt .= '0,0,'; # force extra item if low count
     };
  chop ($opt);  # remove terminal comma
  return ($opt);
}
```

We don't want a null list, so we always force a couple of zeroes if there are no items in the list.

12.3 Monthly data (2)

A variant routine, based on the enhanced daily report. Note that in this coding, only people who have been *discharged* from the service will be counted!

```
 sub PrintMonthlyData2
{
  my ($myODBC, $userid);
     ($myODBC, $userid) = @_;

  # get interval:
  my ($startday, $thisday) = &AskForInterval($ADMINMENU);
  if (  (length $startday < 1)
      ||(length $thisday < 1)
     ){ return; };
  my ($startstamp) = "$startday 00:00:00";
  my ($endstamp) = "$thisday 23:59:59";

  # kick of progress bar:
  $MAINW->focus(); # even focusForce() seems to do nil if focusFollowsMouse ???
```

```
$ADMINMENU->iconify;
$PROGTEXT = 'Preparing report..';
$BARPROGRESS = 0;
+OPTIONAL
$PROGRESSBAR->update();
-OPTIONAL
my $numsteps = 37;  # alter this if alter steps! NB.
my $barstep = $BARWIDTH/$numsteps;

# define SQL query STRING to retrieve cases of interest:
  my ($qi) = "SELECT DISTINCT PERSON from process where \
  proctype = 3 and process > -1 AND \
 cast (rStart as varchar(19)) <=  '$endstamp' and \
 cast(rEnd as varchar(19)) >= '$startstamp'";


# we are interested in patients seen, and total epidurals, pcas, both, neither.
# we will further sub-classify by ward

# [also consider classifying by discipline; and analysis of participating doctor
# including patients seen by these doctors, time spent, start and end time for
# each day, and so forth.
# We should almost certainly look at READMISSIONS in a period.

my (@TOTALCASES) = &SQLManySQL ($myODBC, $qi, "Get patient list");
my ($TOTALCASESNUM) = 1+$#TOTALCASES;
 # now associate cases with wards: %PTWARDS and %WDLIST are globals.

&StepTheBar($barstep); # step #1

 %PTWARDS = (); # clear
 %WDLIST = ();
 my ($pt, $wd);
 foreach $pt (@TOTALCASES)
   { $wd = &LstWard($myODBC, $pt, 1);
     $PTWARDS{$pt} = $wd;
     $WDLIST{$wd} = 0; # initialise counter!
     print '.';
   };

&StepTheBar($barstep); # step #2
my ($CONST_EPIDURAL) = 110;
my (@L_EPITOTAL) = &GetProcessList ($myODBC, $qi, $startstamp, $endstamp, $CONST
my ($EPITOTALNUM) = 1+ $#L_EPITOTAL;

&StepTheBar($barstep); # step #3
my ($CONST_PCA) = 390;
my (@L_PCATOTAL) = &GetProcessList ($myODBC, $qi, $startstamp, $endstamp, $CONST
```

```
my ($PCATOTALNUM) = 1+ $#L_PCATOTAL;

&StepTheBar($barstep); # step #4
my ($CONST_INTERSCALENE) = 120;
my (@L_INTERSCALENETOTAL) = &GetProcessList ($myODBC, $qi, $startstamp, $endstam
my ($INTERSCALENETOTALNUM) = 1+ $#L_INTERSCALENETOTAL;

&StepTheBar($barstep); # step #5
my ($CONST_INTERPLEURAL) = 135;
my (@L_INTERPLEURALTOTAL) = &GetProcessList ($myODBC, $qi, $startstamp, $endstam
my ($INTERPLEURALTOTALNUM) = 1+ $#L_INTERPLEURALTOTAL;

&StepTheBar($barstep); # step #6
my ($CONST_EXTRAPLEURAL) = 134;
my (@L_EXTRAPLEURALTOTAL) = &GetProcessList ($myODBC, $qi, $startstamp, $endstam
my ($EXTRAPLEURALTOTALNUM) = 1+ $#L_EXTRAPLEURALTOTAL;

&StepTheBar($barstep); # step #7
my ($CONST_PARAVERTEBRAL) = 137;
my (@L_PARAVERTEBRALTOTAL) = &GetProcessList ($myODBC, $qi, $startstamp, $endsta
my ($PARAVERTEBRALTOTALNUM) = 1+ $#L_PARAVERTEBRALTOTAL;

&StepTheBar($barstep); # step #8
my ($CONST_FEMORAL) = 140;
my (@L_FEMORALTOTAL) = &GetProcessList ($myODBC, $qi, $startstamp, $endstamp, $C
my ($FEMORALTOTALNUM) = 1+ $#L_FEMORALTOTAL;

&StepTheBar($barstep); # step #9
my ($CONST_SCIATIC) = 145;
my (@L_SCIATICTOTAL) = &GetProcessList ($myODBC, $qi, $startstamp, $endstamp, $C
my ($SCIATICTOTALNUM) = 1+ $#L_SCIATICTOTAL;

&StepTheBar($barstep); # step #10
my ($CONST_CARDIAC) = 1000;
my (@L_CARDIACTOTAL) = &GetProcessList ($myODBC, $qi, $startstamp, $endstamp, $C
my ($CARDIACTOTALNUM) = 1+ $#L_CARDIACTOTAL;

&StepTheBar($barstep); # step #11
my ($CONST_RENAL) = 1010;
my (@L_RENALTOTAL) = &GetProcessList ($myODBC, $qi, $startstamp, $endstamp, $CON
my ($RENALTOTALNUM) = 1+ $#L_RENALTOTAL;

&StepTheBar($barstep); # step #12
my ($CONST_HEPATIC) = 1030;
my (@L_HEPATICTOTAL) = &GetProcessList ($myODBC, $qi, $startstamp, $endstamp, $C
my ($HEPATICTOTALNUM) = 1+ $#L_HEPATICTOTAL;

&StepTheBar($barstep); # step #13
```

```
  my ($CONST_CHRONICPAIN) = 1060;
  my (@L_CHRONICPAINTOTAL) = &GetProcessList ($myODBC, $qi, $startstamp, $endstamp
  my ($CHRONICPAINTOTALNUM) = 1+ $#L_CHRONICPAINTOTAL;

  &StepTheBar($barstep); # step #14
  my ($CONST_CHRONICOPIATES) = 1070;
  my (@L_CHRONICOPIATESTOTAL) = &GetProcessList ($myODBC, $qi, $startstamp, $endst
  my ($CHRONICOPIATESTOTALNUM) = 1+ $#L_CHRONICOPIATESTOTAL;


#  my ($CONST_xxx) = 000;
#  my (@L_xxxTOTAL) = &GetProcessList ($myODBC, $qi, $startstamp, $endstamp, $CONS
#  my ($xxxTOTALNUM) = 1+ $#L_xxxTOTAL;



  # here might include other blocks eg. femoral, sciatic; as well as spinal morphi
  # also is good idea to determine INTERSECTION of PCA and EPI lists, as well
  # as those in TOTALCASES but in neither PCA nor EPI.


  # (what about listing of errors)? Best have separate 'error report'!


  # (what about deviations from normal eg. very long consultation times, other odd


  # what about listing by type of proc? [explore]


  # what about duration of stay?
  # select (rend - rstart) day to hour as fred  from process where proctype = 3  a



 # some frills:
  my ($now) = &GetLocalTime();
  $now =~ /(^.*) (\d\d:\d\d)/;
  my ($NOW) = "$1 $2";
  my ($TODAY) = $1;
  my ($USERNAME) = &GetUserName($myODBC, $userid);

  # slurp in template, fill in values and print PDF:
  my $template = &SlurpFile('long_report.tex');
  if (length $template < 1)
     { &Alert($MAINW, "Long reporting template not found");
       return 0;
     };
```

```
&StepTheBar($barstep); # step #15
# ward names:
my (@WN) = &ListWardNames();
my ($WARDNAMES) = &MakeLatexColumn(@WN); # no leading ampersand

&StepTheBar($barstep); # step #16
# refine data by ward:
&FindWardCounts(@TOTALCASES);
$_ = &MakeLatexColumn(&ListWardCounts());
$TOTALCASESNUM = "$TOTALCASESNUM & $_";

&StepTheBar($barstep); # step #17
# likewise for modalities...
&FindWardCounts(@L_PCATOTAL);
$_ = &MakeLatexColumn(&ListWardCounts());
$PCATOTALNUM = "$PCATOTALNUM & $_";

&StepTheBar($barstep); # step #18
&FindWardCounts(@L_EPITOTAL);
$_ = &MakeLatexColumn(&ListWardCounts());
$EPITOTALNUM = "$EPITOTALNUM & $_";

&StepTheBar($barstep); # step #19
#other:
&FindWardCounts(@L_INTERSCALENETOTAL);
$_ = &MakeLatexColumn(&ListWardCounts());
$INTERSCALENETOTALNUM = "$INTERSCALENETOTALNUM & $_";

&StepTheBar($barstep); # step #20
&FindWardCounts(@L_INTERPLEURALTOTAL);
$_ = &MakeLatexColumn(&ListWardCounts());
$INTERPLEURALTOTALNUM = "$INTERPLEURALTOTALNUM & $_";

&StepTheBar($barstep); # step #21
&FindWardCounts(@L_EXTRAPLEURALTOTAL);
$_ = &MakeLatexColumn(&ListWardCounts());
$EXTRAPLEURALTOTALNUM = "$EXTRAPLEURALTOTALNUM & $_";

&StepTheBar($barstep); # step #22
&FindWardCounts(@L_PARAVERTEBRALTOTAL);
$_ = &MakeLatexColumn(&ListWardCounts());
$PARAVERTEBRALTOTALNUM = "$PARAVERTEBRALTOTALNUM & $_";

&StepTheBar($barstep); # step #23
&FindWardCounts(@L_FEMORALTOTAL);
$_ = &MakeLatexColumn(&ListWardCounts());
$FEMORALTOTALNUM = "$FEMORALTOTALNUM & $_";
```

```
  &StepTheBar($barstep); # step #24
  &FindWardCounts(@L_SCIATICTOTAL);
  $_ = &MakeLatexColumn(&ListWardCounts());
  $SCIATICTOTALNUM = "$SCIATICTOTALNUM & $_";

  &StepTheBar($barstep); # step #25
  &FindWardCounts(@L_CARDIACTOTAL);
  $_ = &MakeLatexColumn(&ListWardCounts());
  $CARDIACTOTALNUM = "$CARDIACTOTALNUM & $_";

  &StepTheBar($barstep); # step #26
  &FindWardCounts(@L_RENALTOTAL);
  $_ = &MakeLatexColumn(&ListWardCounts());
  $RENALTOTALNUM = "$RENALTOTALNUM & $_";

  &StepTheBar($barstep); # step #27
  &FindWardCounts(@L_HEPATICTOTAL);
  $_ = &MakeLatexColumn(&ListWardCounts());
  $HEPATICTOTALNUM = "$HEPATICTOTALNUM & $_";

  &StepTheBar($barstep); # step #28
  &FindWardCounts(@L_CHRONICPAINTOTAL);
  $_ = &MakeLatexColumn(&ListWardCounts());
  $CHRONICPAINTOTALNUM = "$CHRONICPAINTOTALNUM & $_";

  &StepTheBar($barstep); # step #29
  &FindWardCounts(@L_CHRONICOPIATESTOTAL);
  $_ = &MakeLatexColumn(&ListWardCounts());
  $CHRONICOPIATESTOTALNUM = "$CHRONICOPIATESTOTALNUM & $_";

#  &FindWardCounts(@L_xxxTOTAL);
#  $_ = &MakeLatexColumn(&ListWardCounts());
#  $xxxTOTALNUM = "$xxxTOTALNUM & $_";

  # fix up LaTeX table tabs:
  my ($FIXTABS) = '';
  my ($tabcount) = 1 + $#WN; # number of wards
  while ($tabcount > 0)
    { $tabcount --;
      $FIXTABS .= 'c|'; # LaTeX-style tab. See usage.
    };

  # now analyse epochs: ???

  my ($minepoch) = &GetSQL($myODBC,
   "SELECT MIN(epoch) FROM EPOCH WHERE cast (oMade as varchar(19))
     >= '$startstamp' and cast(oMade as varchar(19))
```

```
   < '$endstamp'", '*first relevant epoch');
my ($maxepoch) = &GetSQL($myODBC,
 "SELECT MAX(epoch) FROM EPOCH WHERE cast (oMade as varchar(19))
   >= '$startstamp' and cast(oMade as varchar(19))
   < '$endstamp'", '*last relevant epoch');
print "\n Debug: epoch range is $minepoch to $maxepoch";

# first, pain team members:

&StepTheBar($barstep); # step #30
(@TABLEDATA) = &SQLManySQL($myODBC,
"select pdoforename || ' ' || pdosurname, duration/60, episodes, duration/episod
from (select person as teammember, sum(olength)/1000 as duration, count(olength)
from epoch where olength is not null and epoch >= $minepoch
 and epoch <= $maxepoch
group by person),
persdata where teammember = persdata.pdoperson
  and duration > 3600
order by duration", 'get epoch data');

my (@epochheaders) = ('Name','Total duration (min)','Patient Encounters','Averag
my ($EPOCHTABLE) = &MakeLatexTable(@epochheaders);

&StepTheBar($barstep); # step #31
# by operation type [?]
# [need to refine identification of type of surgery]
(@TABLEDATA) = &SQLManySQL($myODBC,
"select cttext, operations from (select surgtype as st, count(epoch) as operatio
  from surgtypeob where Epoch >= $minepoch AND Epoch <= $maxepoch
  group by surgtype), surgtype where st = surgtype.surgtype
  order by operations desc", 'get counts for various types of surgery');
my (@surgheaders) = ('Type of surgery','Number of operations');
my ($SURGTABLE) = &MakeLatexTable(@surgheaders);

# what about:
# select process.person as patient, count(epoch.epoch) as visits, sum(epoch.olen
#
# select patient, visits, lengthseconds, lengthseconds/visits as averageconsult
#
# select cast(visits*5 as varchar(5)) || '+' as VisitCount, count(1) as NumberOf
#
# select cast(visits*5 as varchar(5)) || '+' as VisitCount, count(1) as NumberOf
# select visitcount, numberofpatients, DurationMin, DurationMin/NumberOfPatients
#
# here are the data for those with five or fewer visits:
# select visitcount, numberofpatients, DurationMin, DurationMin/NumberOfPatients

&StepTheBar($barstep); # step #32
```

```
(@TABLEDATA) = &SQLManySQL($myODBC,
  "select visitcount, numberofpatients, DurationMin, DurationMin/NumberOfPatient
    as AverageTotalDuration from
      (select cast(visits*5 as varchar(5)) || '+' as VisitCount, count(1)
         as NumberOfPatients, sum(lengthseconds)/60 as DurationMin
           from (select process.person as patient, count(epoch.epoch)/5 as visit
              sum(epoch.olength/1000) as lengthseconds
              from process, epoch where epoch.process = process.process and
              epoch.epoch >= $minepoch and
              epoch.epoch <= $maxepoch
              and epoch.olength is not null
              and epoch.olength > 30000
              group by process.person)
         group by visits)",
  'get visit counts and details');
my (@visitheaders) = ('Number of Visits','Patients','Total Time(min)','Avg Total
my ($VISITTBL) = &MakeLatexTable(@visitheaders);

&StepTheBar($barstep); # step #33
# next, break down 'five or fewer visits':
(@TABLEDATA) = &SQLManySQL($myODBC,
  "select visitcount, numberofpatients, DurationMin,
          DurationMin/NumberOfPatients as AverageTotalDuration
            from
    (select visits as VisitCount, count(1) as NumberOfPatients,
            sum(lengthseconds)/60 as DurationMin
       from (select process.person as patient, count(epoch.epoch) as visits,
                sum(epoch.olength/1000) as lengthseconds
                from process,epoch where epoch.process = process.process
                  and epoch.olength is not null and epoch.olength > 30000
                   and epoch.epoch >= $minepoch and epoch.epoch <= $maxepoch
                  group by process.person)
       group by visits having visits < 6)",
  'get visits numbering 5 or less');
my (@visit5headers) = ('Number of Visits','Patients','Total Time(min)','Avg Tota
my ($VISIT5TBL) = &MakeLatexTable(@visit5headers);

&StepTheBar($barstep); # step #34
# substitute in ward tabs and names:
$template =~ s/\$\[FIXTABS\]/$FIXTABS/mg;
$WARDNAMES =~ s/99/Other/;
$template =~ s/\$\[WARDNAMES\]/$WARDNAMES/mg;

# substitute values into template:
$template =~ s/\$\[TOTALCASES\]/$TOTALCASESNUM/mg;
$template =~ s/\$\[PCATOTAL\]/$PCATOTALNUM/mg;
$template =~ s/\$\[EPITOTAL\]/$EPITOTALNUM/mg;
```

```
  $template =~ s/\$\[INTERSCALENE\]/$INTERSCALENETOTALNUM/mg;
  $template =~ s/\$\[INTERPLEURAL\]/$INTERPLEURALTOTALNUM/mg;
  $template =~ s/\$\[EXTRAPLEURAL\]/$EXTRAPLEURALTOTALNUM/mg;
  $template =~ s/\$\[PARAVERTEBRAL\]/$PARAVERTEBRALTOTALNUM/mg;
  $template =~ s/\$\[FEMORAL\]/$FEMORALTOTALNUM/mg;
  $template =~ s/\$\[SCIATIC\]/$SCIATICTOTALNUM/mg;
  $template =~ s/\$\[CARDIAC\]/$CARDIACTOTALNUM/mg;
  $template =~ s/\$\[RENAL\]/$RENALTOTALNUM/mg;
  $template =~ s/\$\[HEPATIC\]/$HEPATICTOTALNUM/mg;
  $template =~ s/\$\[CHRONICPAIN\]/$CHRONICPAINTOTALNUM/mg;
  $template =~ s/\$\[CHRONICOPIATES\]/$CHRONICOPIATESTOTALNUM/mg;

  # various tables, such as epoch table:
  $template =~ s/\$\[EPOCHTABLE\]/$EPOCHTABLE/mg;
  $template =~ s/\$\[SURGTABLE\]/$SURGTABLE/mg;
  $template =~ s/\$\[VISITTBL\]/$VISITTBL/mg;
  $template =~ s/\$\[VISIT5TBL\]/$VISIT5TBL/mg;

  # fix frills:
  $template =~ s/\$\[STARTSTAMP\]/$startstamp/mg;
  $template =~ s/\$\[ENDSTAMP\]/$endstamp/mg;
  $template =~ s/\$\[NOW\]/$NOW/mg;
  $template =~ s/\$\[TODAY\]/$TODAY/mg;
  $template =~ s/\$\[USERNAME\]/$USERNAME/mg;

while ($template =~ /\$\[(.+)\]/)
      { my $dud = $1;
        &Alert($MAINW,
  "Error (oops)! Unknown variable <$dud> in long stats template");
        $template =~ s/\$\[$dud\]/~~~/;
      };

  my $prok = 1;
  my $filename = "LONG$thisday";
  my $pfile="latex/$filename.tex";         # write to 'latex' subdirectory!
  open PFILE, ">$pfile" or $prok = 0; # hmm. Can overwrite!
  if (! $prok)
     { &Alert($MAINW, "Failed to create <$pfile>. Aborted!");
       return 0;
     };

  &StepTheBar($barstep); # step #35
  binmode(PFILE);
  print PFILE $template;
  close PFILE;

  &StepTheBar($barstep); # step #36
  # next convert LaTeX to PDF:
```

```
  &LatexToPdf ($filename);  # written to pdf subdirectory

  &StepTheBar($barstep); # step #37
  # finally, print it:
  &PrintPdf ($filename);

  $PROGTEXT = '';
  $BARPROGRESS = 0;
  $ADMINMENU->deiconify;
  $ADMINMENU->focus();

  &Alert($MAINW, "Report completed");
}
```

Here's the simple progress bar 'stepper':

```
sub StepTheBar
{ my ($barstep) = @_;
  $BARPROGRESS += $barstep;  # step #1
  +OPTIONAL
      $PROGRESSBAR->update();
  -OPTIONAL
}
```

12.3.1 Make LaTeX table

Given a 2-dimensional array, with a header, turn it into a table:

```
sub MakeLatexTable
{ my (@columns) = @_;
  # @TABLEDATA is global containing actual data
  my ($colcount) = 1 + $#columns;

  my ($headerformat) = '|';
  my ($columnames) = '';
  my ($h);           # column header
  foreach $h (@columns)
    { $headerformat .= "l|";
      $columnames .= "\\emph{$h} &";
    };
  chop($columnames); # remove terminal ampersand

  my ($columndata) = '';
  my ($c) = $colcount;
  my ($i);
  foreach $i (@TABLEDATA)
    { $columndata .= "$i &";
      $c --;
```

```
      if ($c <= 0)   # move to next data row..
         { $c = $colcount;
           chop($columndata); # remove terminal '&'
           $columndata .= "\\\\ \n";
    };   };

  my ($tstr) = "\\begin{tabular}{$headerformat}\\hline $columnames \\\\ \
    \\hline $columndata \\hline \\end{tabular}";
 return ($tstr);
}
```

12.3.2 Get date interval (interactive)

```
sub AskForInterval
{ my ($MNU);
     ($MNU) = @_;

  my ($now) = &GetLocalTime();
  my ($endday);
  my ($startday, $thisday);
  # default for $endday is today, default for $startday is month start

  $endday = &GetLocalTime();
  $endday =~ /^(\d{4})-(\d{2})-(\d{2})/;
  $endday = "$1-$2-$3";
  $thisday = $endday;
  $startday = "$1-$2-01";

  $startday = &Ask ($MNU, "Enter FIRST date as YYYY-MM-DD", $startday);
  if ($startday !~ /^(\d{4})-(\d{1,2})-(\d{1,2})/)
     { &Alert($MNU, "Bad start date: $startday");
       return ('', '');
     };
  $startday = &FixDate($1, $2, $3);
  my ($jstart) = &Julian($1, $2, $3, 0, 0, 0, 0);

  $endday = &Ask ($MNU, "Enter SECOND (end) date as YYYY-MM-DD", $endday);
  if ($endday !~ /^(\d{4})-(\d{1,2})-(\d{1,2})/)
     { &Alert($MNU, "Bad end date: $endday");
       return ('', '');
     };
  $endday = &FixDate($1, $2, $3);
  my ($jend) = &Julian($1, $2, $3, 0, 0, 0, 0);
  if ($jend <= $jstart)
     { &Alert($MNU, "End date ($endday) MUST be after start date ($startday)!");
       return ('', '');
     };
  return ($startday, $endday);
```

```
}
```

12.3.3 Find last ward patient was on

Given patient ID, determine most recent ward, from BADOBS table:

```
sub LstWard
{ my($myODBC, $ptID, $truncated)=@_;

 my ($wd) = &GetSQL ($myODBC,
  "SELECT Bed/10000 FROM BADOBS WHERE badobs = (SELECT max(badobs) FROM BADOBS
    WHERE Person = $ptID)", 'get most recent bed');

 if (   ($truncated)
     && (($wd < 29)  || ($wd > 83))
    )
    { $wd = 99;
    };

return ($wd);
}
```

12.3.4 Get Process list

Given the Type of a particular process and a query, with date limits, obtain a list of all patients with such an associated process type.

```
sub GetProcessList
{ my ($myODBC, $qi, $startstamp, $endstamp, $CONST_X);
     ($myODBC, $qi, $startstamp, $endstamp, $CONST_X) =@_;

  my (@L_X) = &SQLManySQL($myODBC,
   "select distinct person from process \
   where person IN ($qi) AND \
    proctype = $CONST_X and \
    process > -1 AND \
          cast(rStart as varchar(19)) < '$endstamp' \
    and ( cast(rEnd as varchar(19)) >= '$startstamp')",
    'Total for process type X');
  return (@L_X);
}
```

12.4 Data tree for a single patient

Here, given the identifier of a single person (we will enter an NHI, and check that it corresponds to just one person), we print a tree structure describing all associated

processes, the EPOCH, STOPPROC and RX entries corresponding to each PROCESS, and all related tables (for EPOCH this means PERSDATA, MEDSCORE, SURGTYPEOB, NONEVENT, MEASURE, PAINSCORE, RXOBS, INFUSIONOBS, PCA, PCASETTINGS, RGNOBS, COMMENT, ISPROBLEM and BADOBS entries.

We won't format things in too fancy a fashion, simply producing tabbed text written using a Perl print statement. We will use our meta (x-) tables to extract dependent tables!

```
sub PrintPatientDataTree
{ my ($myODBC);
     ($myODBC) = @_;

  my $nhi = '';
  my $ERRCOUNT = 0;
  my $PATIENT=0;

  $nhi = &Ask ($ADMINMENU,
     "Enter NHI",
     $nhi);
  if (length $nhi < 7)
     { if ( $nhi =~ /^\d+$/ )  # a key number!!
           { $PATIENT = $nhi;
           } else
           { &Alert($MAINW, "What?? <$nhi>");
             return;
           };
     } else
     { $nhi =~tr/a-z/A-Z/; # uppercase
       my (@ptids) = &GetSQL ($myODBC,
        "SELECT DISTINCT pdoPerson from PERSDATA WHERE pdoHospNo = '$nhi'",
        'get all hosp Nos');
       if ($#ptids < 0)
          {&Alert($MAINW, "No patients found for Hosp. No. <$nhi>");
           return;
          };
       if ($#ptids > 0)
          {&Alert($MAINW, "ERROR! Duplicates (@ptids) for Hosp. No. <$nhi>. Termin
           return;
          };
       $PATIENT = pop(@ptids);
     };
  my $ok = 1;
  open PRINPT, ">data/PatientDataTree.txt" or $ok = 0;
  if (! $ok)
     { &Alert($MAINW, "Failed to open PatientDataTree.txt !");
       return;
```

```
    };

 print PRINPT "Data Tree for NHI 'Snhi', Patient ID $PATIENT";

 my ($cold, $pborn, $pdied, $pmade, $pstatus) = &GetSQL($myODBC,
  "SELECT cold, cast(pBorn as varchar(10)), \
          cast (pDied as varchar(10)), \
          cast (pMade as varchar(19)), \
          pStatus FROM PERSON \
    WHERE person = $PATIENT",
  'get basic PERSON data');
  if (length $cold > 0) { $cold = '*'; };

 print PRINPT "\n $cold $pborn--$pdied (Created: $pmade) status: $pstatus";

 my (@PROCS) = &SQLManySQL ($myODBC,
  "SELECT process from PROCESS WHERE Person = $PATIENT \
   AND process > -1 \
    ORDER BY ProcType, rCreated",
  'get all procs for tree');

  my $pr;
  my $prtot = 1+$#PROCS;
  print PRINPT "\n" . "Processes: ($prtot)";
  foreach $pr (@PROCS)
    {
      print PRINPT "\n PROC-->$pr: ";
      my ($cold, $proctype, $rptNature,
          $rplanner, $rstart, $rend, $rcreated) =
      &GetSQL($myODBC,
       "SELECT PROCESS.cold, PROCESS.ProcType, rptNature, rPlanner, \
         cast(rStart as varchar(19)), \
         cast(rEnd as varchar(19)), \
         cast(rCreated as varchar(19)) FROM \
         PROCESS, PROCTYPE WHERE \
           PROCESS.ProcType = PROCTYPE.proctype \
           AND process = $pr",
       'get proc details');

       # fix 'retired' rEnd values:
       if (  (length $rend == 19)
           &&($rend le '1910')
          )
          { $rend = '[deleted]';
          };

      if (length $cold > 0) { $cold = '*'; };
      my ($AUTHOR) = GetUserName($myODBC, $rplanner);
```

```
        $AUTHOR = "$AUTHOR [$rplanner]";
      print PRINPT
       "$cold $rptNature($proctype), $rstart--$rend, ($rcreated by $AUTHOR )";


      # first, stopprocs:
      my (@STOPPROCS) = &SQLManySQL($myODBC,
       "SELECT stopproc FROM STOPPROC WHERE PROCESS = $pr",
       'get stopproc list');
      my $cnt;
      $cnt = 1+$#STOPPROCS;
      if ($cnt > 0)
         { print PRINPT "\n stop-procs: $cnt ";
         };
      my $stp;
      foreach $stp (@STOPPROCS)
        {  &PrintStopProcData ($myODBC, $stp);
        };

      # next, RX entries:
      my (@RX) =  &SQLManySQL($myODBC,
       "SELECT rx FROM RX WHERE PROCESS = $pr",
       'get rx entry list');
      $cnt = 1+$#RX;
      if ($cnt > 0)
         { print PRINPT "\n Rx entries: $cnt ";
         };
      my $rx;
      foreach $rx (@RX)
        {  &PrintRxData ($myODBC, $rx);
        };

      # finally, epochs:
      my (@EPOCHS) = &SQLManySQL($myODBC,
       "SELECT epoch FROM EPOCH WHERE PROCESS = $pr",
       'get epoch list');
      $cnt = 1+$#EPOCHS;
      if ($cnt > 0)
         { print PRINPT "\n epochs: $cnt ";
         };
      my $ep;
      foreach $ep (@EPOCHS)
        { $ERRCOUNT +=
            &PrintEpochData ($myODBC, $ep, $PATIENT, $rstart, $rend);
        };

   };
print PRINPT "\n\n Error count was $ERRCOUNT";
```

```
  close PRINPT;
  &Alert($MAINW, "Written to data\\PatientDataTree.txt");
}
```

Here are the subsidiary routines:

```
sub PrintStopProcData
{ my ($myODBC, $code);
     ($myODBC, $code) = @_;

  print PRINPT "\n Stop $code: ";

  my ($cold, $whystop) = &GetSQL ($myODBC,
    "SELECT STOPPROC.cold, WHYSTOP.wText FROM STOPPROC, WHYSTOP \
      WHERE STOPPROC.WhyStop = WHYSTOP.whystop \
      AND stopproc = $code", 'get stop code');

  if (length $cold > 0) { $cold = '*'; };

  print PRINPT "$cold $whystop";
  return 0;
}

sub PrintRxData
{ my ($myODBC, $code);
     ($myODBC, $code) = @_;
  print PRINPT "\n Rx $code: (drug conc rate dose(mg) interval) ";

  my ($cold, $drug, $concentration, $rate, $dose, $interval) =
   &GetSQL ($myODBC,
    "SELECT RX.cold, Drug, gConcentration, gRate, gDose, gInterval FROM \
      RX WHERE rx = $code",
    'get RX entry');

   if (length $cold > 0) { $cold = '*'; };

   $dose /= 1000; # convert to mg

   my ($tradename, $formulation) = &GetSQL($myODBC,
    "SELECT dTrade, dFormText FROM DRUG, DRUGFORM \
      WHERE Drug.DrugForm = DRUGFORM.drugform \
      AND drug = $drug", 'get drug details');

  print PRINPT "\n $cold $tradename($formulation), $concentration, ";
  print PRINPT "$rate, $dose, $interval";
  return 0;
}
```

Epochs are more challenging. We might consider automating extraction of data from all the referencing tables, but this would be tricky and perhaps less clear. We plod through manually.

```
sub PrintEpochData
{ my ($myODBC, $code, $PATIENT, $rstart, $rend);
     ($myODBC, $code, $PATIENT, $rstart, $rend) = @_;
  print PRINPT "\n epoch=>$code: ";

  my $ERRCOUNT = 0;

  my ($cold, $made, $duration, $person) = &GetSQL($myODBC,
   "SELECT cold, cast(oMade as varchar(19)), oLength, Person FROM EPOCH WHERE \
     epoch = $code", 'get epoch data');

  if (length $cold > 0) { $cold = '*'; };

  if (length $duration > 0)
     { $duration /= 1000; # convert from ms to sec
     };

  if (  ($made lt $rstart)
      ||( ($made gt $rend) && ($rend gt '1910'))
     )
      { $made = "$made [DATE ERROR]";
        $ERRCOUNT ++;
      };

  my ($AUTHOR) = GetUserName($myODBC, $person);
  $AUTHOR = "$AUTHOR [$person]";

  print PRINPT "$cold $made($duration) $AUTHOR";

  # PERSDATA ======================================
  my ($epoch, $cold, $surname, $forename, $hospno, $gender) = &GetSQL ($myODBC,
   "SELECT Epoch, cold, pdoSurname, pdoForename, pdoHospNo, pdoGender FROM \
     PERSDATA WHERE Epoch = $code", 'get persdata for epoch');
   # theoretically might be several persdata entries: should really check this!

  if (length $epoch > 0)
     { if (length $cold > 0) { $cold = '*'; };
       my ($sex) = '?';
       if ($gender == 1)
          { $sex = 'F';
          }
       elsif ($gender == 2)
          { $sex = 'M';
          };
```

```
      print PRINPT "\n Personal data: $cold, $forename, $surname,";
      print PRINPT " $hospno ($gender)";
    };

# COMMENT ======================================
my (@comments) = &SQLManySQL ($myODBC,
  "SELECT cText FROM COMMENT WHERE Epoch = $code",
  'get comment(s)');
my $c;
foreach $c (@comments)
  {
    print PRINPT "\n Comment: $c";
  };

# SURGTYPEOB ===================================

my (@SURGTYPE) = &SQLManySQL($myODBC,
"SELECT ctText FROM SURGTYPE, SURGTYPEOB WHERE \
  SURGTYPEOB.SurgType = SURGTYPE.surgtype AND \
  Epoch = $code", 'get surgery type(s)');
  # hmm. might document cold/not!
my $s;
foreach $s (@SURGTYPE)
  {
    print PRINPT "\n Surgery type: $s";
  };

# NONEVENT =====================================
my (@NONEVENT) = &SQLManySQL($myODBC,
 "SELECT nonevent FROM nonevent WHERE \
   Epoch = $code", 'get all nonevents for epoch');

 my $n;
 foreach $n (@NONEVENT)
   {
     my ($cold, $noCode, $noValue) = &GetSQL($myODBC,
      "SELECT cold, noCode, noValue FROM NONEVENT \
        WHERE nonevent = $n", 'get non-event data');

     if (length $cold > 0) { $cold = '*'; };

     my $NE='?';
     # here substitute values for noCode:
     if ($noCode == 1)
        { $NE = 'regional';
        }
     elsif ($noCode == 2)
        { $NE = 'IV PCA';
```

```
        }
      elsif ($noCode == 3)
        { $NE = 'oral Rx';
        }
      elsif ($noCode == 4)
        { $NE = 'other Rx';
        };

      print PRINPT "\n 'Non event': $cold $NE = $noValue";
    };

# ISPROBLEM ======================================

my ($ip, $isprob) = &GetSQL ($myODBC,
  "SELECT isproblem, prIsOrNot FROM ISPROBLEM WHERE Epoch = $code",
  'check if problem');
# ideally should check for several. Practically is just 1.
if (length $ip > 0)
   {
     print PRINPT "\n Problem: $isprob";
   };

# BADOBS =====================================

my (@BADOBS) = &SQLManySQL($myODBC,
 "SELECT badobs FROM badobs WHERE Epoch = $code",
 'get badobs for an epoch');

 my ($b);
 foreach $b (@BADOBS)
   {
     my ($cold, $bed, $inactive, $person) = &GetSQL($myODBC,
       "SELECT cold, bed, boInactive, Person FROM BADOBS \
         WHERE badobs = $b", 'get badobs data');

     if (length $cold > 0) { $cold = '*'; };

     my $err = '';
     if ($person != $PATIENT)
        { $err = "***ERROR*** Bad PERSON reference ($person)";
          $ERRCOUNT ++;
        };
     print PRINPT "\n Bed data: $cold $bed (inactive=$inactive)";
   };

# MEDSCORE ===================================

my (@MEDSCORE) = &SQLManySQL($myODBC,
```

```
  "SELECT medscore from MEDSCORE where Epoch = $code",
  'get medscore entries for epoch');

my $m;
foreach $m (@MEDSCORE)
  {
    my ($cold, $msoNature, $msoValue) = &GetSQL($myODBC,
      "SELECT cold, msoNature, msoValue FROM MEDSCORE where \
       medscore = $m", 'get med score for entry');

    if (length $cold > 0) { $cold = '*'; };

    my ($MS) = '?';
    if ($msoNature == 1)
       { $MS = 'ASA E score:';
       }
    elsif ($msoNature == 2)
       { $MS = 'ASA:';
       };
    print PRINPT "\n Medical score: $cold $MS=$msoValue";
  };


# MEASURE  =====================================

my ($measure, $meWt) = &SQLManySQL($myODBC,
 "SELECT measure, meWt FROM MEASURE WHERE Epoch = $code",
 'get weight for epoch');
$meWt /= 1000; # convert to kg.
if (length $measure > 0)
   {
     print PRINPT "\n Weight: $meWt";
   };
# theoretically might be several values. Unlikely.

# PAINSCORE ====================================

my ($cold, $ps, $psoRest, $psoMovement, $psoCough) = &GetSQL($myODBC,
 "SELECT psoRest, psoMovement, psoCough FROM PAINSCORE \
   WHERE Epoch = $code", 'get pain scores');
if (length $ps > 0)
   {
     if (length $cold > 0) { $cold = '*'; };
     print PRINPT
"\n Pain scores: $cold $psoRest, $psoMovement, $psoCough";
   };

# RXOBS ========================================
```

```
my ($cold, $rxobs, $rxoTotal) = &GetSQL($myODBC,
  "SELECT cold, rxobs, rxoTotal FROM RXOBS \
    WHERE Epoch = $code", 'get rx total');
if (length $rxobs > 0)
   {
     if (length $cold > 0) { $cold = '*'; };
     print PRINPT "\n Rx total: $cold $rxoTotal";
   };

# INFUSIONOBS ======================================

my ($cold, $inf, $inoRate, $inoConc) = &GetSQL($myODBC,
  "SELECT cold, infusionobs, inoRate, inoConc FROM \
    INFUSIONOBS WHERE Epoch = $code",
  'get infusionobs data');
if (length $inf > 0)
   {
     if (length $cold > 0) { $cold = '*'; };
     print PRINPT "\n Infusion data: $cold $inoRate ($inoConc)";
   };

# RGNOBS ===========================================

my ($cold, $rgnobs, $rgoPressure, $rgoSite, $rgoMotor, $rgoLevel,
    $rgoMobile, $rgoHeadache, $rgoSitePain, $rgoBackache, $rgoInconti) =
&GetSQL($myODBC, "SELECT cold, rgnobs, rgoPressure, rgoSite, rgoMotor, \
   rgoLevel, rgoMobile, rgoHeadache, rgoSitePain, rgoBackache, rgoInconti \
     FROM RGNOBS where Epoch = $code", 'get ++ region obs data');

if (length $rgnobs > 0)
   {
     if (length $cold > 0) { $cold = '*'; };
     print PRINPT "\n Regional obs: $cold, $rgoPressure, $rgoSite, $rgoMotor, $r
     print PRINPT "$rgoMobile, $rgoHeadache, $rgoSitePain, $rgoBackache, $rgoInc
     print PRINPT "(P site M lvl mob H sitepain back inco)";
   };

# PCA ==============================================

my ($cold, $pca, $pcoTries, $pcoGood) = &GetSQL ($myODBC,
 "SELECT cold, pca, pcoTries, pcoGood FROM PCA \
   WHERE Epoch = $code", 'get pca tries/good');
if (length $pca > 0)
   {
     if (length $cold > 0) { $cold = '*'; };
     print PRINPT "\n PCA tries/good: $cold $pcoTries, $pcoGood";
   };
```

```
  # PCASETTINGS ====================================

  my ($cold, $pcasettings, $pseDose, $pseLockout,
      $pseLimitInterval, $pseDoseLimit) =
    &GetSQL ($myODBC, "SELECT cold, pcasettings, pseDose, pseLockout, \
     pseLimitInterval, pseDoseLimit FROM PCASETTINGS \
       WHERE Epoch = $code", 'get pcasettings');
    if (length $pcasettings > 0) # or SQLOK ! [also in the above]
       {
         if (length $cold > 0) { $cold = '*'; };
         print PRINPT "\n PCA settings: $pseDose, $pseLockout, ";
         print PRINPT "$pseLimitInterval, $pseDoseLimit";
         print PRINPT " (dose/lockout; timeout/dose limit)";
       };
  return $ERRCOUNT;

}
```

12.5 Widows and orphans

It is conceivable that we might render some rows cold but accidentally leave their dependencies 'warm'. This is a potential problem, as we will move the warm records to the PDA, slowing things down and taking up space; in addition it's possible that attempts to view such records on the PDA could result in difficult-to-track errors! We will call these inappropriately warm rows 'orphans'.

The converse problem is 'widows' — tables with cold dependencies! These will generally not be a problem, as on the PDA we always work backwards from leaf to root, but we will list the dependent, cold items anyway!

12.5.1 Widows

Let's look for widowed rows with cold dependencies. (The term 'widows' is inappropriate. We need a term for 'warm mothers with cold children'). It might be smarter to pull out all warm mothers with *only* cold children!

```
sub FindWidows
{ my ($myODBC);
     ($myODBC) = @_;

  my $ok = 1;
  my $tim = time(); # for unique name
  my $widname = "data/$tim" . 'Widows.txt';
  open WIDW, ">$widname" or $ok = 0;
  if (! $ok)
```

```
    { &Alert($MAINW, "Failed to open $widname");
      return;
    };

 my (@WIDOWS)=();

 # first, epochs:
 (@WIDOWS) = &SQLManySQL($myODBC,
   "SELECT DISTINCT PROCESS.process FROM EPOCH,PROCESS WHERE \
     EPOCH.Process = PROCESS.process \
       AND EPOCH.cold IS NOT NULL \
       AND PROCESS.cold IS NULL",
     'get warm processes with cold epochs');
  my $COUNT = 1+$#WIDOWS;
  my $COMMALIST = &CommaList(@WIDOWS);
  print WIDW "\n Widow PROCESSES/epoch: $COUNT ($COMMALIST)";

 #next STOPPROCs:
 (@WIDOWS) = &SQLManySQL($myODBC,
   "SELECT DISTINCT PROCESS.process FROM STOPPROC,PROCESS WHERE \
     STOPPROC.Process = PROCESS.process \
       AND STOPPROC.cold IS NOT NULL \
       AND PROCESS.cold IS NULL",
     'widow stopprocs');
  my $COUNT = 1+$#WIDOWS;
  my $COMMALIST = &CommaList(@WIDOWS);
  print WIDW "\n Widow PROCESSES/stopproc: $COUNT ($COMMALIST)";

#and RX:
 (@WIDOWS) = &SQLManySQL($myODBC,
   "SELECT PROCESS.process FROM RX,PROCESS WHERE \
     RX.Process = PROCESS.process \
       AND RX.cold IS NOT NULL \
       AND PROCESS.cold IS NULL",
     'widow rx');
  my $COUNT = 1+$#WIDOWS;
  my $COMMALIST = &CommaList(@WIDOWS);
  print WIDW "\n Widow PROCESSES/rx: $COUNT ($COMMALIST)";

 # next, examine ALL tables dependent on EPOCH:
 my (@ETABLES) = ('PERSDATA', 'MEDSCORE', 'SURGTYPEOB', 'NONEVENT', 'MEASURE',
                  'PAINSCORE', 'RXOBS', 'INFUSIONOBS', 'PCA', 'PCASETTINGS',
                  'RGNOBS', 'COMMENT', 'ISPROBLEM', 'BADOBS');
 my $t;
 foreach $t (@ETABLES)
   { SeeWidows ($myODBC, $t);
   };
```

```
  # end off:
  close WIDW;
  &Alert($MAINW, "Data written to: $widname");

}

sub SeeWidows
{ my ($myODBC, $t);
     ($myODBC, $t) = @_;

  my (@WIDOWS) = &SQLManySQL($myODBC,
    "SELECT DISTINCT EPOCH.epoch FROM EPOCH,$t WHERE \
      $t.Epoch = EPOCH.epoch \
        AND $t.cold IS NOT NULL \
        AND EPOCH.cold IS NULL",
      "find widow $t");
   my $COUNT = 1+$#WIDOWS;
   my $COMMALIST = &CommaList(@WIDOWS);
   print WIDW "\n Widow $t: $COUNT ($COMMALIST)";
}
```

12.5.2 Orphans

Here's a routine to look for and fix 'orphans'. It may take some time to run.

```
sub FindAndFixOrphans
{ my ($myODBC);
     ($myODBC) = @_;

  my $ok = 1;
  my $tim = time(); # for unique name
  my $orname = "data/$tim" . 'Orphan.txt';
  open ORPH, ">$orname" or $ok = 0;
  if (! $ok)
     { &Alert($MAINW, "Failed to open $orname");
       return;
     };

  my (@ORPHANS)=();

  # here we might test for orphan _processes_ (not attached to a warm person)!
  (@ORPHANS) = &SQLManySQL($myODBC,
    "SELECT process FROM PROCESS,PERSON WHERE \
      PROCESS.Person = PERSON.person \
        AND PERSON.cold IS NOT NULL \
        AND PROCESS.cold IS NULL",
      'get warm procs on cold person');
   my $COUNT = 1+$#ORPHANS;
```

```
  my $COMMALIST = &CommaList(@ORPHANS);
  print ORPH "\n Orphan processes: $COUNT ($COMMALIST)";
  # (do not at this stage attempt to fix)


 # first, epochs:
 (@ORPHANS) = &SQLManySQL($myODBC,
   "SELECT epoch FROM EPOCH,PROCESS WHERE \
     EPOCH.Process = PROCESS.process \
       AND EPOCH.cold IS NULL \
       AND PROCESS.cold IS NOT NULL",
     'get orphan epochs');
  my $COUNT = 1+$#ORPHANS;
  my $COMMALIST = &CommaList(@ORPHANS);
  print ORPH "\n Orphan EPOCHS: $COUNT ($COMMALIST)";

  if ($COUNT > 0)
     { &DoSQL($myODBC,
       "UPDATE EPOCH set cold = 111 WHERE epoch IN ($COMMALIST)",
       'fix orphan EPOCHs');
       print ORPH ' <-fixed';
     };

 #next STOPPROCs:

 (@ORPHANS) = &SQLManySQL($myODBC,
   "SELECT stopproc FROM STOPPROC,PROCESS WHERE \
     STOPPROC.Process = PROCESS.process \
       AND STOPPROC.cold IS NULL \
       AND PROCESS.cold IS NOT NULL",
     'orphan stopprocs');
  my $COUNT = 1+$#ORPHANS;
  my $COMMALIST = &CommaList(@ORPHANS);
  print ORPH "\n Orphan STOPPROCS: $COUNT ($COMMALIST)";

  if ($COUNT > 0)
     { &DoSQL($myODBC,
         "UPDATE STOPPROC set cold = 111 WHERE stopproc IN ($COMMALIST)",
        'fix orphan STOPPROCs');
       print ORPH ' <-fixed';
     };

#and RX:
 (@ORPHANS) = &SQLManySQL($myODBC,
   "SELECT rx FROM RX,PROCESS WHERE \
     RX.Process = PROCESS.process \
       AND RX.cold IS NULL \
       AND PROCESS.cold IS NOT NULL",
```

```
      'orphan rx');
   my $COUNT = 1+$#ORPHANS;
   my $COMMALIST = &CommaList(@ORPHANS);
   print ORPH "\n Orphan RX: $COUNT ($COMMALIST)";
   if ($COUNT > 0)
      { &DoSQL($myODBC,
        "UPDATE RX set cold = 111 WHERE rx IN ($COMMALIST)",
        'fix orphan RX entries');
        print ORPH ' <-fixed';
      };

  # next, update ALL tables dependent on EPOCH:
  # [we might automatically fetch these from the x-tables]
  my (@ETABLES) = ('PERSDATA', 'MEDSCORE', 'SURGTYPEOB', 'NONEVENT', 'MEASURE',
                   'PAINSCORE', 'RXOBS', 'INFUSIONOBS', 'PCA', 'PCASETTINGS',
                   'RGNOBS', 'COMMENT', 'ISPROBLEM', 'BADOBS');
  my $t;
  foreach $t (@ETABLES)
    { FixEpoTable ($myODBC, $t);
    };

  # end off:
  close ORPH;
  &Commit($myODBC);
 # &Alert($MAINW, "Data written to: $orname");

}
```

Here's the subsidiary routine which fixes t, a given table that depends on EPOCH:

```
sub FixEpoTable
{ my ($myODBC, $t);
     ($myODBC, $t) = @_;

  my (@ORPHANS) = &SQLManySQL($myODBC,
    "SELECT $t FROM EPOCH,$t WHERE \
      $t.Epoch = EPOCH.epoch \
        AND $t.cold IS NULL \
        AND EPOCH.cold IS NOT NULL",
      "find orphan $t");
    # (here might check SQLOK for success)
   my $COUNT = 1+$#ORPHANS;
   if ($COUNT < 1)
      { return;
      };

   my $COMMALIST = &CommaList(@ORPHANS);
```

```
   print ORPH "\n Orphan $t: $COUNT ($COMMALIST)";
   &DoSQL($myODBC,
    "UPDATE $t set cold = 111 WHERE $t IN ($COMMALIST)",
    "fix orphan $t");
   print ORPH ’ <-fixed’;
}
```

13 Backup & Restore

First let's look at a simple invocation of an external backup routine. We determine the day of the week and then submit this to an external (here, DOS batch) file.

```
sub SimpleBackup
{
  my ($sec, $min, $hour,
      $mday, $mon, $year,
      $wday, $yday, $isdst) = CORE::localtime(time);

  my @wday = ('Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'S

  my $d = @wday[$wday];

#  &FindAndFixOrphans( $myODBC);

  my $sbat = $CONST{'SIMPLEBACKUP'};
  $sbat = "$sbat $d";
  $_ = '$sbat';  # backticks to invoke
  # &Alert($MAINW, "Debug: the command is <$sbat>");

  &Alert($MAINW, "Database backed up ($d)");
  # here might return outcome!

}
```

In this section we also create rarely used routines which will allow us to completely backup and restore the database. These should be rarely used, because the SQL database you use should have the facility to backup and restore *everything* — if it doesn't, get another database, or write your own backup routines! However, for completeness' sake, we provide these cumbersome routines.

13.1 Backup

For backup we've chosen to write the whole database as SQL! This approach is verbose but means that re-creation of the database is trivial[105] in any compliant SQL database. We do not need to create the actual tables using SQL, as this has already been specified in the SQL used to make the database. All we need to do is write SQL INSERT statements. The name of the file is the current timestamp followed by the suffix '.SQL'.

We've modelled this section closely on the PDB-writing routines.

[105] And slow. And enormous!

13.1.1 BackupAll

As for PDB creation, we first COMMIT and then query the meta-table xTABLE to create the INSERT statement prototype. We write the biiig SQL file to the *data* subdirectory of the current directory which should be *painform*.

```
sub BackupAll
{ my ($myODBC);
     ($myODBC) =@_;

  my($ok);
  $ok = &Confirm($ADMINMENU,
    "Backup database? \n Are you ABSOLUTELY certain? \n \
  ENSURE nobody else is working on the database \n \
 [Long delay, and will overwrite\n any other backup for today]");
  if (! $ok)
     { &Alert($ADMINMENU, "Nothing done");
       return;
     };
  &Commit($myODBC);  # force COMMIT as left pane not yet committed! [??]
```

We also create a series of deletion statements *in reverse* to delete from current tables!

```
  my ($delstring);
  my $bakfile;

  $_ = &GetLocalTime();
  /(\d+-\d+-\d+)/;
  $bakfile="DATA/$1.SQL";
  if (! &ValidatePath("data") )
   { &Alert($MAINW, "Error: directory 'data' does not exist!");
     return;
   };

  open BAKFILE, ">$bakfile" or die
     "*CRASH* Could not open BACKUP $bakfile :$!\n";

  print "\n Backing up to $bakfile";
  $delstring =  "";
  print BAKFILE "-- Backup SQL file: $bakfile \n\n";

  my(@tablenames);
  @tablenames = &SQLManySQL ($myODBC,
     "SELECT xTaKey, xTaName FROM xTABLE ORDER BY xTaKey",
     "fetch table codes/names");
  my ($tcode, $tname);
  my $ltime; # epoch time (raw)
```

We fetch the table names *in order of creation* so that the INSERT statements won't cause key errors.

```
  while ($#tablenames > 0) # [pairs so > 0 is ok hmm]
    { $tcode = shift (@tablenames);
      $tname = shift (@tablenames);
      if ( ($tname !~ /^x/)    # provided not metatables (xTable ...)
         &&( $tname ne 'FUN')  #      and not function table
         &&( $tname ne 'ITEM') #      etc...
         &&( $tname ne 'ICOLTABLE') #
         &&( $tname ne 'MENUITEMS') #
         )
         { &BackupOneTable($tname, $tcode);
           $delstring = "DELETE FROM $tname ; \n $delstring";
         };
    };
  print BAKFILE "\n* --- the last line\n";
  close BAKFILE;
```

We still need to create an accessory *...DEL.SQL* file which will be used to *de-populate* menus prior to the above insertions. This is because we must delete any contents of tables (other than meta-tables and a few other select tables such as those specifying menu details, and the FUN table) prior to restoring the backup. We assume that such restoration will rarely if ever be needed, and that other smarter backups of the whole database using more conventional means will be used preferentially.

```
  $bakfile =~ /(.+)\.SQL/;
  $bakfile="$1DEL.SQL";
  open BAKFILE, ">$bakfile" or die
     "*CRASH* Could not open DELETION FILE $bakfile :$!\n";
  print BAKFILE "-- Deletion script for $bakfile \n\n $delstring";
  print BAKFILE "\n* --- the last line\n";
  close BAKFILE;
  &Alert($ADMINMENU, "Backup completed (includes deletion file $bakfile)");
}
```

13.1.2 BackupOneTable

We insert column values along the lines of:

```
INSERT INTO tablename (list, of, columns) VALUES
  (list, of, values),
  ...
  (list, of, values);
```

Here goes:

```
sub BackupOneTable
{ my ($tname, $tcode);
     ($tname, $tcode) = @_;

  my ($colcount, $colnames, @coltypes);
     ($colcount, $colnames, @coltypes) =
        &FetchColumnData($myODBC, $tname, $tcode);
```

We next fetch all column values and write relevant SQL lines . . .

```
  my (@myrecs);
  @myrecs = &FetchTableData ($myODBC, $tname, $colnames,
                             $colcount, @coltypes);
  my ($rec);
  my ($s);
  $s = "";

  if ($#myrecs > -1) # if any rows
     { print BAKFILE "\n INSERT INTO $tname ($colnames) VALUES \n";
       foreach $rec (@myrecs) #
         {
           print BAKFILE "$s($rec)" ;
           $s = ",\n";
           print ".";
         };
       print BAKFILE ";\n\n" ;  # end off
       return 1; # success
     };
  return 0; # nothing done
}
```

The following sections contain subsidiary routines.

13.1.3 FetchColumnData

We first fetch a list of meta-table keys for all columns into colkeys.

```
sub FetchColumnData
{
  my ($myODBC, $tname, $tcode);
     ($myODBC, $tname, $tcode)=@_;

  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($COLNAMECOMMAS, @COLTYPECOMMAS);
  $COLNAMECOMMAS = '';  # string (list) of names
  @COLTYPECOMMAS = ();

  my ($ck, $cname, $ctype);
  foreach $ck (@colkeys)
    {
      ($cname, $ctype) = &FetchColInfo($myODBC, $ck);
      $COLNAMECOMMAS = $COLNAMECOMMAS . $cname . ',';
      push (@COLTYPECOMMAS, $ctype);
    };

  chop($COLNAMECOMMAS);
  return ($colcount, $COLNAMECOMMAS, @COLTYPECOMMAS);
}
```

13.1.4 FetchTableData

Given a database connection, a comma-delimited list of column names and an array of data types, return an array of SQL rows, each row being a string make up of *formatted*[106] data. Commas separate each datum in an SQL row. Only invoked by BackupOneTable.

```
sub FetchTableData
{
  my ($myODBC, $tname, $colnames, $colcount, @coltypes);
     ($myODBC, $tname, $colnames, $colcount, @coltypes) =@_;

  my @alldata;

  @alldata = &SQLManySQL ($myODBC,
     "SELECT $colnames FROM $tname",
     "get all data for one table");

  my @onerow;
  my @formatted = ();
  my $or;
```

[106] For example, a date will be formatted as *DATE '2006-06-30'*.

```
while ($#alldata > -1)
  {
    @onerow = splice (@alldata, 0, $colcount);
    # here format the data [TO FIX]

    $or = '';
    my ($e, $f, $i);

    $i = 0;
    while ($i < $colcount)
      { $_ = @onerow[$i];
        $f = @coltypes[$i];
        if (length $_ < 1)
           { $e = "NULL";
           } else  # here format item according to type
           {   if ($f eq 'V')
                 { s/'/''/g; # fix single quotes
                   $e = "'$_'";
                 }
               elsif ($f eq 'I')
                 { $e = $_;
                 }
               elsif ($f eq 'N')
                 { $e = $_;
                 }
               elsif ($f eq 'D')
                 { $e = "DATE '$_'";
                 }
               elsif ($f eq 'T')
                 { $e = "TIME '$_'";
                 }
               elsif ($f eq 'S')
                 { $e = "TIMESTAMP '$_'";
                 }
               elsif ($f eq 'F')
                 { $e = $_;
                 }
               else
                 { print "\n Bad datum type: $f ($_)";
                 };
           };
        $or = "$or$e,";
        $i ++;
      };
    chop($or); # remove terminal comma.
    push (@formatted, $or);
  };
```

```
  return(@formatted);
}
```

13.1.5 FetchColInfo

This routine is analogous to MakeColmDescriptor below, but far simpler. It accepts an ODBC connection and the key of the column (in the meta-tables), and returns information about a column — its name and its type.

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

 my ($ctype, $colname);
 ($ctype, $colname) = &GetSQL ($myODBC,
            "SELECT xCoType, xCoName FROM xCOLUMN WHERE xCoKey = $ck",
            "get column type & name");
  return ($colname, $ctype);
}
```

13.2 Restore

This routine should rarely if ever be needed. It will restore the current database by running an SQL backup. Because of the potential lethality of such a routine, we will only allow it to be run if the user has by other means completely deleted the current database. A user-initiated, recent backup copy is assumed to have been made using the above routines — such backup does *not* happen automatically.

The sequence of events is:

1. User has previously created and stored a backup safely;
2. User independently deletes database (for whatever reason);
3. Do not run if database exists;
4. Otherwise, create database as usual (MakeDBandMenus);
5. Then run the deletion file;
6. Finally, run the full SQL re-installation.

```
sub RestoreAll
{ my ($myODBC);
     ($myODBC) =@_;

  $ISDB = -1;
  $ISDB = &IsDbMade($myODBC); # again check if pain db made
  if ($ISDB)
     { &Alert ($ADMINMENU, "You must first DELETE the database \n \
  (CAUTION. Ensure you have a recent, valid SQL backup)!");
       return;
     };

  my($ok);
  $ok = &Confirm($ADMINMENU,
    "Restore database? \n Are you ABSOLUTELY certain? \n \
  This might take ages. \n \
 [It's better to get your database administrator to restore things!]");
  if (! $ok)
     { &Alert($ADMINMENU, "Nothing done");
       return;
     };

  &MakeDBandMenus($myODBC); # re-create menus

  # next, read in the file name
  my $refile = "";
```

```
  while (length $refile < 1)
    { $_ = &GetLocalTime();
      /(\d+-\d+-\d+)/;
      $refile="$1";
      $refile = &Ask($ADMINMENU, "Please enter file name WITHOUT .SQL suffix", $re
    };
  # here might validate existence of files..

  # then run the deletion file to clear all tables
  $ok = &BatchSQL ("$refile" . "DEL" , $myODBC);
  &Commit($myODBC);

  # finally re-populate these tables.
  $ok = &BatchSQL ("$refile" , $myODBC);

  &Commit($myODBC);
  print "\n\n DONE!";
  &Alert($ADMINMENU, "Restored (apparently!)");
}
```

14 CSV initialisation

It's very desirable to be able to load spreadsheet data into our database. We will therefore take common 'CSV' (comma-delimited) files, and import them into our database. The main use of this facility will be in initialisation of the whole database, where we will import the following files:

1. ward.csv
2. room.csv
3. bed.csv
4. proctype.csv
5. person.csv
6. process.csv, epoch.csv, and persdata.csv
7. drug.csv
8. drugusage.csv
9. surgtype.csv
10. surgsite.csv

Here's the routine to read in all of the above CSV files:

```
sub ReadAllCSV
{ my ($myODBC);
     ($myODBC) = @_;

  my $errs = 0;

   $errs += ReadCsv ($myODBC, "ward");
   $errs += ReadCsv ($myODBC, "room");
   $errs += ReadCsv ($myODBC, "bed");
   $errs += ReadCsv ($myODBC, "proctype");
   $errs += ReadCsv ($myODBC, "person");

   $errs += ReadCsv ($myODBC, "process");
   # require processes for the following two:
   $errs += ReadCsv ($myODBC, "epoch");
   $errs += ReadCsv ($myODBC, "persdata");

   $errs += ReadCsv ($myODBC, "drug");
```

```
   $errs += ReadCsv ($myODBC, "drugusage");
   $errs += ReadCsv ($myODBC, "surgtype");

   if ($errs < 1) # on success
      { &Commit($myODBC);
          Alert( $ADMINMENU, "\n CSV data successfully imported.");
      } else
      { &Alert($MAINW, "SERIOUS ERRORS=$errs. \n \
   CSV read(s) failed. COMMIT avoided! See EDLOG.LOG");
      };
}
```

14.1 CSV data file limitations

Having the above initialisation files permits great flexibility in the initial configuration of the database, but certain caveats are in order:

1. Ward database ID (key) numbers are limited to the range 1–99;
2. A particular room in a ward must have a key value in the range $b * 100 - 99 + b * 100$. In other words, the room key can be derived from the ward key using an integer divide by 100. This is not an onerous requirement, as it's difficult to conceive of a ward with more than 100 rooms!
3. A given bed must similarly have an id which is the ward id multiplied by 100, plus 0–99.
4. Initial person import should be limited to staff members, with the appropriate fields filled in.
5. If data are missing, relevant replacements should be present, NULL or zero.
6. The DRUG table is fairly flexible but has some constraints for key values (see documentation below and in *AnalgesiaDB.tex*).
7. If you fiddle with the PROCTYPE table, then extensive script recoding will be required to stop the scripting of the menus from breaking.
8. SURGTYPE is more tractable.

14.2 CSV conventions

The formatting of our CSV files is fairly restrictive. Please adhere to the following conventions:

1. The first line specifies the column names. These must be *identical* to the original database column names, or data import *will* fail! You will have to allocate a *unique key* value to each row, and no identical key value may be present in the database.

2. Subsequent lines contain comma-delimited data, which can be exported as a CSV directly from e.g. Excel.

3. There's another frill to the first line. We specify the column format in this line as well. Simple text variables are in quotes (only in the top column, not in subsequent lines of data). A DATE variable called 'fred' will be specified as follows:

   ```
   DATE 'fred',
   ```

   Likewise for TIMESTAMP and TIME variables. Only use single quotes, not backticks or other modifications.[107]

4. Input continues until end of file (EOF).

5. Blank lines are *ignored*.

6. Lines beginning with two percentage signs (%%) are ignored.

7. A terminal comma at the end of a line is ignored

8. Line feeds should not be used within items, but if they are necessary render them thus: `\n`

9. We unfortunately use CSV formatting which is vaguely Excel-style. We don't usually encase things in quote marks, although these are permitted. If you *must* use a comma within a field preced it by a backslash: `\,`

10. Use of the backtick character (`) within a field is disallowed.

11. Leading and trailing spaces in CSV data fields are suppressed, so if these are required, then the double quote character (") must be used at the start and end of the field to prevent blank suppression. The double quote itself will be ignored.

[107] We must modify things so that a single quote in text is duplicated!

The standard CSV files are included as an appendix (Section 18.7). Here's our code to read in one of our CSV files. All CSV files are located in the *csv* subdirectory of the *painform* directory.

In the ReadCsv routine, we take the given filename (without the .CSV suffix) and write to the data table of the *same name*! We supply an open database handle for the purpose.

```
sub ReadCsv
{
  my ($myODBC, $name);
     ($myODBC, $name) = @_;
  my ($errcount);
  $errcount = 0;

  my($fail);
  $fail = 0;
  my ($CSVFILE);
  $CSVFILE = "csv/$name.csv";

  open CSVFILE, $CSVFILE or $fail = 1;
  if ($fail)
     { &Alert($MAINW, "Failed to open CSV file: $CSVFILE");
       return 0; # fail
     };

  my ($ishead);
  $ishead = 1;
  my (@headarray);
  my (@linearray);
  my ($headlen);

  while( <CSVFILE> )  # until EOF
    {
      chomp;
      if ( /(.*), */ ) # if terminal comma and/or terminal blanks(!)
         { $_ = $1;
         };
      if (  ((length $_) > 1) # minimum length is 2
          &&( ! /^%%/ )
         )
         { if ($ishead)
              { $ishead = 0;
                @headarray = split /,/;
                $headlen = $#headarray; # get size
              } else
              { s/\‘//g;    # get rid of backticks! (hmm)
                s/\\,/\‘/g; # replace \, with backtick ?!
                @linearray = split /,/;
```

```
                if ($#linearray != $headlen)
                   { $errcount ++;
                     print LOGFILE "\n Error($CSVFILE):<$_>";
                   } else
                   { if (! &FormatCsvLine($myODBC, $name, $headlen,
                            \@headarray, \@linearray)) # pass by reference!
                        { $errcount ++;
                          print "?";
                        } else
                        { print ".";
                        };
                   };
              };
         };

    };
  close CSVFILE;

  if ($errcount > 0)
     { &Alert($MAINW, "There were $errcount insertion errors in parsing $CSVFILE")
       return $errcount;
     };
  return 0; # oddly enough, success!
}
```

Here's the routine to parse a single line and write the result to the database. The two arrays are passed by reference.

($myODBC$,name, headarray, linearray))

```
sub FormatCsvLine
{
  my ($myODBC, $name, $headlen, $headArr, $lineArr);
  ($myODBC, $name, $headlen, $headArr, $lineArr) = @_;

  my $i = 0;
  my ($col, $val);
  my ($left, $right);
  $left = "";
  $right = "";

  while ($i <= $headlen)
    {
      ($col, $val) = &FormatOneItem ($headArr->[$i], $lineArr->[$i]);
      $left = "$left $col,";
      $right= "$right $val,";
      $i ++;
    };
```

```
  chop ($left);
  chop ($right); # get rid of terminal commas

  my($sqstmt) = "INSERT INTO $name($left) VALUES ($right)";
#  print LOGFILE "\n DEBUG: $sqstmt";

  if ( &Do2SQL($myODBC, $sqstmt, "submit CSV line") < 0)
     { return 0; # fail
     };
  return 1; # ok
}
```

Finally, we format a single item. If the column name is quoted, then we quote the result (in single quotes). If the column is prefixed with TIME, DATE or TIMESTAMP, then we prefix the result with this, unless of course the result is of null length or NULL, in which case we leave the NULL as is. FormatOneItem is *only* invoked by FormatCsvLine; some of the functionality is duplicated elsewhere (search for `s/''/''/` to see this).

```
sub FormatOneItem
{
  my ($col, $itm);
     ($col, $itm) = @_;

  $itm =~ s/\‘/,/g; # backtick replaced by comma
  $itm =~ s/’/’’/g; # duplicate quotes eg O’Connor!
  $itm =~ s/\\n/\n/g; # LF replaces \n
  if ( $itm =~ /"(.*)"/ ) # if encasing quotes
     { $itm = $1;         #  remove them!
     } else
     { # if not encasing quotes, then trim blanks!!
       $itm =~ / *(.*) */;
       $itm = $1;
     };

  if ($col =~ / *’(.+)’ */)
     { $col = $1;
       if (  ($itm !~ /NULL/i)
           &&(length $itm > 0)
          ) { $itm = "’$itm’";
            };
     }
  elsif ($col =~ / *TIMESTAMP ’(.+)’ */i )
     { $col = $1;
       if (  ($itm !~ /NULL/i)
           &&(length $itm > 0)
          ) { $itm = "TIMESTAMP ’$itm’"; };
```

```
    }
  elsif ($col =~ / *TIME '(.+)' */i )
    { $col = $1;
      if (  ($itm !~ /NULL/i)
          &&(length $itm > 0)
         ) {$itm = "TIME '$itm'"; };
    }
  elsif ($col =~ / *DATE '(.+)' */i )
    { $col = $1;
      if (  ($itm !~ /NULL/i)
          &&(length $itm > 0)
         ) {$itm = "DATE '$itm'"; };
    };
  # default is to leave unchanged.

  if (length $itm < 1)
    { $itm = "NULL";
    };
  return ($col, $itm);
}
```

We return *two* values, the first being the undecorated column name, the last the formatted value.

15 Importing from an external database

Here we obtain patient information from an external anaesthetic database. The database we use is that of SaferSleep (aka the Integrated Drug Administration System, IDAS), so our ODBC calls to this database won't of course work on other proprietary databases. Because the IDAS database is proprietary, we refer to constant SQL expressions which are stored in the file CONSTANTS.CONST. These SQL statements are (unfortunately) not in the public domain and will need to be replaced with SQL appropriate to the database you are using.

Our plan of attack is as follows:

1. Identify all relevant anaesthetics (in our example, on all patients from Level 8 anaesthesia who have either an epidural or are on PCA).

2. Store each such anaesthetic. This involves:

   (a) Identifying whether the patient exists in our database, based on a search for the NHI. If the NHI doesn't exist in our database, then create a unique entry for the patient (PERSON), recording the Date of Birth along the lines of ...

   ```
   INSERT INTO PERSON (person, pBorn, pMade, pStatus)
               VALUES (10001, TIMESTAMP '1950-01-01 00:00:00',
                       TIMESTAMP '2007-01-01 00:00:00', 1);
   ```

   (b) We will also need a data observation PROCESS (code 1) entry for the patient, which we can then use to record an observation of the NHI, gender, forename, and surname.

   (c) We will then require a separate process for the operation (process code 500), recording the start time[108]. We will attach the type of operation.

   (d) If the destination ward is documented, we will also record this in the BADOBS table; otherwise we will place the patient in the 'generic' ward 1.

[108] At present, the end time is rather problematic in IDAS!

(e) We also record weight and ASA rating . . .

(f) Epidural presence and details (as well as other regionals) . . .

(g) . . . and PCA presence and details.

This last-mentioned item is now obtainable from IDAS.

3. Remember to COMMIT the SQL!

15.1 Identifying recent anaesthetics

We open the external database, submit the relevant string(s), and obtain an array of anaesthetic IDs. We then query this same database using each anaesthetic ID to obtain information about the patient and anaesthetic. We write data to our database, and commit the data.

If the $CANSKIP value is 1, then we check for a recent fetch (within an hour) and skip the fetch if this is the case; with a value of 0, we fetch regardless.

```
sub ExtDbImport
{
  my ($myODBC, $CANSKIP);
  ($myODBC, $CANSKIP) = @_;

  my ($externalODBC, $fail);
  $fail = 0;
  $externalODBC = new Win32::ODBC($CONST{'XDBCONNECT'}) or $fail = 1;
  if ($fail)
     {
      $fail = Win32::ODBC::Error();
      &Alert($MAINW, "Connection failed. Message was '$fail'");
      return; #exit
     };

  my (@anaesthetics);
  my ($getanaestring, $firsttimestamp, $secondtimestamp, $ortemplate);
  my ($today, $target, $jd, $yyyy, $mm, $dd);
```

Previously we were constrained into gettting cases from the previous day.[109] With the help of our IS chaps we can now access the Database in ‘real time’, so we record the timestamp for the previous access and use this in our retrieval:

```
$today =  &GetLocalTime();
$secondtimestamp = $today;

my $lfok = 1;
open LASTFETCH, 'LASTFETCH.DATA' or $lfok = 0;

if ($lfok)
   { $target = <LASTFETCH>;  # YYYY-MM-DD HH:MM:SS
     if ($target !~ /(\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2})/)
        { $lfok = 0; # bad format: force failure
          &Alert($MAINW,
"Woops. Bad timestamp YYYY-MM-DD HH:MM:SS <$target>.");
        } else
        { $target = $1;
          $firsttimestamp = $target;
          # here might check that time is not > say 2 days [!?]

          # v 0.95 (18/3/2008)
          if ($CANSKIP) # the following section checks for
                        # lastfetch within the past hour:
             { # at present we do NOT repeat the import if this happened.
               # things are rather tricky as we have to ADD 12 hours to the
               # time in $target, as explained later.
               my ($recentjd, $nowjd, $deltah);
               $target =~ /(\d{4})-(\d{2})-(\d{2}) (\d{2}):(\d{2}):(\d{2})/;
               $recentjd = &Julian($1,$2,$3,$4,$5,$6,0);
               $recentjd += 0.5; # fix up 'compensation' by adding 12 hr!
               $today =~ /(\d{4})-(\d{2})-(\d{2}) (\d{2}):(\d{2}):(\d{2})/;
               $nowjd = &Julian($1,$2,$3,$4,$5,$6,0);
               $deltah = 24*($nowjd - $recentjd);
               if ($deltah < 1) # if under an hour
                  {
                    print LOGFILE "\n Ext db fetch too recent! $today ; $target
                    close LASTFETCH;
                    return;
                  };
               };
        };
     close LASTFETCH;
   };

if (! $lfok) # if failed
```

[109] In fact, until 7pm of the previous day!

```
    { &Alert($MAINW,
"Failed to load fetch time (LASTFETCH.DATA)!");
      $today =~ /(\d{4})-(\d{2})-(\d{2})/;
      $jd = &Julian($1, $2, $3, 0, 0, 0, 0); # julian day
      $jd --;
      ($yyyy, $mm, $dd) = &Gregorian($jd);
      $target = &FixDate($yyyy, $mm, $dd);
      $firsttimestamp = "$target 00:00:00";
    };

 # ---------------------------------------------------------
 # the following is debugging, allows wider search interval:
 # $jd -= 3;
 # ($yyyy, $mm, $dd) = &Gregorian($jd);
 # $firsttimestamp = &FixDate($yyyy, $mm, $dd);
 # $firsttimestamp = "$firsttimestamp 00:00:00'";
 # end debugging
 # ---------------------------------------------------------

 my $tim = time(); # for unique name
 my $importlog = 'log/IMPORT' . $tim . '.LOG';
     # log most recent imports/rejections
 open IMPORTLOG, ">$importlog" or die
    "*CRASH* Could not open LOG $importlog :$!\n";
 print IMPORTLOG
   "\n IMPORT LOG BETWEEN '$firsttimestamp' AND '$secondtimestamp'\n";

 my $epiduralstring = $CONST{'XDBQUERYEPIDURAL'};
 my $pcastring      = $CONST{'XDBQUERYPCA'};
 my $ortemplateA = "'A8%'";
 my $ortemplateB = "'A4%'";

 $epiduralstring =~ s/\$STARTTIME/'$firsttimestamp'/;
 $epiduralstring =~ s/\$ENDTIME/'$secondtimestamp'/;
 $epiduralstring =~ s/\$ORA/$ortemplateA/;
 $epiduralstring =~ s/\$ORB/$ortemplateB/;


 $pcastring =~ s/\$STARTTIME/'$firsttimestamp'/;
 $pcastring =~ s/\$ENDTIME/'$secondtimestamp'/;
 $pcastring =~ s/\$ORA/$ortemplateA/;
 $pcastring =~ s/\$ORB/$ortemplateB/;

 my (@epidurals) = &SQLManySQL($externalODBC, $epiduralstring,
                    "get epidurals");
 my (@pca) = &SQLManySQL($externalODBC, $pcastring,
                    "get pca instances");

 print IMPORTLOG "\n Epidural import list <@epidurals>";
```

```
print IMPORTLOG "\n PCA import list <@pca>";

my (@anaesthetics) = (@epidurals, @pca); # join the two
 @anaesthetics = sort { $a <=> $b } @anaesthetics; # numeric sort

print IMPORTLOG "\n TOTAL import list <@anaesthetics>";
```

We sort the anaesthetic ID numbers in numeric order, so that we can easily eliminate duplicates! We assume that the generated list is one of all 'relevant' anaesthetic IDs within the external database. We next process each anaesthetic in turn.

Epidurals can be derived both from the IDAS tables EpiduralAnalgesia and (oddly enough) from the RegionalBlock table, an uncomfortable denormalisation.[110] The EpiduralAnalgesia table is now active, so it's the one we'll use, at least for now. (Later on we might write routines to cross-check against the other table!)

Patients with PCA might be derived from the IDAS PCA table, or another external database (that in the 24 hour centre). Ideally we should have our own 'data warehouse', and repeatedly update this warehouse with data from a multiplicity of databases, and then query the *warehouse* when we need data for the painform database.[111]

```
my ($an, $isregnl, $ispca, $useme, $descrip, $surgtype, $surgsite);
my $importcount = 0;
my $rejects = 0;
$isregnl = 0;   #
$ispca = 0;     # default NO

my $lastan;
$lastan = 0;

foreach $an (@anaesthetics)
  {
    print IMPORTLOG "\n\n PROCESSING ANAESTHETIC, Id = $an";

    next if ($an == $lastan); # ignore if duplicate!
    $lastan = $an;
    $isregnl = 0;   #
    if ( &InArray($an, @epidurals) )
```

[110] Getting them from the RegionalBlock table we will have to remove rubbish like toe blocks, finger blocks, penile blocks etc. We really want SPINAL, EPIDURAL, CSE, and perhaps other regional blocks like intercostal blocks and femoral nerve catheters.

[111] Such an approach should minimise the likelihood that we will interfere with the other databases in an unpredictable fashion, as well as allowing us to integrate information from may different databases without keeping them open for long periods of time.

```
        { $isregnl = 110; # epidural alone, for now.
        };
      $ispca =  &InArray($an, @pca);

      my $descripstmt = $CONST{'XDBDESCRIP'};
      $descripstmt =~ s/\$ANAESTHETIC/$an/; # fill in anaesthetic
      my ($descrip) = &GetSQL($externalODBC, $descripstmt,
                        "get op descriptn");
      ($useme, $surgtype, $surgsite) = &IWantYou($descrip);
      # at present we ignore both $useme and $surgsite!
      # a value of zero in $surgtype is 'unspecified'
      print IMPORTLOG "\n Description: $descripstmt ($surgtype)";


    #  &Alert($MAINW, "Debug: accepted <$descrip>");
     my $impok;
        $impok = &ImportExternalAnaesthetic($myODBC, $externalODBC,
                $an, $isregnl, $ispca, $descrip,
                $surgtype, $surgsite);
        if ($impok == 1)
             { $importcount ++;
             }
        elsif ($impok == 0)
             { # duplicate!
             }
        else { $rejects ++;
               &Alert($MAINW,
               "Failed to import anaesthetic (ID $an : <descrip>)");
                print IMPORTLOG "\n Failed import: <$an> $descrip";
             };
    };

  my $duplicates = 1+ $#anaesthetics - ($rejects + $importcount);
  &Alert($MAINW,
 "Import count was $importcount ($rejects rejected, $duplicates duplicate(s))");

  &Test_Fixup($myODBC); # fix up important duplicates [SEE RTN BELOW]

  &Commit($myODBC);
  $externalODBC->Close();
  close IMPORTLOG;
```

There is a problem here with recording the time that data were last fetched, arising from a defect in the current version of the SaferSleep database. The database currently leaves the AnaestheticFinish timestamp blank, so no end time is recorded. As data are only written to the server on completion of the anaesthetic, any query will miss a currently running anaesthetic. If we record the current time-

stamp as the last time, then we will miss these anaesthetics, and there is no way of recovering them short of looking back to a time before they were started! As a temporary hack, we subtract 12 hours from the last time, knowing that this will result in prompts about duplicate anaesthetics, and also knowing that if the anaesthetic lasted over 12 hours it will still be missed :-(The correct solution is to fix the SaferSleep database; failing this, we will ultimately have to keep a record of all anaesthetic IDs, and then re-interrogate the database, excluding duplicate ids!

```
  my ($halfbefore);
  my ($hh, $min, $sec);

       $secondtimestamp =~ /(\d{4})-(\d{2})-(\d{2}) (\d{2}):(\d{2}):(\d{2})/;
       $hh = $4;
       $min = $5;
       $sec = $6;
       $jd = &Julian($1, $2, $3, 0, 0, 0, 0); # julian day
       if ($hh < 12)
          { $hh += 24;
            $jd --;
          };
       $hh -= 12;
       $hh = &DoubleDigit($hh);
       ($yyyy, $mm, $dd) = &Gregorian($jd);
       $target = &FixDate($yyyy, $mm, $dd);
       $halfbefore = "$target $hh:$min:$sec"; # half day previously!

  my $wrok = 1;
  open LASTFETCH, ">LASTFETCH.DATA" or $wrok = 0;
  if (! $wrok)
     { &Alert($MAINW,
  "Oops! Could not store 'fetch timestamp' (LASTFETCH.DATA)");
     } else
     { print LASTFETCH $halfbefore; # see note above [hack]
       close LASTFETCH;
     };

  &SimpleBackup();  # added 2007-12-16.

}
```

Here's a minor routine which will principally fix up single quotes within strings, as this will otherwise mess up our INSERT statements:

```
sub FancyUp
{ ($_) = @_;
  s/'/''/g;
  return ($_); # redundant stmt.
}
```

Here's my cumbersome InArray (there must be a better way):

```
sub InArray
{
  my ($i, @a);
     ($i, @a) = @_;

  foreach (@a)
    {
     if ($_ == $i)
        { return 1;
        };
    };
  return 0;
}
```

15.2 Identifying an 'interesting' anesthetic

We look for interesting anaesthetics based on whether PCA or a relevant regional procedure was used, or perhaps on whether a major (painful) procedure was performed! This routine has been disabled.

```
sub RelevantAnaesthetic
{
  my ($externalODBC, $an);
     ($externalODBC, $an) = @_;

  my ($useme, $isregnl, $ispca, $surgtype, $surgsite);
  $useme = 0;
  $isregnl = 0;
  $ispca = 0;
  # we will also

  # we will set $useme if:
  #   it's an epidural
  #   pca is being used
  #   a suitably 'impressive' (painful) operation has been performed!

  # first check the regional table to see whether an epidural
  # (or other regional) has been used, setting the relevant flag
  my $regnstmt = $CONST{'XDBREGNL1'};
  $regnstmt =~ s/\$ANAESTHETIC/$an/; # fill in anaesthetic
#  print LOGFILE "\n DEBUG regional stmt: <$regnstmt>";
  my ($regid) = &GetSQL($externalODBC, $regnstmt, "identify regional");
     # hmm. what if more than one ??? [check this out]
     #
  if ($SQLOK) # if succeeded
```

```
    { $regnstmt = $CONST{'XDBREGNL2'};
      # [proliferation of block names is major liability in IDAS]
      $regnstmt =~ s/\$BLOCK/$regid/; # fill in block [BlockNameId]
      # &Alert($MAINW, "regnl query is <$regnstmt>");
      my ($regname) = &GetSQL($externalODBC, $regnstmt, "get block name");
      # &Alert($MAINW, "Debug: Block is <$regname>");
      $isregnl = &ChooseBlock($regname);
    };

  # repeat check for PCA [inactive for now]

    # [TO FILL IN HERE]



  # get procedure name, and scan for relevant procs.
  my $descripstmt = $CONST{'XDBDESCRIP'};
  $descripstmt =~ s/\$ANAESTHETIC/$an/; # fill in anaesthetic
  my ($descrip) = &GetSQL($externalODBC, $descripstmt, "get op descriptn");

  # we next check for valid/invalid strings, setting $useme (or not)
  # at the same time, we will set values for: $surgtype, $surgsite

  my $wanted = ($isregnl) || ($ispca);
  ($useme, $surgtype, $surgsite) = &IWantYou($descrip, $wanted);
  if ($useme < 0) # definite no:
     { $useme = 0;
      # &Alert($MAINW, "Debug: rejected <$descrip>");
     } else
     { if ( $wanted )
          { $useme = 1; # overrides user refusal of a particular op ??
          };
     };

  return ($descrip, $useme, $isregnl, $ispca, $surgtype, $surgsite);
}
```

In IDAS, the description of the operation can be up to 200 characters long.

The 'IWantYou' routine

The following routine might even be modified to return several values based on the type of operation detected. For now, we simply filter out 'unwanted operations', along the lines of:

```
upper(operationdescription) not like '%HYSTER%' AND
upper(operationdescription) not like '%OVARI%' AND
```

```
upper(operationdescription) not like '% TOE%' AND
upper(operationdescription) not like '%VULV%'
```

...but actually use Perl to do the selection, ie based on text in operationdescription. This routine will return zero for no, 1 for yes and -1 for a 'definite no'. At present, we have 'tuned' it for general (Level 8) surgery, excluding e.g. gynaecologic surgery which although done on the same level, is moved to Level 9.[112]

Here are the tables from *AnalgesiaDBpart1.tex*.

First, type of surgery:

Code	*Meaning*
0	unspecified
1	general
49	endoscopy
99	plastics
149	orthopaedic
199	neurosurgery
249	ophthalmic
299	dental
399	ORL
449	cardiac
499	thoracic
599	vascular
649	hepatobiliary
699	colorectal
799	urology/renal
899	gynaecology
999	obstetrics

Table 3: Coding of types of surgery

Next surgical site:

[112] These comments are specific for Auckland City Hospital.

Code	*Meaning*
0	unspecified
1	upper abdomen
2	lower abdomen
3	abdomen
119	lumbar region
149	pelvis
199	perineum
299	thorax
399	upper limb
499	lower limb
599	eye
699	head
799	neck

Table 4: Coding of surgical sites

The coding is somewhat clumsy, but not as clumsy as our methods of identification below, which would benefit from considerable refinement. We return a 'useme' flag, followed by the type and site, with coding as in the above tables.

```
sub IWantYou
{
 ($_) = @_;

# some rather arbitrary exclusions:
 if ( /HYSTER/i )
    { return (-1, 899, 2);
    };
 if ( /OVARI/i )
    { return (-1, 899, 2);
    };
 if ( /VULV/i )
    { return (-1, 899, 199);
    };
 if ( / TOE/i )
    { return (-1, 0, 499);
    };
 if ( / FINGER/i )
    { return (-1, 0, 399);
    };
 if ( /NEPHROSTOMY/i )
    { return (-1, 799, 119);
    };
```

```
if ( /MASTEC/i || /BREAST/i )
   { return (-1, 1, 299);
   };
if ( /DURAL/i )
   { return (-1, 199, 699);
   };
if ( /VENTRIC/i || /EVD/i )
   { return (-1, 199, 699);
   }; # woops. 'ventric' watch out for C-T.
if ( /CRANI/i )
   { return (-1, 199, 699);
   };
if ( /BURR/i )
   { return (-1, 199, 699);
   };
if ( /COIL/i || /CLIP/i ) # hmm.
   { return (-1, 199, 699);
   };
if ( /PITU/i || /HYPOPH/i )
   { return (-1, 199, 699);
   };
if ( /EYE/i || /BLEPH/i )
   { return (-1, 249, 599);
   };

# now for the positives:


if ( /AAA/i )
   { return (1, 599, 3);
   };
if ( /ANEURY/i )
   { if (/ABD/i || /FEM/i || /POPL/i )
        { return (1, 599, 3);
        };
     if (/THOR/i || /ARCH/i )
        { return (1, 499, 299);
        };
     return (0, 0, 0); # ?neuro/?vasc
   };
if ( /OSTOMY/i )
   {
     if ( /ureter/i )
        { return (1, 799, 3);
        };
     if ( /colo/i || /ileo/i )
        { return (1, 699, 3);
        };
```

```
     return (1, 0, 3);
   };
if ( /GALL/i || /CHOLEC/i || /BILI/i )
   { return (1, 649, 1);
   };
if ( /CONDUIT/i )
   { return (1, 799, 2);
   };
if ( /VESIC/i || /BLADDER/i || /STONE/i )
   { return (1, 799, 2);
   };
if ( /PANCRE/i )
   { return (1, 649, 1);
   };
if ( /LAPAROT/i )
   { return (1, 1, 3);
   };
if ( /COLECT/i )
   { return (1, 699, 3);
   };
if ( /ANTERIOR.+RESEC/i )
   { return (1, 699, 2);
   };
if ( /A.?P.+RESEC/i )
   { return (1, 699, 2);
   };
if ( /ABDOMINO.+RESEC/i )
   { return (1, 699, 2);
   };
if ( /TRANSPL/i )
   { if ( /liver/i )
        { return (1, 649, 1);
        };
     if ( /renal/i || /kidney/i )
        { return (1, 799, 3);
        };
     if ( /heart/i || /cardi/i )
        { return (1, 449, 299);
        };
     if ( /lung/i )
        { return (1, 499, 299);
        };
     if ( /pancr/i )
        { return (1, 649, 1);
        };
     return (1, 0, 0);
   };
if ( /NEPHRE/i || /KIDNEY/i )
```

```
   { return (1, 799, 3);
   };
if ( /HEPAT/i || /LIVER/i )
   { return (1, 649, 1);
   };
if ( /WHIPPLE/i )
   { return (1, 649, 1);
   };
if ( /GASTREC/i )
   { return (1, 1, 1);
   };
if ( /AMPUT/i )
   {
     return (1, 149, 0);
   };
if ( /AKA/i )
   { return (1, 149, 499);
   };
if ( /BKA/i )
   { return (1, 149, 499);
   };
if ( /HEPAT/i )
   { return (1, 649, 1);
   };
if ( /RADICAL/i )
   { return (1, 0, 0);
   };
if ( /JOINT.+REPL/i || /ARTHROP/i || /THJ/i || /TKJ/i )
   { return (1, 149, 0);
   };
if ( /ANASTOM/i)
   { return (1, 0, 0);
   };

# here might insert other 'diagnostics':
# e.g.:


if ( /CLOSURE/i || /REVERSAL/i || /REVIS/i )
   { return (0, 0, 0);
   };
if ( /EXPLOR/i )
   { return (0, 0, 0);
   };
if ( /DEBRID/i )
   { return (0, 0, 0);
   };
if ( /CATH/i )
```

```
   { return (0, 0, 0);
   };
if ( /FISTUL/i )
   { return (0, 0, 0);
   };
if ( /WIRE/i || /WIRI/i || /METAL/i || /NAIL/i || /PLATE/i || /SCREW/i || /ROD/i
   { return (0, 149, 0);
   };
if ( /REMOVAL/i )
   { return (0,0,0);
   };
if ( /WASH/i )
   { return (0, 0, 0);
   };
if ( /ORIF/i )
   { return (0, 0, 499);
   };
if ( /EUA/i )
   { return (0, 0, 0);
   };
if ( /OSCOP/i )
   { return (0, 0, 0);
   };
if ( /LAPAROSC/i )
   { return (0, 0, 3);
   };
if ( /APPEND/i )
   { return (0, 0, 2);
   };
if ( /FIXATION/i )
   { return (0, 0, 499);
   };
if ( /INGU.+HERN/i )
   { return (0, 1, 2);
   };
 if ( /LACERAT/i )
   { return (0, 0, 0);
   };
 if ( /ABSC/i || /INCIS/i || /DRAIN/i || /BOIL/i )
   { return (0, 0, 0);
   };
 if ( /BIOPSY/i )
   { return (0, 0, 0);
   };
if ( /TURB/i )
   { return (0, 799, 149);
   };
if ( /TURP/i )
```

```
    { return (0, 799, 149);
    };
 if ( /PCNL/i )
    { return (0, 799, 119);
    };
 if ( /URETER/i )
    { return (0, 799, 119);
    };
 if ( /PENI/i || /ORCHI/i || /SCROT/i )
    { return (0, 799, 199);
    };
 if ( /EMBOLEC/i || /THROMB/i )
    { return (0, 599, 0);
    };
 if ( /AVF/i )
    { return (0, 599, 0);
    };
 if ( /HERNIA/i )
    { return (0, 1, 0);
    };
 if ( /PARAT/i )
    { return (0, 399, 699);
    };
 if ( /NECK.+DISSEC/i )
    { return (0, 399, 699);
    };
 if ( /SLING/i )
    { return (0, 799, 199);
    };
 if ( /DHS/i || /DHCS/i )
    { return (0, 149, 499);
    };
if ( /MANIP/i )
    { return (0, 0, 499);
    };
if ( /BALLOON/i )
   { return (0, 0, 0);
   };
 if ( /SHUNT/i)
   { # might subclassify..
     return (0, 0, 0);
   };
 if ( /FEMUR/i || /FEMORA/i )
   { return (0, 0, 499)
   };
 if ( /NECK/i)
   { return (0, 0, 799);
   };
```

```
if ( /SHIN/i || /TIBIA/i || /FIBUL/i || /PATEL/i || /KNEE/i || /HEEL/i || /FOOT/i
  { return (0, 0, 499);
  };
if ( /ICP/i)
  { return (0, 0, 799);
  };
if ( /DRESSING/i)
  { return (0, 0, 0);
  };
if (/LAP /i)
  { return (0, 0, 0);
  };
if (/URETHRO/i)
  { return (0, 799, 199);
  };
if ( /HAND/i || /PALM/i || / ARM/i || /ELBOW/i || /SHOULD/i || /RADIUS/i ||
     /ULNA/i || /HUMER/i || /RADIAL/i || /OLEC/i || /CLAVI/i )
  { return (0, 0, 399);
  };
if ( / LEG/i )
  { return (0, 0, 499);
  };
if ( /LIMB/i )
  { if ( /UPPER/i )
      { return (0, 0, 399);
      };
    if ( /LOWER/i )
      { return (0, 0, 499);
      };
    return (0, 0, 0);
  };
if ( /DISC/i )
   { return (0, 199, 699);
   };
if ( /CORD/i ) # hmm.
   { return (0, 199, 0);
   };
if ( / RECTUM/i || /RECTA/i )
  { return (0, 699, 2);
  };
if ( /STENT/i )
   { return (0, 599, 0); # or cardiothoracic
   };
if (/ENDARTER/i )
   { return (0, 599, 0);
   };
if ( /DEBRID/i )
   { return (0, 0, 0);
```

```
    };
 if ( /TEN.+OFF/i )
    { return (0, 1, 3); # nasty: Tenckhoff & variants.
    };
 if ( /OSTEOT/i )
    { return (0, 149, 0);
    };
 if ( /LAMINEC/i )
   { return (0, 199, 0);
   }; # hmm.
 if ( /ABDO/i )
   { return (0, 0, 3);
   };
 if ( /PELVI/i )
   { return (0, 0, 149);
   };
 if ( /LUMBAR/i )
   { return (0, 0, 119);
   };

 # ====================================
 # the following might be commented out:
# if (! $wanted)
#    { if ( &Confirm ($MAINW, "Include the following operation <$_> ?") )
#         { print LOGFILE "\n\n *** OPERATION INCLUDED: $_";
#           return (1, 0, 0);
#         };
#    };
 # ====================================

 return (0, 0, 0); # default = fail!
}
```

Even better would be to have a template file (or array derived from a stored file) and simply go through the array excluding or including based on a regex match. Note that depending on where the painform database is deployed, the above will change.

Identifying the block

We won't want to see every block on the pain service (ring blocks of fingers, blocks for circumcisions??) The following routine looks at the name[113] and decides, returning 0 if we don't want the block, and a number over 0 if we do. The numeric code also says what type of block. In Table 5, derived from *AnalgesiaDBpart1.tex*, we identify spinals, CSE and epidurals, but add a wide range of

[113] As IDAS has no easy way of distinguishing between the various blocks other than the name

regional catheters and infusions, and even code the ability to give PCA via these catheters. As the coding provided is sadly only free text, we manipulate this text to identify whether a catheter was present, or whether only the block is recorded.

Process code	*Meaning*
100	Spinal
102	infraclavicular block
103	axillary block
104	interpleural block
105	femoral block
106	sciatic block
107	incisional block
108	interscalene block
110	Epidural catheter
109	CSE
120	interscalene catheter
220	interscalene infusion
320	interscalene PCA
125	infraclavicular catheter
225	infraclavicular infusion
325	infraclavicular PCA
130	axillary catheter
230	axillary infusion
330	axillary PCA
135	interpleural catheter
235	interpleural infusion
335	interpleural PCA
140	femoral catheter
240	femoral infusion
340	femoral PCA
145	sciatic catheter
245	sciatic infusion
345	sciatic PCA
150	incisional catheter
350	incisional PCA
250	incisional infusion

Table 5: Codes for regional procedures

Here's the code. At present we will not flag everyone with a block, concentrat-

ing on spinals, epidurals, CSEs and those where a catheter is identified as having been placed.

```
sub ChooseBlock
{
 ($_) = @_;

 if ( /tenon/i )
    { return 0;
    };
 if ( /failed/i )
    { return 0;   # heh!
    };
 if ( /attempted/i )
    { return 0;
    };
 if ( /abandon/i )
    { return 0;
    };
 if ( /bulb/i )  # as in bulbar
    { return 0;
    };
 if ( /ring/i )
    { return 0;
    };
 if ( /digit/i )
    { return 0;
    };

 if ( /cse/i )
    { return 109;
    };
 if ( /combined/i ) # as in CSE. Place before 'epidural'!
    { return 109;
    };
 if ( /epidural/i )
    { return 110;
    };
 if ( /spinal/i )
    { return 100;
    };

 #-------------------#
 if ( $_ !~ /cath/i )
    { return 0; # get rid of misc stuff!
    };
 #-------------------#
```

```
 if ( /pleur/i || /costa/i )
    { return 135;  # hmm??
    };
 if ( /wound/i || /incis/i )
    { return 150;
    };
 if ( /sciatic/i )
    { return 145;
    };
 if ( /femoral/i )
    { return 140;
    };
 if ( /scalen/i )
    { return 120;
    };
 if ( /infraclav/i )
    { return 125;
    };
 if ( /axil/i )
    { return 130;
    };

  return 0; # not interested,  by default.
}
```

We need to expand the above to include intrathecal (IT) catheters, psoas compartment block with catheter, paravertebral catheter, lumbar plexus catheter, brachial plexus, caudal, ...

15.3 Individual data for one anaesthetic

At present this routine ignores the value in 'surgsite'. The value in $an is the external (SaferSleep) primary key of the anaesthetic concerned. At present $isregnl will always contain the decimal code for an epidural (110), as we only specify epidurals, but note that other regional procedures can incorrectly be stored in the relevant epidural table in SaferSleep. It is possible that in the future we may add in spinal morphine (code 101).

```
sub ImportExternalAnaesthetic
{
  my ($myODBC, $externalODBC, $an, $isregnl, $ispca, $descrip,
      $SURGTYPE, $SURGSITE);
  ($myODBC, $externalODBC, $an, $isregnl, $ispca, $descrip,
      $SURGTYPE, $SURGSITE) = @_;
  print IMPORTLOG "\n Importing.. <$an> $descrip ";
```

```
  my ($dataproc, $bedproc);

my $extDB = $CONST{EXTDBCODE};
```

The value of EXTDBCODE was introduced on 2007-11-08 as a way of uniquely representing data acquired from the external database.[114]

First, obtain the NHI and DOB, and if this doesn't exist in our database, insert the patient. Also (in this case) record the surname and forename. If however the NHI does exist, validate the recorded DOB/surname/forename/gender, and *warn* the user if any datum differs.

Invocation of the FancyUp routine is a clumsy fix to permit insertion of quotes without SQL errors. This functionality is represented elsewhere in my Perl code.

```
  my $idstmt = $CONST{'XDBGETID'};
  $idstmt =~ s/\$ANAESTHETIC/$an/; # fill in anaesthetic
   # &Alert($MAINW, "Debug: Target <$idstmt>");
  my ($nhi, $dob, $sex, $forename, $surname) =
     &GetSQL($externalODBC, $idstmt, "get nhi, DOB");
  $forename = &FancyUp($forename);
  $surname  = &FancyUp($surname);
  print IMPORTLOG
   "\n name: $forename $surname, born on: $dob, gender ($sex) NHI: $nhi ";


  ## &Alert($MAINW, "Debug: IMPORTING $forename $surname: DOB $dob sex $sex ($desc
  # should we validate $dob [or if NULL, replace with NULL in stmt?] [seems unnece
  # note that for IDAS $sex is returned as 'M' or 'F'.
  if (length $sex < 1)
     { $sex = 0; # unknown
     } else
     {  if ($sex =~ /F/i)
           { $sex = 1;
           } else
           { $sex = 2;
           };
     };
  # ensure NHI is uppercase: (might have other checks too)
  $nhi =~ tr/a-z/A-Z/; # force uppercase.

  my ($person) = &GetSQL($myODBC,
    "SELECT pdoPerson from PERSDATA where pdoHospNo = '$nhi'",
    "check for NHI in our db");
  # pdoPerson is a deliberate denormalisation.
  # we might check for duplication. [!!]
  # [we must ensure lowercase nhi cannot enter database, or use UPPER(pdoHospNo)]
```

[114] For SaferSleep, we use a code value of 5.

```
# (even consider referring to other databases)
my $newperson = 0;
my $now = &GetLocalTime();

if ($SQLOK) # if found, 'warm up' the person in our database:
   { print IMPORTLOG "[EXISTS]";
     ($dataproc, $bedproc) =
       &FixColdCase($myODBC, $person, $extDB); # extDB is planner. ???
   } else
```

Alternatively, if we failed to find the person, we insert a new person, with appropriate attributes:

```
   {
     print IMPORTLOG "[NEW]";
     if (length $dob > 1) # or might fully valiDate.
        { $dob = "TIMESTAMP '$dob'";
        } else
        { $dob = 'NULL';
        };
     $person = &AutoKey($myODBC, "PERSON");
     &DoSQL($myODBC, "INSERT INTO PERSON (person, pBorn, pMade, pStatus) \
          VALUES ($person, $dob, TIMESTAMP '$now', 1)",
          "create new person");
     # a pStatus of 1 signals a 'patient';
     # Next create new data process (type 1):
     $dataproc = &AutoKey($myODBC, "PROCESS");
     &DoSQL($myODBC, "INSERT INTO PROCESS \
        (process, Person, ProcType, rStart, rCreated, rPlanner ) \
     VALUES ($dataproc, $person, 1, TIMESTAMP '$now', TIMESTAMP '$now', $CURRENT
        "create data process");
     # The rPlanner for this process is the current user
     # below the type 3 planner becomes the external database ($extDB)
```

A 'type 1' process is a generic *observation* process. We will also create a 'type 3' or hospital admission process, which allows us to record bedspace movements of the patient. It is possible that the IDAS database records the ward, in which case we use this (as a bedspace entry); alternatively we allocate a ward code of 1 (For 'unallocated').

```
  $bedproc =  &AutoKey($myODBC, "PROCESS");
      &DoSQL($myODBC, "INSERT INTO PROCESS \
         (process, Person, ProcType, rStart, rCreated, rPlanner ) \
VALUES ($bedproc, $person, 3, TIMESTAMP '$now', TIMESTAMP '$now', $extDB)",
         "create admission process");
  print IMPORTLOG "\n our ID: $person (#1:$dataproc)(#3:$bedproc)";
```

```
   # finally, set the following flag for later:
       $newperson = 1;
     };

  # now record the bedspace:
   &InsertBedObservation ($myODBC, $externalODBC, $bedproc,
        $now, $extDB, $person, $an);   # [2007-12-13: fixed] ???
```

Record weight and ASA rating, with the appropriate timestamp.

```
  # first, if data process not created above, retrieve it:
  if (! $dataproc) # hmm. should never be used [can this occur: check?] ???
     { &Alert($MAINW, "Hmm. Missing data process??");
       print LOGFILE ("\n **MISSING DATA PROCESS: person is $person");
       ($dataproc) = &GetSQL($myODBC,
        "SELECT process from PROCESS where proctype = 1 \
        AND process > -1 \
        AND Person = $person",
        "get data process");
       # what if cold? [check] ???
     };
  # Next: observation on the process:
  my $datobs = &AutoKey($myODBC, "EPOCH");
  &DoSQL($myODBC, "INSERT INTO EPOCH (epoch, oMade, Person, Process) \
     VALUES ($datobs, TIMESTAMP '$now', $extDB, $dataproc)",
     "create epoch on proc");

  # If new person, insert personal data:
  if ($newperson)
     { my $persdat = &AutoKey($myODBC, "PERSDATA");
       &DoSQL($myODBC, "INSERT INTO PERSDATA \
(persdata, Epoch, pdoSurname, pdoForename, pdoHospNo, pdoPerson, pdoGender)
VALUES ($persdat, $datobs, '$surname', '$forename', '$nhi', $person, $sex)",
          "insert personal data");
     };
  # [thought: ensure that if sex is 0 this is *consistently* shown as '?']

  # In any case, insert ASA, Weight:
  # retrieve these from IDAS ...
  my ($wt, $asa, $asaE);
  $asaE = 0;
  my $asastmt = $CONST{'XDBGETASAWT'};
  $asastmt =~ s/\$ANAESTHETIC/$an/; # fill in anaesthetic
   # &Alert($MAINW, "Debug: ASA/wt <$asastmt>");
  my ($asa, $wt) = &GetSQL($externalODBC, $asastmt, "get ASA,Wt");
   print IMPORTLOG "\n ASA: $asa, Weight: $wt";

  # first, weight:
```

```
if (length $wt > 0) # hmm?
   { $wt *= 1000;  # convert kg to grams (our unit of measure!)
     my $mekey = &AutoKey($myODBC, "MEASURE");
     &DoSQL($myODBC, "INSERT INTO MEASURE (measure, Epoch, meWt) \
     VALUES ($mekey, $datobs, $wt)", "record weight");
   };
# next, ASA rating:
if (length $asa > 0)
   {
     # ... must also fix up E value:
     if ($asa =~ /(.)e/i)
        { $asa = $1;
          $asaE = 1;
        };
     my $mskey = &AutoKey($myODBC, "MEDSCORE");
     &DoSQL($myODBC, "INSERT INTO MEDSCORE \
      (medscore, epoch, msoNature, msoValue) \
       VALUES ($mskey, $datobs, 1, $asaE)", "insert asa E");
     # ignoring absent E value for ASA is not attractive.
     my $mskey = &AutoKey($myODBC, "MEDSCORE");
     &DoSQL($myODBC, "INSERT INTO MEDSCORE \
      (medscore, epoch, msoNature, msoValue) \
       VALUES ($mskey, $datobs, 2, $asa)", "insert asa");
   };
```

Then check the painform database for a record of *this* operation/anaesthetic. If the timestamp is similar to a previous operation, confirm with the user that a new operation should be created, otherwise just do this. We must also record the type of operation (if possible).

```
my $optimes = $CONST{'XDBOPTIMES'};
$optimes =~ s/\$ANAESTHETIC/$an/; # fill in id.
 # &Alert($MAINW, "Debug: Op times <$optimes>");
my ($opstart, $opend) = &GetSQL($externalODBC, $optimes, "get op start,end");
# IDAS cannot at present deliver a decent operation end timestamp, another
# reason for us ultimately to data warehouse. For now, we hack things!
# first, check whether a similar operation exists in OUR database:
print IMPORTLOG "\n Operation: start $opstart, end $opend (SIMILAR: ";

my ($opmin, $opmax, $opday);
# We must ensure length is 19 chars ie "YYYY-MM-DD HH:MM:SS" :
$opstart =~ /(\d{4}-\d{2}-\d{2})( \d{2}:\d{2}:\d{2})/; # pull out date
# [here might check opstart not null??? is this possible]
$opday = $1;
$opmin = "$1 00:00:00";
$opmax = "$1 23:59:59";
$opstart = "$1$2"; # trim any fractional seconds
# [hmm there is still possibility of phantom operation just after midnight]
```

```
 # 17/3/2008: the following gave a lot of trouble due to Ocelot defect.
 # (the BETWEEN clause behaved erratically).
 # Fixed by casting to varchar()

 my $found = 0;
 my ($st, $pr);
 $pr = 0; # pr is PROCESS. Default 'not found'
 my @oldops;
 (@oldops) = &SQLManySQL ($myODBC,
    "SELECT cast(rStart as varchar(19)), process FROM PROCESS \
      where ProcType = 500 AND \
       process > -1 AND \
      Person = $person AND \
      cast(rStart as varchar(10)) = '$opday' ",
    "get similar operation processes"); # ???: check me!

 if ($SQLOK) # if similar surgery already exists:
    { while (  ($#oldops > 0)
             &&(! $found)
            )
        { $st = shift(@oldops);
          $pr = shift(@oldops);
          if ($st eq $opstart)
             { $found = 1;
             } else
             { print IMPORTLOG "$st;";
               if (&Confirm ($MAINW,
 "I've found a new operation dated $opstart, \n\
 for patient $forename $surname but there's \n\
  a similar operation with timestamp '$st'! \n\
Are the two the same? [If unsure, say 'YES'] ") )
                  { $found = 1;
                  };
             };
        };
    };
print IMPORTLOG ")";

# FIX THE FOLLOWING:
if ($pr)
   { return 0; # duplicate ???
   };

# if no prior record of this operation, record it:
if (! $pr)
   {
     print IMPORTLOG "* ";
```

```
      $pr = &AutoKey($myODBC, "PROCESS");
       &DoSQL($myODBC, "INSERT INTO PROCESS \
 (process, Person, ProcType, rStart, rCreated, rEnd, rPlanner ) \
 VALUES ($pr, $person, 500, TIMESTAMP '$opstart', TIMESTAMP '$now', \
 TIMESTAMP '$opend', $extDB)",
        "create surgical process");
      # also make OBServation on the process:
      my $surgobs = &AutoKey($myODBC, "EPOCH");
      &DoSQL($myODBC, "INSERT INTO EPOCH (epoch, oMade, Person, Process) \
        VALUES ($surgobs, TIMESTAMP '$now', $extDB, $pr)",
        "create epoch on surg proc");


    # finally, record site and type of surgery [as best can!]
      my $surgtypeob = &AutoKey ($myODBC, "SURGTYPEOB");
      &DoSQL($myODBC, "INSERT INTO SURGTYPEOB(surgtypeob,SurgType,Epoch) \
       VALUES ($surgtypeob, $SURGTYPE, $surgobs)",
       "record surg type");

# WE HAVE REMOVED THE RECORDING OF SURGICAL SITE:
# (at least, for now).
#
#      my $surgsiteob = &AutoKey ($myODBC, "SURGSITEOB");
#      &DoSQL($myODBC, "INSERT INTO SURGSITEOB(surgsiteob,SurgSite,Epoch) \
#       VALUES ($surgsiteob, $SURGSITE, $surgobs)",
#       "record surg site");

      # finally, add in the operation description:
      my $opcomment = &AutoKey ($myODBC, "COMMENT");
      $descrip = &FancyUp($descrip);
      &DoSQL($myODBC, "INSERT INTO COMMENT(comment, Epoch, cText) \
       VALUES ($opcomment, $surgobs, '$descrip')",
       "operation description");

    };
```

Determine whether an epidural is present (and if so, get details). Do the same for other significant regional procedures (spinal, CSE).[115]

```
  # at present, depend on $isregnl.
  # For spinals we must create a spinal process (code 100), AND do the 24 hour que
  # To force a 24 hour query, we create a type 99 process!
  # For epidurals we simply create the process. (epidural code is 110).
  # [look into checking for spinal morphine]

  if ($isregnl)
```

[115] At present we have not included other regional infusions, which clearly needs to be done.

```
     { # suppress the following spinal process and 24 hour check, for now:
       # ---

       # There is a problem in SaferSleep. Other regionals can be recorded in the
       # So we have this clumsy query:

       my ($rgnq) = $CONST{'XDBREGIONALTYPE'};
       $rgnq =~ s/\$ANID/$an/;  # fill in SaferSleep anaesthetic ID
       my ($rt) = &GetSQL($externalODBC, $rgnq, 'get rgn type');
       if ( $rt =~ /scalen/i )
          { $isregnl = 120;
          }
       elsif ( $rt =~ /femor/i )
          { $isregnl = 140;
          }
       elsif ( $rt =~ /stump/i )
          { $isregnl = 150;
          }
       elsif ( $rt =~ /wound/i )
          { $isregnl = 150;
          }
       elsif ( $rt =~ /rectus/i )
          { $isregnl = 150;
          }
       elsif ( $rt =~ /sciat/i )
          {$isregnl = 145;
          }
       elsif ( $rt =~ /verteb/i )
          { $isregnl = 137;
          }
       elsif ( $rt =~ /pleur/i )
          { $isregnl = 135;
          };

     # what about brachial, tibial, ... ? perhaps add as options
     # &Alert($MAINW, "Regional code was $isregnl");


       print IMPORTLOG "\n Regional: code $isregnl";

  # [2007-12-13: another problem. If epidural/pca already exists as was
  # entered on the PDA, then a duplicate proc will be created. PREVENT THIS (?)
  # [hmm: what if the regional is removed on that day, and data are THEN fetched
  # from idas. explore this possible 're-activation' of the epidural?
       # [2007-12-13: fix : first ensure no active regional process:]
       my ($regnlisactive) = &GetSQL ($myODBC,
        "SELECT process FROM PROCESS WHERE Person = $person AND \
          ProcType BETWEEN 100 AND 159 AND \
```

```
         process > -1 AND \
         rEnd IS NULL", 'get current regnl');
      if (length $regnlisactive < 1)
         {
            if (($isregnl == 100) || ($isregnl == 101)) # spinal (plain/morphine)
               { # here would create type 100/101 AND type 99 process: ???

               } else
               { # all of the others:
                 my $regproc = &AutoKey ($myODBC, "PROCESS");
                 &DoSQL($myODBC, "INSERT INTO PROCESS \
                 (process, Person, ProcType, rStart, rCreated, rPlanner ) \
 VALUES ($regproc, $person, $isregnl, TIMESTAMP '$opstart', TIMESTAMP '$now', $ex
               "create data process");
                 # note we do NOT start an infusion yet!
                 # use of $opstart may be inaccurate.
                 # Perhaps get 'actual' time from IDAS ??
                 # i.e. EpiduralAnalgesia.PrescriptionDate
               };
         };
    };
```

Likewise, determine whether IV PCA was ordered. The process code for IV PCA is 390.

```
  if ($ispca)
     { print IMPORTLOG "\n IV PCA: $ispca";
       # [2007-12-13] a fix for dup. PCA, as for epidural above!
       my ($pcaisactive) = &GetSQL ($myODBC,
        "SELECT process FROM PROCESS WHERE Person = $person AND \
         process > -1 AND \
         ProcType = 390 AND \
         rEnd IS NULL", 'get current pca');
       if (length $pcaisactive < 1)
         { my $pcaproc = &AutoKey ($myODBC, "PROCESS");
           &DoSQL($myODBC, "INSERT INTO PROCESS \
             (process, Person, ProcType, rStart, rCreated, rPlanner ) \
 VALUES ($pcaproc, $person, 390, TIMESTAMP '$opstart', TIMESTAMP '$now', $extDB)",
               "create data process");
           # note we do NOT start an infusion yet!
           # use of $opstart may be inaccurate.
           # Perhaps get 'actual' time from IDAS PCA table?! ???
         };
     };
```

Finish off . . .

```
  return 1; # success
}  # the end.
```

At present the above routine either returns 0 (duplicate) or 1 (success). Later might return -1 on failure.

15.3.1 InsertBedObservation

Whether a case is ‘new’ or has had previous surgery, we need to retrieve and record the current bed status. Note that we should always ensure that a current, active bed observation doesn’t already exist!

We supply open connections to our database and the external database, as well as the current (active) type 3 process which refers to admission, the time ($now), the *ID* of the external database ($extDB), and the patient ($person) we are interested in, and finally the actual anaesthetic ID in the external database, $an.

```
sub InsertBedObservation # [2007-12-03] ???
{
  my ($myODBC, $externalODBC, $bedproc,
        $now, $extDB, $person, $an);
     ($myODBC, $externalODBC, $bedproc,
        $now, $extDB, $person, $an)=@_;

  my ($isbadobs) = &GetSQL ($myODBC,
   "SELECT badobs FROM BADOBS WHERE person = $person \
     AND cold IS NULL AND boInactive IS NULL",
   'get existing bed observation');
  if (length $isbadobs > 0)
     { return; # a bed obs exists. Don't fiddle!
     };

  # get ward from external database [IDAS]
  my $wardstmt = $CONST{'XDBGETWARD'};
  $wardstmt =~ s/\$ANAESTHETIC/$an/; # fill in anaesthetic
   # &Alert($MAINW, "Debug: Target <$idstmt>");
  my $wardcode;
  my ($ward) =
     &GetSQL($externalODBC, $wardstmt, "get ward");
  if ($SQLOK)
     { ($wardcode) = &GetSQL($myODBC,
      "SELECT ward FROM WARD where swrdText LIKE '%$ward%'",
      "find ward in OUR db");
        # this is debugging:
           # if (! $SQLOK)
           #   { &Alert ($MAINW, "DEBUG: Bad ward <$ward>");
           #   };
     }; # NOT 'else..'
  if (! $SQLOK) # note resetting!
```

```
      { $wardcode = 1; # 'unknown'
      };
    print IMPORTLOG "\n ward: $ward ($wardcode)";

    my $bobs = &AutoKey($myODBC, "EPOCH");
    &DoSQL($myODBC, "INSERT INTO EPOCH(epoch, oMade, Person, Process) \
      VALUES ($bobs, TIMESTAMP '$now', $extDB, $bedproc)",
      "create epoch on bed proc");

  my $badobs = &AutoKey($myODBC, "BADOBS");
  my $bed = $wardcode  * 10000; # our convention: generic bedspace
  &DoSQL($myODBC, "INSERT INTO BADOBS(badobs,Bed,Epoch,Person,boFlag) \
    VALUES($badobs, $bed, $bobs, $person, 1)",
    "make bedspace observation");
  # we set boflag as the patient is by default 'interesting'!
  # Person is denormalisation
}
```

15.3.2 FixColdCase

Here we have identified that a person already exists within our database, based on their hospital number already being recorded here. There are two possibilities:

1. A current, active record exists, with active (non-cold) type 1 observation and type 3 admission processes;
2. The records are historical. We would have converted old type 1 to type 2 (historical) processes, and there will be no active PERSON, PERSDATA, PROCESS (or dependent EPOCH), or BEDSPACE entries.

In the former case we wish to attach all new data (etc) to the existing processes, but we may need to create new operation processes etc.

In the latter case, we will need to make new type 1 and 3 processes

In addition we must activate relevant old processes, in particular associated rows in PERSON, PERSDATA, EPOCH and PROCESS. By using type 2 processes we avoid slow MAX(process) WHERE ProcType = 1 constructs on the PDA.

So our plan is:

1. If both type 1 and 3 processes exist identify these, else
2. If neither one exists, create both.
3. Whatever happened above, we need to go through and activate previous rows (make them un-cold ie warm) so that the relevant data will be transferred to the PDA, preventing a crash.

Arguments are self-explanatory, and the routine returns the keys of the data (type 1) and admission (type 3) processes:

```
sub FixColdCase
{
  my ($myODBC, $person, $planner);
     ($myODBC, $person, $planner) = @_;

  # 1. First, if processes 1,3 don't exist, make them:
  my ($old1) = &GetSQL($myODBC,
    "SELECT process FROM PROCESS WHERE Person = $person \
      AND process > -1 \
      AND cold IS NULL AND ProcType = 1", 'get existing proc 1');
      # for extra security might say MAX(process). [no]
  my ($old3) = &GetSQL($myODBC,
    "SELECT process FROM PROCESS WHERE Person = $person \
      AND process > -1 \
      AND cold IS NULL AND ProcType = 3", 'get proc3');

  my $now = &GetLocalTime();  # ??? 8/4/2008
  if (length($old3) < 1)
     { $old3 = &AutoKey($myODBC, "PROCESS");
       &DoSQL($myODBC, "INSERT INTO PROCESS \
          (process, Person, ProcType, rStart, rCreated, rPlanner ) \
         VALUES ($old3, $person, 3, TIMESTAMP '$now', \
                 TIMESTAMP '$now', $planner)",
          "new proc3");
       print LOGFILE "\n\n New PROC3 made: $old3";
     } else
     { # print LOGFILE "\n Old PROC3: $old3";
     };

  if (length($old1) < 1)
     { $old1 = &AutoKey($myODBC, "PROCESS");
       &DoSQL($myODBC, "INSERT INTO PROCESS \
          (process, Person, ProcType, rStart, rCreated, rPlanner ) \
         VALUES ($old1, $person, 1, TIMESTAMP '$now', \
                 TIMESTAMP '$now', $planner)",
          "new proc1");
       print LOGFILE "\n New PROC1 made: $old1";
     } else
     { print LOGFILE "\n Old PROC1: $old1";
     };

  # in any case, activate all old persdata!
  # ideally we should amalgamate all old persdata into one RECENT record:

  my ($lastsurid) = &GetSQL($myODBC,
```

```
    "SELECT MAX(persdata) FROM PERSDATA WHERE pdoPerson = $person \
      AND pdoSurname IS NOT NULL GROUP BY persdata", 'get last surname');
  my ($lastforeid) = &GetSQL($myODBC,
    "SELECT MAX(persdata) FROM PERSDATA WHERE pdoPerson = $person \
      AND pdoForename IS NOT NULL GROUP BY persdata", 'get last forename');
  my ($lastsexid)= &GetSQL($myODBC,
    "SELECT MAX(persdata) FROM PERSDATA WHERE pdoPerson = $person \
      AND pdoGender IS NOT NULL GROUP BY persdata", 'get last gender');
  # hospno is tricky: the FIRST one should take priority [check me?] ???
  # [[?? is the major NHI always the first one ???]]
  my ($firstNHIid) = &GetSQL($myODBC,
    "SELECT MIN(persdata) FROM PERSDATA WHERE pdoPerson = $person \
      AND pdoHospNo IS NOT NULL GROUP BY persdata", 'get FIRST hosp No');
  my ($MAXid) = &GetSQL($myODBC,
    "SELECT MAX(persdata) FROM PERSDATA WHERE pdoPerson = $person \
     GROUP BY persdata", 'get last persdata record');

#  print LOGFILE "\n debug: MAX ID is $MAXid";

# ??? CAREFULLy explore what happens in the following with ABSENT entries in PERSD
# is this possible? does it hurt?

  # the following is cumbersome and should also ideally be logged in
  # a separate AUDIT table!
  # in addition we are introducing a denormalisation (ugh)!
  if ($lastsurid != $MAXid)
     { my ($surn) = &GetSQL($myODBC, "SELECT pdoSurname FROM PERSDATA \
         WHERE persdata = $lastsurid", 'get recent surname');
       &DoSQL ($myODBC, "UPDATE PERSDATA SET pdoSurname = '$surn' \
         WHERE persdata = $MAXid", 'set recent surname');
     };
  if ($lastforeid != $MAXid)
     { my ($foren) = &GetSQL($myODBC, "SELECT pdoSurname FROM PERSDATA \
         WHERE persdata = $lastforeid", 'get recent forename');
       &DoSQL ($myODBC, "UPDATE PERSDATA SET pdoForename = '$foren' \
         WHERE persdata = $MAXid", 'set recent forename');
     };
  if ($lastsexid != $MAXid)
     { my ($sex) = &GetSQL($myODBC, "SELECT pdoGender FROM PERSDATA \
         WHERE persdata = $lastsexid", 'get recent sex');
       &DoSQL ($myODBC, "UPDATE PERSDATA SET pdoGender = $sex \
         WHERE persdata = $MAXid", 'set recent gender');
     };
  if ($firstNHIid != $MAXid)
     { my ($nhi) = &GetSQL($myODBC, "SELECT pdoSurname FROM PERSDATA \
         WHERE persdata = $firstNHIid", 'get recent hosp No');
       &DoSQL ($myODBC, "UPDATE PERSDATA SET pdoSurname = '$nhi' \
         WHERE persdata = $MAXid", 'set recent NHI');
```

```
    };

  # MAXid is now an up to date reference. Fix all its 'referents':

  my ($epo) = &GetSQL($myODBC,
    "SELECT Epoch FROM PERSDATA WHERE persdata = $MAXid",
    'get epoch for persdata');
  my ($pro) = &GetSQL($myODBC,
    "SELECT Process FROM EPOCH WHERE epoch = $epo",
    'get process for epoch');
  &DoSQL($myODBC, "UPDATE EPOCH SET cold = NULL WHERE epoch = $epo",
    'warm up epoch');
  &DoSQL($myODBC, "UPDATE PROCESS SET cold = NULL WHERE process = $pro",
    'warm process'); # (type 2)
  &DoSQL($myODBC, "UPDATE PERSON SET cold = NULL WHERE person = $person",
    'warm person');
  &DoSQL($myODBC, "UPDATE PERSDATA SET cold = NULL WHERE persdata = $MAXid",
    'NB also warm persdata'); # else fails catastrophically

  # NOTE that we do NOT create/activate a BADOBS row, nor do we COMMIT.

  return ($old1, $old3);
}
```

16 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.[116]

16.1 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 6.

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 6: PDB header fields

The recordList has a header that is usually only six bytes long. The first four bytes are almost always zero,[117] 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.[118]

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

[117] 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!

[118] 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 7: 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!

16.2 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 8. 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 8: 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 offset 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 9.

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 9: Our column format

Although the first entry in Table 9 (offset of column) is redundant, we retain it, and *mandate* its use![119]

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.[120]

16.2.1 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[121] 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 10 describes the types.

[119]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.

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

[121]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 10: 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.[122] For decimal, use numeric. We eschew use of a fixed length character format, as this is inefficient.[123]

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.[124]

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'.

16.3 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 11.

[122] We may eventually implement a BLOB type.

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

[124] 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 11: 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.[125]

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.

16.4 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'.

16.4.1 MakeAllPDBs

First up, we COMMIT the database, as the current desktop user may have been entering data but normally a COMMIT would not be performed until they actually Quit the left pane (and program).[126]

[125] 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.

[126] We could consider confirming this COMMIT.

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.

```
sub MakeAllPDBs
{ my ($myODBC, $localdir);
     ($myODBC, $localdir) =@_;

  &CheckWarm($myODBC, 1); # use sysop as author?!

  chomp $_;  # ?? what

  &Commit($myODBC);  # force COMMIT as left pane not yet committed!
#  $PROGTEXT = 'Exporting PDB files..';
  $PROGTEXT = 'Exporting data..';
  $BARPROGRESS = 0;
+OPTIONAL
  $PROGRESSBAR->update();
-OPTIONAL

  my $xlog="XLOG.LOG";
  open XLOG, ">$xlog" or die "*CRASH* Can't open XLOG :$!\n";
  print XLOG "% A list of PDB files for later manual editing";

  print SYNCFILE "\n\n *** EXPORTING PDB FILES *** \n";
  my(@tablenames);
  @tablenames = &SQLManySQL ($myODBC,
     "SELECT xTaKey, xTaName FROM xTABLE ORDER BY xTaKey",
     "fetch table codes/names");
  my ($tcode, $tname);
  my $ltime; # epoch time (raw)

  my $barstep;
  if ($#tablenames > 0)
     { $barstep = 2*$BARWIDTH/$#tablenames;
     }; # *2 because we have pairs!

  while ($#tablenames > 0)
    { $tcode = shift (@tablenames);
      $tname = shift (@tablenames);
      $ltime = time(); # since 1970
      $BARPROGRESS += $barstep;
+OPTIONAL
      $PROGRESSBAR->update();
-OPTIONAL
      MakeOnePDB($tname, $tcode, $localdir, $ltime);
      print XLOG "\n$tname.PDB"; # store table name
    };
```

```
  print XLOG "\n*";
  # Alert( $MAINW,  "\n PDBs created.");
  close XLOG;

  &WriteSyncTime($localdir);
  $PROGTEXT = '';
  $BARPROGRESS = 0;
+OPTIONAL
  $PROGRESSBAR->update();
-OPTIONAL
}
```

We've modified the above code to order the SELECT from XTABLE; we also now write all of the table names to an *XLOG.LOG* file, which we will subsequently pare down to a short list of names contained in the manually created file *XLOG.LST*. The last-mentioned file will also be used in re-importing data from the PDA to the database (Section 17).

We use WriteSyncTime to write the local (epoch) time to the file LASTSYNCH.TIME, adding ten seconds to allow for caching issues delaying the timestamp on the last few files written.[127]

```
sub WriteSyncTime
{ my ($localdir);
  ($localdir) = @_;

  my $ltime;
  my $LS = "$localdir/LASTSYNCH.TIME";
  open LS, ">$LS"
    or die "Failed on synch time: $!\n";
  print LS "% This is the (raw, local) last synch time in $localdir";
  $ltime = time(); # epoch time (raw)
  $ltime += 10; # allow for caching...?
  print LS "\n$ltime";
  close LS;
  $_ = ctime($ltime);
  print SYNCFILE "\n\n * Local synchronisation time (+10s): $_ ($ltime) *";
}
```

16.4.2 Export all PDBs and PRCs

ExportAllPDBsAndPRCs is a minor administrative routine which allows you to export all PDB files to a directory of your choice, which must already exist. It

[127] We're probably being over-cautious over here, and could conceivably introduce problems with this trick, but believe it's the smarter option, despite the fact that even if the file timestamp occurs after the local timestamp, examination of the files won't result in any new records being written to the database.

then invokes the external local file (eugh) *moveover.bat* which copies PRC files from where they were generated under Cygwin to the same directory. Hmm, a nasty hack which requires fixing.[128] Useful in setting up the database on a new PDA without clutter.

```
sub ExportAllPDBsAndPRCs
{
  my ($myODBC);
  ($myODBC) = @_;

#  my $expdir = "data/copied";
  my $expdir = "/Progra~1/Handspring/Level8/backup";

   $expdir = &Ask ($ADMINMENU, "Enter DESTINATION directory", $expdir);


 # here might check it's a valid directory???
  if (! &ValidatePath($expdir))
   { &Alert($MAINW, "Error: directory $expdir does not exist");
     return;
   };

   if (length $expdir < 1)
      { &Alert ($ADMINMENU, "Nothing done!");
        return;
      };

  $MAINW->focus(); # even focusForce() seems to do nil if focusFollowsMouse ???
  $ADMINMENU->iconify;
  &MakeAllPDBs ($myODBC, $expdir);
  $ADMINMENU->deiconify;
  $ADMINMENU->focus();

  my $pbat;
  $pbat = $CONST{PAINEXPORTBATFILE};
    # NEXT add the destination as the path ?!
    # the batch file must accept the path as %1 : the first argument!
  # Next, because MS Windows/DOS is picky, we translate forward slashes to backsla
  $_ = $expdir;
  tr /\//\\/; # ugly
  $pbat = "$pbat $_";
  system ( $pbat );

  &Alert ($ADMINMENU, "Exported to <$expdir>");
}
```

[128] For starters, we now must also copy *MathLib.prc* from the local directory to that directory!

16.4.3 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 ($tname, $tcode, $localdir, $ltime);
     ($tname, $tcode, $localdir, $ltime) = @_;

  print SYNCFILE "\n $tname --->";  # debug
  my ($colcount, $myhdr, $myQUERY, $dTYPE);

  open PDBFILE, ">$localdir/$tname.PDB" or
      die "*CRASH* Could not open PDB $tname \
      \n(Does <$localdir> subdirectory exist?) :$!\n";
  binmode PDBFILE;
```

The assumption is made that the file will be written in an existing directory specified as LOCALDATADIR. See Section 18.6.

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 =~ /^UIDS$/i);
  my($myhdrA, $myhdrB, $negoff);
  ($colcount, $myhdrA, $myhdrB, $myQUERY, $dTYPE) =
     &MakeOurHeader($myODBC, $tcode, $isuids);
```

The sneakiest (and therefore most error-prone) aspect of MakeOurHeader is the third item it returns — the comma-delimited list of column names, myQUERY, which is used below by FetchAllRecs. 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, $isuids);
```

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

```
  $negoff = -(2 + $#myrecs);
  $negoff = pack("N", $negoff);
  $myhdr = $myhdrA . $negoff . $myhdrB;

 $ph = &MakePalmDBHeader ($myhdr, $tname, $ltime, @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.

16.4.4 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, $ctype) =
         &MakeColmDescriptor ($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 MakeColmDescriptor: 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 ($myhdrA, $myhdrB);
#    Offset    Size     Description
#    +0         4       CRC32
    $myhdrA = &Print4 (0);
#    +4         2       Flags.
    $myhdrA = $myhdrA .  &Print2 (1);
#    +6         2       Total header length
    $myhdrA = $myhdrA .  &Print2 ( 0x10 +
                              length($COFF) +
                              length($CH) );
```

```
#    +8         4       This number must be < 0:
#    $myhdrA = $myhdrA .  &Print4 (0x80000000);
# we'll leave the above till later!!


#    +C         2       all zeroes:
    $myhdrB = &Print2 (0);
#    +E         2       Number of columns
    $myhdrB = $myhdrB .  &Print2 ($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,
          $myhdrA,
          $myhdrB . $COFF . $CH,
          $COLNAMECOMMAS,
          $COLTYPECOMMAS);
}
```

16.4.5 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 = &SQLManySQL ($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.

16.4.6 MakeColmDescriptor

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.[129]

MakeColmDescriptor 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 MakeColmDescriptor
{ 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) { $ctype = 'I'; }; # force 32 bit integer
 ($colname) = &GetSQL ($myODBC,
            "SELECT xCoName FROM xCOLUMN WHERE xCoKey = $ck",
            "get name of this column");
 if ($ctype eq 'I')
    { $cwidth = 4;
    };
```

[129] 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!

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 SYNCFILE ("\n     $colname($ctype) :\
          $cwidth($cscale) -> $ctable");
```

In the above we might check that key reference is type I, and fail (or coerce) if not![130] The SYNCFILE 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);
    #  +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
```

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

```
    $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.[131] 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.[132]

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.[133] Finally, we concatenate all strings into a descriptor:

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

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

[132] 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.

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

16.4.7 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).

We also submit the mysterious `isuids`. This variable has a very specific meaning — if set, it signals we are dealing with the UIDS table, which is used to generate new keys on the PDA. PDA keys are always greater than or equal to 900 000 000, and every time we rewrite to the PDA, we restart all keys at this number. The `isuids` variable permits this — see FormatDatum usage below!

```
sub FetchAllRecs
{ my ($myODBC, $colcount, $tname, $myQUERY, $dTYPE, $isuids);
     ($myODBC, $colcount, $tname, $myQUERY, $dTYPE, $isuids) = @_;

  if ($colcount < 1)
     { return ""; # fail [check me]
     };

  print SYNCFILE "\n * Fetching records: columns = $colcount";

  if ($isuids) # non-zero
     {
      $isuids = $UIDBASE;
     };

  my (@dtypes); # column 'data types'
  $_ = $dTYPE;
  @dtypes = split /,/; # array of column data types
  my ($keyname);

     $myQUERY =~ /([^,]+),/; # get key column
     $keyname = $1;
```

The first column is *always* the primary key.

We set $isuids to the value of UIDBASE, which for a single PDA should be 900 000 000. This value is then progressively incremented for each generator key ('seed'), allowing us to establish a particular key-space for each table on the PDA! (See section 17.5).

Next, we select the columns. We now have a *cold* field in every table we export. This allows us only to export records which have a NULL cold field, suppressing export of all other cold fields! (See also section 5.6). From February

2008, we also allow for the presence of tables with negative key values (which must never be exported to the PDA).

```
my ($thisSQL);
$thisSQL = "SELECT $myQUERY FROM $tname WHERE \
cold IS NULL AND $keyname > -1 \
ORDER BY $keyname";
my($reccnt);  # total number of rows
my(@myrecs); # output array
my @outrecs = ();

(@myrecs) = &SQLManySQL ($myODBC,
                    $thisSQL, "get ROW values");
print SYNCFILE "\n QUERY RESULT(debug) < @myrecs >";
$reccnt = (1 + $#myrecs)/$colcount;
print SYNCFILE ("\n Records=$reccnt ");
print SYNCFILE ("QUERY: <$myQUERY>");

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.[134]

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

```
while ($i < $reccnt)
  {

    # strip off first row:
    @c = @myrecs[0..($colcount-1)];
    @myrecs = @myrecs[$colcount..$#myrecs]; # clumsy
    # print SYNCFILE "\n record($i) values <@c>";     #
    # print SYNCFILE "\n REMAINING values <@myrecs>"; # debug only

    $offs = 0x10 + 2 + 2*$colcount; # offset of 1st datum
    my ($prim);
    $prim = $c[0];  # primary key (clumsy but explicit)
    $formrec = &Print2(8);
    $dat = '';
    $ci = 1;
```

[134] Every row has its own tiny header.

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.[135]

```
  while ($ci < $colcount)
    { $formrec = $formrec . &Print2($offs); # write offset
      my ($formd);
      if (! defined $c[$ci])
         { $formd = "";
         } else
         { $formd = &FormatDatum($dtypes[$ci], $c[$ci], $isuids);
           print SYNCFILE (
              "\n   $c[$ci]: $dtypes[$ci] -> $formd");
           if (defined $formd)
              { $offs += length $formd;
                $dat = $dat . sprintf ("%s", $formd);
              } else
              { print SYNCFILE
"\n Bad formatting: $dtypes[$ci]:$c[$ci] (@dtypes)";
              };
         };
      $ci ++;
      if ($isuids) # non-zero
         { $isuids += $UIDINCREMENT;
         };
    }; # end outer loop
  $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.[136] 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.

At the point where we're about to loop around in the outer loop, we increase the value of $isuids if we're dealing with the UIDS table, so that the generator value for each PDA key is unique and separate!

Finally we make the header:

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

[136] We also keep a record of what we've done in SYNCFILE, for debugging purposes.

```
      # 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]
      $outrecs[$i] = $formrec;
      $i ++;  # BUMP record count
    };
  return (@outrecs);
}
```

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.[137] In line `[8]` we complete the line by prepending the header.[138]

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

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

[137] 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.

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

```
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.[139] 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')
   { $_ = &SqueezeTime($_);
   }
```

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 fourteen 'squeezed' digits.[140] The varchar (character varying) field is easy as we do precisely nothing:

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

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

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

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:[141]

```
  elsif ($dtype eq 'I')
     {  if ($isuids)
          {$_ = $isuids; # force special key 'seed'
          };
       $_ = &Print4($_);
     }
```

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 SYNCFILE
             "\n Bad datum type: $dtype ($_)";
     };
return $_;
}
```

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

16.4.9 SqueezeDate

Here's the first of the squeezing routines. We trim the dashes out of a date:[142]

```
sub SqueezeDate
{ ($_) = @_; # redundant.
  my ($y,$m,$d);
  if (! /(\d{4})-(\d{1,2})-(\d{1,2})/ )
          { print SYNCFILE "\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.
}
```

16.4.10 SqueezeTime

Similarly for time:

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

[142] Hideous code, I'm afraid!

```
     { $s = "0" . $s;
     };
  while (length $f < 6)
     { $f = "$f" . "0";
     };
  $_ = "$h$m$s$f";
  return ($_);
}
```

16.4.11 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![143]

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, $ltime,  @myrecs);
     ($myhdr, $tname, $ltime, @myrecs) = @_;
  my($reccount);
  $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 = $ph . &Print4 ($ltime);
    #    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);
```

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

```
#    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.[144] 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:

[144] PalmOS fills these in.

```
  my($c);
  $c = 0;
  print SYNCFILE ("\n Creating $reccount-1 records: ");
  while ($c < ($reccount-1)) # -1 as header already done
    { $ph = $ph . &Print4 ( $uptotop );
      print SYNCFILE (" $c:$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.

16.4.12 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 --;
       };
 return $s;
}
```

16.4.13 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;
}
```

16.4.14 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.

16.5 Validating Exports

It is possible that a patient about to be exported may have been re-admitted. As we archive old information by setting the **cold** flag in relevant records, such data will not be exported to the PDA (with catastrophic results) if we don't 'warm them up' again!

The following routine checks all active people (in other words, those in whom there is a type 3 (admission) process with a **cold** flag that is NULL. The corresponding tables must be interrogated:

BADOBS There should be an active entry here;

PERSON The record should not be cold;

PERSDATA Not only should the record not be cold, but we (ideally) desire amalgamation of all of the most recent changes into one record (for ease and speed of access on the PDA). These changes are identical to the ones enacted in *FixColdCase*.

PROCESS The process to which the persdata are attached must be warm, as must be the EPOCH which links the two. The alterations are as for FixColdCase.

Before we do anything else, we ensure that all cold type 1 (observation) processes are converted to type 2 ('old observation') processes! This allows us to export such processes to the PDA (iff required) without worrying about them being misidentified as type 1 processes (In other words, we don't have to SELECT MAX on the PDA).

Next we identify the relevant people, and then call *FixColdCase* for every one. This will slow things down somewhat, and we need to look into a more robust fix. In order to identify the relevant people, we look at the BADOBS table (not PROCESS) as this is what will be displayed on the PDA, and there's a chance that a BADOBS entry might not have a type 3 entry (if an error occurred), with horrendous consequences.

Things are more complex. It's conceivable that we somehow have an error state where somebody has been discharged but an errant BADOBS row describing that patient is still active (warm). This is a real problem, as if we note this BADOBS entry, we will automatically re-admit the patient within FixColdCase?! The real problem is that we are trying to coerce FixColdCase into doing two things, firstly allowing re-activation of a process on import from SaferSleep; secondly, within MakeAllPDBs.

Our clumsy hack is to only select people with an active type 3 PROCESS. Better would be to kill the BADOBS problem at source and/or to here invoke a variant of FixColdCase which doesn't permit creation of new type 1/3 processes. We *do* use the latter strategy with the modified routine FixColdCase2.

Here's the tiny routine:

```
sub CheckWarm
{
  my ($myODBC, $planner);
     ($myODBC, $planner) = @_;

  # here we wish to simply cool ALL BADOBS entries
  #    not attached to a warm process
#  &DoSQL ($myODBC,
#    "UPDATE BADOBS SET cold = 1 WHERE badobs IN (SELECT \
#      badobs FROM BADOBS, EPOCH, PROCESS WHERE \
#        BADOBS.Epoch = EPOCH.epoch AND \
#        EPOCH.Process = PROCESS.process AND \
#        BADOBS.cold IS NULL AND \
#        PROCESS.cold IS NOT NULL)",
#    'ensure no aberrant badobs rows. NASTY');
## fix 2008-10-29 by remming this out.
## MUST STILL NOW WATCH FOR DUPS, AND ID HOW THEY OCCUR. HMM. $jvs$
```

```
&TempFixWarmOnCold($myODBC, $MAINW); # temp fix replaces the above. Hmm.

  &DoSQL ($myODBC,
    "UPDATE PROCESS SET ProcType = 2 \
      WHERE cold IS NOT NULL AND \
      ProcType = 1", 'retire old procs');

  my (@warmlist) = &SQLManySQL ($myODBC,
    "SELECT DISTINCT BADOBS.Person FROM BADOBS, PROCESS WHERE \
      BADOBS.Person = PROCESS.Person AND \
      PROCESS.cold IS NULL AND \
      PROCESS.ProcType = 3 AND \
      BADOBS.cold IS NULL AND \
      boInactive IS NULL",
    'get warm admissions');  # the join may be slow.
    # we ensure an active type 3 process exists!

 my ($person);

 foreach $person (@warmlist)
   { &FixColdCase2($myODBC, $person, $planner);
   };

  # if very slow might have display ??

  &Commit($myODBC); # hmm.
}
```

A temporary hack

Here's the temporary hack (2008-10-31) that addresses two problems:

1. Intermittently, a new PDA-generated epoch (related to a BADOBS entry) is attached to a cold process. We must still investigate how this occurs. If we detect this, we transfer the incorrectly attached epoch to the most recent (active) type 3 process;

2. Conversely, a BADOBS entry may not have been terminated despite the patient having been discharged! This presumably occurs where the patient has been moved to another ward/bed on one PDA, is then discharged on a 2nd PDA, and the first PDA is resynchronised after the second! (Check me!) Here we simply make the BADOBS entry cold.

```
# ------------------------------------------------------------------------------
sub TempFixWarmOnCold              # 2008-10-31
```

```
{
  my ($myODBC, $MAINW);
     ($myODBC, $MAINW) = @_;

my (@warmoncold) = &SQLManySQL($myODBC,
 "SELECT EPOCH.epoch FROM BADOBS,EPOCH,PROCESS WHERE
   BADOBS.epoch = EPOCH.epoch AND EPOCH.process = PROCESS.process AND
   BADOBS.cold IS NULL AND
   BADOBS.boInactive IS NULL AND
   PROCESS.cold IS NOT NULL",
 'get warm epochs on cold processes'); # ensure only active BADOBS!

if ($#warmoncold > -1)
   { my ($wc);
     foreach $wc (@warmoncold)
       { my ($pt) = &GetSQL ($myODBC,
          "SELECT PROCESS.Person FROM EPOCH, PROCESS WHERE
            EPOCH.Process = PROCESS.process and EPOCH.epoch = $wc",
          'get person for this EPOCH. ugh.'); # rather unify with the above!
         my ($warmpr) = &GetSQL($myODBC,
         "SELECT process FROM PROCESS WHERE PROCESS.Person = $pt
           AND PROCESS.cold IS NULL
           AND PROCESS.ProcType = 3",
         'get warm admission process for this pt');
         if (length $warmpr < 1)
            { # &Alert ($MAINW, "Oops. Missing admission data. Patient code $pt. T
              print LOGFILE "\n Unable to fix: Patient code $pt. Epoch $wc. COOLED
              # LIKELY problem is residual BADOBS that was not cooled (WHY?)
              # thus chill this observation
              &DoSQL ($myODBC,
               "UPDATE BADOBS SET cold=33 WHERE Epoch=$wc",
               'cool orphan bad observation');
            } else
            { print LOGFILE "\n Error fix: Patient $pt. Epoch $wc. New PROCESS: $w
              &DoSQL ($myODBC,
               "UPDATE EPOCH set Process = $warmpr WHERE epoch = $wc",
               'fix bad process attribution in epoch');
            };
       }
   }
}
```

Here's the variant routine, which will *not* create new type 1 or 3 processes, and in addition will balk if no non-terminated process exists:

```
sub FixColdCase2
{
  my ($myODBC, $person, $planner);
```

```
    ($myODBC, $person, $planner) = @_;

  # 1. First, if processes 1 and 3 don't exist, warn and exit:
  my ($old1) = &GetSQL($myODBC,
    "SELECT process FROM PROCESS WHERE Person = $person \
      AND cold IS NULL \
      AND rEnd IS NULL \
      AND process > -1 \
      AND ProcType = 1", 'get existing proc 1');
  my ($old3) = &GetSQL($myODBC,
    "SELECT process FROM PROCESS WHERE Person = $person \
      AND cold IS NULL \
      AND rEnd IS NULL \
      AND process > -1 \
      AND ProcType = 3", 'get proc3');
    if (  (length($old3) < 1)
        ||(length($old1) < 1)
       )
       { &Alert($MAINW, "Anomaly! Orphan BADOBS for $person");
         # here turn off all BADOBS entries:
         &DoSQL ($myODBC,
           "UPDATE BADOBS SET cold = 1 WHERE person = $person",
           'fix BADOBS anomaly');
         return -1;
       };

  my $now = &GetLocalTime();
#  print LOGFILE "\n Old procs: ($old1, $old3)";

  # in any case, activate all old persdata!
  # ideally we should amalgamate all old persdata into one RECENT record:
  my ($lastsurid) = &GetSQL($myODBC,
    "SELECT MAX(persdata) FROM PERSDATA WHERE pdoPerson = $person \
      AND pdoSurname IS NOT NULL GROUP BY persdata", 'get last surname');
  my ($lastforeid) = &GetSQL($myODBC,
    "SELECT MAX(persdata) FROM PERSDATA WHERE pdoPerson = $person \
      AND pdoForename IS NOT NULL GROUP BY persdata", 'get last forename');
  my ($lastsexid)= &GetSQL($myODBC,
    "SELECT MAX(persdata) FROM PERSDATA WHERE pdoPerson = $person \
      AND pdoGender IS NOT NULL GROUP BY persdata", 'get last gender');
  # hospno is tricky: the FIRST one should take priority [check me?]
  # [[?? is the major NHI always the first one ???]]
  my ($firstNHIid) = &GetSQL($myODBC,
    "SELECT MIN(persdata) FROM PERSDATA WHERE pdoPerson = $person \
      AND pdoHospNo IS NOT NULL GROUP BY persdata", 'get FIRST hosp No');
  my ($MAXid) = &GetSQL($myODBC,
    "SELECT MAX(persdata) FROM PERSDATA WHERE pdoPerson = $person \
     GROUP BY persdata", 'get last persdata record');
```

```
#  print LOGFILE "\n debug: MAXIMUM ID is $MAXid";

  # Issues with the following are as for FixColdCase:
  if ($lastsurid != $MAXid)
     { my ($surn) = &GetSQL($myODBC, "SELECT pdoSurname FROM PERSDATA \
         WHERE persdata = $lastsurid", 'get recent surname');
       &DoSQL ($myODBC, "UPDATE PERSDATA SET pdoSurname = '$surn' \
         WHERE persdata = $MAXid", 'set recent surname');
     };
  if ($lastforeid != $MAXid)
     { my ($foren) = &GetSQL($myODBC, "SELECT pdoSurname FROM PERSDATA \
         WHERE persdata = $lastforeid", 'get recent forename');
       &DoSQL ($myODBC, "UPDATE PERSDATA SET pdoForename = '$foren' \
         WHERE persdata = $MAXid", 'set recent forename');
     };
  if ($lastsexid != $MAXid)
     { my ($sex) = &GetSQL($myODBC, "SELECT pdoGender FROM PERSDATA \
         WHERE persdata = $lastsexid", 'get recent sex');
       &DoSQL ($myODBC, "UPDATE PERSDATA SET pdoGender = $sex \
         WHERE persdata = $MAXid", 'set recent gender');
     };
  if ($firstNHIid != $MAXid)
     { my ($nhi) = &GetSQL($myODBC, "SELECT pdoSurname FROM PERSDATA \
         WHERE persdata = $firstNHIid", 'get recent hosp No');
       &DoSQL ($myODBC, "UPDATE PERSDATA SET pdoSurname = '$nhi' \
         WHERE persdata = $MAXid", 'set recent NHI');
     };
  # heck, or just perform a single update thus?:
  # "UPDATE PERSDATA set pdoSurname = '$nhi', \
  #      pdoGender = $sex, \
  #      SET pdoForename = '$foren'
  #      SET pdoSurname = '$surn'
  #      WHERE persdata = $MAXid" ...

  # MAXid is now an up to date reference. Fix all its 'referents':

  my ($epo) = &GetSQL($myODBC,
    "SELECT Epoch FROM PERSDATA WHERE persdata = $MAXid",
    'get epoch for persdata');
  my ($pro) = &GetSQL($myODBC,
    "SELECT Process FROM EPOCH WHERE epoch = $epo",
    'get process for epoch');
  &DoSQL($myODBC, "UPDATE EPOCH SET cold = NULL WHERE epoch = $epo",
    'warm up epoch');
  &DoSQL($myODBC, "UPDATE PROCESS SET cold = NULL WHERE process = $pro",
    'warm process'); # (type 2) [WHAT IF TYPE 1 ??]
  &DoSQL($myODBC, "UPDATE PERSON SET cold = NULL WHERE person = $person",
    'warm person');
```

```
  &DoSQL($myODBC, "UPDATE PERSDATA SET cold = NULL WHERE persdata = $MAXid",
    'NB also warm persdata'); # else fails catastrophically

  return ($old1, $old3);
}
```

17 PDB Retrieval

The opposite of writing PDBs (allowing them to be synchronised to the hand-held) is retrieving the information. We need to be careful here, making sure that we don't corrupt our database. Ideally, we should back-up the entire database on a regular basis, prior to hand-held synchronisation.[145]

This process of retrieval should involve:

1. Processing all modified tables, *in the correct order*. The correct order involves reading the tables in order of creation, as we'll then get the dependencies right. This order can be identified from the meta-tables documenting table creation. (In other words, we need to ORDER BY xTaKey within xTABLE). If a table hasn't been modified, it should not be read, and in addition there are certain tables we will not normally synchronise *from* the PDA, notably those defining menu structures!

2. Replacing temporary keys (which by our convention will be in the range 900 000 000 – 900 999 999) with permanent keys. Generally a temporary key will be replaced with an appropriately generated database key (under 900 000 000) but the one circumstance where we must be particularly careful is where a 'new person' has been entered on the PDA. Here we should ideally ensure that the hospital number is nowhere duplicated within the 'desktop' database; we might even go to the extent of looking for similar numbers using a single character check and then checking for surname identity, but this is probably overkill![146] In any case, we should validate a newly entered patient against the hospital database.[147]

As the row structures on the PDA and main database are identical, updates are trivial. The above suggests a single modification to our procedures in the preceding export section (Section 16). This is:

- Set up the generator table to create large keys (above 900000000). Reset all generator values to these large initial numbers with every synchronisation *to* the PDA![148]

[145] In addition, when we re-export back to the hand-held, we must only re-export information likely to be used, i.e. current patients, and perhaps those recently discharged from the pain service.

[146] In the case of NHIs, which have seven characters, this means seven single-wildcard-character searches, with follow-up confirmatory searches!

[147] Perhaps also against theatre lists, if the patient has undergone surgery!

[148] Subsequent to the update of the main database, we should *re*-synchronise *to* the PDA, sending small numbers for the 'final' generated keys created from the temporary ones originally supplied by the PDA!

We can then identify all rows generated on the PDA. If the key is big, the row needs updating, otherwise not.[149] A further convenient modification we might introduce is to identify whether a file was modified on the PDA, and if not, we don't hot-synch it. File synchronisation should be trivial, as we can simply store the date of last synchronisation, and then compare file dates to this. If they are greater, then we know the file has been altered, and we must examine it.[150]

17.1 Data integrity checking

There's a clear need to make sure that there has not been any loss of relational integrity in the PDA database. Every PDB file offset should thus be checked to see if it makes sense, and if there is even one error, some sort of error recovery should be performed. One possible approach is:[151]

1. Rollback the main database to the point where PDA import started;
2. Make a copy of all of the PDB files being synchronised, and ensure this copy is protected as write only;
3. Re-acquire all PDB files from the PDA (if possible) and compare these with the first lot to exclude a communication error. If different, perform a *third* acquisition, and where relevant, use a 'two out of three ain't bad' rule;
4. Mandate that the main database be backed up in full before the PDA synch/ error recovery proceeds;[152]
5. Try to repair the defect, with relevant user-prompting for sense.
6. Be particularly careful where referential integrity might be compromised;

Note that for the above to work optimally, we also need to implement CRC-32 checking in each database row on the PDA. (This hasn't been done yet).

[149] Hey wait a mo, what about *alterations* to rows? Well, remember our database structure? All EPOCHs are closed after a particular set has been stored, so even if somebody fiddles with a row, it *cannot* be re-synchronised back to the main database! If they generate a new key, well, then it has a new serial number, even if the timestamp is crooked! A security feature. Remember how alterations to important things like surnames and so forth are also stored as *observations*, so even here, updates are not an issue! **But** we still have a problem with at least one thing, and that is *closure of a PROCESS*. See section 17.1.2.

[150] But be careful that the relevant timestamps are correct!

[151] We don't yet do this!

[152] In the worst case, the user might be permitted to trash the entire PDA contents, but this is undesirable, to say the least!

17.1.1 Repair strategies

Where a single data field is defective, for example has the wrong length (integer, date or whatever), we should try to determine whether there are any orphan bytes corresponding to the missing/defective field, ie. whether a pointer is defective. If the field refers to the primary key of another table, make an intelligent guess as to what we are referring to, and get user confirmation.

Where a whole database row is defective, find all references to that row, and ensure referential integrity. Check on number of fields, possible impact on other rows, etc.

Where a whole table is defective, try to repair this intelligently, with user input. Check on number of rows, what prior data we have from this table, what we expected to change, etc.

Many data items (e.g. dates) have a limited range of likely values, and in addition have some internal redundancy.

We **must** always log each error, the corrective actions taken (with a time-stamp), and try to identify the mechanism of error, e.g. bad code design, attempts to alter data, random errors on the PDA (memory errors, cosmic radiation..?), corruption of data during communication, and so forth. It makes sense to once more acquire all relevant PDB files from the PDA, to eliminate the possibility that comms errors were the problem — if identical garbage comes through the second time, then we've excluded comms errors, so this repeat data acquisition from the PDA should always be performed if an error is found. Multiple backups should be made of error/corrective logs.

17.1.2 Other biig issues

WE NEED TO ADDRESS:

- PERSON.pDied
- PROCESS.rEnd

PERSON.pTokens could conceivably be an issue too, but we will *not* permit the user to alter their password on the PDA, eliminating one source of concern, keeping things simpler, and preventing some trickery. Each of the two items above is however a potential problem. We need to record when somebody dies, and we need to document process termination.

We could redesign our database to eliminate these problems. Many null-hating database experts would favour such an approach in the case of the PROCESS table, but I have two problems with recording process termination as a separate datum joined to the PROCESS table. These are:

1. Extra complexity is introduced (a join on another table) whenever we wish to test whether a process is active or not;

2. I find use of a NULL to indicate an active process attractive, simple, and appropriate. Nulls are not always bad.

Our solution is simply to set a flag (Bit 1 of the 4 flag bytes in the PDA database row)[153] and test for this in addition to the key value being 900000000 or more. If *either* the key value is large or the bit is set, we process the row.[154]

It would be possible to record the death of a person as an observation rather than directly altering the `pDied` field in the PERSON table, but such an approach, despite some merit, introduces an uncomfortable asymmetry between the PDA and main database. Note that detection of the death of a person when the main database acquires new data from the PDA might prompt several actions. Here's the ideal:

- This patient record is flagged for review;
- A human should confirm the death, and this fact should also be checked on other hospital databases and with the staff on the relevant ward;
- Ideally, the death certificate (and cause of death) should be obtained and recorded;
- At the next PDA synchronisation (once the death has been confirmed), all of the patient data should *not* be re-exported from the main database back to the PDA, and existing data for that patient should of course be removed from the PDA.

17.2 PDB file parsing

We've explored the PDB file structure in the previous section (16). In extracting new data, we employ similar strategies to writing the data, but things are a lot simpler. We commit the database, and then simply grab each new row from the relevant table, check it for sense, and thrust it into the main database. If an error is encountered, we might rollback and proceed as described above!

[153] Remembering that the lowest order bit bit 0, is used to signal absence of a CRC32!

[154] We will limit flag usage to selected tables, to prevent cheating! In other words, we will only check for the setting of this bit flag in the PERSON and PROCESS tables, and even there we will be restrictive in what changes we permit.

17.2.1 Main Resynchronisation routine

We need to perform several duties:

1. Import data files (PDB) from the PDA.
2. Resynchronise the database using the data in the PDB files.
3. Re-export the files back to the PDA. This involves rewriting the files to the */painform/export* directory and then exporting them to the PDA. The routine used is the same as the original installation routine, but PRC files are not re-exported.

Here's the code.

```
sub ResynchFromPda  # ???
{
  my ($myODBC);
     ($myODBC)=@_;

  # right at start, check for unresolved temporary keys in vital tables:
  # if any one is present in the PERSON table, fail:

  my $q = "SELECT count(person) FROM PERSON WHERE person >= 900000000";
  my ($c) = &GetSQL($myODBC, $q, 'ensure no remaining temp keys');
  if ($c == 1)
     {  $_ = &SlurpFile ('SQLRECENTFAILURE.DATA');
        print LOGFILE "\n\n Trying to fix PERSON temp key error using SQL: <$_>";
        &DoSQL($myODBC, $_, 'try to fix temp key');
        if ($SQLOK)  # iff success:
           { ($c) = &GetSQL($myODBC, $q, 'recheck temp keys');
             if ($c == 0)
                { &Commit($myODBC);
                  print LOGFILE "... success!";
                  # ideally should also delete SQLRECENTFAILURE.DATA here!
                };
           };
     };
  if ($c > 0)
     { print LOGFILE "\n\n ***Residual temp key errors*** = $c";
       &Alert($MAINW, "Error. There is/are $c unresolved temporary key(s) in \
     the database. Cannot resynch. Please call Jo on 021 385 927 and discuss!");
     return (0);
     # with Ocelot the problem will likely be with "update person set person" ...
     };

  # first, import from external db (unless have done so within 1 hr):
  ExtDbImport ($myODBC, 1); # allow skip if recent...
```

```
  my $expdir = $CONST{EXPORTDIR}; # usually  /painform/export
  if (! &ValidatePath($expdir))
   { &Alert($MAINW, "Error: export directory $expdir does not exist");
     return;
   };
  my $importdir = $CONST{IMPORTDIR}; # usually  /painform/import
  if (! &ValidatePath($importdir))
   { &Alert($MAINW, "Error: import directory $importdir does not exist");
     return;
   };

  my $pbat;
  $pbat = $CONST{BATFETCHFROMPDA};
  $pbat = "$pbat $expdir $importdir";
  $pbat =~ s/\//\\/g; # replace slash with backslash for DOS
  # 'source' is first argument, 'destination' 2nd:
  # we will ultimately retrieve PDA files into the 'destination' $importdir.
  # $expdir is used as a source of template files!
 my $r = `$pbat`; # system call using backticks.
 if ($r =~ /Something went wrong/im)
    {  &Alert ($MAINW, "Error in import from PDA. Aborted.");
       return; # fail
    };

# ============================================================
# HERE IMPORT DATA:
  my $opt = &ImportPdbData($myODBC, $importdir);
  if (!$opt) # ugly.
     { &Alert($MAINW, "Import failed ($opt)");
       return;
     };
#  &Alert($MAINW,  "Imported $ok \
#  \n We now send data BACK to the PDA \n to keep it in synch! \
#  \n(Preparation may take two minutes). Click on OK to do so!");
#  &Commit($myODBC); # only commit if no error

# ============================================================
# The following is a hack to prevent process duplication.
# WE MUST STILL EXAMINE DUPLICATION MECHANISMS IN GREAT DETAIL.

  &Test_Fixup($myODBC);

# ============================================================
# NEXT, EXPORT AGAIN!:
  my $localdir = $CONST{'LOCALDATADIR'};
  &MakeAllPDBs ($myODBC, $expdir);
  $pbat = $CONST{BATWRITETOPDA};
```

```
    # batch file accepts %1 as the first argument
  $pbat = "$pbat $expdir";
  $pbat =~ s/\//\\/g; # replace slash with backslash for DOS
 $r = `$pbat`;
 # We then look for the string 'Something went wrong'
 if ($r !~ /Something went wrong/im)
    { $r = "Success! ($opt) On PDA please click 'Abort+Disconnect', then press 'Ho
      &Commit($myODBC);  # NB commit
    } else
    { $r = "Ooops. An error occurred. Files NOT transferred";
    };
 &Alert ($MAINW, $r);
}
```

17.3 Test and Fix

The following is an important routine that scans for 'trouble'. It is invoked from within ResynchFromPda and ExtDbImport, that is, whenever there has been external importation of data. Despite countermeasures, there is the potential for duplicate processes to occur, for example if data about a person have been entered in separate locations. In addition, if data are entered on PDA 'A' and 'B', and the patient is discharged from 'A' but this PDA is synchronised before 'B', orphan processes or other data may be recorded.

```
sub Test_Fixup
{ my ($myODBC);
     ($myODBC)=@_;

   print LOGFILE "\n\n ***FIXING DUPLICATE PEOPLE***";
   # here must check for duplicate person VIA duplicate NHI
     # NB. do NOT go further if the pdoPerson value is the same. We want same
     # NHI but distinct person!
   # if so, must:
   #   a. replace all references to the duplicate person with the first entry!
   #   b. archive the new entry
   #   c. log what we've done.
   &FixDuplicatePeople($myODBC);

   print LOGFILE "\n\n ***FIXING DUPLICATE PROCESSES***";
   # and ensure that duplicate type 1 or type 3 process hasn't been made?!
   # if we identify a new type 1 or type 3 process, we MUST:
   #   a. Attach all references to this new process to the oldest one;
   #   b. archive the new process
   #   c. log our doings.
   &EschewTwins($myODBC);  # hide ended type 1 processes.
   &FixDuplicateProcs ($myODBC, 1);    # observations
```

```
   &FixDuplicateProcs ($myODBC, 3);    # admissions
   &FixDuplicateProcs ($myODBC,110);   # epidurals!
   &FixDuplicateProcs ($myODBC, 390);  # PCA

   print LOGFILE "\n\n ***FIXING DUPLICATE BEDS***";
   # If duplicate bedspaces exist [hack] then hide all but most recent!
   &FixDoubleBeds($myODBC);
   ## &Commit($myODBC); # no. only in debug.
}
```

In the above we might also scan through for bed entries still active on patients who have already been discharged. This will occur if entries are made on a PDA synchronised after someone has been discharged and that PDA has been resynchronised!

FixDoubleBeds

Here scan through all BADOBS entries for duplicate, active bedspace observations. If the same person has two active entries (that is, cold is null and boInactive is null for each entry), then we inactivate all but the most recent.[155]

```
sub FixDoubleBeds
{
  my ($myODBC);
     ($myODBC) = @_;

  my @DBLBED;
  my $pers;

  (@DBLBED) = &SQLManySQL($myODBC,
  "SELECT Person FROM BADOBS WHERE cold IS NULL AND boInactive IS NULL GROUP BY Pe
  'get double bed entries');

 foreach $pers (@DBLBED)
   {
     my (@DBL) = &SQLManySQL($myODBC,
      "SELECT badobs FROM BADOBS WHERE cold IS NULL AND boInactive IS NULL AND per
      'get bed entries');
      while ($#DBL > 0)  # ignore LAST entry
        {
          my ($bo) = shift(@DBL);
```

[155] Note that a problem might arise where the most recent entry is not temporally the most recent, for example where someone has been walking around with the PDA in their pocket for ages, recording an older entry, and then synchronises after a more recent movement. It would be best to research this possibility. A solution might be to SELECT badobs from a join of BADOBS on EPOCH, and order by CAST(oMade AS VARCHAR(19)) to get the most recent bed event.

```
           print EDLOG "\n Fix dup BED observation: $bo";
          &DoSQL ($myODBC,
           "UPDATE BADOBS SET boInactive = 1 WHERE badobs = $bo",
           'clear extra entry');
          print LOGFILE "\n EXTRA BADOBS CLEARED: $bo";
        };
   };
  # &Commit($myODBC);  # No. not yet!
}
```

We should also here scan for a BADOBS entry attached to someone who has already been discharged, rather than relying on a later pre-export scan.

FixDuplicatePeople

```
sub FixDuplicatePeople
{ my ($myODBC);
     ($myODBC) = @_;

   my (@DUPNHI) = &SQLManySQL($myODBC,
     "SELECT pdoHospNo FROM PERSDATA WHERE pdoPerson > 999 \
      GROUP BY pdoHospNo HAVING count(pdoHospNo) > 1",
     'GET multiple NHIs');  # [could do with some work: speed issues here]

   my $nhi;
   foreach $nhi (@DUPNHI)
     {
       my (@DUPPERS) = &SQLManySQL($myODBC,
        "SELECT DISTINCT pdoPerson from PERSDATA WHERE pdoHospNo = '$nhi' ORDER BY
        'get people for a duplicated NHI ??');
       my $mainp = shift(@DUPPERS); # get primary person!
       while ($#DUPPERS > -1) # if any duplicates
          {
            my $dupp = pop (@DUPPERS);  # get new person ref
            print EDLOG "\n Fix duplicate pt: $dupp -> $mainp";
            # we might have a more global continuous error log too [2008-11-01]

            # now, update all references using $mainp:
            &DoSQL($myODBC,
             "UPDATE PROCESS SET Person = $mainp WHERE Person = $dupp",
             'refer PROCESS to older person');
            &DoSQL($myODBC,
             "UPDATE BADOBS SET Person = $mainp WHERE Person = $dupp",
             'refer BADOBS to older person');
            &DoSQL($myODBC,
             "UPDATE PERSDATA SET pdoPerson = $mainp WHERE pdoPerson = $dupp",
             'refer PERSDATA to older person');
            &DoSQL($myODBC,
```

```
            "UPDATE PERSON SET cold = 1 WHERE person = $dupp",
            'retire duplicate person');
         };
    };
}
```

Note that if the database is restructured with addition of extra tables that reference a patient (as PERSON) then these tables too would have to be updated by FixDuplicatePeople.

FixDuplicateProcs

In the following, we *must* ensure that the 'fixed' items are not cold, otherwise we will harm our database.

```
sub FixDuplicateProcs
{ my ($myODBC, $procTYPE);
     ($myODBC, $procTYPE) = @_;

   my (@DUPPT) = &SQLManySQL($myODBC,
     "SELECT Person FROM PROCESS WHERE cold IS NULL \
       AND procType = $procTYPE \
       AND process > -1 \
       AND rEnd IS NULL \
       GROUP BY Person HAVING count(Person) > 1",
     "GET duplicate type $procTYPE entries");
   # hmm. what about type 1 process which is not cold but has rEnd completed??

   my $pt;
   foreach $pt (@DUPPT)
     {
       my (@DUPPRO) = &SQLManySQL($myODBC,
        "SELECT process from PROCESS WHERE \
          cold IS NULL \
          AND rEnd IS NULL \
          AND process > -1 \
          AND procType = $procTYPE AND Person = $pt ORDER BY process",
        "get dup type $procTYPE procs for a patient");

       # THE FOLLOWING IS A PROBLEM [2008-11-01]:
       # perhaps we should take _most recent_ process!
       # particularly with the type 3 (aagh, what??)
       # perhaps: my $mainp = pop(@DUPPRO);
       my $mainp = shift(@DUPPRO); # get earlier process
       while ($#DUPPRO > -1) # if any duplicates
          {
            my $dupp = pop (@DUPPRO);  # get new proc
```

```
            print EDLOG "\n Fix proc dup: $dupp -> $mainp";
            # now, update all references using this proc
            &DoSQL($myODBC,
             "UPDATE EPOCH SET Process = $mainp WHERE Process = $dupp",
             'fix EPOCH proc');
            &DoSQL($myODBC,
             "UPDATE RX SET Process = $mainp WHERE Process = $dupp",
             'fix RX proc');
            # &DoSQL($myODBC,
            # "UPDATE STOPPROC SET Process = $mainp WHERE Process = $dupp",
            # 'fix STOPPROC proc');  # Rem this out as should never happen!

            # We should get rid of the following, simply NEGATING the process ID
            # [2008-11-01: fix later]
            &DoSQL($myODBC,
             "UPDATE PROCESS SET cold = 1, rEnd = TIMESTAMP '1900-01-01 00:00:00'
             'retire duplicate proc');
          };
     };
}
```

There was a problem in the above. If we retire a process and don't set the rEnd value, then this causes no end of problems later on, especially with type 1 and type 3 processes. We rather clumsily flag these troublesome processes by setting the rEnd value to a ridiculous one (1900-01-01 00:00:00). Better will be to negate the primary key value, but check the implications of this. Would we ever need to cascade, (surely not as we've detatched everything) and what would happen if this failed?

EschewTwins

The following routine, EschewTwins is much more specific. It looks for and eliminates all type 1 processes which have an rEnd value, altering them to type 2. This routine is only ever called from Test_Fixup.

```
sub EschewTwins
{ my ($myODBC);
     ($myODBC) = @_;

   &Do2SQL ($myODBC,
    "UPDATE PROCESS SET procType = 2 WHERE \
      procType = 1 \
      AND rEnd IS NOT NULL",
    'retire old type 1 procs');
    # that's it!
}
```

The idea is that we cannot have more than 1 type 1 process. If it's ended, then we have no need for it, so get rid of it!

17.3.1 ImportPdbData

Here's the subsidiary routine ImportPdbData. The routine reads the manually created file *XLOG.LST* for all of the filenames. It then checks the date of each file (locating the file in the relevant subdirectory, LOCALDATADIR), and if the date is after the last recorded synchronisation date, updates the database.

Note that the last synchronisation date is a single number (local system epoch date) stored in the file *LASTSYNCH.TIME* which should be present within the local directory LOCALDATADIR, together with the PDB files. This date should be written to the file, after obtaining it using `time()`, every time PDB files are written *from* the main database to that directory![156]

When we resynchronise, we translate *all* temporary keys from their PDA value to a *new* desktop value. The only exception to this rule is where the PDA key is already represented on the desktop in the case of the key for a patient who has already been admitted in the past! This case is easily identified because it is only under this circumstance that the patient ID key on the desktop [??] will be a 999 nnn nnn number. If encountered, such a number should be replaced by the previous number, or (if no number is found) by a generated number in the usual fashion. Comparison will be on the basis of the NHI; it is attractive to cross-check with the name, and ultimately to integrate this check with intranet databases or even national ones!

When a temporary key is first encountered (always as the key at the start of a row, unless an error has been made in table processing order), a new permanent desktop key is generated and the association is stored in the array KEYHASH. Future encounters are resolved by an associative lookup in KEYHASH.

```
sub ImportPdbData
{ my ($myODBC, $localdir);
     ($myODBC, $localdir) =@_;

  $PROGTEXT = 'Importing PDB data..';
  $BARPROGRESS = 0;
+OPTIONAL
  $PROGRESSBAR->update();
-OPTIONAL
```

[156] We should probably add checking routines which, at the very least, look for improbable dates based on previous dates used. E.g. if the date skips a week or moves back in time, the user should confirm the change! Such confirmation should be logged. Ideal would be to add frequent synchronisation via intranet or Internet with a reliable time source.

```
  %KEYHASH = ();
  @RETAINEDPROCS = ();
  $DebugRECNO = 0;
  my $XLIST = 'XLOG.LST';
  my $errcount = 0;

  print SQLLOG "\n\n--IMPORTING DATA: Source $localdir";
  my $now = ctime();
  my $ltime = time(); # epoch time (raw)
  print SYNCFILE "\n Current time is $now ($ltime)";

  my $LS = "$localdir" . "LASTSYNCH.TIME";
  my $lastsynch;
  my $fail = 0;

  open LS, $LS
    or $fail = 1;
  if ($fail)
     { &Alert ($MAINW,  "WARNING: No last synch time ($LS)! :$!\n");
       $lastsynch = 0;
     } else
     {  <LS>; # waste first (comment) line
         $lastsynch = <LS>;
        chomp $lastsynch; # might also ensure numeric?
        close LS;
     };
  $_ = ctime($lastsynch);
  print SYNCFILE "\n Last synch (+10s) at: $_ ($lastsynch)";

  my $ok = 1;
  open XLIST, $XLIST or $ok = 0;
  if (! $ok)
     { &Alert ($MAINW, "*CRASH* Can't open $XLIST :$!\n");
       return 0;
     };

    <XLIST>;       #discard first line
  my $more = 1;
  my $PDBFILE;

  while ( $more )  # till end
    { $_ = <XLIST>;
      $BARPROGRESS += 7;
+OPTIONAL
      $PROGRESSBAR->update();
      chomp $_;
-OPTIONAL
```

```
     if ( /^\*/ ) # last line
        { $more = 0;
        } else
        { if (  (! /\%/ )
              &&( length $_ > 2)
             ) # length ok, not commented out
             { print SYNCFILE "\n\n Filename: $_";
               # &Alert($MAINW, "Value is <$PDBFILE>");
               $PDBFILE = $_;
               my $myage = stat("$localdir/$PDBFILE")->mtime;
               $_ = ctime($myage);
               print SYNCFILE " ($_)";

               if ($myage > $lastsynch) # of little utility
                  { $errcount += &SyncOneFile($myODBC,
                        $localdir, $PDBFILE, $lastsynch);
                   # debugging: [2007-10-23 13:52]
                   print SYNCFILE ("\n File: $PDBFILE. err count=$errcount");
                   # errcount = 0; # [HACK TO ALLOW CONTINUATION] [2007-10-23 14:
                  } else
                  { print SYNCFILE " *skipped* ";
                  };
             };
        };
   };
close XLIST;

 $errcount += &FixPERSONkeys ($myODBC);
 $errcount += &FixRetainedProcs ($myODBC);

my $okstring;
$okstring = "$DebugRECNO lines";

 # debug [2007-10-23 13:49]
 # print SYNCFILE "\n ERROR COUNT ($errcount)"; # [2007-10-23 13:22]
 # &Alert($MAINW, "Error count is $errcount");

 if ($errcount > 0)
    { $okstring = "Found $okstring, Errors = $errcount. \
 \n Sync terminated due to errors. \
 \n Discuss this with an expert before ANY \
 \n further attempts at synchronisation!";

     &Alert($MAINW, $okstring);
      return 0;
    };

 print SQLLOG "\n\n --- Fixing cold cases ($okstring)"; # [2007-10-23 13:22]
```

```
 &MakeCold($myODBC); # fix up cold cases! THIS COMMITS

 $BARPROGRESS = $BARWIDTH;
+OPTIONAL
 $PROGRESSBAR->update();      chomp $_;
-OPTIONAL
  print SYNCFILE "\n Cold cases done. ($okstring)"; # [2007-10-23 13:22]

 return $okstring; # success: return a comment (cannot be "0")
}
```

The FixPERSONkeys routine is described below (Section 17.5.1).

There is potential for this routine to malfunction if the system time is altered on the desktop following the last PDB write, or there are date/time problems with the files.[157]

At the end, the invocation of MakeAllPDBs does just that, writing the PDB files to $localdir.

Finally, we use the Perl *system* function to invoke the external *PInstall.exe* program (Pilot install), installing PDBs on the PDA.[158] Here the invocation specifies the full path (from root) of a Windows batch file which will need to be modified in CONSTANTS.CONST depending on where PainForm has been installed. An appropriate sub-directory also needs to be created in the painform directory. As things stand, *export.bat* clumsily copies all of the requisite files from the PDA backup directory to this sub-directory *\painform\export*.

17.4 Importing one PDB file

Here, given the path (and name) of a single PDB file (in the variable PDBFILE), presumably one obtained from the handheld, we go through the entire file and pull out any new data.

This process involves:

1. Parsing the PDB header and extracting the offset of each record;
2. Obtaining the first record, which contains the names of the database columns, and their characteristics. We have details of data items we will write to the main database. The details we will use are the type of item, and its name;
3. Using the above information to parse each database row, and write it to the main database.

[157] We might check for such issues.

[158] Similar functionality can be achieved under Linux using pilot-xfer.

There is some trickery with certain tables: PROCESS, PERSON and BADOBS. This is because it is possible with these tables (and these tables only) for the hand-held user to add information to existing rows. The information is respectively PROCESS termination, the death of a PERSON, and a change in the active status of a BADOBS bedspace observation — these require special handling. After some pre-processing, where $uidx and $ucname are used to flag this special state, ParseWriteRow does the actual database write.

```
sub SyncOneFile
{ my($myODBC, $localdir, $PDBFILE, $lastsynch);
    ($myODBC, $localdir, $PDBFILE, $lastsynch) = @_;

  my $pdbok = 1;
  my ($hPDB);
  open ($hPDB, "$localdir/$PDBFILE") or $pdbok = 0; #hPDB is handle
  if ($pdbok)
     { binmode ($hPDB);
       my ($keepterm) = $/; # retain
       undef $/;            # undefine
       my ($wholefile) = <$hPDB>; # slurp
       $/ = $keepterm;      # restore
       close ($hPDB);

       my $flen = length $wholefile;
       print SYNCFILE " Size=$flen";

       my ($numrecs, $toprec);
       $PDBFILE =~ /(.+)\.PDB/; # has .PDB suffix
       $PDBFILE = $1;           # clip this off
       ($numrecs, $toprec) = &CheckPdbHeader ($PDBFILE,
                             $wholefile, $lastsynch);
       if (! $numrecs)
          { return 1; # fail
          };

       my @colmdata;
       my $colnamestring;
       ($colnamestring, @colmdata) = &ParsePdbCols($wholefile,
                                                   $toprec);
       # note @colmdata contains both types and scales!

       my @rowdata;
       @rowdata = &GetPdbRows($wholefile, $numrecs-1, $toprec+8);

       my $uidx;
       my $ucname; # update column name.
       if ($PDBFILE eq 'PROCESS')
```

```
          { $ucname = 'rEnd';
            $uidx = GetUIndex($colnamestring, $ucname);
          }
        elsif ($PDBFILE eq 'PERSON')
          { $ucname = 'pDied';
            $uidx = GetUIndex($colnamestring, $ucname);
          }
        elsif ($PDBFILE eq 'BADOBS')
          { $ucname = 'boInactive';
            $uidx = GetUIndex($colnamestring, $ucname);
          }
        # we have a potential problem with boFlag: should we also
        # update this on the desktop from the PDA [NO!]
        else
          { $ucname = '';
            $uidx = 0; # signal 'do not use'
          };


        print SYNCFILE
          "\n column data(@colmdata), names: $colnamestring";
        $colnamestring =
          "INSERT INTO $PDBFILE ($colnamestring) VALUES ";

        my $rw;
        foreach $rw (@rowdata)
          {
            if (! &ParseWriteRow ($myODBC, $PDBFILE,
              $wholefile, $rw, $colnamestring, $uidx, $ucname, @colmdata))
               { print SYNCFILE "\n Premature end: ERROR";
                 return 1; # fail, will increment error count
               };
            $DebugRECNO ++; # bump record count
          };
        return 0; # no error
      } else
      { print SYNCFILE "\n *Failed to open file, <$PDBFILE> $!";
      };
  return 1; # error
}
```

In the above we play around with a more modern approach to file handles.[159]

Here's GetUIndex, which given the name of a column, returns the index of this column in the comma-delimited list *colnamestring*.

```
sub GetUIndex
```

[159] Initially we contemplated passing the handles to several routines, but eventually chose the slurp option.

```
{
  my ($colnamestring, $ucname);
     ($colnamestring, $ucname)=@_;

  $colnamestring = "$colnamestring,"; # add comma
  my $cc = 0;
  while (length $colnamestring > 0)
    { $colnamestring =~ /^(.*?),(.*)/;
      if ($1 eq $ucname)
         { return ($cc);
         };
      $cc ++;
      $colnamestring = $2;
    };
 return 0; # fail.
}
```

Here follow three important routines mentioned above: ParsePdbCols, GetPdbRows and ParseWriteRow. The CheckPdbHeader routine is explored later (Section 17.4.2).

First, column parsing. We pass the whole file, together with the offset of the first eight-byte record descriptor. This record descriptor comprises a four-byte 'local chunk ID', which, shorn of pretension, is the offset of the contents of the record from the *start* of the PDB, and thus the offset in `wholefile`. Following this is a single attribute byte, and then a three-byte unique PalmOS ID.

```
sub ParsePdbCols
{ my ($wholefile, $toprec);
     ($wholefile, $toprec)=@_;
  my @mycols = ();

  my($localoff);
  $localoff = &GetDoubleBe ($wholefile, $toprec);
  print SYNCFILE "\n Local offset: $localoff";
  if (($localoff < $toprec+8) || ($localoff > length $wholefile))
     { print SYNCFILE "\n Error: bad offset = $localoff";
       return @mycols; # fail
     };
  # more correct would be "< toprec+(numrecs*8)
  # the next is for interest's sake:
  my ($nums);
  $nums = &GetDoubleBe ($wholefile, $toprec+4);
  print SYNCFILE "\n Numeric codes: $nums";
```

Next, we enter *our* formatting scheme as we pull out details of all of the columns. The record starting at `localoff` is formatted according to Table 8.

At present we have not implemented the CRC32. We might also (but don't) check the internal byte length with 'reality'.

All we do for now is:

1. Read the (2 byte) byte length `headsize` at offset +6;
2. Read the 2 byte column count `ccount` at offset +0E hex;
3. Sequentially access the column descriptors, by reading their 2-byte offsets starting at offset +10 hex. There are cc $+ 1$ offsets, allowing us to find the length of the last descriptor.
4. At each stage we must check that the values make sense! We store three values for each column: the type and scale are pushed onto `mycols` in that order, and the column names are concatenated into a string `colnamelist`, separated by commas.

```
my $headsize;
my ($cc, $ccount, $colPoff, $colmdesc);
my ($coltype, $colname, $colnamelen, $colnameoff, $scale);
my ($tophead, $minhead);
my ($colnamelist);
$cc = 0;
$colnamelist = '';

$headsize = &GetBeWord($wholefile, $localoff+6);
$ccount = &GetBeWord($wholefile, $localoff+0x0E);
$colPoff = $localoff + 0x10;
$tophead = $localoff + $headsize;
$minhead = $localoff + 0x10 + 2*$ccount;
print SYNCFILE " top=$tophead columns=$ccount";

while ($cc < $ccount)
  {
    $colmdesc = $localoff + &GetBeWord($wholefile, $colPoff);
    if (($colmdesc > $tophead)||($colmdesc < $minhead))
       { print SYNCFILE
           "\n ** Error: bad pointer in header ($colmdesc)";
         return (); # fail
       };

    $coltype = &ClipText($wholefile, $colmdesc+6, 1);
    $scale = ord (&ClipText($wholefile, $colmdesc+7, 1));
    $colnameoff = &GetBeWord($wholefile, $colmdesc + 0);
    $colnamelen = &GetBeWord($wholefile, $colmdesc + 2);
    $colname = &ClipText ($wholefile,
```

```
                    $colmdesc + $colnameoff, $colnamelen);


      print SYNCFILE
         ( "\n  name=<$colname> type=$coltype at:$colmdesc ");

      push (@mycols, $coltype);
      push (@mycols, $scale);
      $colnamelist = "$colnamelist$colname,";

      $colPoff += 2;
      $cc ++;
    };

  chop($colnamelist); # remove terminal comma!
  return ($colnamelist, @mycols);
}
```

Here's the subsidiary GetPdbRows routine:

```
sub GetPdbRows
{ my ($wholefile, $reccount, $thisrec);
     ($wholefile, $reccount, $thisrec)=@_;

  my @offsets = ();
  my($minoff, $cnt, $flen);
  $minoff = $thisrec + 8*$reccount;
  $flen = length $wholefile;
  $cnt = 0;

  my($localoff);
  while ($cnt < $reccount)
    {
      $localoff = &GetDoubleBe ($wholefile, $thisrec);
      push (@offsets, $localoff);
      print SYNCFILE "\n Local offset ($cnt): $localoff";
      if (($localoff < $minoff) || ($localoff > $flen))
         { print SYNCFILE " * Error: bad offset($cnt) $localoff!";
           return (@offsets); # fail
         };
      $cnt++;
      $thisrec += 8;
    };
  return (@offsets);
}
```

In ParseWriteRow, we explore a database row in a PDB file, extracting values and writing them to the main database.

The routine accepts an ODBC handle, the name of a PDB FILE, a pointer to the contents of that whole file, the offset of a row ($rw), a comma-separated string of column names in the correct order ($insstring), two variables sometimes used in updating files rather than writing a new row ($uidx, $ucname), and an array of information about columns.

We use this last-mentioned `colmdata` array to tell us how to process each datum we obtain from `wholefile`. Our starting offset in `wholefile` is the numeric value contained in `rw`. Our data row format is contained in Table 11. At present we don't use the CRC32.

ParseWriteRow returns 1 on *success*, 0 on failure.

```
sub ParseWriteRow
{ my ($myODBC, $PDBFILE, $wholefile, $rw, $insstring, $uidx,
      $ucname, @colmdata);
     ($myODBC, $PDBFILE, $wholefile, $rw, $insstring, $uidx,
      $ucname, @colmdata)=@_;

  my ($ccount, $cc, $ok);
  $ccount = int((1+$#colmdata)/2);
  $cc = 0+1; # +1 past primary key
  my ($rowptr, $rowPitem);
  $rowptr = $rw;
  my ($rowsize, $rowmax, $rowkey, $rowPitm, $rk);
  $rowkey = &GetDoubleBe($wholefile, $rowptr+8);
  $rk = -1;
  # now have primary key
  # for later, get the flags:
  my ($uflags);
  $uflags = &GetBeWord($wholefile, $rowptr+4);
```

There's another catch — if we are dealing with an *updated* row, the row key may well be an existing value, and we will have to update the row, not insert a new row. However, we must first check for a new row key (value over 900 million) and then only if this fails, check for an updated row (bit 1 of uflags is set).

```
  my ($retain) = 0;
  my ($testk);
  if ($rowkey > 900_000_000) # new item
     {
       $rk = $rowkey; # retain temporary value
       if ($PDBFILE eq 'PROCESS')
          { $testk = &ScanDuplicates($myODBC, $wholefile, $rw);

          # New response to the above: v0.95 2008-02-24 21:42
            if ($testk > 0) # if duplicate found
               { $KEYHASH{$rk} = $testk; # establish association,
```

```
            return (1);             # and discard (ok).
         };

       if ($testk < 0) # signal to retain process
         { $retain = 1;
         };
   };
if ($PDBFILE ne 'PERSON')
   { $rowkey = &AutoKey($myODBC, $PDBFILE); # use table name
   }; # see comments below!
$KEYHASH{$rk} = $rowkey;
if ($retain) # only retain dubious processes:
   { push (@RETAINEDPROCS, $rowkey); # for later check!!
   };
```

In the above we obtain a permanent key value to substitute for the temporary number in $rowkey, retaining the association between the two in the associative array KEYHASH.[160]

There is a problem with new unique IDs for patients *entered on the PDA*. It is conceivable that a 'new' patient may already exist in the desktop database.[161] We cannot check the NHI when we first encounter the patient in the PERSON table, as the NHI will only enter our ken later, so we *retain* the temporary key and only update *after* we've completed our update. During this final update, we will either replace a temporary key with a new key or (if the key already exists), with an existing key. Because we wisely divided the whole of the keyspace from 900 million up *among* participating PDAs, these temporary keys should not collide when we finally implement a program where multiple simultaneous synchronisations can be performed by different users.

We however still have a problem, because when we now *look up* the 'permanent' value of a 900-key representing a patient, we won't have one! Or will we — we sneakily make the association but retain the original key value!

A final problem is addressed by ScanDuplicates: it is conceivable that a duplicate process might have been entered on one of multiple PDAs (for example, a duplicate operation, or a duplicate PCA process). If such a process is present, we must accommodate it by making all references *to it* (EPOCHs, etc) refer to the original process. If ScanDuplicates couldn't establish whether a process is a duplicate or not, the *new* process ID is stored in the RETAINEDPROCS array.

```
$rowsize = &GetBeWord($wholefile, $rowptr+6);
```

[160] For extra-paranoid security, we might even check that there is no collision in KEYHASH, and that the temporary value corresponds to the correct table, but we don't go to these extremes!

[161] We must still research duplicate entries entered on two PDAs and synched more or less at the same time!

```
$rowmax = $rowptr + $rowsize;
$rowPitem = $rowptr + 0x10+2;
# we add 2 to skip past primary key!!
print SYNCFILE ( "\n  at $rw($rowsize) k=$rowkey {$rk} ->");
my ($typ, $nam, $lasto, $nexto);
my $values = '';
$nexto = $rowptr + &GetBeWord($wholefile, $rowPitem);
  # 1st offset

shift (@colmdata); # waste first type
# (might check it is indeed 'I')
shift (@colmdata); # also waste scale (0)
my ($i, $scale);

while ($cc < $ccount)
  { $typ = shift (@colmdata);
    $scale = shift (@colmdata);
    $rowPitem += 2;
    $lasto = $nexto;
    $nexto = $rowptr + &GetBeWord($wholefile, $rowPitem);
    ($i, $ok) = &ReformatDatum ($wholefile, $lasto,
                          $nexto-$lasto, $typ, $scale);
    print SYNCFILE ( " $typ,$nam<$i>");
    $values = "$values$i,";
    $cc ++;
    if (! $ok)
       { return 0; #fail
       };
  };
chop ($values); # remove terminal comma
$values ="$rowkey,$values"; # prepend primary key

my ($insstmt);
$insstmt = "$insstring ($values)";
print SYNCFILE "\n $insstmt"; # [debug only]
```

In the above it is best to return a success value ('ok') from ReformatDatum in addition to the i value, and fail completely on failure, notably if relational integrity is compromised.

In the following section we must submit the SQL INSERT statement to the database[162] The major problem is with temporary keys.

```
  &Do2SQL ($myODBC, $insstmt, "");
  return 1;
}
```

[162] Advanced features of ODBC might be used to speed things considerably, but for now we'll be simple.

Here's the resolution of our problem with the updated row:

```
  elsif ($uflags & 0x2) # updated row?
  {
    print LOGFILE "\n Debug: Update row $rw index $uidx($ucname) in table $PDBFILE

    # here must perform update. For now only affects 1 field in each
    # of two tables:
    # PROCESS : rEnd
    # PERSON  : pDied

    my $updcmd;
    my $uvalu;
    if ($uidx > 0)
       { $uvalu = &FetchUpd($uidx, $rw, $wholefile, $colmdata[2*$uidx],
                            $colmdata[1+2*$uidx]);
         if (
             (length $uvalu < 1)
              || ($uvalu =~ /^NULL$/i )
              # do NOT allow value to be replaced by null !!
              # NB. if alter DB to allow fetching of boInactive, then
              # must also alter the above stmt to permit fetch of boInactive as NU
            )
            { print SYNCFILE "\n NULL update avoided: ($PDBFILE($ucname) : row $ro
            } else
            { $updcmd =
              "UPDATE $PDBFILE SET $ucname=$uvalu WHERE $PDBFILE = $rowkey";
              # By our convention, primary key has same name as file!
              print SYNCFILE "\n Debug special UPDATE: <$updcmd>";
            };
       }
    elsif ($PDBFILE eq 'UIDS')
       { return 1; # ignore, but don't fail!
       } else
       { print SYNCFILE "\n Error! Bad UPDATE on file <$PDBFILE>";
         return 1; # fail  2008-12-03 TEMP AMENDMENT: return 1. was 0.
       };
    &Do2SQL ($myODBC, $updcmd, 'Special update');
  };
  return 1; # success.
}
```

Here's FetchUpd, which given an index of a column pulls out the column data:

```
sub FetchUpd
{ my ($uidx, $rw, $wholefile, $typ, $scale);
     ($uidx, $rw, $wholefile, $typ, $scale) = @_;
```

```
#  &Alert($MAINW, "Debug: index is $uidx, type is $typ, scale $scale");
  my $rowP = $rw + 0x10 + 2*$uidx;
  my $thisitem = $rw + &GetBeWord($wholefile, $rowP);
  my $nextitem = $rw + &GetBeWord($wholefile, $rowP+2);
  my ($i, $ok);
     ($i, $ok) = &ReformatDatum ($wholefile, $thisitem,
                                 $nextitem-$thisitem, $typ, $scale);
  if (! $ok)
     { return '';
     };
  return ($i);
}
```

17.4.1 Datum Reformat

The idea here is to take an encoded datum and turn it into an ASCII string. We submit the whole data file, an offset, the datum length (which may be zero for null items), and the type of item.

```
sub ReformatDatum
{ my ($wholefile, $offs, $datlen, $typ, $scale);
     ($wholefile, $offs, $datlen, $typ, $scale) = @_;
  my ($dat);

  # first, if null length, return null
  if ($datlen <= 0)
     { return ('NULL',1); # ok
     };

  # here read datum depending on type:
  if ($typ eq 'V')  # varchar
     {
       $dat = &ClipText ($wholefile, $offs, $datlen);
       if ($dat =~ /'/)
     # might check for other database funnies eg pipes, backticks???
     # [CHECK ME]
          { $dat =~s/'/''/g; # double each one! Hmm. See FancyUp, etc?
          };
       return ("'$dat'", 1); # ok
     }
  elsif ($typ eq 'I')
     {
       if ($datlen != 4)
          {
            print SYNCFILE ("\n!BAD INT LENGTH($datlen)");
            return ('NULL', 0); # fail
          };
       $dat = &GetDoubleBe ($wholefile, $offs);
```

```
      # If temp key, look up final value:
      if ($dat >= 900_000_000)
         {
           if (exists $KEYHASH{$dat})
              {
                print SYNCFILE "{$dat}";
                $dat = $KEYHASH{$dat};
              } else
              { print SYNCFILE ( "\n *** ERROR! Loss of relational integrity \
(temporary key $dat NOT known)" );
              &Alert($MAINW, "Relational problem! datum = $dat. Please tell Jo!")
                return (’NULL’, 0);  # fail [restored v0.95 2008-02-24 21:05]
               #  return (0, 1);  # [temporary hack 2007-10-23 14:14]
              };
         };
      return ($dat, 1);
   }
elsif ($typ eq ’N’)
   {  $dat = &ClipText ($wholefile, $offs, $datlen);
      $dat = &FormatNumeric($dat, $scale);
      return ($dat, 1);

   }
elsif ($typ eq ’D’)
   {
     if ($datlen != 8)
        {
          print SYNCFILE (" ! Bad length of date ($datlen)");
          return (’NULL’, 0); # fail
        };
      $dat = &ClipText ($wholefile, $offs, $datlen);
      $dat = &FormatDate($dat);
      return ("DATE ’$dat’", 1); # ok

   }
elsif ($typ eq ’T’)
   {
     if ($datlen != 6)  # [not: 12) ]
        {
          print SYNCFILE (" ! Bad length of time ($datlen)");
          return (’NULL’, 0); # fail
        };
      $dat = &ClipText ($wholefile, $offs, $datlen);
      $dat = &FormatTime($dat);
      return ("TIME ’$dat’", 1); # ok

   }
elsif ($typ eq ’S’)
```

```
     {
       if ($datlen != 14) # [not: 20)]
          {
            print SYNCFILE (" ! WARNING: Bad length of timestamp ($datlen)");
            if ($datlen > 14)
               { $datlen = 14;
               } else
               { $dat = &ClipText ($wholefile, $offs, $datlen);
                 print SYNCFILE (" Bad datum: <$dat> at offset $offs");
                 return (’NULL’, 1); # fail [temp hack 1=ok 16/1/2008]
               };
          };
        $dat = &ClipText ($wholefile, $offs, $datlen);
        $dat = &FormatTimestamp($dat);
        return ("TIMESTAMP ’$dat’", 1);

     }
  elsif ($typ eq ’F’)
    {
       if ($datlen != 8)
          {
            print SYNCFILE (" ! Bad length of float ($datlen)");
            return (’NULL’, 0); # fail
          };
        $dat = &FetchFloat ($wholefile, $offs);
        return ($dat, 1); # ok

    }
  else
    { print SYNCFILE (" Bad data type=$typ at:$offs, length $datlen ");
    };
  return (’NULL’, 0); # fail.
}
```

The following trivial routines format dates, times, and timestamps. There is at present *no* checking for validity! First convert 8 character date YYYYMMDD to YYYY-MM-DD:

```
sub FormatDate
{ my ($dat);
     ($dat)=@_;

  my ($yy,$mm,$dd);
  ($yy,$mm,$dd) = unpack ("a4 a2 a2", $dat);

  $dat = "$yy-$mm-$dd";
  return ($dat);
}
```

Next, time in format HHMMSSPPPPPP to HH:MM:SS.PPPPPP.

```
sub FormatTime
{ my ($tim);
     ($tim)=@_;

  my ($hh,$min,$sec);
  ($hh,$min,$sec) = unpack ("a2 a2 a2", $tim);
  $tim = "$hh:$min:$sec";

## formerly:
#  my ($hh,$min,$sec,$p);
#  ($hh,$min,$sec,$p) = unpack ("a2 a2 a2 a6", $tim);
#  $tim = "$hh:$min:$sec.$p";


  return ($tim);
}
```

Timestamp, which is a combination of the two. The output has a space character separating the date and time.

```
sub FormatTimestamp
{ my ($ts);
     ($ts) = @_;

  my ($d,$t);
  ($d,$t) = unpack ("a8 a6", $ts);
  ## formerly:
  # ($d,$t) = unpack ("a8 a12", $ts);
  $d = &FormatDate($d);
  $t = &FormatTime($t);

  $ts = "$d $t";
  return($ts);
}
```

Finally, numeric reformatting, which is a little more tricky. The scale represents the number of significant digits after the decimal point. The catch is that *all* leading zeroes are suppressed, and no decimal point is stored. As numbers are right aligned with the relevant number of zeroes *after* the [imaginary] point, this is fine. There is always at least one digit, so zero is represented as a single zero, come what may![163]

[163] This cumbersome routine could be far shorter!

```
sub FormatNumeric
{ my ($num, $scale);
     ($num, $scale)=@_;

  if ($scale == 0) # explicit
     { return $num;
     };
  # might warn if -ve!

  my ($diff);
  $diff = (length $num) - $scale;

  while ($diff < 0) # if too short:
    { $num = "0$num"; # pad
      $diff ++;
    };

  if ($diff == 0)  # eg scale of 4, value 1234 forces 0.1234
     { return ("0.$num");
     };

  my ($p, $q);
  ($p, $q) = unpack ("a$diff a$scale", $num);
  return ("$p.$q");
}
```

17.4.2 Checking the PDB header

Of note is that we submit the pdb file name *without* the `.PDB` suffix! We must do several things here:

1. ensure the name contained in the header at offset zero is the same as the supplied name (!)

2. Warn if the modification date is under the supplied timestamp (there is likely to be poor synchronisation between the clock on the PDA and that on the desktop)! The offset of the date is at 28 hex, and it is a four-byte big-endian number relative to the PalmOS epoch;

3. Ensure the type is ‘DATA’, the creator is ‘JoVS’ Each occupies four bytes, starting at hex offset 3C.

4. Check the recordList header starting at hex offset 48. At offset +4 within this header (absolute hex offset 4C) is the number of records in the PDB.

5. Determine the offset of the first record *descriptor*. [THIS SHOULD ALWAYS BE at hex offset 4E but watch out is it 50 sometimes due to the padding. Check me ???!!!]

```
sub CheckPdbHeader
{ my ($PDBFILE, $wholefile, $lastsynch);
     ($PDBFILE, $wholefile, $lastsynch)=@_;

  my ($clip);
  my($numrecs, $toprec);
  $numrecs = 0;
  $toprec = 0;

  # 1. compare names:
  if (! &SameText ($wholefile, 0, $PDBFILE))
     { $clip = &ClipText ($wholefile, 0, 32);
       print SYNCFILE "\n * File mismatch! $clip";
       return ($numrecs, $toprec);
     };

  #2. Get modification date:
  my ($moddate, $crdate);
  $moddate = &GetDoubleBe ($wholefile, 0x28);
  print SYNCFILE "\n Mod. date: $moddate";
  $crdate = &GetDoubleBe ($wholefile, 0x24);
  print SYNCFILE "\n Cr. date: $crdate";

  #3. Verify types:
  if (! &SameText ($wholefile, 0x3C, "DATA"))
     { $clip = &ClipText ($wholefile, 0x3C, 4);
       print SYNCFILE "\n * Bad type: $clip";
       # don't fail. Hmm?
     };
  if (! &SameText ($wholefile, 0x40, "JoVS"))
     { $clip = &ClipText ($wholefile, 0x40, 4);
       print SYNCFILE "\n * Bad author: $clip";
       # don't fail. Hmm?
     };

  #4. get number of records:
  $numrecs = &GetBeWord ($wholefile, 0x4C);
  print SYNCFILE "\n Record count: $numrecs";

  #5. set toprec:
  $toprec = 0x4E;
  my ($junk);
  $junk = &GetBeWord ($wholefile, $toprec);
  print SYNCFILE "\n Value at $toprec is $junk";
```

```
  $junk = &GetBeWord ($wholefile, $toprec+2);
  print SYNCFILE "\n Value at $toprec+2 is $junk";

  # here might check validity?

  return ($numrecs, $toprec);
}
```

Here are the required subroutines. SameText gets a string of the same length as the submitted string `mytxt` and compares the two, returning 0/1 as the result.

```
sub SameText
{ my ($wholefile, $offs, $mytxt);
     ($wholefile, $offs, $mytxt)=@_;

  my ($fred, $mylen);
  $mylen = length $mytxt;
  $fred = unpack ( "x$offs a$mylen", $wholefile);
  # from the template string
  # (x) skips $offs characters,
  #   then we rip out $mylen characters (a)
  # debugging:
  # print SYNCFILE "\n Comparing: <$fred> <$mytxt>";

  return ($fred eq $mytxt); # the same?
}
```

In the following, we clip out `len` characters starting at the given offset:

```
sub ClipText
{ my ($wholefile, $offs, $len);
     ($wholefile, $offs, $len)=@_;
  my $rslt;

  $rslt = unpack ("x$offs a$len", $wholefile);
  return ($rslt);
}
```

Next we retrieve a 32-bit big-endian long number from the stated offset.

```
sub GetDoubleBe
{ my ($wholefile, $offs);
     ($wholefile, $offs)=@_;

  my $i;
  $i = unpack ("x$offs N", $wholefile);

  return ($i);
}
```

The following routine is similar, but obtains a 16-bit ‘short’ big-endian number.

```
sub GetBeWord
{ my ($wholefile, $offs);
     ($wholefile, $offs)=@_;

  my $i;
  $i = unpack ("x$offs n", $wholefile);

  return ($i);
}
```

Finally, read a IEEE754 style double precision float (d is for double precision):

```
sub FetchFloat
{ my ($wholefile, $offs);
     ($wholefile, $offs)=@_;

  my $f;
  $f = unpack ("x$offs d", $wholefile);
  return ($f);
}
```

17.5 Fixing the temporary keys

There are several possible approaches to synchronising the main (PC) database and the PDA database(s). Our approach reasonably assumes the following:

- There are no more than 100 actively updated tables on the PDA;
- Between updates, no table will acquire more than 10 000 new keys;
- No more than 100 PDAs will be actively used in garnering data
- All temporary keys exist in the space from 900 000 000 to 999 999 999.
- All (valid) permanent keys exist between 0 and 900 000 000 (non-inclusive)

We then proceed as follows:

1. Divide the temporary key space into 100 equal parts, one per PDA.
2. For a given PDA (say in the space 900 000 000 to 900 999 999) we have one million available keys. For each table, allocate a block of 10 000 potential keys, and start the ‘seed’ key value (stored in the UIDS table) with the first key in that block at each resynchronisation.

3. We can now identify all keys belonging to that PDA by simple inspection, as well as knowing to which table they belong

4. When we resynchronise FROM the PDA TO the PC, we examine each key reference (starting with a number over 899 999 999) and substitute in a relevant permanent key value. We hold such keys in an associative array, creating the permanent key and storing the association when we first encounter the new temporary key in the relevant table.[164]

17.5.1 PERSON table fix-up

After we've uploaded the new data from the PDA, we still must check through the PERSON table for temporary (900-) keys. For each of these keys we need to find out whether the person already exists in our main database, or whether a new key must be generated. We then fix all references (cascading).

We find all people, identify all associated NHIs (hospital numbers), and then if these already exist in the database, replace the temporary key with the existing one. The following SQL is *really* clumsy and inefficient.

```
sub FixPERSONkeys
{ my ($myODBC);
     ($myODBC) = @_;

my @ppl;
my $prsn;

  @ppl = &SQLManySQL ($myODBC,
     "SELECT person FROM PERSON WHERE person >= 900000000",
     "Get temp keys from PERSON");

my $nhi;
my $dupli;

  foreach $prsn (@ppl)
    {($nhi) = &GetSQL ($myODBC,
       "SELECT pdoHospNo FROM PERSDATA WHERE pdoPerson = $prsn",
     "Get new NHI");
     # &Alert($MAINW, "Debug: new NHI is $nhi");

      ($dupli) = &GetSQL ($myODBC,
        "SELECT PROCESS.Person FROM PERSDATA,EPOCH,PROCESS \
          WHERE PERSDATA.Epoch = EPOCH.epoch \
           AND EPOCH.Process = PROCESS.process \
```

[164] We know that this first encounter will be with the key as the primary key in the relevant row, as we have ordered our tables to allow this approach.

```
            AND PERSDATA.pdoHospNo = '$nhi' \
            AND PROCESS.person < 900000000",
      "Look for old matching NHI");

      if (length $dupli <= 0) # if no duplicate
         { $dupli = &AutoKey($myODBC, "PERSON"); # new permanent key
           &DoSQL ($myODBC,
             "UPDATE PERSON SET person = $dupli \
               WHERE person = $prsn",
             "Replace temp with NEW key"); # cascading
         } else
         { &FixDuplicatePerson ($myODBC, $prsn, $dupli);
         };
    };
return 0; # success
}
```

There are two distinct scenarios for each temporary key:

1. The person is new. This is simpler, and we 'merely' generate a new key, and replace the temporary key with the new one. The catch is that there are multiple tables which depend on the old, temporary key! We've already anticipated this — in the definition of those tables (PROCESS, EPOCH, ACTOR2 and BADOBS), we used 'ON UPDATE CASCADE' in the definition of the relevant rows, so as we replace the temporary key with the new one, the update will take place automagically in SQL.

2. The person already exists in the database. This is a problem, as we need to first redirect all references to the temporary key to the older one, and then remove the temporary person. In an ugly fashion we find the old key, update the relevant tables, and then delete the new temporary entry in the PERSON table. As (at least in our current configuration) the only tables containing a reference to a *patient* are PROCESS (Person) and BADOBS (Person), we confine our attentions to the relevant rows of these tables.

In the above, it would be smart (now or later) to check the NHI (and even the surname) against other local databases, or even a national one, to ensure a perfect patient match, with a user caution at the least if the match isn't there![165]

17.5.2 Fixing duplicates

In the following, $prsn is the 900-key for the 'new' person entry, and $dupli is the existing (older) value for that same person, as adjudged by the hospital number.

[165] Look into this!

All 900-keys *apart* from this PERSON 900-key have already been updated to permanent keys (less than 900 000 000).

This routine caused substantial problems with re-admissions, due to a bug — we initially suppressed duplicates despite rEnd values existing, with substantial consequences!

Note the routine FixProcByType, which only addresses the case where a new (duplicate) person has been established. We create a second routine to deal with duplication of entries for the same person due to multiple new entries coming from different PDA sources! We call this ScanDuplicates (Section 17.6).

```
sub FixDuplicatePerson
{ my ($myODBC, $prsn, $dupli);
     ($myODBC, $prsn, $dupli) = @_;

      #  &Alert($MAINW, "Debug: OLD key $dupli <- $prsn"); # debug ???
      &DoSQL ($myODBC,
        "UPDATE BADOBS SET cold=1, boInactive=1 WHERE Person = $dupli",
        "retire original BADOBS");
      # ideally we should ensure new BADOBS exists! [2008-11-01]

      # next check for NEW type 1 process and OLD types 1 and 3
      # --- attach all relevant EPOCHs etc to OLD process!
      # the idea is it's unacceptable to have *active* DUPLICATE type 1, 3 procs.
      &FixProcByType($myODBC, $prsn, $dupli, 1);
      &FixProcByType($myODBC, $prsn, $dupli, 3);

      #likewise for IV PCA:
      &FixProcByType($myODBC, $prsn, $dupli, 390); # IV PCA
      # ANOTHER POTENTIAL PROBLEM: need to check whether 2 diff
      # drugs for PCA. If so, remove older one?!

      # regionals a bit more tricky:
      &FixProcBetween($myODBC, $prsn, $dupli, 100,159); #includes 100,159
      &FixProcBetween($myODBC, $prsn, $dupli, 200,259);
      &FixProcBetween($myODBC, $prsn, $dupli, 300,359);
      # [We should check whether PDA has PCEA and the
      #  (say) desktop has plain epidural. Delete older one!?]
      # THIS IS AN ERROR WAITING TO HAPPEN, eventually!

      # [we might even look for duplicate operations, oral drug Rx?]

      # at present we retain and update other PROCESSes containing
      # $prsn.

      &DoSQL ($myODBC,
        "DELETE FROM PERSDATA WHERE pdoPerson = $prsn",
         "Delete 'dup' in PERSDATA. Ideally should check if dissimilar!");
```

```
         # this is nasty: there must be a better way. Try to avoid deletion!
         # we could 'retire' the key by generating a new key
         # NEGATING it, and then updating the row in PERSDATA!

      &DoSQL ($myODBC,
        "UPDATE PROCESS SET Person = $dupli \
          WHERE Person = $prsn",
        "Replace temp with OLD key");
      &DoSQL ($myODBC,
        "UPDATE BADOBS SET Person = $dupli \
          WHERE Person = $prsn",
        "Also replace temp with OLD key");
      &DoSQL ($myODBC,
        "DELETE FROM PERSON WHERE person = $prsn",
        "Remove redundant row");
        # this too is nasty. Ideally we should not DELETE. [explore options]
        print LOGFILE "\n Duplicate: OLD key $dupli <- $prsn";
}
```

Other tables apart from the above reference PERSON but not as a patient! Here's the subsidiary FixProcByType. The value in prsn reflects the new entry for that person, and dupli is the *old* key.

```
sub FixProcByType
{ my ($myODBC, $prsn, $dupli, $TYP);
     ($myODBC, $prsn, $dupli, $TYP) = @_;

      my ($NEWpr) = &GetSQL($myODBC,
      "SELECT process FROM PROCESS WHERE person = $prsn \
         AND process > -1 \
         AND rEnd is NULL \
         AND ProcType = $TYP", "get new type $TYP proc");

      my ($OLDpr) = &GetSQL($myODBC,
      "SELECT process FROM PROCESS WHERE person = $dupli \
       AND process > -1 \
       AND rEnd is NULL \
        AND ProcType = $TYP", "get OLD type $TYP");

      # ... and update associations to OLD type
      if ( (length $NEWpr > 0)
         &&(length $OLDpr > 0)
         )
         { &DoSQL($myODBC,
            "UPDATE EPOCH SET Process = $OLDpr \
              WHERE Process = $NEWpr", 'fix new obs');
           &DoSQL($myODBC,
            "UPDATE RX SET Process = $OLDpr \
```

```
            WHERE Process = $NEWpr", 'fix new rx');
          &DoSQL($myODBC,
           "UPDATE STOPPROC SET Process = $OLDpr \
            WHERE Process = $NEWpr", 'fix new stopproc');
          #finally, retire NEW process:
          &DoSQL($myODBC,
            "UPDATE PROCESS SET rEnd = TIMESTAMP '1890-01-01 00:00:00' WHERE proc
            'retire NEW proc');
          # consider negating the NEWpr key rather than the timestamp hack. Hmm.
        };
  # RETURN.
}
```

Formerly we didn't test for rEnd is null, thereby obliterating necessary processes (We've now also taken the kinder approach of 'retiring' such processes with a low-valued timestamp, rather than deleting them and leaving keyspace gaps).

FixProcBetween is very similar, but identifies a single process in a range of types:

```
sub FixProcBetween
{ my ($myODBC, $prsn, $dupli, $LO, $HI);
     ($myODBC, $prsn, $dupli, $LO, $HI) = @_;

      my ($NEWpr, $TYP) = &GetSQL($myODBC,
  "SELECT process, ProcType FROM PROCESS WHERE person = $prsn \
        AND process > -1 \
        AND rEnd IS NULL \
        AND ProcType BETWEEN $LO AND $HI", "new type");

    if (length $NEWpr < 1)
       { return;
       };

      my ($OLDpr) = &GetSQL($myODBC,
      "SELECT process FROM PROCESS WHERE person = $dupli \
        AND rEnd IS NULL \
        AND process > -1 \
        AND ProcType = $TYP", "get OLD type $TYP");

      # ... and update associations to OLD type
      if (length $OLDpr > 0) # know NEWpr is OK!
         { &DoSQL($myODBC,
            "UPDATE EPOCH SET Process = $OLDpr \
              WHERE Process = $NEWpr", 'fix new obs');
           &DoSQL($myODBC,
            "UPDATE RX SET Process = $OLDpr \
              WHERE Process = $NEWpr", 'fix new rx');
           &DoSQL($myODBC,
```

```
            "UPDATE STOPPROC SET Process = $OLDpr \
              WHERE Process = $NEWpr", 'fix new stopproc');
           #finally, retire old process:
           &DoSQL($myODBC,
             "UPDATE PROCESS SET rEnd = TIMESTAMP '1880-01-01 00:00:00' WHERE proc
             'get rid of NEW proc');
         };
  # RETURN.
}
```

17.6 Scan for and Fix duplicates

There is a very real problem where we import data from multiple PDAs — it is possible for somebody to create duplicate, active processes for a given patient. This can occur with almost all processes. The basic idea is that a process (say a type 290 PCA process) is created on two separate PDAs, and then both get imported, one after the other.

Every time we create a process (other than, perhaps a drug Rx process [QV] where multiple oral medicines may have similar processes [LOOK AT THIS]) we need to check whether there already exists a similar process for that person which is not closed. If this is the case, we must alter all database rows which reference the new process to reference the old one, and abandon the new process.

This transformation is straightforward for EPOCH: merely create a table entry [QV] to refer to the *old* PROCESS rather then the new one. There should be no STOPPROC reference (as then the new PROCESS would be closed and thus not harmful), but the really interesting part comes with the RX table, which also references a process.

The handling of the various processes should be as follows:

1 A potential problem. What about old v. new patient information?

3 Move the new bedspace observation to the old process, *and* (this is important) turn off the old bedspace observation, to prevent there being two active bedspaces;

5 Post-discharge — should not happen!

50 Orals. Multiple instances permissible.

99 24 hour check — consider deleting [CHECK ME], as should have been closed if used;

100/101 Spinals. Should *not* be open [CHECK]

103–159 Spinal catheter, epidural etc. Remove duplicate, attach as normal.

160–499 Everything from IVs to PCA. Remove and attach.

500 Surgery. If duplicate, should ignore! But won't identify here, as is closed.

501+ For all others, remove and attach.

In other words, we have special handling of a few types. For all others, we simply attach new data to the old, open process.

We submit an ODBC handle, and an offset ($rw) within binary data ($wholefile).

We return zero if there is no duplicate process, negative numbers to indicate that we're not sure (due to deferred data), and a positive value (ID of the duplicate) if a duplicate exists. If ScanDuplicates returns a negative number, that process *must* have a null rEnd value.

```
sub ScanDuplicates
{
  my($myODBC, $wholefile, $rw);
    ($myODBC, $wholefile, $rw) = @_;
  # table is PROCESS. We need offsets of Person, ProcType, rStart, rEnd.
  # at present we hard code these, ideally should _obtain_:
  # name=<process>  0
  # name=<cold>     1
  # name=<Person>   2
  # name=<ProcType> 3
  # name=<rStart>   4
  # name=<rEnd>     5
  # name=<rCreated> 6
  # name=<rPlanner> 7
  print SYNCFILE "\n\n Scanning for PROCESS duplicates at $rw: ";

  my($PERSOFF)    = 2;
  my($PROCTYPOFF) = 3;
  my($STARTOFF)   = 4;
  my($ENDOFF)     = 5;

  # get start, end time. End must be NULL.
  my ($starttime, $endo, $startlen); #hmm. Text pointers
  my($geto) = $rw + 0x10 + (2*$STARTOFF);
  $endo = $rw + 0x12 + (2*$STARTOFF);
  $geto = GetBeWord($wholefile, $geto);
  $endo = GetBeWord($wholefile, $endo);
  $startlen = $endo - $geto;        # get length,
  $starttime = &ClipText ($wholefile, $geto, $startlen);
  # [here might check valid start time] [unhelpful]
```

```
$geto = $rw + 0x10 + (2*$ENDOFF);
$endo = $rw + 0x12 + (2*$ENDOFF);
$geto = GetBeWord($wholefile, $geto);
$endo = GetBeWord($wholefile, $endo);
if (($endo - $geto) > 0) # if end time exists
   { print SYNCFILE "(ended)";
     return (0); # do nothing!
   };
# as this is first test to fail, all of the
# following MUST have a NULL end time (NB)!

# get person. if >= 900 000 000, can't fix here!
# [note: here might check that rowsize, row max are ok]
my($person, $geto);
$geto = $rw + 0x10 + (2*$PERSOFF);
$geto = GetBeWord($wholefile, $geto);
$geto += $rw;
$person = &GetDoubleBe ($wholefile, $geto);
if ($person >= 900000000)
   { print SYNCFILE "(new: $person)";
     return (-999_999_999);
   }; # will process LATER

# obtain process type: some need special mx.
# needn't test for type 500 (always has non-null rEnd)!
my($proctype);
$geto = $rw + 0x10 + (2*$PROCTYPOFF);
$geto = GetBeWord($wholefile, $geto);
$geto += $rw;
$proctype = &GetDoubleBe ($wholefile, $geto);
if ($proctype == 1)
   { print SYNCFILE "(proc #1)";
     return (-1);   # will process LATER
   }
elsif ($proctype == 3)
   { print SYNCFILE "(proc #3)";
     return (-3);   # will process LATER
   }
elsif(  ($proctype > 259)
      &&($proctype < 300)
     )
   { print SYNCFILE "(odd stuff)";
     return(0); # ignore odd stuff.. [CHECK ME!]
   }
elsif ($proctype == 50) # enteral?!
   { print SYNCFILE "(enteral)";
     return(0);     # don't bother [CHECK ME!? a mess?]
```

```
    };

  # does duplicate process exist?
  my($qry) = "SELECT process FROM PROCESS WHERE \
    Person = $person AND \
    ProcType = $proctype AND \
    process > -1 AND \
    rEnd IS NULL";
  # need NOT test for:
  #  "AND cast(rStart as varchar(19)) > '$starttime'";
  # as processes ARE contemporaneous!
  # but what if start within range of ended process?
  # (separate, manual garbage rtns will check this!)

  my ($oldproc) = &GetSQL ($myODBC, $qry, 'get dup');
  if (length $oldproc < 1)
     { print SYNCFILE "(ok)";
       return (0); # no duplicate
     };
  print SYNCFILE "... OVERLAP PROCESS FOUND: $oldproc";
  return ($oldproc);
}
```

Returning a negative value in the preceding ScanDuplicates routine forces the caller to put the new process (created subsequent to return) into the RETAINED-PROCS array. Here's the routine to process this array. At present ScanDuplicates has conveniently excluded all oral processes (type 50) and other odds and ends (types 260–299), so we don't test for these here, at least at present! We will not here be able to identify duplicate surgery, as this was excluded due to a non-null rEnd value. All processes in RETAINEDPROCS must have null rEnd values.

```
sub FixRetainedProcs
{ my($myODBC);
    ($myODBC) = @_;
    my ($r);
  foreach $r (@RETAINEDPROCS)
    { my($qry) = "SELECT Person, ProcType FROM PROCESS WHERE \
                 process = $r";
      my($person, $proctype) =
         &GetSQL($myODBC, $qry, 'get pr detail');
      $qry = "SELECT process FROM PROCESS WHERE \
             rEnd IS NULL AND Person = $person AND \
             ProcType = $proctype"; # contemporaneous proc?
      my ($oldp) = &GetSQL($myODBC, $qry, 'get overlap proc');
      if (length $oldp > 0) # if duplicate:
         { if ($proctype == 3)
              { # [ensure new, valid bedspace]
```

```
                &HideBed($myODBC, $r, $oldp);
              };  # for now, don't get fancy for proctype = 1 [CHECK ME?]
          };
       &RetireProc($myODBC, $oldp, $r);
     };
  return (0); # 'no errors'
}
```

Here's the subsidiary RetireProc, which points new epochs and RX to the older of two concurrent processes, and 'retires' the new process by setting its key to a negative value![166]

```
sub RetireProc
{ my($myODBC, $oldp, $newp);
    ($myODBC, $oldp, $newp) = @_;
  # first, epochs:
  my($qry) = "UPDATE EPOCH SET Process = $oldp \
     WHERE Process = $newp";
##  &DoSQL ($myODBC, $qry, 'upd dup epoch');
  print SYNCFILE "\n debug: fixing(1) <$qry>";
  # next, RX:
  $qry = "UPDATE RX SET Process = $oldp \
     WHERE Process = $newp";
##  &DoSQL ($myODBC, $qry, 'upd dup rx');
  print SYNCFILE "\n debug: fixing(2) <$qry>";
  # then retire:
  $qry = "UPDATE PROCESS SET process = -$newp \
     WHERE process = $newp"; # 'invalidate' but retain!
##  &DoSQL($myODBC, $qry, 'retire proc');
  print SYNCFILE "\n debug: retiring: <$qry>";
}
```

NOTE that once we actually institute the above processes, we need to examine *all* of our code to prevent:

- Negative IDs from being exported to the PDA;
- Reporting of items with negative IDs.

Checking and hiding an old bed:

```
sub HideBed
{ my ($myODBC, $newp, $oldp);
     ($myODBC, $newp, $oldp) =@_;
```

[166] 'Retirement' will fail if a table still references the process, but the only possibility here is STOPPROC, which shouldn't reference this process!

```
  print SYNCFILE "\n Debug: hiding bed, procs =($newp, $oldp)";

  # first ensure old bed exists:
  my ($oldbed, $bo) = &GetSQL($myODBC,
     "SELECT Bed, badobs FROM BADOBS, EPOCH WHERE \
     BADOBS.Epoch = EPOCH.epoch AND \
     EPOCH.Process = $newp AND \
     boInactive IS NULL AND \
     BADOBS.cold is NULL", 'check valid old bed');
  if (length $oldbed < 1)
     { return; # assume new bed will suffice [?]
     };

  # obtain new bed:
  my ($newbed, $nb) = &GetSQL($myODBC,
       "SELECT Bed, badobs FROM BADOBS, EPOCH WHERE \
       BADOBS.Epoch = EPOCH.epoch AND \
       EPOCH.Process = $newp AND \
       boInactive IS NULL AND \
       BADOBS.cold is NULL", 'check valid bed');
  if (length $newbed < 1)
     { return; # rely on old bed.
     };
  # if NEW bed is rubbish and OLD is valid, retain old:
  my ($q);
  $q = "UPDATE BADOBS SET boInactive = 1 WHERE badobs = $nb";
  if ($newbed < 2_00_00)  # if artificial bed
     { # &DoSQL ($myODBC, $q, 'retire NEW bed!'); # temp removal 2008-02-05
       print SYNCFILE "\n Debug retire new bed: <$q>"; # temp debug
       return; # what about 'cold' ?
     };
  # else retire old:
  $q = "UPDATE BADOBS SET boInactive = 1 WHERE badobs = $bo";
  # &DoSQL ($myODBC, $q, 'retire old bed.');  # tem removal 2008-02-05
  print SYNCFILE "\n Debug retire old bed: <$q>"; # temp debug
}              # what about 'cold' ?
```

17.7 Export and Import of data

At present we enthusiastically endorse the use of AutoIt which is freely available, available free, and appears to work well on a variety of MS Windows platforms. Note that there may be some restrictions to the use of Pilot Install, which you will have to obtain separately.[167] AutoIt has an option to compile such scripts to exe

[167] We would welcome a robust GPL equivalent!

files, which means that you don't have to install AutoIt on the desktop machine.[168]

17.7.1 Initialisation of the PDA

As discussed previously (Section 7.2), we require a batch file and an AutoIt program to install to the PDA. First the batch file, which simply moves some PRC files around, and then invokes AutoIt:

```
rem Windows batch file for installing files on PDA:
rem echo off
rem cls
rem NB: Accepts the working directory (path) in %1
copy \painform\prc\*.prc %1
echo PRC Files copied to %1
\progra~1\AutoIt3\AutoIt3 \painform\install2pda.au3 %1
if errorlevel 1 goto FAILURE
GOTO END
:FAILURE
echo Something went wrong
:END
echo The end
```

Very similar is the *write2pda.bat* file, which however ensures that only PDB files (and not PRC) files are present and therefore written to the PDA:

```
rem Windows batch file for installing only PDB files on PDA:
rem echo off
rem cls
rem NB: Accepts the working directory (path) in %1
rem the path ends in a backslash.
rem ensure no PRC files present in this directory:
del %1*.PRC
\progra~1\AutoIt3\AutoIt3 \painform\install2pda.au3 %1
if errorlevel 1 goto FAILURE
GOTO END
:FAILURE
echo Something went wrong
:END
echo The end
```

Next, the AutoIt file *install2pda.au3*, which does the following

1. Runs the file *PFB.exe* and makes a connection to the PDA, with relevant checks and alerts.

[168] It might then be a reasonable idea to submit paths like "C:\painform\export\" as command line parameters to AutoIt! We haven't included such an executable as a UUencoded file here, as it runs to 2700 lines, which is a little over the top.

2. Transfers all files from the PC to the PDA using manipulation of PFB menus and commands.

```
; AutoIt program transfers files from PC to PDA.
; On PC files are normally in /painform/export, but the source
; directory is passed to AutoIt by a DOS batch file!
; AutoIt takes control of the program PFB.EXE to achieve its task.
; On the PDA the file "filepc2pda.prc" is essential, and must be
; running with a "port speed" set to e.g. 57600 (NOT "USB").
; if you choose a speed other than 57600, au3 files must be altered!

Dim $source = $CmdLine[1];
Dim $ok = 1;
Dim $retcode = 0;
Dim $timeout = 4;
Dim $slowly = 30;   ; minimal delay at first

While ($ok And $timeout > 0)
   Run("\painform\PFB.exe", "\painform")
   WinWaitActive("Connect to port:", "")

   WinActivate("Connect to port:", "")

   ; hmm assumes 115200 as default. ?? [amended 2007-11-28]
   ; NOOO. Let's try this at half speed (57600)
   ; ie:
    Send ("!s");
    Sleep ($slowly);
    Send ("5");
    Sleep ($slowly);

   Send("!P");     ; activate left hand window
   Sleep($slowly);
   Send("CU");     ; ...then force Select USB(Palm)
   Sleep($slowly);
   Send("!C");     ; Connect

   Sleep($slowly + 300);     ; short delay
   $ok = WinExists("PFB", "You're not connected with a palm device");
     ; check this first, just in case.
   If $ok Then
      WinClose("PFB", "You're not connected" );  ; close this
   Else
      $ok = WinWaitActive("PFB (C) PMrogan", "", 7);  ;time out after 7 s
      If ($ok) Then
          $retcode = MoveStuff("C:" & $source, "xTABLE.PDB" );
                    ; retcode is still 0 if succeeded
          $ok = 0;  ; force exit
```

```
      Else
         $ok = 1;   ;if $ok==1, can RETRY!
         $retcode = CheckForErrors();
         If ($retcode <> 0) Then
            $ok = 0; ; fail
         EndIf;
      EndIf;
   EndIf

   If Not WinExists("PFB", "") Then
      MsgBox(0, "Woops!", "Connection failed");
      $retcode = 6;
      $ok = 0;
   EndIf
   WinClose("PFB");

  $timeout = $timeout -1;
  If ($timeout < 1) Then
     MsgBox(0, "I'm bored.", "Bye!");
     $ok = 0;
     $retcode = 20;
  EndIf

  If $ok Then
      MsgBox(0, "Problem", "Connection failed! Please try once more..");
  EndIf
  $slowly = $slowly + 500; ; bump delay by 500ms

Wend; ; retry if $ok is nonzero...

Exit($retcode);
; ----------------------end main section---------------------;
; CheckForErrors does just that returning 0 if there is potential
; to retry.

Func CheckForErrors()
  Dim $retcode = 1;
  Dim $mycount = 10;

While $mycount > 0 And $retcode <> 1

  Sleep (500);  ;500 milliseconds
  If $retcode == 1 AND WinExists("Error", "USBPort") Then
    WinClose("Error");
    MsgBox(0, "Error", "Bad/missing USBPort.dll. This is NOT good!");
    $retcode = 8;    ; fatal
  EndIf
```

```
  If $retcode == 1 AND WinExists("PFB", "You're not connected with a palm device")
    WinClose("PFB");
    MsgBox(0, "Bah!", "Computer has trouble connecting..");
    $retcode = 0;
  EndIf

  If $retcode == 1 AND WinExists("Error", "") Then
    WinClose("Error");
    MsgBox(0, "Error", "Miscellaneous (but nasty) error!");
    $retcode = 9; ; fatal
  EndIf

  $mycount = $mycount -1;
Wend;

  If $retcode == 1 Then
    MsgBox(0, "Error (very tired)", "Most odd!");
    $retcode = 10;  ; just force failure
  EndIf

  Return($retcode);  ; nonzero return forces Exit!
EndFunc;


; ----------------------Move files TO the pda----------------;
Func MoveStuff ($SRCD, $LASTFILE) ; source dir is $SRCD

  ; First, open relevant directories on PDA, PC:
  Dim $retcode = SetDirectories($SRCD, "YCOORDINATES.DATA");
  If ($retcode <> 0) Then
     Return ($retcode);
  EndIf

  ; activate the left transfer button (send a space)
  ControlFocus("PFB", "", 1005);
  ControlClick ("PFB", "", 1005); ;;clumsy hack ? WHY needed
  Send(" ");

  ; wait for completion:
  Dim $endmsg = "";
  Dim $safety = 60;   ; max wait 60 seconds (??)
  Dim $sec = 0;

  ; note: PFB automatically orders files, so 'last file' works!
  While ($sec < $safety AND $endmsg <> "File was saved under 0:\" & $LASTFILE & "!
    Sleep (1000);   ;milliseconds
    $sec = $sec+1;
    WinSetTitle ("PFB", "", "PFB Installing files: elapsed time " & $sec & " secon
```

```
    $endmsg= ControlGetText("PFB", "", 1012);
  Wend

  If $sec >= $safety Then
     $retcode = 11;   ;error
  EndIf
  Return ($retcode);
EndFunc



Func SetDirectories ($SRCD, $DATAFILE) ; source dir
  Dim $retcode = 0;
  Dim $wsize = WinGetPos("PFB");
  Dim $winX = $wsize[0];
  Dim $winY = $wsize[1];
  Dim $winW = $wsize[2];
  Dim $winH = $wsize[3];
  Dim $ARBITRARYYOFFSET = 20;
  $winY = $winY + $ARBITRARYYOFFSET;

  ; home:
  Dim $xclick = 40 + $winX;
  Dim $RightX=  40 + $winX + $winW/2;

  ; a. get position of control:
  ControlFocus("PFB (C) PMrogan", "", 1018);
  Dim $boxcoords = ControlGetPos("PFB (C) PMrogan", "", 1018);
  Dim $ctly = $winY + $boxcoords[1];
  Dim $ctlyMAX = $ctly + $boxcoords[3] -10;  ;;be conservative.
  Dim $yclickD = 35 + $ctly;

; FIRST the RIGHT box (PDA):
; we can use Y coordinates of LEFT control as they are the same!
  Dim $COORDS[5];  ;;max 5 deep, for now.
  $COORDS[0]=0;    ;; no coordinates here!
  If Not GoDirectory(1019, "0:\", $RightX, $ctly, $ctlyMAX, $COORDS) Then
    Return(13); ;;fail
  EndIf

; NEXT the left box
  LoadCoordinates ($COORDS, $DATAFILE);
  If Not GoDirectory(1018, $SRCD, $xclick, $ctly, $ctlyMAX, $COORDS) Then
    Return(14); ;;fail
  EndIf
  ; also write y coordinates to file "\painform\YCOORDINATES.DATA":
  ; [2207-10-23 13:15] Have turned off this line temporarily. Slow but more sure!
  ;  SaveCoordinates($COORDS, $DATAFILE);
```

```
; finally, select everything in this directory!
  MouseClick("left", $xclick, $yclickD, 1, 3);
  Dim $dbg = ControlListView("PFB (C) PMrogan", "", 1018, "GetSelected");
  $val = ControlListView("PFB (C) PMrogan", "", 1018, "GetText", $dbg);
  If $val == "[..]" Then
     ControlSend("PFB (C) PMrogan", "", 1018, "{DOWN}");
     ; prevent error if right at top!
  EndIf
  ControlSend("PFB (C) PMrogan", "", 1018, "^+{END}");
  ; Control+SHIFT+End selects everything!
  ; NOTE: a major error will occur (crash PDA+PC) if we click
  ; on the ".." at the top of the directory and not the item below
  ; so be careful of the value in $yclickD.

; BRACES & BELT:
; before we send stuff we might here examine the text at the top
; to ensure that we are transferring the right stuff
; (OTHERWISE FAIL)!
  Dim $srcdir = ControlGetText("PFB (C) PMrogan", "", 1020)
  If $srcdir <> $SRCD Then
    MsgBox(0, "Error", "Bad source (" & $srcdir & " [want: " & $SRCD & "]" & "). B
    Return (12);  ; fail
  EndIf

  Dim $destdir = ControlGetText("PFB (C) PMrogan", "", 1021)
  If $destdir <> "0:\" Then
    MsgBox(0, "Error", "Bad destination (" & $destdir & "). Bad mouse positioning?
    Return (13);  ; fail
  EndIf

 Return(0);   ;success
EndFunc


;------------------------------
; Write coordinates to a file:
Func SaveCoordinates ($COORDS, $DATAFILE)
  Dim $hYCOFILE = FileOpen ($DATAFILE, 2); ;;overwrite
  If $hYCOFILE <> -1 Then
    Dim $i;
    For $i = 0 to $COORDS[0]
      FileWriteLine($hYCOFILE, $COORDS[$i]);
    Next
    FileClose($hYCOFILE);
  EndIf
EndFunc
```

```
;----------------------------
; Reload coordinates:
Func LoadCoordinates (ByRef $COORDS, $DATAFILE)
  If FileExists ($DATAFILE) Then
     Dim $hYCOFILE = FileOpen ($DATAFILE, 0); ;;read
     Dim $lines = FileReadLine($hYCOFILE, 1);  ;;first line=line count!
     Dim $i;
     For $i = 1 to $lines+1
       $COORDS[$i-1] = FileReadLine($hYCOFILE, $i);
     Next
    Return 1; (ok)
  EndIf
  $COORDS[0] = 0;  ;;signal no lines
  Return 0; ;;fail (redundant signal)
EndFunc

; ------------------------------
; move to a particular directory
; using mouse clicks.

Func GoDirectory($id, $DIR, $x, $y0, $yMAX, ByRef $COORDS)
  Dim $y = $y0+12;
  Dim $myDir = StringSplit($DIR, "\");
  Dim $i;
  Dim $posn;

  For $i = 1 to ($myDir[0]-1)  ;;ignore the final backslash!
   ;; MsgBox(0, "Debug: going to..", $myDir[$i]);
    If $COORDS[0] >= $i Then
       $y = $COORDS[$i];  # pre-determinated coordinates
    EndIf
    $posn = GoOneDir ($id, $myDir[$i], $x, $y, $y0, $yMAX);
    If Not $posn Then
      Return 0;  ;;fail.
    EndIf
    $COORDS[$i] = $posn;
  Next
  $COORDS[0] = ($myDir[0]-1);
  Return 1;  ;;success!
EndFunc

Func GoOneDir($id, $target, $x, $y, $y0, $yMAX)
  $y = ConfirmMyFocus($id, $x, $y, $target, $y0, $yMAX);
  If Not $y Then
    MsgBox(0, "Failed", "Failed to identify " & $target);
    Return(0);  ;fail
  EndIf
  ControlFocus("", "", $id);
```

```
  MouseClick("left", $x, $y, 2, 3); ;;doubleclick
  Sleep(300);
  Return($y);
EndFunc

; ------------------------------------------------------------------;
; the following slowly and repetitively examines each item in a
; list to see if it's the target item. Because we cannot enter the path,
; and can only select an item using a mouse double click :-(
; we have to look for each component of the path name in "[square brackets]"!

; <=====================================
;  We must still:
;    1. Rename directories to A-painform, A-export, A-import

Func ConfirmMyFocus($ITEMCODE, $XCO, $YCO, _
                    $TEXTTARGET, $YMIN, $YMAX)

Dim $TITLE = "PFB (C) PMrogan";
Dim $SUBTEXT = "";
$TEXTTARGET = "[" & $TEXTTARGET & "]";   ; put in square brackets
Dim $dbg=0;
Dim $val="";
Dim $limit = 50;  ;max tries to prevent endless loop on error!

If $YCO > $YMAX Then
  $YCO = $YMAX - 12;   ;;ensure in range
EndIf

If $YCO < $YMIN Then
  $YCO = $YMIN + 12;   ;;likewise
EndIf

While $limit > 0 And $YCO > $YMIN And $YCO < $YMAX
  MouseClick("left", $XCO, $YCO, 1, 2);
  $dbg = ControlListView($TITLE, $SUBTEXT, $ITEMCODE, "GetSelected"); ;first item
  $val = ControlListView($TITLE, $SUBTEXT, $ITEMCODE, "GetText", $dbg); ;first ite
  If $val == $TEXTTARGET Then
    Return ($YCO);   ;success
  EndIf
  If $val > $TEXTTARGET Then   ;if need to move UP:
    $YCO = $YCO - 12;
  Else
    $YCO = $YCO + 12;
  EndIf
  $limit = $limit -1;
Wend;   ;if While FAILS, then must try to shift contents of List Box
```

```
  ControlFocus($TITLE, $SUBTEXT, $ITEMCODE);
  ; the above line prevents sequential clicks being seen as a
  ; doubleclick (disaster)!

If $YCO < $YMIN Then
  $YCO = $YCO + 12;  ;;back down
  MouseClick("left", $XCO, $YCO, 1, 10);
  $dbg = ControlListView($TITLE, $SUBTEXT, $ITEMCODE, "GetSelected"); ;first item
  While $dbg > 0 ; while after first item... ie can scroll up
    ControlSend($TITLE, $SUBTEXT, $ITEMCODE, "{UP}");  ;keep in the window
    $dbg = $dbg - 1;
    $val = ControlListView($TITLE, $SUBTEXT, $ITEMCODE, "GetText", $dbg); ;first i
    If $val == $TEXTTARGET Then
       Return ($YCO);   ;success
    EndIf
  Wend;
  Return (0);   ;fail
EndIf

; else must try to move down...
$YCO = $YCO - 12;  ;;back up
Dim $itemcount = ControlListView($TITLE, $SUBTEXT, $ITEMCODE, "GetItemCount");
MouseClick("left", $XCO, $YCO, 1, 2);
$dbg = ControlListView($TITLE, $SUBTEXT, $ITEMCODE, "GetSelected");
While $dbg < $itemcount
  $dbg = $dbg + 1;
  ControlSend($TITLE, $SUBTEXT, $ITEMCODE, "{DOWN}");
  $val = ControlListView($TITLE, $SUBTEXT, $ITEMCODE, "GetText", $dbg); ;first ite
  If $val == $TEXTTARGET Then
       Return ($YCO);   ;success
  EndIf
Wend

Return (0);   ;fail
EndFunc
```

17.7.2 Synchronisation of the PDA

This is much more of a challenge than simple installation, but the process is similar:

1. First move relevant PDB files *from* the PDA;
2. Then synchronise them with database (parse the PDB files and update the SQL database);
3. Then only can we write the files back to the PDA.

We perform the above using a lot of the functionality described in the previous section (17.7.1). We will invoke a distinct AutoIt script (*fetchfrompda.au3*) to obtain data once the user has established a connection. Then we will leave this script, read the PDB files and synchronise (normally from the directory */painform/import*), followed by a write to the export directory and an export identical to the installation process, apart from the fact that we don't copy any PRC files to the export directory!

First, the DOS batch file *fetchfrompda.bat*, which is similar to *install2pda.bat*.

```
rem Windows batch file for synching with PDA:
rem This one ONLY does the retrieval bit..
rem echo off
rem cls
rem NB: Accepts the destination directory (path) in %2
rem     *and* the SOURCE directory in %1
rem Both directory paths end in a backslash
rem FIRST delete destination directory contents
del /Q %2
rem as usual in DOS we copy from LEFT to right:
rem we copy .PDB files:
copy %1*.PDB %2
rem and of course LASTSYNCH.TIME:
copy %1*.TIME %2
\progra~1\AutoIt3\AutoIt3 \painform\fetchfrompda.au3 %2
if errorlevel 1 goto FAILURE
GOTO END
:FAILURE
echo Something went wrong
:END
echo The end
```

In obtaining selected files from the PDA, the key AutoIt function is ControlListView. The functionality of the following script *fetchfrompda.au3* is very similar to *install2pda.au3* but we must take great pains to fetch only the correct files from the PDA We submit the *destination* directory which now contains template files which are only there to provide us with names — each is overwritten by a file from the PDA!

```
; AutoIt Program to fetch PDB files from a PDA using PFB program
; Get DOS path for destination (which contains 'template' files to be overwritten!

#include <Array.au3>

Dim $source = $CmdLine[1]; ;actually 'destination' too!
Dim $ok = 1;
Dim $retcode = 0;
```

```
Dim $timeout = 4;
Dim $slowly = 30;   ; minimal delay at first

While ($ok And $timeout > 0)
   Run("\painform\PFB.exe", "\painform")
   WinWaitActive("Connect to port:", "")
   WinActivate("Connect to port:", "")

   ; hmm assumes 115200 as default. ?? [amended 2007-11-28]
   ; NOOO. Let's try this at half speed (57600)
   ; ie:
    Send ("!s");
    Sleep ($slowly);
    Send ("5");
    Sleep ($slowly);

   Send("!P");     ; activate left hand window
   Sleep($slowly);
   Send("CU");     ; ...then force Select USB(Palm)
   Sleep($slowly);
   Send("!C");     ; Connect

   Sleep($slowly + 300);     ; short delay
   $ok = WinExists("PFB", "You're not connected with a palm device");
     ; check this first, just in case.
   If $ok Then
      WinClose("PFB", "You're not connected" );  ; close this
   Else
      $ok = WinWaitActive("PFB (C) PMrogan", "", 4);
      If ($ok) Then
          $retcode =  MoveOut("C:" & $source);
                    ; retcode is still 0 if succeeded
          $ok = 0;  ; force exit
      Else
         $ok = 1;   ;if $ok==1, can RETRY!
         $retcode = CheckForErrors();
         If ($retcode <> 0) Then
            $ok = 0; ; fail
         EndIf;
      EndIf;
   EndIf

   If Not WinExists("PFB", "") Then
      MsgBox(0, "Error", "Connection failed");
      $retcode = 6;
      $ok = 0;
   EndIf
   WinClose("PFB");
```

```
  $timeout = $timeout -1;
  If ($timeout < 1) Then
     MsgBox(0, "I'm bored.", "Bye!");
     $ok = 0;
     $retcode = 20;
  EndIf

  If $ok Then
      MsgBox(0, "Problem", "Connection failed! Please try once more..");
  EndIf
  $slowly = $slowly + 500; ; bump delay by 500ms

Wend; ; retry if $ok is nonzero...


Exit($retcode);
; ----------------------end main section---------------------;

Func CheckForErrors()
  Dim $retcode = 1;
  Dim $mycount = 10;

While $mycount > 0 And $retcode <> 1

  Sleep (500);  ;500 milliseconds
  If $retcode == 1 AND WinExists("Error", "USBPort") Then
    WinClose("Error");
    MsgBox(0, "Error", "Bad/missing USBPort.dll");
    $retcode = 8;
  EndIf

  If $retcode == 1 AND WinExists("PFB", "You're not connected with a palm device")
    WinClose("PFB");
    $retcode = 0;
  EndIf

  If $retcode == 1 AND WinExists("Error", "") Then
    WinClose("Error");
    MsgBox(0, "Error", "Miscellaneous error!");
    $retcode = 9;
  EndIf

  $mycount = $mycount -1;
Wend;

  If $retcode == 1 Then
    MsgBox(0, "Error", "Most odd!");
```

```
    $retcode = 10;  ; just force failure
  EndIf

  Return($retcode);  ; nonzero return forces Exit!
EndFunc;


; ------------ move data from PDA to PC ----------------- ;

Func MoveOut ($DST) ;
; Here we MUST fetch only those PDA files with matches in
; the directory $DST (the DESTINATION directory).
; We overwrite all files in $DST.
; We must go through the left hand (PC) directory list, get all of
; the file names, THEN flag corresponding files on the right,
; and finally move from right (PDA) to left.

  ; First, open relevant directories on PDA, PC:
  Dim $retcode = SetDirectories($DST, "OUTCOORDINATES.DATA");
  If ($retcode <> 0) Then
     Return ($retcode);
  EndIf

  ; Next WE MUST:
  ; 1. Examine all files in the left directory, apart from . and ..
  ; 2. For each file, find and mark the corresponding file on the right
  ;   (Warn if file not found on PDA)
  ; 3. Then activate transfer from PDA to PC, waiting for termination (the last fi

  ;1. Get file list:
  Dim $itemcount = ControlListView ("PFB (C) PMrogan", "", 1018, "GetItemCount");
  Dim $aItems [$itemcount];
  Dim $c;  ; fetch all item values (text):
  For $c = 0 to $itemcount-1
    $aItems[$c] = ControlListView ("PFB (C) PMrogan", "", 1018, "GetText", $c);
  Next
  _ArraySort($aItems); ;ensure it's sorted! (but should be anyway)
     ; note the requirement for #include <Array.au3> above.
  Dim $LASTFILE = $aItems[$itemcount-1];

  ;2. Do something similar on the right (PDA) but SELECT each item iff it
  ; corresponds to an item in aItems:
  Dim $Rcount = ControlListView ("PFB (C) PMrogan", "", 1019, "GetItemCount");
  Dim $rItem;
  Dim $found;
  Dim $hits = 0;
    ; consider each item:
  For $c = 0 to $Rcount-1
```

```
    $rItem = ControlListView ("PFB", "", 1019, "GetText", $c);
    $found = _ArrayBinarySearch($aItems, $rItem);
    If Not @error Then
      ControlListView ("PFB", "", 1019, "Select", $c);
      $hits = $hits + 1;
      WinSetTitle ("PFB", "", "PFB --- Importing: " & $hits & " files");
      Sleep(30);
    EndIf
  Next

  ; MsgBox (0, "Debug", "Number of hits was " & $hits);
  ; activate the RIGHT transfer button (send a space)
  Sleep(1000); ; necessary [? why]
  ControlFocus("PFB", "", 1006);
  ControlClick("PFB", "", 1006);
  Send(" ");

  ; finally wait for completion...
  Dim $endmsg= "";
  Dim $safety = 60;   ; max wait 60 seconds (??)
  Dim $sec = 0;
  While $sec < $safety AND _
    $endmsg <> _
    StringUpper("File was saved under " & $DST & $LASTFILE & " gespeichert!")
    Sleep (1000);   ;milliseconds
    $sec = $sec+1;
    WinSetTitle ("PFB", "", "PFB Transferring data to PC: elapsed time " & $sec &
    $endmsg= StringUpper( ControlGetText("PFB", "", 1012) );
  Wend
;;  "File was saved under C:\painform\import\xTABLE.pdb gespeichert!"

  If ($sec >= $safety) Then
   MsgBox(0, "Time-out", "Trigger failed: " & $DST & $LASTFILE);
  EndIf
  Return (0);  ;success!
EndFunc

; ------------------------------------------------------
; SetDirectories is identical to the Fx found in install2pda.au3.
; Note the use of a different coordinates file: OUTCOORDINATES.DATA

Func SetDirectories ($SRCD, $DATAFILE) ; source dir
  Dim $retcode = 0;
  Dim $wsize = WinGetPos("PFB");
  Dim $winX = $wsize[0];
  Dim $winY = $wsize[1];
  Dim $winW = $wsize[2];
  Dim $winH = $wsize[3];
```

```
  Dim $ARBITRARYYOFFSET = 20;
  $winY = $winY + $ARBITRARYYOFFSET;

  ; home:
  Dim $xclick = 40 + $winX;
  Dim $RightX=  40 + $winX + $winW/2;

  ; a. get position of control:
  ControlFocus("PFB (C) PMrogan", "", 1018);
  Dim $boxcoords = ControlGetPos("PFB (C) PMrogan", "", 1018);
  Dim $ctly = $winY + $boxcoords[1];
  Dim $ctlyMAX = $ctly + $boxcoords[3] -10;  ;;be conservative.
  Dim $yclickD = 35 + $ctly;

; FIRST the RIGHT box (PDA):
; we can use Y coordinates of LEFT control as they are the same!
  Dim $COORDS[5];  ;;max 5 deep, for now.
  $COORDS[0]=0;    ;; no coordinates here!
  If Not GoDirectory(1019, "0:\", $RightX, $ctly, $ctlyMAX, $COORDS) Then
    Return(13); ;;fail
  EndIf

; NEXT the left box
  LoadCoordinates ($COORDS, $DATAFILE);
  If Not GoDirectory(1018, $SRCD, $xclick, $ctly, $ctlyMAX, $COORDS) Then
    Return(14); ;;fail
  EndIf
;;  SaveCoordinates($COORDS, $DATAFILE);
;; 2007-11-01 : disable this too!

; finally, select everything in this directory!
  MouseClick("left", $xclick, $yclickD, 1, 3);
  Dim $dbg = ControlListView("PFB (C) PMrogan", "", 1018, "GetSelected");
  $val = ControlListView("PFB (C) PMrogan", "", 1018, "GetText", $dbg);
  If $val == "[..]" Then
     ControlSend("PFB (C) PMrogan", "", 1018, "{DOWN}");
     ; prevent error if right at top!
  EndIf
  ControlSend("PFB (C) PMrogan", "", 1018, "^+{END}");

; BRACES & BELT:
; before we send stuff we might here examine the text at the top
; to ensure that we are transferring the right stuff
; (OTHERWISE FAIL)!
  Dim $srcdir = ControlGetText("PFB (C) PMrogan", "", 1020)
  If $srcdir <> $SRCD Then
    MsgBox(0, "Error", "Bad source (" & $srcdir & " [want: " & $SRCD & "]" & "). E
    Return (12);  ; fail
```

```
  EndIf

  Dim $destdir = ControlGetText("PFB (C) PMrogan", "", 1021)
  If $destdir <> "0:\" Then
    MsgBox(0, "Error", "Bad destination (" & $destdir & "). Bad mouse positioning?
    Return (13);  ; fail
  EndIf

 Return(0);   ;success
EndFunc


;------------------------------
; Write coordinates to a file:
Func SaveCoordinates ($COORDS, $DATAFILE)
  Dim $hYCOFILE = FileOpen ($DATAFILE, 2); ;;overwrite
  If $hYCOFILE <> -1 Then
    Dim $i;
    For $i = 0 to $COORDS[0]
      FileWriteLine($hYCOFILE, $COORDS[$i]);
    Next
    FileClose($hYCOFILE);
  EndIf
EndFunc

;----------------------------
; Reload coordinates:
Func LoadCoordinates (ByRef $COORDS, $DATAFILE)
  If FileExists ($DATAFILE) Then
     Dim $hYCOFILE = FileOpen ($DATAFILE, 0); ;;read
     Dim $lines = FileReadLine($hYCOFILE, 1);  ;;first line=line count!
     Dim $i;
     For $i = 1 to $lines+1
       $COORDS[$i-1] = FileReadLine($hYCOFILE, $i);
     Next
    Return 1; (ok)
  EndIf
  $COORDS[0] = 0;  ;;signal no lines
  Return 0; ;;fail (redundant signal)
EndFunc

; ------------------------------
; move to a particular directory
; using mouse clicks.

Func GoDirectory($id, $DIR, $x, $y0, $yMAX, ByRef $COORDS)
  Dim $y = $y0+12;
  Dim $myDir = StringSplit($DIR, "\");
```

```
  Dim $i;
  Dim $posn;

  For $i = 1 to ($myDir[0]-1)  ;;ignore the final backslash!
    If $COORDS[0] >= $i Then
       $y = $COORDS[$i];  # pre-determinated coordinates
    EndIf
    $posn = GoOneDir ($id, $myDir[$i], $x, $y, $y0, $yMAX);
    If Not $posn Then
      Return 0;  ;;fail.
    EndIf
    $COORDS[$i] = $posn;
  Next
  $COORDS[0] = ($myDir[0]-1);
  Return 1;  ;;success!
EndFunc

Func GoOneDir($id, $target, $x, $y, $y0, $yMAX)
  $y = ConfirmMyFocus($id, $x, $y, $target, $y0, $yMAX);
  If Not $y Then
    MsgBox(0, "Failed", "Failed to identify " & $target);
    Return(0);  ;fail
  EndIf
  ControlFocus("", "", $id);
  MouseClick("left", $x, $y, 2, 3); ;;doubleclick
  Sleep(300);
  Return($y);
EndFunc

; ------------------------------------------------------------------;
; the following slowly and repetitively examines each item in a
; list to see if it's the target item. Because we cannot enter the path,
; and can only select an item using a mouse double click :-(
; we have to look for each component of the path name in "[square brackets]"!

; <====================================
;  We must still:
;    1. Rename directories to A-painform, A-export, A-import

Func ConfirmMyFocus($ITEMCODE, $XCO, $YCO, _
                    $TEXTTARGET, $YMIN, $YMAX)

Dim $TITLE = "PFB (C) PMrogan";
Dim $SUBTEXT = "";
$TEXTTARGET = "[" & $TEXTTARGET & "]";   ; put in square brackets
Dim $dbg=0;
Dim $val="";
Dim $limit = 50;  ;max tries to prevent endless loop on error!
```

```
If $YCO > $YMAX Then
  $YCO = $YMAX - 12;   ;;ensure in range
EndIf

If $YCO < $YMIN Then
  $YCO = $YMIN + 12;   ;;likewise
EndIf

While $limit > 0 And $YCO > $YMIN And $YCO < $YMAX
  MouseClick("left", $XCO, $YCO, 1, 2);
  $dbg = ControlListView($TITLE, $SUBTEXT, $ITEMCODE, "GetSelected"); ;first item
  $val = ControlListView($TITLE, $SUBTEXT, $ITEMCODE, "GetText", $dbg); ;first ite
  If $val == $TEXTTARGET Then
    Return ($YCO);   ;success
  EndIf
  If $val > $TEXTTARGET Then   ;if need to move UP:
    $YCO = $YCO - 12;
  Else
    $YCO = $YCO + 12;
  EndIf
  $limit = $limit -1;
Wend;   ;if While FAILS, then must try to shift contents of List Box

  ControlFocus($TITLE, $SUBTEXT, $ITEMCODE);
  ; the above line prevents sequential clicks being seen as a
  ; doubleclick (disaster)!

If $YCO < $YMIN Then
  $YCO = $YCO + 12;  ;;back down
  MouseClick("left", $XCO, $YCO, 1, 10);
  $dbg = ControlListView($TITLE, $SUBTEXT, $ITEMCODE, "GetSelected"); ;first item
  While $dbg > 0 ; while after first item... ie can scroll up
    ControlSend($TITLE, $SUBTEXT, $ITEMCODE, "{UP}");  ;keep in the window
    $dbg = $dbg - 1;
    $val = ControlListView($TITLE, $SUBTEXT, $ITEMCODE, "GetText", $dbg); ;first i
    If $val == $TEXTTARGET Then
       Return ($YCO);   ;success
    EndIf
  Wend;
  Return (0);   ;fail
EndIf

; else must try to move down...
$YCO = $YCO - 12;  ;;back up
Dim $itemcount = ControlListView($TITLE, $SUBTEXT, $ITEMCODE, "GetItemCount");
MouseClick("left", $XCO, $YCO, 1, 2);
$dbg = ControlListView($TITLE, $SUBTEXT, $ITEMCODE, "GetSelected");
```

```
While $dbg < $itemcount
  $dbg = $dbg + 1;
  ControlSend($TITLE, $SUBTEXT, $ITEMCODE, "{DOWN}");
  $val = ControlListView($TITLE, $SUBTEXT, $ITEMCODE, "GetText", $dbg); ;first ite
  If $val == $TEXTTARGET Then
       Return ($YCO);   ;success
  EndIf
Wend
Return (0);   ;fail
EndFunc
```

17.8 Another PDA file move

Here's the clumsy DOS batch file to copy PRC files to the backup directory for the PDA (on the desktop). It copies directly from the generated files in Palmdev/testing (and subdirectories), assuming that these PRC files exist.

```
rem NB: Accepts the destination path in %1 with NO terminal backslash
rem The following ensures old stuff not re-installed ipo new programs!?
del %1\pain*.prc
del %1\err*.prc
del %1\sc*.prc
del %1\nu*.prc
del %1\sq*.prc
del %1\consol*.prc
del %1\os*.prc
del %1\IDX*.prc
del %1\Cache*.prc
rem watch out for similar names as we have not been very selective!
rem NOW copy over new prc files:
copy \PalmDev\testing\*.prc           %1
copy \PalmDev\testing\err\*.prc       %1
copy \PalmDev\testing\scripting\*.prc %1
copy \PalmDev\testing\numeric\*.prc   %1
copy \PalmDev\testing\sql3\*.prc      %1
copy \PalmDev\testing\console\*.prc   %1
copy \PalmDev\testing\osbox\*.prc     %1
copy \PalmDev\testing\idx\*.prc       %1
copy \PalmDev\testing\cache\*.prc     %1
copy \painform\MathLib.prc %1
echo PRC Files copied to destination
```

17.9 PalmOS testing: Emulator synchronisation

Under MS Windows (2000 or XP) it is possible to 'HotSync' the POSE emulator to a local directory without using a null modem cable, or even being connected to

a network.[169] Run through the following laborious process once, and keep a copy of the POSE configuration:

1. Run the Palm Hotsync manager, so that its icon appears on the Windows taskbar. Right click on this icon, and select 'Network'. [170]

2. Run POSE, right click on the emulator, choose Settings/Properties, and then ensure that the "Redirect NetLib calls to host TCP/IP" box is checked. Click OK.

3. Still in POSE, click on the Hotsync icon, followed by a click on the title at the top to activate the 'Options' menu. You will need to do some configuring:

   (a) First, choose "Modem Sync Prefs" and click on the Network button, followed by OK;

   (b) Next, select the "LANSync Prefs" menu option, and choose Local HotSync and OK;

   (c) Finally in this section, choose "Primary PC Setup" where you will need to enter the 'network' name of your PC,[171] followed by typing in `localhost` as the "Primary PC Address". Leave the subnet mask blank.

4. Your next POSE task (remember you only have to do all of this once!) is to click on "Modem" in the main Hotsync screen, and below the Hotsync logo click on 'Select Service'. Enter the service as `POSE`, 'Tap to Enter Phone' and type in a number of zero, and click OK. Don't fuss about the User name, password or connection. Click 'done'.

Save this copy of POSE, and when you click on the modem hotsync logo, with luck you should be able to hosync to a local PC directory! You'll find the PDB (and other) files stored in a local directory such as
`Program Files\Handspring\Fred\Backup`

[169] Depending on the level of paranoia in your system, you may need to be the, or be friends with your, network administrator to get this to work. Firewall software may also get in the way, although we found that AVG worked fine.

[170] You may have to first create a user within the PalmOS desktop, and then click in a similar fashion on the Hotsync manager, select 'Setup' and then check the relevant box in the 'Network' tab.

[171] You can obtain this by right clicking on 'My Computer' on your PC desktop, choosing 'Network Identification', and then looking at the name under 'Properties'.

...depending on your installation of the PalmOS desktop, and the user name you've chosen (here Fred).[172] If you store PDB files within this directory, the next time you Hotsync, these should be loaded *into* the POSE emulator. Most convenient!

18 Several files: batch files, constants, xlog & CSV

18.1 Pain batch file

Under Windows, we run a batch file (*pain.bat* to kick things off:

```
echo off
cls
perl pain2.pl -font "Times 11" %1
```

It's convenient to set the perl font. In addition we allow addition of a parameter, which is the user name of the person logging on. (In real life, I've written a front end in AutoIt which controls logging on, and submits this single parameter to the perl program, which accesses the parameter using $ARGV[0]).

18.2 PDF creation

As promised above, we need a batch file to invoke PdfLatex, turning a LaTeX file into a PDF file. Here it is, *pdfify.bat*. We submit the name of the relevant file, and assume that our current directory is the main PainForm directory.

```
rem Batch file for turning LaTeX into PDF files
pdflatex -interaction=nonstopmode latex/%1.tex %1.pdf
rem second pass hmm
pdflatex -interaction=nonstopmode latex/%1.tex %1.pdf
move %1.pdf pdf/%1.pdf
rem here might clean up all auxiliary files
del %1.log
del %1.aux
```

We use the -interaction=nonstopmode with PdfLaTeX to force it to continue if errors occur — the alternative is to wait for user input, which hangs our whole program!

[172] There are many ways to find where the files have been stored, but the easiest is to right click on the taskbar Hotsynch icon, and view the Log after synchronising.

18.3 PDF printing

It's possible to invoke e.g. Acrobat Reader from the command line, and get it to print a document thus:

```
CALL START /MIN "C:\Program Files\Adobe\Acrobat 5.0\Reader\AcroRd32"
"C:\Program Files\Adobe\Acrobat 5.0\Reader\AcroRd32" /p pdf\%1
```

The batch file must be given the file to be printed (with path and .PDF suffix) as %1. The /h switch is meant to open Acrobat in hidden mode, but is not reliable.

Alternatively one might suppress the printer dialogue by using the /t switch. The four command-line arguments for AcroRd32 with the /t switch are:

1. The file to print;
2. The printer name (as it appears in Windows);
3. The printer driver (In Windows try Start—Settings—Printers and then right click on the desired printer, and Properties—Advanced to see the driver);
4. The port. (As above, but click on the Port tab).

For convenience we've broken the following long line into two but *do not* break the line if you wish to try this in DOS:

```
"C:\Program Files\Adobe\Acrobat 5.0\Reader\AcroRd32" /t pdf\%1
  "Canon LBP-800" "Canon LBP-800" "LPT1"
```

The single line needs to be rewritten for each individual printer, printer driver and port. If the port contains a slash (illegal!) then the default port is used for that printer.

The problem with the above is that Acrobat seems a little bit unreliable as to whether it will close after printing.

A smarter idea is to create an AutoIt script which will run AcroRd32 with the appropriate arguments, and then close it down after printing; the AutoIt script can handle all of the printing. So we first create *batprint.bat* thus:

```
batprint.au3 %1
```

And then create the invoked AutoIt script, along the lines of:

```
; AutoIt program to run AcroRd32, wait a while and then close the bugger!

 Dim $pdffile = $CmdLine[1];
 If WinExists ("Acrobat Reader") Then
    Sleep(3000);
    WinClose("Acrobat Reader" );
    WinWaitClose("Acrobat Reader" );
 EndIf
 Run('"C:\Program Files\Adobe\Acrobat 5.0\Reader\AcroRd32.exe" /t pdf\' _
    & $pdffile & ' "Canon LBP-800" "Canon LBP-800" "LPT1"');
 WinWaitActive("Acrobat Reader", "")
 Sleep(5000);                          ; sleep for 5s
 WinClose("Acrobat Reader" );          ; close Acrobat HMM?
```

Note that in AutoIt we continue a statement onto the next line with an underscore character. The above file will work rather variably depending on whether you're using a network printer or not, what version of Adobe Acrobat Reader you're using, and so forth. Here's a more general solution, which is slightly less pretty:

```
; AutoIt program to run AcroRd32 loading a file, then print, close.

 Dim $pdffile = $CmdLine[1];
 If WinExists ("Acrobat Reader") Then
    Sleep(3000);
    WinClose("Acrobat Reader" );
    WinWaitClose("Acrobat Reader" );
 EndIf
 Run('"C:\Program Files\Adobe\Acrobat 5.0\Reader\AcroRd32.exe"  pdf\' & $pdffile);
 WinWaitActive("Acrobat Reader", "");
 Send("!fp");
 Sleep(1000);
 Send("{ENTER}");
 ;; WinSetState("Acrobat Reader", "", @SW_MINIMIZE); ;; noo!
 Sleep(3000);                          ; sleep for 3s
 WinClose("Acrobat Reader" );          ; close Acrobat HMM?
 WinWaitClose("Acrobat Reader" );
```

This should work, but you have flashing windows opening and closing. Sending commands to the minimised Acrobat doesn't seem to work.

18.4 Print templates

The following template files are customised for our institution. The LaTeX is crude.

18.4.1 daily_report.tex

A simple overview of daily activity.

```
\documentclass[a4paper,11pt]{article}
% NB. use PdfLatex to create a PDF document from this file!
% This LaTeX file must be pre-processed, replacing variables with an appropriate v
\pagestyle{headings}
\usepackage{float}
\usepackage{graphicx}
\usepackage{fancyhdr}
\usepackage{longtable}
\usepackage{float}
\usepackage{times}
\usepackage{rotating}  % use this for landscape, but needs ++ tweaking!
\usepackage[ % dvipdfm,   %remove the dvipdfm reference for PdfLatex, Yap
            bookmarks=false,pdfborder=0,colorlinks=true,%
            linkbordercolor={1 1 1},%   % white border is invisible
            linkcolor=red]{hyperref}   % THIS PACKAGE MUST BE THE *LAST* ONE!
\begin{document}
% Copyright (C) Johan Michael van Schalkwyk, 2005--2007 (J van Schalkwyk)
% This code is released under the Gnu Public Licence (GPL), a copy of which can
% be obtained at:
% \href{http://www.gnu.org/copyleft/gpl.html}{http://www.gnu.org/copyleft/gpl.html
% Please note the conditions of this licence.
\pagestyle{plain}
\begin{sideways}
\begin{minipage}{1.65\textwidth}
\section*{Pain Team Daily report: $[TODAY]}
\subsection*{Breakdown of cases by ward}
\begin{table}[H]
{\small
\begin{tabular}{|l|l|$[FIXTABS]}
\hline
  \emph{Metric} & \emph{Total} & $[WARDNAMES] \\
\hline
 Number of patients &   $[ACTIVEPATIENTS]  \\
 Patients seen &   $[SEEN]                  \\
 Total epidurals &   $[EPITOTAL]            \\
 Epidurals started &   $[EPISTART]          \\
 Epidurals removed &   $[EPIEND]            \\
 Total PCAs &   $[PCATOTAL]                 \\
 PCAs started &   $[PCASTART]               \\
 PCAs removed &   $[PCAEND]                 \\
 Combined PCA+epidural &   $[BOTH]          \\
 Admissions &   $[ADMIT]                    \\
 Discharges &   $[DISCHARGE]                \\
\hline
\end{tabular}}
```

```
\end{table}

Patients not seen $[NOTSEEN]. Patients with neither PCA nor epidural:
$[NEITHER].
\end{minipage}
\end{sideways}

\newpage
\pagestyle{fancy}
\lhead{PAIN TEAM DAILY REPORT} \rhead{FOR DATE: $[TODAY]}
\section*{List of patients not seen}
$[UNSEENTABLE]
\subsection*{Disclaimer}
The SQL code used to extract the above data is still undergoing checking and refin
\lfoot{Printed: $[NOW]}
\rfoot{Printed by $[USERNAME]}
\end{document}
```

18.4.2 daily_unseen.tex

A component of the above table, referenced by UNSEENTABLE.

```
% This is a partial document, to be included in daily_report.tex
\begin{tabular}{|l|l|l|l|}
\hline
  \emph{Ward} & \emph{Surname} & \emph{Forename} & \emph{Hospital Number} \\
\hline
 $[UNTEXT]
\hline
\end{tabular}
```

18.4.3 monthly_report.tex

A crude overview of monthly activity. Needs some work.

```
\documentclass[a4paper,11pt]{article}
% NB. use PdfLatex to create a PDF document from this file!
% This LaTeX file must be pre-processed, replacing variables with an appropriate v
\pagestyle{headings}
\usepackage{float}
\usepackage{graphicx}
\usepackage{fancyhdr}
\usepackage{longtable}
\usepackage{float}
\usepackage{times}
\usepackage[ % dvipdfm,   %remove the dvipdfm reference for PdfLatex, Yap
            bookmarks=false,pdfborder=0,colorlinks=true,%
```

```
            linkbordercolor={1 1 1},%   % white border is invisible
            linkcolor=red]{hyperref}   % THIS PACKAGE MUST BE THE *LAST* ONE!
\begin{document}
% Copyright (C) Johan Michael van Schalkwyk, 2005--2008 (J van Schalkwyk)
% This code is released under the Gnu Public Licence (GPL), a copy of which can
% be obtained at:
% \href{http://www.gnu.org/copyleft/gpl.html}{http://www.gnu.org/copyleft/gpl.html
% Please note the conditions of this licence.
\setlength{\headheight}{15pt}
\pagestyle{fancy}
\lhead{PAIN TEAM REPORT} \rhead{FOR $[FIRSTDATE] to $[SECONDDATE]}
%\renewcommand{\section}{ }
% \thispagestyle{headings}
% \setcounter{secnumdepth}{0}
\section*{Summary statistics for the pain database}

These summary data cover the interval $[FIRSTDATESTAMP] to $[SECONDDATESTAMP].

\vskip 5mm

\begin{tabular}{|l|l|}
\hline
  \emph{Metric} & \emph{Number of patients} \\
\hline
 Number of patients &   $[ACTIVEPATIENTS]  \\
 Patients seen &   $[SEEN]                  \\
 Total epidurals &   $[EPITOTAL]            \\
 Epidurals started &   $[EPISTART]          \\
 Epidurals removed &   $[EPIEND]            \\
 Total PCAs &   $[PCATOTAL]                 \\
 PCAs started &   $[PCASTART]               \\
 PCAs removed &   $[PCAEND]                 \\
 Combined PCA+epidural &   $[BOTH]          \\
 Neither PCA nor epidural &   $[NEITHER]    \\
 Admissions &   $[ADMIT]                    \\
 Discharges &   $[DISCHARGE]                \\
\hline
\end{tabular}
\lfoot{Printed: $[NOW]}
\rfoot{Printed by $[USERNAME]}
\end{document}
```

18.4.4 template_summary.tex

A summary template for a single patient. Provides an overview of activity related to that patient. Fairly complex, with multiple included components.

```
\documentclass[a4paper,11pt]{article}
```

```
% NB. use PdfLatex to create a PDF document from this file!
% This LaTeX file must be pre-processed, replacing variables with an appropriate v
% Current $ variables are:
% INI, SURNAME, NHI, FORENAME, DOB, WT, GENDER, AGE, ASA, ALIVE, NOW,
% PAINVISITTABLE, OPERATIONTABLE, PAINALERTTABLE,
% PROBLEMITEMLIST, ORALRXLIST, OTHERAGENTLIST,
% GENPAINNOTE,
% TEMPLATERGN
% FINALCOMMENT
% We also include the files (potentially several copies):
%  template_ivpca.tex
%  template_rgn.tex
% ... WITH THEIR ASSOCIATED $ variables!
\pagestyle{headings}
\usepackage{float}
\usepackage{graphicx}
\usepackage{fancyhdr}
\usepackage{longtable}
\usepackage{float}
\usepackage{times}
\usepackage[ % dvipdfm,   %remove the dvipdfm reference for PdfLatex, Yap
            bookmarks=false,pdfborder=0,colorlinks=true,%
            linkbordercolor={1 1 1},%   % white border is invisible
            linkcolor=red]{hyperref}   % THIS PACKAGE MUST BE THE *LAST* ONE!
\begin{document}
% Copyright (C) Johan Michael van Schalkwyk, 2005--2007 (J van Schalkwyk)
% This code is released under the Gnu Public Licence (GPL), a copy of which can
% be obtained at:
% \href{http://www.gnu.org/copyleft/gpl.html}{http://www.gnu.org/copyleft/gpl.html
% Please note the conditions of this licence.
 \setlength{\oddsidemargin}{4mm}
 \setlength{\evensidemargin}{4mm}
 \setlength{\textwidth}{145mm}
 \setlength{\headheight}{15pt}
 \pagestyle{fancy}
\lhead{PAIN TEAM DISCHARGE SUMMARY}
\rhead{$[INI] $[SURNAME]: $[NHI]}
 \begin{figure}[H]
 \vskip -35mm
 \hskip 160mm
 \includegraphics{images/DischargeCR8850.png}
 \vspace{-200mm}
 \end{figure}
\vskip -95mm

 \begin{figure}[H]
 \vskip -10mm
 \includegraphics[width=38mm,height=22mm]{images/ADHBpoor.jpg}
```

```
 \end{figure}

 \begin{figure}[H]
 \vskip -20mm
 \hskip -20mm
 \includegraphics[width=9mm,height=67.5mm]{images/CR8850.png}
 \end{figure}
 \vskip -45mm

\begin{tabular}{llll}
 Patient: $[FORENAME] $[SURNAME]  & NHI: \textbf{$[NHI]}     & Weight: $[WT] & War
 Birth Date: $[DOB]               &    Gender: $[GENDER]     & Age: $[AGE] & \\
 Discharged: $[ENDDATE]           & Alive on discharge: $[ALIVE] & ASA rating: $[A
 \end{tabular}
\section*{Surgery}
$[TEMPLATEOP]
$[TEMPLATEVISIT]
$[TEMPLATEALERT]
$[PROBLEMITEMLIST]
\section*{Analgesic modalities}
$[GENPAINNOTE]
% e.g. 'Interventions in DCCM are generally not detailed here.'
$[TEMPLATERGN]
% MUST HAVE TYPE OF REGIONAL HERE, AND OPTION OF LISTING SEVERAL SEQUENTIAL REGION
$[TEMPLATEIVPCA]
% having each daily count for each drug is probably overkill!
$[ORALRXLIST]
% should we have daily dose total here for major agents such as Sevredol, Oxynorm,
$[OTHERAGENTLIST]
% in final version must trim out duplicates
\lfoot{Printed on $[TODAY]}
\rfoot{Printed for $[USERNAME]}
% \section*{Discharge comment}
% $[FINALCOMMENT]
% What about discharge referrals?
% [IDEALLY HAVE FINAL COMMENT AT BOTTOM]
\end{document}
```

18.4.5 template_visit.tex

A component of the preceding summary file, as are all of the following .TEX files.

```
% include this template in template_summary.tex.
\section*{Pain team visits/consultation}
\begin{table}[H] % 30/1/2008
\vskip -5mm
\begin{tabular}{|l|l|p{70mm}|l|}
\hline
```

```
\emph{When} & \emph{Time*} & \emph{Comments} & Seen\\
            & \emph{(min)} & & by: \\
\hline
$[PAINVISITTABLE]
\multicolumn{4}{|l|}{\footnotesize *Rounded up to nearest 5~min. Brief,
uncommented consultations are not documented here.} \\ \hline
\end{tabular}
\end{table}
```

18.4.6 template_alert.tex

Alerts.

```
% This partial image is included in template_summary.tex as TEMPLATEALERT:
\section*{Alerts, flags and pain scores}
% only create a line if at least one flag is set!
\begin{table}[H] % 30/1/2008
\vskip -8.5mm
\begin{longtable}{|l|l|l|l|l|}
\kill % try for single pass
\hline
\emph{Date} & \emph{‘Concern’} & \emph{PAIN: Rest} & \emph{Movement} & \emph{Adequ
$[PAINALERTTABLE]
\multicolumn{5}{|l|}{\footnotesize \textbf{+} indicates a ‘general concern’. Pain
\hline
\hline
\emph{Date} & \emph{Low BP} & \emph{Sedation} & \emph{Nausea} &  \emph{Bowels oper
$[PAINPROBLEMTBL]
\multicolumn{5}{|l|}{\footnotesize \textbf{Y} indicates ‘Yes’. } \\
\hline
\end{longtable}
\end{table}
```

18.4.7 template_op.tex

Operation details.

```
% This partial image is included in template_summary.tex as TEMPLATEOP:
\begin{table}[H] % 30/1/2008
\vskip -6.5mm
\begin{tabular}{|l|l|p{77mm}|}
\hline
\emph{Date} & \emph{Type} & \emph{Note} \\
\hline
$[OPERATIONTABLE]
\end{tabular}
\end{table}
```

18.4.8 template_rgn.tex

Regional anaesthesia.

```
% This document must be included within template_summary.tex as TEMPLATERGN
% variables to be filled in include:
% RGNRXTABLE, REGIONOBSTABLE,
% REGIONALTYPE, STARTRGN, STOPRGN, WHYSTOPRGN,
\subsection*{Regional anaesthesia}
\begin{table}[H] % 30/1/2008
\vskip -7mm
\begin{longtable}{|l|l|l|l|l|l|}
\kill
\hline
\multicolumn{3}{|l|}{Date started: $[STARTRGN]} &
\multicolumn{3}{|l|}{Date stopped: $[STOPRGN]} \\
\multicolumn{3}{|l|}{Type: $[REGIONALTYPE]} &
\multicolumn{3}{|l|}{Reason: $[WHYSTOPRGN]} \\
\hline
\multicolumn{6}{|l|}{Regional observations/problems:
{\footnotesize \textbf{-} indicates normality, \textbf{x} a concern, blank = unrec
$[TEMPLATERGNOBS]
$[TEMPLATERGNRX]
\end{longtable}
\end{table}
% Really need to have results of 24-hour check here.
```

18.4.9 template_rgn_obs.tex

Regional observations.

```
% this template should be included within template_rgn.tex
\hline
\emph{Date/time} & \emph{Pressure areas} & \emph{Motor} & \emph{Block} & \emph{Sit
$[REGIONOBSTABLE]
```

18.4.10 template_rgn_rx.tex

Regional management.

```
% this template should be included within template_rgn.tex
\hline
\emph{Date/time} & \emph{Mix} & \emph{Rate} & \emph{Top-ups} & \emph{PCA: doses} &
$[RGNRXTABLE]
```

18.4.11 template_ivpca.tex

Intravenous PCA.

```
% this is a partial file, to be included within template_summary.tex
% Current $ variables are:
%    IVPCARXTABLE,
% STARTIVPCA, STOPIVPCA, WHYSTOPIVPCA, IVPCAMIX
% BASALINFUDATA
\subsection*{Intravenous PCA}
\begin{table}[H] % 30/1/2008
\vskip -7.5mm
\begin{longtable}{|l|l|l|l|l|l|}
\kill
\hline
\multicolumn{3}{|l|}{Date started: $[STARTIVPCA]} &
\multicolumn{3}{|l|}{Date stopped: $[STOPIVPCA]} \\
\multicolumn{3}{|l|}{Mix: $[IVPCAMIX]} &
\multicolumn{3}{l|}{Reason: $[WHYSTOPIVPCA]} \\
\hline
$[TEMPLATEIVPCARX]
\end{longtable}
\end{table}
$[BASALINFUDATA]
% \vskip 1mm ~ \\
```

18.4.12 template_ivpca_rx.tex

Intravenous PCA therapy.

```
% this template must be included in template_ivpca.tex as TEMPLATEIVPCARX
\emph{Date/time} & \emph{Doses} & \emph{Attempts} & \emph{Total(mg)} & \emph{Bolus
$[IVPCARXTABLE]
```

18.5 Long report

This is a ‘comprehensive’ report covering cases admitted and discharged between two dates:

```
\documentclass[a4paper,11pt]{article}
% NB. use PdfLatex to create a PDF document from this file!
% This LaTeX file must be pre-processed, replacing variables with an appropriate v
\pagestyle{headings}
\usepackage{float}
\usepackage{graphicx}
\usepackage{fancyhdr}
\usepackage{longtable}
```

```
\usepackage{float}
\usepackage{times}
\usepackage{rotating}  % use this for landscape, but needs ++ tweaking!
\usepackage[ % dvipdfm,   %remove the dvipdfm reference for PdfLatex, Yap
            bookmarks=false,pdfborder=0,colorlinks=true,%
            linkbordercolor={1 1 1},%   % white border is invisible
            linkcolor=red]{hyperref}   % THIS PACKAGE MUST BE THE *LAST* ONE!
\begin{document}
% Copyright (C) Johan Michael van Schalkwyk, 2005--2007 (J van Schalkwyk)
% This code is released under the Gnu Public Licence (GPL), a copy of which can
% be obtained at:
% \href{http://www.gnu.org/copyleft/gpl.html}{http://www.gnu.org/copyleft/gpl.html
% Please note the conditions of this licence.
\pagestyle{plain}
\begin{sideways}
\begin{minipage}{1.65\textwidth}
\vskip -25mm
\section*{Pain Team Long report}
\subsection*{Breakdown of cases by ward}
This report covers patients admitted and then discharged between $[STARTSTAMP] and
$[ENDSTAMP].
\begin{table}[H]
% \hskip -25mm
{\small
\begin{tabular}{|l|l|$[FIXTABS]}
\hline
  \emph{Metric} & \emph{Total} & $[WARDNAMES] \\
\hline
 Patients &   $[TOTALCASES]  \\
 Epidurals &   $[EPITOTAL]            \\
 PCAs &   $[PCATOTAL]                \\
 Interscalene &  $[INTERSCALENE] \\
 Interpleural &  $[INTERPLEURAL] \\
 Extrapleural &  $[EXTRAPLEURAL] \\
 Paravertebral &  $[PARAVERTEBRAL] \\
 Femoral &  $[FEMORAL] \\
 Sciatic &  $[SCIATIC] \\
 Cardiac Dis. &  $[CARDIAC] \\
 Renal Dis.&  $[RENAL] \\
 Liver Dis.&  $[HEPATIC] \\
 Chr. pain &  $[CHRONICPAIN] \\
 Chr. opiates &  $[CHRONICOPIATES] \\
\hline
\end{tabular}}
\end{table}

‘Other’ includes patients in the Emergency Department, Admission Planning Unit,
Wards 96--98, Tamaki, Rakino, Remuera and Rangitoto wards.
```

```
\end{minipage}
\end{sideways}

\newpage
\pagestyle{fancy}
\lhead{PAIN TEAM LONG REPORT} \rhead{$[TODAY]}
\subsection*{Patient Encounters}
\begin{table}[H]
{ \small $[EPOCHTABLE] }
\end{table}
\subsection*{Types of Surgery}
\begin{table}[H]
{ \small $[SURGTABLE] }
\end{table}

The preceding table is based on automatic classification of types of surgery, and
therefore not be considered definitive, or even very reliable.

\subsection*{Visit Counts}
\begin{table}[H]
{ \small $[VISITTBL] }
\end{table}

The above table groups patients by number of visits from the Pain Team, in groups
visits, 5--9 visits and so forth, with the corresponding total time recorded as sp
Pain Team. Here's the breakdown for patients having five or fewer visits:

\begin{table}[H]
{ \small $[VISIT5TBL] }
\end{table}

\subsection*{Disclaimer}

The above represent automated SQL operations on largely raw data. Ideally the
entered data should be regularly 'sanitised', but time constraints have limited my
ability to perform this function.

\lfoot{Printed: $[NOW]}
\rfoot{Printed by $[USERNAME]}
\end{document}
```

18.6 Constants and things

Here's a tiny but important file which contains Perl 'constants'. It will be placed in the *data* subdirectory of the main painform directory (usually *painform*). Note that with Dogwagger 2.0 and below, there is no automatic creation of such directories,

which should already exist.

```
<PAINDATABASE>='PAIN04' % default database to connect to! (Ocelot)
<BASEX>='15' %
<BASEY>='20' %
<BASEW>='320' %
<BASEH>='320' %
<RED>='red' %
<WHITE>='white' %
<GREY>='lightgrey' %
<WARNABOVE>='8' %
<THREEDAYS>='7' % extend grace period to 7 days
<EXTDBCODE>='5' %
<LOCALDATADIR>='../Progra~1/Handspring/Level8/Backup' %
<EXPORTDIR>='/painform/export/' %
<IMPORTDIR>='/painform/import/' %
<PDAINSTALLBATFILE>='/painform/install2pda.bat' %
<PAINSYNCBATFILE>='/painform/export.bat' %
<BATFETCHFROMPDA>='/painform/fetchfrompda.bat' %
<BATWRITETOPDA>='/painform/write2pda.bat' %
<PAINEXPORTBATFILE>='/painform/moveover.bat' %
<XDBCONNECT>='Driver=SQL Server;Database=SaferSleep;Server=AHSL260;ServerName=IDAS
<XDBQUERY1>='select anaestheticid from anaesthetic where creationtime between $STA
<XDBQUERYEPIDURAL>='select AnaestheticId FROM ANAESTHETIC, EpiduralAnalgesia, OPER
<XDBREGIONALTYPE>='SELECT InsertionSite FROM Anaesthetic,EpiduralAnalgesia WHERE A
<XDBQUERYPCA>='SELECT AnaestheticID FROM ANAESTHETIC,PCA,OPERATINGROOM WHERE Anaes
<XDBDATA1>='select DOB, Sex, FirstName, LastName, NHI from patient, anaestheticpat
<XDBREGNL1>='select BlockNameId from RegionalBlock where AnaestheticId = $ANAESTHE
<XDBREGNL2>='select RegionalName from RegionalBlockName where BlockNameId = $BLOCK
<XDBDESCRIP>='select OperationDescription from Anaesthetic where AnaestheticId = $
<XDBGETID>='select NHI,DOB,Sex,FirstName,LastName from Patient, AnaestheticPatient
<XDBGETWARD>='select Description from Anaesthetic, Ward where Ward.WardId = Anaest
<XDBOPTIMES>='select CreationTime,CreationTime from Anaesthetic where AnaestheticI
<XDBGETASAWT>='select ASA,PatientWeight from AnaestheticPatient where AnaestheticI
<SIMPLEBACKUP>='simplebackup.bat' %
<USERMENUBACKGROUNDCOLOUR>='#BFEFFF' %
* The last line begins with a star.
% optional comments afterwards.
```

Important 'constants' are the name of the database, the (x,y) coordinates of the database menu displayed on the screen, as well as its width and height. There's also the option to play with colours. The WARNABOVE variable should largely be disregarded, as it's an atavistic remnant. The last relevant line of the file is identified because it starts with a star, and should always be present within the file. Lines may be commented out as in LaTeXwith an initial percentage sign.

A recently added constant (January 2006) is relevant to database synchronisation as described in Section 17: LOCALDATA. Note that we prefer UNIX-style

file pathnames, and use forward slashes and (under MS Windows) appropriately mangled directory names if they include spaces! It's a good idea to make the path name relative to the current directory from which the *pain2.pl* program will be running.[173]

Here's the simple backup batch file (for DOS/Windows):

```
echo Backing up database ...
copy c:\pain04\*.* p:\pain04\%1
```

Finally, here are two files which are provided as demonstration files only — the former should be generated, and the latter should ideally be manually configured (Both will however work *as is*). First, the generated file *XLOG.LOG*:

```
% A list of PDB files for later manual editing
xTABLE.PDB
xCOLUMN.PDB
xLIMIT.PDB
UIDS.PDB
PERSON.PDB
TYPEOFPROC.PDB
PROCESS.PDB
EPOCH.PDB
ACTROLE.PDB
ACTOR2.PDB
DRUGFORM.PDB
DRUG.PDB
DRUGUSAGE.PDB
RX.PDB
WARD.PDB
ROOM.PDB
BED.PDB
BADOBS.PDB
PERSDATA.PDB
MEDSCORE.PDB
SURGTYPE.PDB
SURGTYPEOB.PDB
SURGSITE.PDB
PAINSCORE.PDB
RXOBS.PDB
INFUSIONOBS.PDB
PCA.PDB
PCASETTINGS.PDB
```

[173] Because of the lack of a common mount point for MS Windows drives, you may encounter some pain in accessing different drives from the current one, but usually you will find that specifying the drive name, a colon and the relevant backslashes will work; for networking, Google [Universal Naming Convention] to see what you can do! Also check out [use File::Spec]

```
RGNOBS.PDB
COMMENT.PDB
ITEM.PDB
MENUITEMS.PDB
ICOLTABLE.PDB
FUN.PDB
NONEVENT.PDB
MEASURE.PDB
ISPROBLEM.PDB
UIDS.PDB
```

We have removed the generated file *XLOG.LST* as this should be generated manually. Remove all names of files such as *MENUITEMS.PDB* which are not involved in reporting of data back to the main database.[174]

```
% A list of PDB files: Manually revised 8-5-2005 21:10
PERSON.PDB
PROCESS.PDB
EPOCH.PDB
ACTOR2.PDB
DRUG.PDB
RX.PDB
BADOBS.PDB
PERSDATA.PDB
MEDSCORE.PDB
SURGTYPEOB.PDB
PAINSCORE.PDB
RXOBS.PDB
INFUSIONOBS.PDB
PCA.PDB
PCASETTINGS.PDB
RGNOBS.PDB
COMMENT.PDB
NONEVENT.PDB
MEASURE.PDB
ISPROBLEM.PDB
STOPPROC.PDB
UIDS.PDB
PAINERROR.PDB
*
```

Note that whenever a new table is added to our database, we must seriously consider whether *XLOG.LST* should be altered — usually the answer is 'yes'! *In*

[174] It's generally acceptable to use the following, but if one has restructured the database then it should of course be rewritten.

addition, if the table has a key in UIDS (which is usually the case) we must manually insert[175] an entry describing this column into the XCOLUMN table defined in *AnalgesiaDBpart1.tex*, an exacting task.

18.7 Standard CSV files

Here are the standard files. We have modified our CSV reader to ignore lines which start with %% as a mechanism of introducing comments into CSV files.[176]

18.7.1 ward.csv

```
ward,'swrdText',
21,ED,
22,APU,
31,31,
41,41,
46,46/42,
48,CVICU,
61,61,
62,62,
63,63,
64,64,
65,65,
66,66,
67,67,
68,68,
71,71,
72,72,
73,73,
74,74,
75,75,
76,76,
77,77,
78,78,
81,81,
82,DCCM,
83,83,
96,96,
97,97,
98,98,
10,Tamaki,
11,Rakino,
12,Remuera,
13,Rangi,
```

[175] At least, at present!

[176] There is no commenting standard for CSV, just as there is no meaningful CSV standard.

18.7.2 room.csv

For CVICU we allocate HDU patients to room ’7’ (4807)

```
room,ward,’srmtext’,
3100,31,?,
4100,41,?,
4800,48,?,
6100,61,?,
6200,62,?,
6300,63,?,
6400,64,?,
6500,65,?,
6600,66,?,
6700,67,?,
6800,68,?,
7100,71,?,
7200,72,?,
7300,73,?,
7400,74,?,
7500,75,?,
7600,76,?,
7700,77,?,
7800,78,?,
8100,81,?,
8200,82,?,
8300,83,?,
9600,96,?,
9700,97,?,
9800,98,?,
3101,31,1,
4101,41,1,
4801,48,1,
6101,61,1,
6201,62,1,
6301,63,1,
6401,64,1,
6501,65,1,
6601,66,1,
6701,67,1,
6801,68,1,
7101,71,1,
7201,72,1,
7301,73,1,
7401,74,1,
7501,75,1,
7601,76,1,
7701,77,1,
7801,78,1,
```

```
8101,81,1,
8201,82,1,
8301,83,1,
9601,96,1,
9701,97,1,
9801,98,1,
3102,31,2,
4102,41,2,
4802,48,2,
6102,61,2,
6202,62,2,
6302,63,2,
6402,64,2,
6502,65,2,
6602,66,2,
6702,67,2,
6802,68,2,
7102,71,2,
7202,72,2,
7302,73,2,
7402,74,2,
7502,75,2,
7602,76,2,
7702,77,2,
7802,78,2,
8102,81,2,
8202,82,2,
8302,83,2,
9602,96,2,
9702,97,2,
9802,98,2,
3103,31,3,
4103,41,3,
4803,48,3,
6103,61,3,
6203,62,3,
6303,63,3,
6403,64,3,
6503,65,3,
6603,66,3,
6703,67,3,
6803,68,3,
7103,71,3,
7203,72,3,
7303,73,3,
7403,74,3,
7503,75,3,
7603,76,3,
```

```
7703,77,3,
7803,78,3,
8103,81,3,
8203,82,3,
8303,83,3,
9603,96,3,
9703,97,3,
9803,98,3,
3104,31,4,
4104,41,4,
4804,48,4,
6104,61,4,
6204,62,4,
6304,63,4,
6404,64,4,
6504,65,4,
6604,66,4,
6704,67,4,
6804,68,4,
7104,71,4,
7204,72,4,
7304,73,4,
7404,74,4,
7504,75,4,
7604,76,4,
7704,77,4,
7804,78,4,
8104,81,4,
8204,82,4,
8304,83,4,
9604,96,4,
9704,97,4,
9804,98,4,
3105,31,5,
4105,41,5,
4805,48,5,
6105,61,5,
6205,62,5,
6305,63,5,
6405,64,5,
6505,65,5,
6605,66,5,
6705,67,5,
6805,68,5,
7105,71,5,
7205,72,5,
7305,73,5,
7405,74,5,
```

```
7505,75,5,
7605,76,5,
7705,77,5,
7805,78,5,
8105,81,5,
8205,82,5,
8305,83,5,
9605,96,5,
9705,97,5,
9805,98,5,
3106,31,6,
4106,41,6,
4806,48,6,
6106,61,6,
6206,62,6,
6306,63,6,
6406,64,6,
6506,65,6,
6606,66,6,
6706,67,6,
6806,68,6,
7106,71,6,
7206,72,6,
7306,73,6,
7406,74,6,
7506,75,6,
7606,76,6,
7706,77,6,
7806,78,6,
8106,81,6,
8206,82,6,
8306,83,6,
9606,96,6,
9706,97,6,
9806,98,6,
3107,31,7,
4107,41,7,
4807,48,HDU,
6107,61,7,
6207,62,7,
6307,63,7,
6407,64,7,
6507,65,7,
6607,66,7,
6707,67,7,
6807,68,7,
7107,71,7,
7207,72,7,
```

```
7307,73,7,
7407,74,7,
7507,75,7,
7607,76,7,
7707,77,7,
7807,78,7,
8107,81,7,
8207,82,7,
8307,83,7,
9607,96,7,
9707,97,7,
9807,98,7,
3108,31,8,
4108,41,8,
6108,61,8,
6208,62,8,
6308,63,8,
6408,64,8,
6508,65,8,
6608,66,8,
6708,67,8,
6808,68,8,
7108,71,8,
7208,72,8,
7308,73,8,
7408,74,8,
7508,75,8,
7608,76,8,
7708,77,8,
7808,78,8,
8108,81,8,
8208,82,8,
8308,83,8,
9608,96,8,
9708,97,8,
9808,98,8,
3109,31,9,
4109,41,9,
6109,61,9,
6209,62,9,
6309,63,9,
6409,64,9,
6509,65,9,
6609,66,9,
6709,67,9,
6809,68,9,
7109,71,9,
7209,72,9,
```

```
7309,73,9,
7409,74,9,
7509,75,9,
7609,76,9,
7709,77,9,
7809,78,9,
8109,81,9,
8209,82,9,
8309,83,9,
9609,96,9,
9709,97,9,
9809,98,9,
3110,31,10,
4110,41,10,
6110,61,10,
6210,62,10,
6310,63,10,
6410,64,10,
6510,65,10,
6610,66,10,
6710,67,10,
6810,68,10,
7110,71,10,
7210,72,10,
7310,73,10,
7410,74,10,
7510,75,10,
7610,76,10,
7710,77,10,
7810,78,10,
8110,81,10,
8210,82,10,
8310,83,10,
9610,96,10,
9710,97,10,
9810,98,10,
3111,31,11,
4111,41,11,
6111,61,11,
6211,62,11,
6311,63,11,
6411,64,11,
6511,65,11,
6611,66,11,
6711,67,11,
6811,68,11,
7111,71,11,
7211,72,11,
```

```
7311,73,11,
7411,74,11,
7511,75,11,
7611,76,11,
7711,77,11,
7811,78,11,
8111,81,11,
8211,82,11,
8311,83,11,
9611,96,11,
9711,97,11,
9811,98,11,
3112,31,12,
4112,41,12,
6112,61,12,
6212,62,12,
6312,63,12,
6412,64,12,
6512,65,12,
6612,66,12,
6712,67,12,
6812,68,12,
7112,71,12,
7212,72,12,
7312,73,12,
7412,74,12,
7512,75,12,
7612,76,12,
7712,77,12,
7812,78,12,
8112,81,12,
8212,82,12,
8312,83,12,
9612,96,12,
9712,97,12,
9812,98,12,
3113,31,13,
4113,41,13,
6113,61,13,
6213,62,13,
6313,63,13,
6413,64,13,
6513,65,13,
6613,66,13,
6713,67,13,
6813,68,13,
7113,71,13,
7213,72,13,
```

```
7313,73,13,
7413,74,13,
7513,75,13,
7613,76,13,
7713,77,13,
7813,78,13,
8113,81,13,
8213,82,13,
8313,83,13,
9613,96,13,
9713,97,13,
9813,98,13,
4600,46,?,
4601,46,1,
4602,46,2,
4603,46,3,
4604,46,4,
4605,46,5,
4606,46,6,
4607,46,7,
4608,46,8,
4609,46,9,
4610,46,10,
4611,46,11,
4612,46,12,
4613,46,13,
4614,46,14,
4615,46,15,
4616,46,16,
4617,46,17,
4618,46,18,
4619,46,19,
4620,46,20,
4621,46,21,
4622,46,22,
4623,46,23,
4624,46,24,
4625,46,25,
4626,46,26,
4627,46,27,
2100,21,?,
2101,21,1,
2102,21,2,
2103,21,3,
2104,21,4,
2105,21,5,
2106,21,6,
2107,21,7,
```

```
2108,21,8,
2109,21,9,
2110,21,10,
2111,21,11,
2112,21,12,
2113,21,13,
2114,21,14,
2115,21,15,
2116,21,16,
2117,21,17,
2118,21,18,
2119,21,19,
2120,21,20,
2121,21,21,
2122,21,22,
2123,21,23,
2124,21,24,
2125,21,25,
2126,21,26,
2127,21,27,
2200,22,?,
2201,22,1,
2202,22,2,
2203,22,3,
2204,22,4,
2205,22,5,
2206,22,6,
2207,22,7,
2208,22,8,
2209,22,9,
2210,22,10,
2211,22,11,
2212,22,12,
2213,22,13,
2214,22,14,
1000,10,?,
1001,10,1,
1002,10,2,
1003,10,3,
1004,10,4,
1005,10,5,
1006,10,6,
1007,10,7,
1008,10,8,
1009,10,9,
1010,10,10,
1011,10,11,
1012,10,12,
```

```
1013,10,13,
1014,10,14,
1015,10,15,
1016,10,16,
1017,10,17,
1018,10,18,
1019,10,19,
1020,10,20,
1021,10,21,
1022,10,22,
1023,10,23,
1024,10,24,
1025,10,25,
1026,10,26,
1027,10,27,
1028,10,28,
1029,10,29,
1030,10,30,
1031,10,31,
1032,10,32,
1033,10,33,
1034,10,34,
1035,10,35,
9614,96,14,
9615,96,15,
9616,96,16,
9617,96,17,
9814,98,14,
9815,98,15,
9816,98,16,
9817,98,17,
1100,11,?,
1101,11,1,
1102,11,2,
1103,11,3,
1104,11,4,
1105,11,5,
1106,11,6,
1107,11,7,
1108,11,8,
1109,11,9,
1110,11,10,
1111,11,11,
1112,11,12,
1113,11,13,
1200,12,?,
1201,12,1,
1202,12,2,
```

```
1203,12,3,
1204,12,4,
1205,12,5,
1206,12,6,
1207,12,7,
1208,12,8,
1209,12,9,
1210,12,10,
1211,12,11,
1212,12,12,
1213,12,13,
1300,13,?,
1301,13,1,
1302,13,2,
1303,13,3,
1304,13,4,
1305,13,5,
1306,13,6,
1307,13,7,
1308,13,8,
1309,13,9,
1310,13,10,
1311,13,11,
1312,13,12,
1313,13,13,
```

18.7.3 bed.csv

Even though every one of the seven hundred or so potential beds isn't detailed, the following file is still a biggie, as it has one generic bed per ward:

```
Bed,Room,'Sname',
650101,6501,65/1/1,
310000,3100,31/?--,
410000,4100,41/?--,
480000,4800,48/?--,
610000,6100,61/?--,
620000,6200,62/?--,
630000,6300,63/?--,
640000,6400,64/?--,
650000,6500,65/?--,
660000,6600,66/?--,
670000,6700,67/?--,
680000,6800,68/?--,
710000,7100,71/?--,
720000,7200,72/?--,
730000,7300,73/?--,
740000,7400,74/?--,
```

```
750000,7500,75/?--,
760000,7600,76/?--,
770000,7700,77/?--,
780000,7800,78/?--,
810000,8100,81/?--,
820000,8200,82/?--,
830000,8300,83/?--,
960000,9600,96/?--,
970000,9700,97/?--,
980000,9800,98/?--,
310100,3101,31/1--,
410100,4101,41/1--,
480100,4801,48/1--,
610100,6101,61/1--,
620100,6201,62/1--,
630100,6301,63/1--,
640100,6401,64/1--,
650100,6501,65/1--,
660100,6601,66/1--,
670100,6701,67/1--,
680100,6801,68/1--,
710100,7101,71/1--,
720100,7201,72/1--,
730100,7301,73/1--,
740100,7401,74/1--,
750100,7501,75/1--,
760100,7601,76/1--,
770100,7701,77/1--,
780100,7801,78/1--,
810100,8101,81/1--,
820100,8201,82/1--,
830100,8301,83/1--,
960100,9601,96/1--,
970100,9701,97/1--,
980100,9801,98/1--,
310200,3102,31/2--,
410200,4102,41/2--,
480200,4802,48/2--,
610200,6102,61/2--,
620200,6202,62/2--,
630200,6302,63/2--,
640200,6402,64/2--,
650200,6502,65/2--,
660200,6602,66/2--,
670200,6702,67/2--,
680200,6802,68/2--,
710200,7102,71/2--,
720200,7202,72/2--,
```

```
730200,7302,73/2--,
740200,7402,74/2--,
750200,7502,75/2--,
760200,7602,76/2--,
770200,7702,77/2--,
780200,7802,78/2--,
810200,8102,81/2--,
820200,8202,82/2--,
830200,8302,83/2--,
960200,9602,96/2--,
970200,9702,97/2--,
980200,9802,98/2--,
310300,3103,31/3--,
410300,4103,41/3--,
480300,4803,48/3--,
610300,6103,61/3--,
620300,6203,62/3--,
630300,6303,63/3--,
640300,6403,64/3--,
650300,6503,65/3--,
660300,6603,66/3--,
670300,6703,67/3--,
680300,6803,68/3--,
710300,7103,71/3--,
720300,7203,72/3--,
730300,7303,73/3--,
740300,7403,74/3--,
750300,7503,75/3--,
760300,7603,76/3--,
770300,7703,77/3--,
780300,7803,78/3--,
810300,8103,81/3--,
820300,8203,82/3--,
830300,8303,83/3--,
960300,9603,96/3--,
970300,9703,97/3--,
980300,9803,98/3--,
310400,3104,31/4--,
410400,4104,41/4--,
480400,4804,48/4--,
610400,6104,61/4--,
620400,6204,62/4--,
630400,6304,63/4--,
640400,6404,64/4--,
650400,6504,65/4--,
660400,6604,66/4--,
670400,6704,67/4--,
680400,6804,68/4--,
```

```
710400,7104,71/4--,
720400,7204,72/4--,
730400,7304,73/4--,
740400,7404,74/4--,
750400,7504,75/4--,
760400,7604,76/4--,
770400,7704,77/4--,
780400,7804,78/4--,
810400,8104,81/4--,
820400,8204,82/4--,
830400,8304,83/4--,
960400,9604,96/4--,
970400,9704,97/4--,
980400,9804,98/4--,
310500,3105,31/5--,
410500,4105,41/5--,
480500,4805,48/5--,
610500,6105,61/5--,
620500,6205,62/5--,
630500,6305,63/5--,
640500,6405,64/5--,
650500,6505,65/5--,
660500,6605,66/5--,
670500,6705,67/5--,
680500,6805,68/5--,
710500,7105,71/5--,
720500,7205,72/5--,
730500,7305,73/5--,
740500,7405,74/5--,
750500,7505,75/5--,
760500,7605,76/5--,
770500,7705,77/5--,
780500,7805,78/5--,
810500,8105,81/5--,
820500,8205,82/5--,
830500,8305,83/5--,
960500,9605,96/5--,
970500,9705,97/5--,
980500,9805,98/5--,
310600,3106,31/6--,
410600,4106,41/6--,
480600,4806,48/6--,
610600,6106,61/6--,
620600,6206,62/6--,
630600,6306,63/6--,
640600,6406,64/6--,
650600,6506,65/6--,
660600,6606,66/6--,
```

```
670600,6706,67/6--,
680600,6806,68/6--,
710600,7106,71/6--,
720600,7206,72/6--,
730600,7306,73/6--,
740600,7406,74/6--,
750600,7506,75/6--,
760600,7606,76/6--,
770600,7706,77/6--,
780600,7806,78/6--,
810600,8106,81/6--,
820600,8206,82/6--,
830600,8306,83/6--,
960600,9606,96/6--,
970600,9706,97/6--,
980600,9806,98/6--,
310700,3107,31/7--,
410700,4107,41/7--,
480700,4807,48/HDU-,
610700,6107,61/7--,
620700,6207,62/7--,
630700,6307,63/7--,
640700,6407,64/7--,
650700,6507,65/7--,
660700,6607,66/7--,
670700,6707,67/7--,
680700,6807,68/7--,
710700,7107,71/7--,
720700,7207,72/7--,
730700,7307,73/7--,
740700,7407,74/7--,
750700,7507,75/7--,
760700,7607,76/7--,
770700,7707,77/7--,
780700,7807,78/7--,
810700,8107,81/7--,
820700,8207,82/7--,
830700,8307,83/7--,
960700,9607,96/7--,
970700,9707,97/7--,
980700,9807,98/7--,
310800,3108,31/8--,
410800,4108,41/8--,
610800,6108,61/8--,
620800,6208,62/8--,
630800,6308,63/8--,
640800,6408,64/8--,
650800,6508,65/8--,
```

```
660800,6608,66/8--,
670800,6708,67/8--,
680800,6808,68/8--,
710800,7108,71/8--,
720800,7208,72/8--,
730800,7308,73/8--,
740800,7408,74/8--,
750800,7508,75/8--,
760800,7608,76/8--,
770800,7708,77/8--,
780800,7808,78/8--,
810800,8108,81/8--,
820800,8208,82/8--,
830800,8308,83/8--,
960800,9608,96/8--,
970800,9708,97/8--,
980800,9808,98/8--,
310900,3109,31/9--,
410900,4109,41/9--,
610900,6109,61/9--,
620900,6209,62/9--,
630900,6309,63/9--,
640900,6409,64/9--,
650900,6509,65/9--,
660900,6609,66/9--,
670900,6709,67/9--,
680900,6809,68/9--,
710900,7109,71/9--,
720900,7209,72/9--,
730900,7309,73/9--,
740900,7409,74/9--,
750900,7509,75/9--,
760900,7609,76/9--,
770900,7709,77/9--,
780900,7809,78/9--,
810900,8109,81/9--,
820900,8209,82/9--,
830900,8309,83/9--,
960900,9609,96/9--,
970900,9709,97/9--,
980900,9809,98/9--,
311000,3110,31/10--,
411000,4110,41/10--,
611000,6110,61/10--,
621000,6210,62/10--,
631000,6310,63/10--,
641000,6410,64/10--,
651000,6510,65/10--,
```

```
661000,6610,66/10--,
671000,6710,67/10--,
681000,6810,68/10--,
711000,7110,71/10--,
721000,7210,72/10--,
731000,7310,73/10--,
741000,7410,74/10--,
751000,7510,75/10--,
761000,7610,76/10--,
771000,7710,77/10--,
781000,7810,78/10--,
811000,8110,81/10--,
821000,8210,82/10--,
831000,8310,83/10--,
961000,9610,96/10--,
971000,9710,97/10--,
981000,9810,98/10--,
311100,3111,31/11--,
411100,4111,41/11--,
611100,6111,61/11--,
621100,6211,62/11--,
631100,6311,63/11--,
641100,6411,64/11--,
651100,6511,65/11--,
661100,6611,66/11--,
671100,6711,67/11--,
681100,6811,68/11--,
711100,7111,71/11--,
721100,7211,72/11--,
731100,7311,73/11--,
741100,7411,74/11--,
751100,7511,75/11--,
761100,7611,76/11--,
771100,7711,77/11--,
781100,7811,78/11--,
811100,8111,81/11--,
821100,8211,82/11--,
831100,8311,83/11--,
961100,9611,96/11--,
971100,9711,97/11--,
981100,9811,98/11--,
311200,3112,31/12--,
411200,4112,41/12--,
611200,6112,61/12--,
621200,6212,62/12--,
631200,6312,63/12--,
641200,6412,64/12--,
651200,6512,65/12--,
```

```
661200,6612,66/12--,
671200,6712,67/12--,
681200,6812,68/12--,
711200,7112,71/12--,
721200,7212,72/12--,
731200,7312,73/12--,
741200,7412,74/12--,
751200,7512,75/12--,
761200,7612,76/12--,
771200,7712,77/12--,
781200,7812,78/12--,
811200,8112,81/12--,
821200,8212,82/12--,
831200,8312,83/12--,
961200,9612,96/12--,
971200,9712,97/12--,
981200,9812,98/12--,
311300,3113,31/13--,
411300,4113,41/13--,
611300,6113,61/13--,
621300,6213,62/13--,
631300,6313,63/13--,
641300,6413,64/13--,
651300,6513,65/13--,
661300,6613,66/13--,
671300,6713,67/13--,
681300,6813,68/13--,
711300,7113,71/13--,
721300,7213,72/13--,
731300,7313,73/13--,
741300,7413,74/13--,
751300,7513,75/13--,
761300,7613,76/13--,
771300,7713,77/13--,
781300,7813,78/13--,
811300,8113,81/13--,
821300,8213,82/13--,
831300,8313,83/13--,
961300,9613,96/13--,
971300,9713,97/13--,
981300,9813,98/13--,
460000,4600,46/?--,
460100,4601,46/1--,
460200,4602,46/2--,
460300,4603,46/3--,
460400,4604,46/4--,
460500,4605,46/5--,
460600,4606,46/6--,
```

```
460700,4607,46/7--,
460800,4608,46/8--,
460900,4609,46/9--,
461000,4610,46/10--,
461100,4611,46/11--,
461200,4612,46/12--,
461300,4613,46/13--,
461400,4614,46/14--,
461500,4615,46/15--,
461600,4616,46/16--,
461700,4617,46/17--,
461800,4618,46/18--,
461900,4619,46/19--,
462000,4620,46/20--,
462100,4621,46/21--,
462200,4622,46/22--,
462300,4623,46/23--,
462400,4624,46/24--,
462500,4625,46/25--,
462600,4626,46/26--,
462700,4627,46/27--,
210000,2100,21/?--,
210100,2101,21/1--,
210200,2102,21/2--,
210300,2103,21/3--,
210400,2104,21/4--,
210500,2105,21/5--,
210600,2106,21/6--,
210700,2107,21/7--,
210800,2108,21/8--,
210900,2109,21/9--,
211000,2110,21/10--,
211100,2111,21/11--,
211200,2112,21/12--,
211300,2113,21/13--,
211400,2114,21/14--,
211500,2115,21/15--,
211600,2116,21/16--,
211700,2117,21/17--,
211800,2118,21/18--,
211900,2119,21/19--,
212000,2120,21/20--,
212100,2121,21/21--,
212200,2122,21/22--,
212300,2123,21/23--,
212400,2124,21/24--,
212500,2125,21/25--,
212600,2126,21/26--,
```

```
212700,2127,21/27--,
220000,2200,22/?--,
220100,2201,22/1--,
220200,2202,22/2--,
220300,2203,22/3--,
220400,2204,22/4--,
220500,2205,22/5--,
220600,2206,22/6--,
220700,2207,22/7--,
220800,2208,22/8--,
220900,2209,22/9--,
221000,2210,22/10--,
221100,2211,22/11--,
221200,2212,22/12--,
221300,2213,22/13--,
221400,2214,22/14--,
100000,1000,10/?--,
100100,1001,10/1--,
100200,1002,10/2--,
100300,1003,10/3--,
100400,1004,10/4--,
100500,1005,10/5--,
100600,1006,10/6--,
100700,1007,10/7--,
100800,1008,10/8--,
100900,1009,10/9--,
101000,1010,10/10--,
101100,1011,10/11--,
101200,1012,10/12--,
101300,1013,10/13--,
101400,1014,10/14--,
101500,1015,10/15--,
101600,1016,10/16--,
101700,1017,10/17--,
101800,1018,10/18--,
101900,1019,10/19--,
102000,1020,10/20--,
102100,1021,10/21--,
102200,1022,10/22--,
102300,1023,10/23--,
102400,1024,10/24--,
102500,1025,10/25--,
102600,1026,10/26--,
102700,1027,10/27--,
102800,1028,10/28--,
102900,1029,10/29--,
103000,1030,10/30--,
103100,1031,10/31--,
```

```
103200,1032,10/32--,
103300,1033,10/33--,
103400,1034,10/34--,
103500,1035,10/35--,
961400,9614,96/14--,
961500,9615,96/15--,
961600,9616,96/16--,
961700,9617,96/17--,
981400,9814,98/14--,
981500,9815,98/15--,
981600,9816,98/16--,
981700,9817,98/17--,
110000,1100,10/?--,
110100,1101,10/1--,
110200,1102,10/2--,
110300,1103,10/3--,
110400,1104,10/4--,
110500,1105,10/5--,
110600,1106,10/6--,
110700,1107,10/7--,
110800,1108,10/8--,
110900,1109,10/9--,
111000,1110,10/10--,
111100,1111,10/11--,
111200,1112,10/12--,
111300,1113,10/13--,
120000,1200,10/?--,
120100,1201,10/1--,
120200,1202,10/2--,
120300,1203,10/3--,
120400,1204,10/4--,
120500,1205,10/5--,
120600,1206,10/6--,
120700,1207,10/7--,
120800,1208,10/8--,
120900,1209,10/9--,
121000,1210,10/10--,
121100,1211,10/11--,
121200,1212,10/12--,
121300,1213,10/13--,
130000,1300,10/?--,
130100,1301,10/1--,
130200,1302,10/2--,
130300,1303,10/3--,
130400,1304,10/4--,
130500,1305,10/5--,
130600,1306,10/6--,
130700,1307,10/7--,
```

```
130800,1308,10/8--,
130900,1309,10/9--,
131000,1310,10/10--,
131100,1311,10/11--,
131200,1312,10/12--,
131300,1313,10/13--,
```

18.7.4 proctype.csv

A list of process types, restricted to codes of 1000 and more (the remaining processes were specified as SQL in the file *AnalgesiaDBpart1.tex*!

```
proctype,'rptnature',
1000,substantial renal dysfunction,
1010,heart failure/dysfunction,
1020,respiratory failure/dysfunction,
1030,liver dysfunction,
1040,anticoagulation,
1050,coagulopathy,
1060,chronic pain,
1070,chronic opiate use,
1080,allergy alert,
1090,CNS dysfunction,
1100,PM observation,
1110,Sedation level,
1120,BP monitoring,
1130,Monitoring for nausea,
1140,Bowel opening,
```

18.7.5 person.csv

Here we populate our database with nurses and doctors. First, references to people via the PERSON table. This table is very simple, as *observations* on names, gender etc are stored as such — observations. Here we insert the first person who is the default organiser of everything, as well as five nurses and five doctors.

Note that the initial setup of personnel is 'reserved' in the sense that these initial persons (up to 1000) can be given ID numbers between 0 and 1000. Subsequently, such people are allocated sequential numbers based on the primary key generator table UIDS, where values start at 1000.

In our CSV importation process, leading and trailing blanks are suppressed, so the following columns are fine (and made more legible).

```
person,pstatus,pflags,
100,10,0,
101,10,0,
```

```
102,10,0,
103,10,0,
104,10,0,
105,10,0,
106,10,0,
107,10,0,
108,10,0,
109,10,0,
110,10,0,
111,10,0,
112,10,0,
113,10,0,
114,10,0,
115,10,0,
116,10,0,
117,10,0,
118,10,0,
119,10,0,
120,10,0,
121,10,0,
122,10,0,
123,10,0,
124,10,0,
1,10,0,
93,6,0,
94,6,0,
95,6,0,
96,6,0,
97,6,0,
2,0,0,
3,0,0,
4,0,0,
5,0,0,
```

The 'virtual persons' 2, 3 and 4 correspond to institutions: TARPS, and in-patient and out-patient palliative care respectively. They have corresponding processes and epochs, and even persdata entries! On 2007-11-8 we finally added SaferSleep as an entity, with ID code 5.

We need processes associated with each person, and an observation on each such process to which we attach personal data via the PERSDATA table. The PROCESS type is always type 1. The four vital fields in the PROCESS field are the process type, the person to whom the process refers, the planner of the process (Here we use the generic person 1), and the primary process key.

We have conveniently set the process key and the person key to the same values in the following table, but this will otherwise not be the case. (Note that 'persons' 12–14 are virtual persons, representing transfer of patients to other services)!

```
process,Person,ProcType,rPlanner,
100,100,1,1,
101,101,1,1,
102,102,1,1,
103,103,1,1,
104,104,1,1,
105,105,1,1,
106,106,1,1,
107,107,1,1,
108,108,1,1,
109,109,1,1,
110,110,1,1,
111,111,1,1,
112,112,1,1,
113,113,1,1,
114,114,1,1,
115,115,1,1,
116,116,1,1,
117,117,1,1,
118,118,1,1,
119,119,1,1,
120,120,1,1,
121,121,1,1,
122,122,1,1,
123,123,1,1,
124,124,1,1,
93,93,1,1,
94,94,1,1,
95,95,1,1,
96,96,1,1,
97,97,1,1,
2,2,1,1,
3,3,1,1,
4,4,1,1,
5,5,1,1,
```

Now for the EPOCH(ervation) table, with an observation on each process. We again conveniently equate the keys. The person referred to in EPOCH is the observer.

```
epoch,Process,Person,
100,100,1,
101,101,1,
102,102,1,
103,103,1,
104,104,1,
105,105,1,
106,106,1,
```

```
107,107,1,
108,108,1,
109,109,1,
110,110,1,
111,111,1,
112,112,1,
113,113,1,
114,114,1,
115,115,1,
116,116,1,
117,117,1,
118,118,1,
119,119,1,
120,120,1,
121,121,1,
122,122,1,
123,123,1,
124,124,1,
93,93,1,
94,94,1,
95,95,1,
96,96,1,
97,97,1,
2,2,1,
3,3,1,
4,4,1,
5,5,1,
```

18.7.6 persdata.csv

And associated personal data (a file that we've truncated, for obvious reasons).

```
persdata,epoch,'pdosurname','pdoforename','pdohospno',pdoperson,pdogender
121,121,Van Schalkwyk,Jo,X12345678,1,2
2,2,TARPS, ,T001,2,0
3,3,Palliative Care,Inpatient,T002,3,0
4,4,Palliative Care,Outpatient,T003,4,0
5,5,SaferSleep, ,T004,5,0
```

18.7.7 pharm.csv

The table PHARM.csv has been removed.

18.7.8 drug.csv

. . . DRUG names and formulations. Note that certain groups of drugs *must* have primary key (drug) values within certain ranges, notably oral analgesics (2048–

5119), anti-emetics (19456–20479), and 'special infusions' (22528–100351). See AnalgesiaDB2 for details in the section describing the function **ListDrugs**. Other drugs should be *outside* these ranges!

```
drug,'dtrade',drugform,
20,PR morphine,20,
21,PR paracetamol,20,
22,PR Voltaren,20,
50,TTS fentanyl,30,
100,Bupiv/Fentanyl Std Mix,1,
101,Bupivacaine 0.25% plain,1,
102,Bupivacaine 0.125% plain,1,
103,Pethidine 5mg/ml,1,
120,Morpine 1mg/ml,4,
121,Tramadol 10mg/ml,4,
122,Fentanyl 10microg/ml,4,
123,Pethidine 10mg/ml,4,
124,Methadone,4,
1120,Morphine SC,9,
1122,Fentanyl SC,9,
1124,Methadone SC,9,
2049,Paracetamol,10,
2050,Sevredol,10,
2051,Morphine syrup,17,
2052,M-Eslon,13,
2053,LA Morph,13,
2054,Kapanol,13,
2080,Oxynorm,13,
2081,Oxycontin,10,
2100,Methadone,10,
2110,Tramadol,10,
2111,SR Tramadol,13,
2130,Gabapentin,13,
2150,Amitriptyline,13,
2151,Nortriptyline,13,
2170,Diclofenac SR,13,
2171,Ibuprofen,10,
2172,Tenoxicam,13,
2200,Mexiletine,13,
5000,IV Lignocaine 1mg/ml,4,
5001,IV Ketamine 1mg/ml,4,
5002,IV calcitonin 1U/ml,4,
5100,IV tenoxicam,4,
5110,IV parecoxib,4,
5125,Celebrex,13,
19457,IV cyclizine,4,
19459,IV metoclopramide,4,
19460,IV prochlorperazine,4,
```

```
19461,IV ondansetron,4,
19463,IV droperidol,4,
19464,IV dexamethasone,4,
19466,IV aprepitant,4,
19969,PO cyclizine,10,
19971,PO metoclopramide,10,
19973,PO ondansetron,10,
20479,scopolamine patch,30,
29697,clonidine patch,30,
```

The formulations are 1 epidural, 4 IV, 9 SC, 10/13/17 oral (tabs/caps/syrup), 20 rectal, and 30 transdermal. We previously distinguished between IV and PCA, but our more mature code is to make this distinction using the DRUGUSAGE table:

```
drugusage,Drug,drUsage,
1,120,1,
2,121,1,
3,122,1,
4,123,1,
5,121,2,
6,122,2,
7,123,2,
8,5000,3,
9,5001,3,
10,5002,3,
22,5001,2,
23,2049,5,
24,2050,5,
25,2051,5,
26,2052,5,
27,2053,5,
28,2054,5,
29,2080,5,
30,2081,5,
31,2100,5,
32,2110,5,
33,2111,5,
34,2130,5,
35,2150,5,
36,2151,5,
37,2170,5,
38,2171,5,
39,2172,5,
40,2200,5,
41,120,2,
11,19457,4,
12,19459,4,
```

```
13,19460,4,
14,19461,4,
15,19463,4,
16,19464,4,
17,19466,4,
18,19969,4,
19,19971,4,
20,19973,4,
21,20479,4,
50,5100,2,
51,5110,2,
52,1120,3,
53,1122,3,
54,1124,3,
55,5125,5,
```

The coding (as per *AnalgesiaDBpart1.tex*) is 1=PCA, 2=IV bolus, 3=special infusion, 4=anti-nauseant, 5=oral analgesic

18.7.9 surgtype.csv

Generic surgery types. The code 1 for ‘general’ is already represented in the database (inserted in *AnalgesiaDBpart1.tex* so is not repeated here.

```
surgtype,'cttext',
49,endoscopy,
99,plastics,
149,orthopaedic,
199,neurosurg.,
249,ophthalmic,
299,dental,
399,ORL,
449,cardiac,
499,thoracic,
599,vascular,
649,hepatobil.,
699,colorectal,
799,urol./renal,
899,gynaecol.,
999,obstetrics,
```

19 Some debugging

The biggest problem with C (and C++) is probably inappropriate memory access. Under PalmOS, the second biggest is orphaned chunks of memory left behind after a program has finished executing. In order to trap these errors, we take all suspicious chunks and label them with an identifying number (which we keep) when we request them using xNew. This ID number can be retrieved, flagged and kept when we Delete the chunk.

The library routine ErrorWrite is used to write such ID numbers to a cyclical buffer called CYCERROR. We can then export this file from the PDA (stored as *CYCERROR.PDB* on the desktop machine) and match up New and Deleted codes.

Here's a clumsy Perl program (CycMatch.pl) to do just this matching.

19.1 An introduction to CycMatch

This program performs the following steps

1. Locate *CYCERROR.PDB* in the current directory (usually */painform*);
2. Move to the offset of the start of the data, which should be 0x5C;
3. Read in pairs of hexadecimal digits, taking each pair as a big-endian number ('opening' number) and storing each such 16-bit number in an array;
4. Match *closure* numbers, written when a memory block is freed, with the opening numbers which were written when the block was opened. A closure number has the highest bit set, for example 0x8005 is the closure number corresponding to the opening number 0x0005.
5. At the end, write unmatched opening numbers — these blocks will still have been left open.

If we encounter a closure number without an opening number, we signal this anomaly. We have stolen much of the following code from our SyncOneFile routine.

```
#!/usr/local/bin/perl -w

  my $pdbok = 1;
  my ($hPDB);
  open ($hPDB, "CYCERROR.PDB") or $pdbok = 0; #hPDB is handle
  if (! $pdbok)
     { die "Could not open CYCERROR.PDB :$!\n";
```

```
    };
  binmode ($hPDB);
  my ($keepterm) = $/; # retain
  undef $/;            # undefine
  my ($wholefile) = <$hPDB>; # slurp
  $/ = $keepterm;      # restore
  close ($hPDB);

  my $flen = length $wholefile;
  print SYNCFILE " Size=$flen";

  my $howfar;
  $howfar = 0x5C; # HARD CODED START OFFSET ??
  my $e;
  my @ENTRIES;  # ugly global
  my $total = 0;

  while ($howfar < $flen)
    { $e = &GetBeWord ($wholefile, $howfar);
      # read big-endian word from current offset $howfar

      if ($e < 32768)
         { # print "$i\n";
           if ($e != 0x3F3F) # signals junk
              { push(@ENTRIES, $e);
              } else
              { $howfar = $flen; # force end
              };
           $total ++;
         } else
         { $e &= 0x7FFF;
           # print "<$e>";
           $r = &ExciseFirstMatch($e); # does this preserve $_ ?
           if (! $r)
              { print "\n UNKNOWN! $e";
              };
         };
      $howfar += 2;
    };

#print "\n OUTCOME WAS [@ENTRIES]";
  my $n;
  $n = 1 + $#ENTRIES;

  print "\n There were $total entries. OUTCOME:";
  if ($n < 1)
     { print " Perfect match. Hooray.";
     } else
```

```
     { if ($n == 1)
          { print " There was one bad match: ";
          } else
          { print " There were $n bad matches: \n";
          };
     };

  my $i;
  my $h;
  foreach $i (@ENTRIES)
    {
      $h = sprintf ("%X", $i); # convert to hex
      # another option would be
      #   $hex = unpack("H*", pack("N", $i));
      print "$i: <$h>; ";
    };

end;
```

When we go through the remaining array of unmatched ENTRIES in the above, we print each number followed by its hex value. Here's the clumsy ExciseFirstMatch routine which as the name suggests clips out a single matching item:

```
sub ExciseFirstMatch
{ my ($itm);
     ($itm) = @_;

  my ($cnt);
  $cnt = 0;

  foreach $m (@ENTRIES)
    {
      last if ($m == $itm); # terminate on match
      $cnt ++;
    };
  if ($cnt <= $#ENTRIES)     # unless not found
     { splice (@ENTRIES, $cnt, 1);
      # print "\n DEBUG: $itm excised from [@ENTRIES]";
      return 1;
     }
  return (0);
}
```

We return 0 if no matching element was found. GetBeWord is as before.

```
sub GetBeWord
```

```
{ my ($wholefile, $offs);
     ($wholefile, $offs)=@_;

  my $i;
  $i = unpack ("x$offs n", $wholefile);

  return ($i);
}
```